



中国科学技术大学
University of Science and Technology of China

计算系统概论A

Introduction to Computing Systems
(CS1002A.02)

Chapter 1-1 Why Take This Course?

计算机科学与技术学院
School of Computer Science and Technology



- 1 Course and Crew
- 2 What's the difference between "Big" and "Small"?
- 3 Why Take This Course?
- 4 Great Ideas in Computing Systems
- 5 What' s This Course All About ?
- 6 Summary



- 1 **Course and Crew**
- 2 What's the difference between "Big" and "Small"?
- 3 Why Take This Course?
- 4 Great Ideas in Computing Systems
- 5 What' s This Course All About ?
- 6 Summary

第一次开课: 2011年,暑期小学期

■ 2011年,暑期小学期

- 全校160人选修,10级计算机英才班必修
- 主讲: Yale Patt, 中国科大客座教授, 安虹教授
- 助教: 8名研究生。孙荪 (USTC Ph.D, 华为), 张海博(PennState Ph.D), 汤旭龙(PennState Ph.D), 魏学超(北大 Ph.D), 王涛, 吴石磊, 孙公瑾, 冷鹏



第一次开课：2011年,暑期小学期



中国科大《计算机系统概论(H)》课程



课程开设情况

■ 2012年,秋季学期

- 计算机学院48人选修,11级计算机英才班必修
- 主讲:安虹教授
- 助教: 张海博, 王涛

■ 2013年,秋季学期

- 计算机学院47人选修,12级计算机英才班必修
- 主讲:安虹教授
- 助教: 张海博, 程亦超, 彭毅

■ 2014年,秋季学期

- 计算机学院57人选修,13级计算机英才班必修
- 主讲: 安虹教授
- 助教: 彭毅, 邱晓杰, 迟孟贤

■ 2015年,秋季学期

- 计算机学院65人选修,14级计算机英才班必修
- 主讲:安虹教授
- 助教: 迟孟贤, 金旭

■ 2016年,秋季学期

- 计算机学院65人选修,15级计算机英才班必修
- 主讲:安虹教授
- 助教: 苏志超, 冯诗影

■ 2017年,秋季学期

- 计算机学院94人选修,16级计算机英才班必修
- 主讲:安虹教授
- 助教: 苏志超, 冯诗影, 徐青青

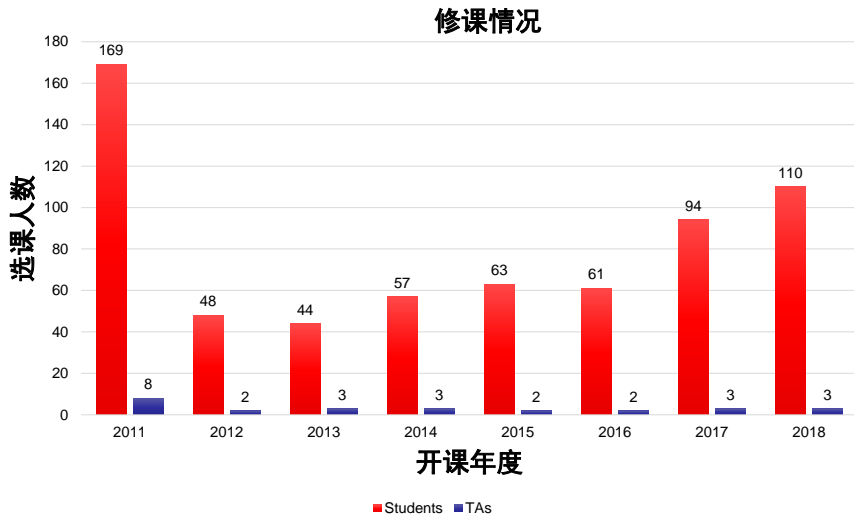
■ 2018年,秋季学期

- 全校110人选修,17级计算机英才班必修
- 主讲:安虹教授
- 助教:武铮, 金旭, 邓静恒, 张子豫

■ 2019年,秋季学期

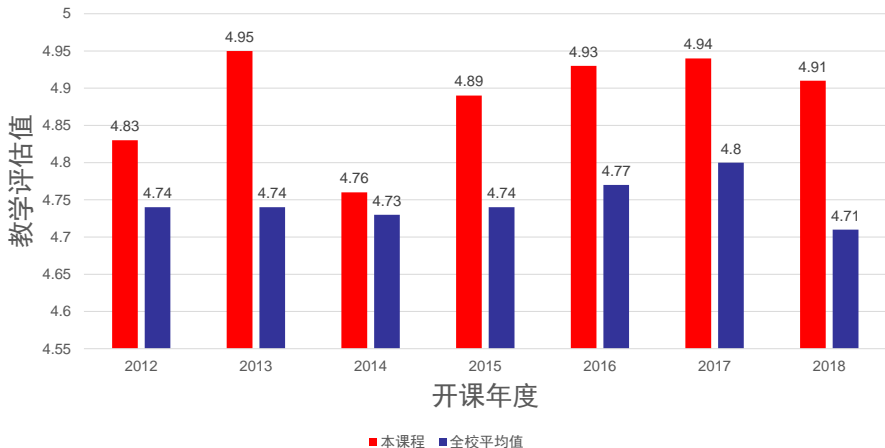
- 全校83人选修,18级计算机英才班必修
- 主讲:安虹教授
- 助教:张子豫,姜庆彩,许乐

选课情况



教学评估

教学评估



计算机学院第一门国家级一流本科课程(2020年)



CECC2019 中国·厦门 2019.12.6-8

2019中国计算机教育大会

Computer Education Conference of China
新时代·新计算·新教育

报告专家：安虹

教授，博士生导师，安徽省教学名师，现任中国科学技术大学计算机科学与技术学院先进计算机系统结构实验室主任，计算机系统结构专业学位点负责人，学生高寒院团新教育委员会主任，本科生系统结构课程组组长，教育基础学科交叉专业培养计划研究课题负责人，获得过国家级科技进步二等奖、国家教学成果二等奖、中科院杰出科技成就奖（首届）、中科院科技进步二等奖、中科院教书育人成果一等奖、安徽省教学成果一等奖、中国科大教学成果一等奖、中科院李兆华优秀青年教师奖、宝钢优秀教师奖【两次】、华为优秀教师奖、中国科大平基金教育奖、中国科大西区精神教学成果奖、安徽省教学名师、安徽省优秀教师等荣誉和荣誉，指导33支队伍参加国内外大学生竞赛，其中一等奖、金奖或第一名10次，二等奖、银奖或第二名7次，三等奖、铜奖或第三名7次，特等奖7次，在国赛影响力最大的SC16 SCC赛项决赛中，成为有史以来首个包揽全部奖项的冠军队。

二维码

计算机专业教学指导委员会

《上好一门计算机系统入门课》

物联网和云计算、大数据和人工智能等新兴计算机应用领域推动了计算机学科的迅猛发展，使得与计算机相关的学科和研究领域不断增加，各种新型计算技术和应用层出不穷；与此同时，计算机学科不断与其它学科交叉，在完善自身发展理论和技术的同时，也不断地拓宽计算机的应用领域，引发思维的深刻变革。今天的计算机学科已经越来越像数学和物理等学科一样，成为高等学校本科生通识教育的一部分。面对这种变化形势，对当代计算机系统全貌和计算本质的完整了解，不仅对计算机专业的学生是必需的，而且对众多计算机交叉专业的学生也是必需的。因此，如何让更多的低年级本科生、包括非计算机专业的学生，尽早地理解计算机系统最核心的基本概念，在有限课时迅速理清和抽象的计算思维方法，掌握计算机系统的软件协同设计思想，初步接触指令、数据流、线程和进程并行技术，为深入计算机专业后续课程增强牵引力和打下坚实的基础，是当代计算机教育值得重点关注的课题。本报告从中国科大计算机学院为全校低年级本科生开设的《计算机系统概论》课程的教学实践出发，对比分析美国一流高校计算机专业的同类课程，探讨计算机本科专业四年有限课时如何规划课程也是该教学内容的更加基础和必要前提？课程需要经常更新和如何更新？设计什么样的计算基础课程和核心课程体系能够更好地适应迅猛发展的计算技术变革，满足前沿交叉学科人才培养的需要。

计算机专业教学指导委员会

Course crew: Instructors

| Instructor | Classroom | Course time | E-mail | Office | Phone |
|---------------|------------------------------|---------------------|--|---|--------------------|
| Hong An | 3A112, CS1002A.01 | Monday 8,9,10 | han@ustc.edu.cn | Room 1409, Sci. & Lab Building(west) | 13500507406 |
| Hui Zhang | 3A112, CS1002A.02 | Friday 3,4,5 | fzhh@ustc.edu.cn | Room 409,High Performance Computing Center(east) | 13956969596 |
| Fuyou Miao | 3B102, CS1002A.03 | Tuesday 3,4,5 | mfy@ustc.edu.cn | Room 517, Electronic Teaching Building #3(west) | 13866166896 |

ICS Course 2021

Information HW Lab Exam Download Discuss

Introduction to Computing Systems

From bits and gates to C and beyond

What we will do
Our intent is to introduce you to the world of computing, give you a stronger foundation for later courses.

Motivated bottom-up approach
Advocated by Yale N. Patt. Build your knowledge bottom-up. Learn one thing building on what you already know step by step.

About LC-3
LC-3 is a computer simulator which captures the important structures of a modern computer, while keeping it simple enough to allow full understanding.

About RISC-V(new)
RISC-V is an open-source hardware instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles.

课程主页: <http://acsa.ustc.edu.cn/ics/>

课程群

计算机系统概论课程...



该二维码 7 天内 (9/13前)有效 [更改有效期](#)

该群组为 计算机系统概论课程 内部群，仅限
内部成员扫码加入

分享

保存图片

<http://ics-ustc.feishu.cn/>

中国移动 5.28K/s

下午4:16



设置



2班学生群



群成员

121 >



群应用



群公告



任务



Pin



群成员日历

消息搜索



消息



云文档



文件



图片



链接

群管理



群机器人



群昵称



Course crew: TAs

| Name | E-mail | Office | Phone |
|------|--------------------------------|--------------------------------------|-------------|
| 金旭 | jinxu@mail.ustc.edu.cn | Room 1406, Sci. & Lab Building(west) | 13511610125 |
| 石军 | shijun18@mail.ustc.edu.cn | Room 1411, Sci. & Lab Building(west) | 17756023365 |
| 王朝晖 | wangzh95@mail.ustc.edu.cn | Room 1411, Sci. & Lab Building(west) | 15273133022 |
| 钟书锐 | zsr1341864378@mail.ustc.edu.cn | Room 1411, Sci. & Lab Building(west) | 17873661361 |

■ Office hours and Discussion

- 140+ students, Chaired by 4 TAs, fellow students
- Fact to face help
- See web page for times
- <http://acsa.ustc.edu.cn/ics/information.html>



- 1 Course and Crew
- 2 What's the difference between "Big" and "Small"?
- 3 Why Take This Course?
- 4 Great Ideas in Computing Systems
- 5 What' s This Course All About ?
- 6 Summary

Questions

- How powerful are today's computers? Why are they so powerful?

What can computers do?

- Is an abacus a computer? How to understand Turing's contribution to computers?

How are they done?

- What are the serious flaws in today's computers?

What can't computers do?

What is the definition of a computer?

- A computer is a machine that can be programmed to carry out sequences of arithmetic or logical operations automatically.
- Modern computers can perform generic sets of operations known as programs.
- A broad range of industrial and consumer products use computers as **control systems**.
 - Simple special-purpose devices like microwave ovens and remote controls
 - Factory devices like industrial robots and computer-aided design
 - General-purpose devices like personal computers and mobile devices like smartphones.
- Early computers were meant to be used **only for calculations**.
 - Simple manual instruments like the abacus have aided people in doing calculations since ancient times.
- A modern computer consists of at least one processing element along with some type of computer memory.

Today, computer is in everything!



Games



Pad



Refrigerators



Robots



Laptops



Routers



Smart Phones



Media Players



Set-top boxes



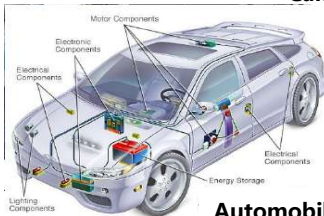
Cameras



Servers



Sensor Nets



Automobiles



Supercomputers

Today, computer is in everything !



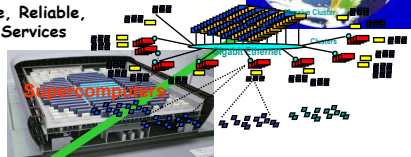
Today, computer is in everything !



Vast infrastructure behind them: from the small to the big



Scalable, Reliable,
Secure Services



Internet
Connectivity

Laptops



Servers

Databases
Information Collection
Remote Storage
Online Games
Commerce

Routers

Cameras



Sensor
Nets

MEMS for
Sensor Nets



Cars



Games



Robots



你点击一次Google搜索按钮，结果是如何呈现给你的？

■ 所花费的时间

- 不足1秒钟，可以提供你所要找东西的1000+个链接

■ 所访问的服务器

- 1个搜索请求会发往数千台服务器，平均往返2400公里

■ 所消耗的能量

- 可以让一只100瓦的灯泡工作1小时

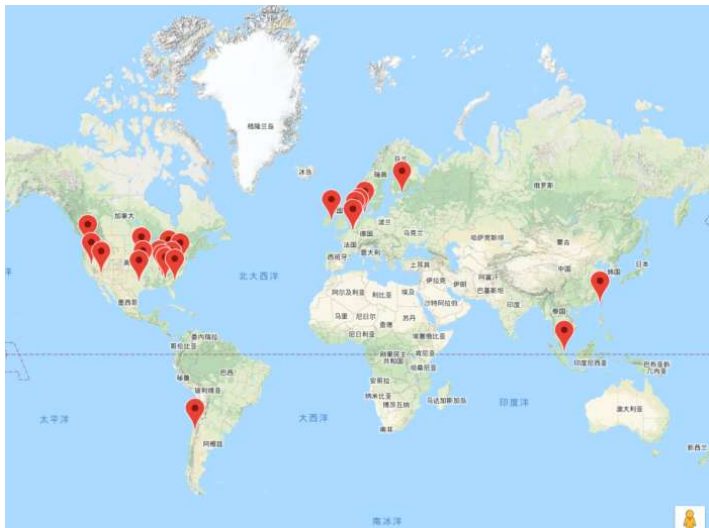
■ 所使用的技术

- 信息检索技术 如何找到最佳的信息匹配
- 网络技术 如何用最新的网络技术传送信息
- 信息采集技术 如何收集分散的各种信息资源
- 硬件技术 如何为海量信息处理提供计算和存储资源
- 并行处理技术 如何快速分类、检索和组织信息
- 等等

每时每刻都有200+项相关改进算法在实践

Google全球数据中心 (2021年)

全球最高端数据中心-谷歌的数据中心是什么样的



"Big " vs. "Small" Computer



Personal mobile devices/smart terminal devices

Small

- High performance calculation
- High-performance communications and I/O
- Power consumption constraints
- The volume constraint



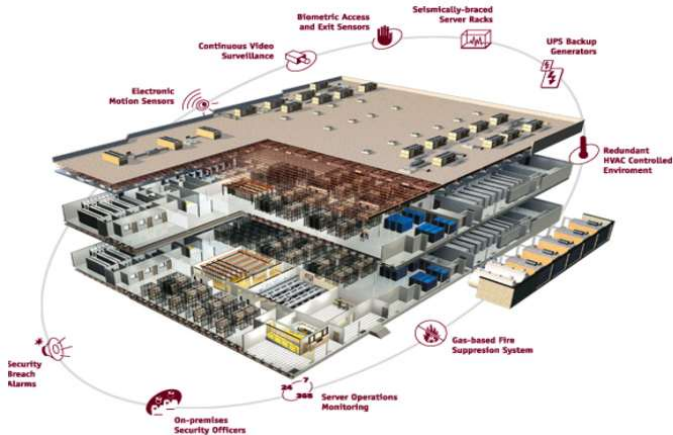
Supercomputing center (calculation)/data center (storage)

Big

- Capable of high-performance data processing more than demand
- Reliability requirement

"Big " Computer Inside

■ Data center: a "warehouse" supercomputer



Each data center covers an average of about 45,000 square meters and cost about \$600 million to build

Sunway Taihulight

- June 2016, national supercomputing wuxi center

- [神威太湖之光书写自主创新传奇！运算速度每秒亿亿次](#)
- [科技之光，神威太湖之光芯片竟如此强大，造价更是昂贵到这种程度](#)



Sunway Taihulight

- June 2016, national supercomputing wuxi center

Table 1: Sunway TaihuLight System Summary

| | |
|-----------------------------------|---|
| CPU | Shenwei-64 |
| Developer | NRCPC |
| Chip Fab | CPU vendor is the Shanghai High Performance IC Design Center |
| Instruction set | Shenwei-64 Instruction Set (this is NOT related to the DEC Alpha instruction set) |
| Node Processor cores | 256 CPEs (computing processing elements) plus 4 MPEs (management processing elements) |
| Node Peak Performance | 3.06 TFlop/s |
| Clock Frequency | 1.45 GHz |
| Process Technology | N/A |
| Power | 15.371 MW (average for the HPL run) |
| Peak Performance of system | 125.4 Pflop/s system in Wuxi |
| Targeted application | HPC |
| Nodes | 40,960 |
| Total memory | 1.31 PB |
| Cabinets | 40 |
| Nodes per cabinet | 1024 Nodes |
| Cores per node | 260 cores |
| Total system core count | 10,649,600 |

How are "very large" computer systems built?

-Sunway Taihulight

■ 运算节点板

- 1CPU, 260计算核, 系统基本构成单元, 众核处理器+存储器

■ 运算插件板

- 4 运算节点板高密组装, 4CPU。运算节点+网络接口板

■ 运算超节点

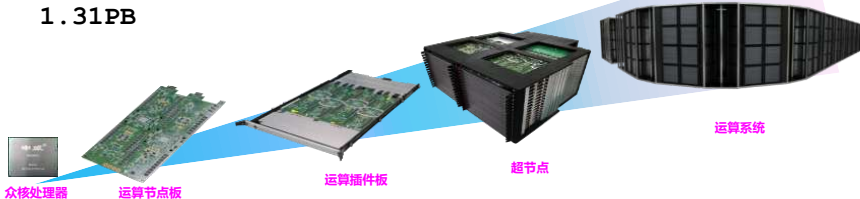
- 64 运算插件板, 256CPU, 超节点内部采用紧耦合弹性互连

■ 运算机仓 (Cabinet)

- 4运算超节点, 256运算插件板, 1024运算节点 (CPU)

■ 整机系统

- 40运算机仓, 160超节点, 40960节点, 10649, 600计算核, 1.31PB



How are "very large" computer systems built?

-Sunway Taihulight

■ 运算节点板

- 1CPU, 260计算核, 系统基本构成单元, 众核处理器+存储器

■ 运算插件板

- 4 运算节点板高密组装, 4CPU。运算节点+网络接口板

■ 运算超节点

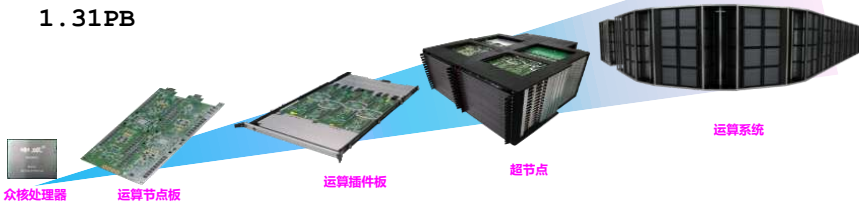
- 64 运算插件板, 256CPU, 超节点内部采用紧耦合弹性互连

■ 运算机仓 (Cabinet)

- 4运算超节点, 256运算插件板, 1024运算节点 (CPU)

■ 整机系统

- 40运算机仓, 160超节点, 40960节点, 10649, 600计算核, 1.31PB

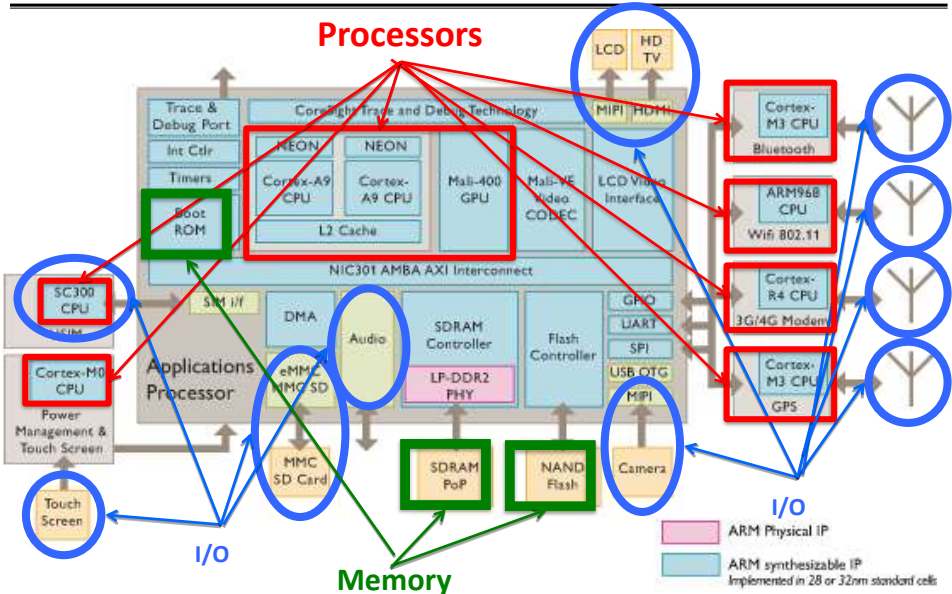


"Small " Computer

■ Personal mobile devices : Huawei P30



"Small" Computer Inside





1 Course and Crew

2 What's the difference between "Big" and "Small"?

3 Why Take This Course?

4 Great Ideas in Computing Systems

5 What's This Course All About?

6 Summary

Why Take This Course?

- 掌握计算机系统（晶体管器件、数字逻辑、计算机组成原理、高级语言的编译与汇编、高级语言的硬件实现、操作系统）核心概念和思想的最小集
- 理解硬件和软件在计算机系统中的作用和相互关系
- 理解如何构建完整的计算机系统
- 塑造与众不同的程序员：将算法设计与硬件设计融会贯通

枝繁叶茂



树大根深

UCB EECS Bachelors Curriculum Overview

Calculus: Math 1A & 1B
Multi-variable Calculus: Math 53

A two course physics sequence:
Physics 7A/7B, or Physics 5A/5B/5BL

A natural science course with lab from {Physics, Astronomy, Biology, Chemistry, Earth and Planetary Science, Integrative Biology, Molecular and Cell Biology, Physics or Plant and Microbial Biology }

4 unit STEM elective from {Astro, Chem, Data Sci, EPS, IB, Math, MCB, Physics, PMB, Stat, or any Engin dept.}

**CS
Bachelor
of Arts**

**EECS
Bachelor
of Science**

Lower Division CS Requirements

Math 1A (Calculus I) , Math 1B (Calculus II), Math 54
CS 70 (Discrete Mathematics and Probability Theory)
CS 61A (Structure and Interpretation of Computer Programs)
CS 61B/BL (Data Structures)

CS 61C (Machine Structures)

Upper Division CS Requirements

20 Upper Division Units*

4 units of a Design Course from {CS 152, 160, 161, 162, 164, 169, 182, 184, 186/W186} or
{EE C128, 130, 140, 143, 192, EECS C106A/C106B, 149 (formerly EE/CS 149), 151 (formerly CS 150/EE 141) }

8 units upper-division CS courses

8 units upper division CS/EE/EECS courses^{1 2}

7 units of Upper Division Technical Electives:

can be upper division CS/EE/EECS courses^{1 2}

Lower Division EECS Requirements

CS 70 Discrete Mathematics and Probability Theory
EE 16A Designing Information Devices and Systems I
EE 16B Designing Information Devices and Systems II
CS 61A (Structure and Interpretation of Computer Programs)
CS 61B/BL (Data Structures)

CS 61C (Machine Structures)

Upper Division EECS Requirements

EE C106A, C106B, C128, 130, 140, 143, C149, 192
CS C149, 160, 162, 164, 169, 182, 184, 186, W186
EECS 149, 151 and 151LA (must take both), 151 and 151LB (must take both)
CS 161 will fulfill the design requirements for students who took the class in Spring 2019 or later.

Stanford CS Bachelors Curriculum Overview

| Math.&Sci. | Intro & Core | Track |
|---|---|---|
| Mathematics (26 units minim) | CS 1C. Introduction to Computing at Stanford. 1 Unit. | Artificial Intelligence track(choose one) |
| Science (11 units minimum) | Theory <ul style="list-style-type: none">• <i>Mathematical Foundations of Computing (CS103) (5 units)</i>• <i>Introduction to Probability for Computer Scientists (CS109) (5 units)</i>• <i>Data Structures and Algorithms (CS161) (5 units)</i> | Biocomputation track (choose one) |
| Technology in Society (3-5 units) | Systems <ul style="list-style-type: none">• <i>Programming Abstractions (CS106A, CS106 B or CS106X)</i>• <i>Principles of Computer Systems (CS110) (5 units)</i>• <i>Computer Organization and Systems (CS107)(5 units)</i> | Computer Engineering track(choose one) |
| Engineering Fundamentals (13 units) | | Graphics track (choose one) |
| | | Human-Computer Interaction track (choose one) |
| | | Information track (choose one) |
| | | Systems track (choose one) |
| | | Theory track (choose one) |
| | | Unspecialized track (choose one) |
| | | Individually Designed track |

MIT EECS Bachelors Curriculum Overview

6-2. Electrical Engineering and Computer Science

Intro: 1 of {6.01,6.02,6.03,6.S08}

Prog. Skills: 1 of {6.0001,6.S080}

Math: 1 of {18.03,2.087}

Foundation: 3 of {6.002,6.003,**6.004**,6.006,6.007,6.008,6.009}

Header: 3 of {6.011,6.012,6.013,6.014,6.021,6.031,6.033,6.034,6.036,6.045,6.046}

Other: 1 of {6.UAT,6.UAR}, AUS2,AUS2,EECS,EECS

Semester 1: Programming skills, Differential Equations
Semester 2: Introduction to EECS, Foundation #1
Semester 3: Foundation #2, Foundation #3
Semester 4: Header #1, Header #2
Semester 5: Header #3, AUS #1
Semester 6: AUS #2, Course 6 Elective #1
6.UAT or 6.UAR and the second Course 6 elective

2025/2/24

6-3. Computer Science and Engineering

Intro: 1 of {6.01,6.02,6.03,6.S08}

Prog. Skills: 1 of {6.0001,6.S080}

Math: 1 of 6.042

Foundation: 3 of {**6.004**,6.009,6.006}

Header: 6.031,6.033, 1 of {6.045,6.046}, 1 of {6.034,6.036}

Other: 1 of {6.UAT,6.UAR}, AUS2,AUS2,EECS

Semester 1: Programming skills, Discrete math
Semester 2: Introduction to EECS, Foundation #1
Semester 3: Foundation #2, Foundation #3
Semester 4: Header #1, Header #2
Semester 5: Header #3, Header #4
Semester 6: AUS #1, AUS #2
6.UAT or 6.UAR and the second Course 6 elective

6-7. Computer Science and Molecular Biology

Intro: (6.009 and 6.031),6.042J [Note 4]
Intro Lab: 1 of {7.02J, 20.109, 20.129 [Note 14]}

Math Requirement: (6.00 [Note 13] and 6.009)

Foundation Bio: 7.03, 7.05
Foundation CS: 6.006

Foundation Bio: 7.06
Foundation CS: 6.0046J

Bio and Computational Bio
Restricted Electives: 2 of {1.S993, 6.047, 6.049, 6.503, 6.802J, 7.09, 18.418}

Other: 1 of {6.UAT,6.UAR}

Semester 1: Programming skills, Discrete math
Semester 2: Introduction to EECS, Foundation #1
Semester 3: Foundation #2, Foundation #3
Semester 4: Header #1, Header #2
Semester 5: Header #3, Header #4
Semester 6: AUS #1, AUS #2
6.UAT or 6.UAR

6-14. Computer Science, Economics and Data Science

Prog. Skills: 1 of {6.0001, 6.0002, 6.009}

Math: 18.06,6.041, 6.042, 14.30 or 18.600

Foundation EDS: 1 of {14.01,14.03}, 14.32
Foundation CS: 6.006, 6.036

Foundation EDS: 1 of {14.05,14.18, 14.33}, 1 of {6.207,6.207, 15.053},
Foundation CS: 6.046

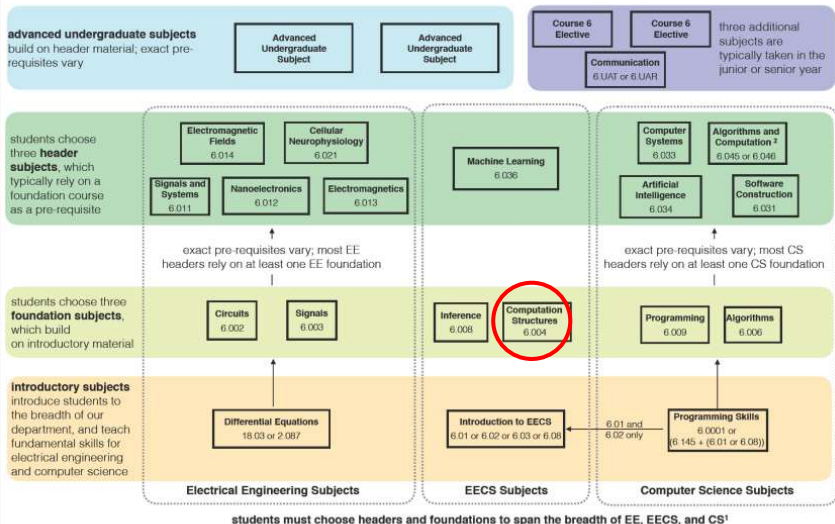
Data Science Elective: 1
Economics Theory Elective: 1
Data Science or Theory Elective: 1

Other: 1 of {6.UAT,6.UAR,15.276}

Semester 1: Linear Algebra, Discrete Math, Programming #1 + #2 (if 6.0002)
Semester 1: Linear Algebra, Discrete Math, Programming #1 + #2 (if 6.0002)
Semester 2: Probability and Statistics, Programming #2 (if 6.009), Microeconomics
Semester 3: Algorithms, Economics
Semester 4: Machine Learning, Advanced Algorithms
Semester 5: Intermediate Economics, Networks and Optimization, Elective #1
Semester 6: Elective #2, Elective #3

MIT EECS Bachelors Curriculum Overview

The 6-2 curriculum builds primarily on the **Physics II and Calculus II GIRs**; not all courses require a GIR as a pre-requisite



¹ of the headers and foundations, two must be from EE, two from CS, and one from EECS

² 6.045 and 6.046 also require 6.042, either as a direct pre-req or as a pre-req to 6.006

CMU CS Bachelors Curriculum Overview

Computing @ Carnegie Mellon

Humanities and Arts
(64 units)

15-128: Freshman Immigration Course*

15-122: Principles of Imperative Computation**

15-150: Principles of Functional Programming

15-151: Mathematical Foundations for Computer Science***

15-210: Parallel and Sequential Data Structures and Algorithms

15-213: Introduction to Computer Systems

15-251: Great Ideas in Theoretical Computer Science

15-451: Algorithm Design and Analysis

a concentration within SCS

a minor outside of SCS

Technical Communications Course (choose one)

Logics & Languages Elective (choose one)

Software Systems Elective (choose one)

Artificial Intelligence Elective (choose one)

Domains Elective (choose one)

Computer Science Electives (choose two)

Mathematics & Probability (choose four)

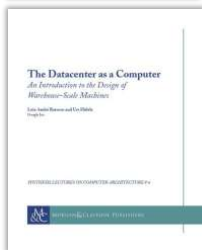
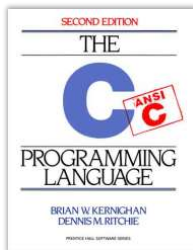
Science and Engineering (choose four)

USTC CS Bachelors Curriculum Overview

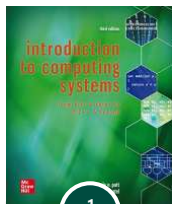
| Math.&Sci. | Intro & Core | Track |
|----------------------|---|------------------------------------|
| Mathematics sequence | 011044. Introduction to Computing at USTC (1 Unit) | Computer System Track |
| Physics sequence | Theory(Intro) <ul style="list-style-type: none">Discrete Mathematics(A,B,C,9 units)Data Structures (4 units)Fundamentals of Algorithms (3.5 units) | Computer Software and Theory Track |
| English sequence | Systems(Intro) <ul style="list-style-type: none">Fundamentals of Programming (4 units)Introduction to Computer Systems (4 units) | Computer Application Track |
| Politics sequence | System(Core) <ul style="list-style-type: none">Computer Organization and DesignPrinciple and Design of Operating SystemPrinciples and Techniques of CompilationComputer Architecture | Information Security Track |
| Physical Education | | Individually Designed Track |

Textbooks in CS 61C at UCB

- Great Ideas in Computer Architecture (Machine Structures)
- 30+ TAs
- <https://cs61c.org/fa21/>
 - Computer Organization and Design RISC-V Edition, 1st ed. by David Patterson, and John Hennessy
 - The C Programming Language, 2nd ed. by Brian Kernighan and Dennis Ritchie
 - The Datacenter as a Computer by Luiz André Barroso and Urs Hölzle, freely available [here](#)



优化课程内容



**Introduction to
Computing
Systems: From Bits
& Gates to C/C++ &
Beyond**

Yale N. Patt and
Sanjay J. Patel, June
2003, McGraw-Hill
Higher Education



**Computer
Organization and
Design: The
Hardware/Software
Interface,**

David A. Patterson,
John L. Hennessy,
5th edition.
Morgan Kaufmann
Publishers, Inc.,
2017



中国科学技术大学
University of Science and Technology of China

计算机系统概论

Introduction to Computing Systems
(CS1002A.02)

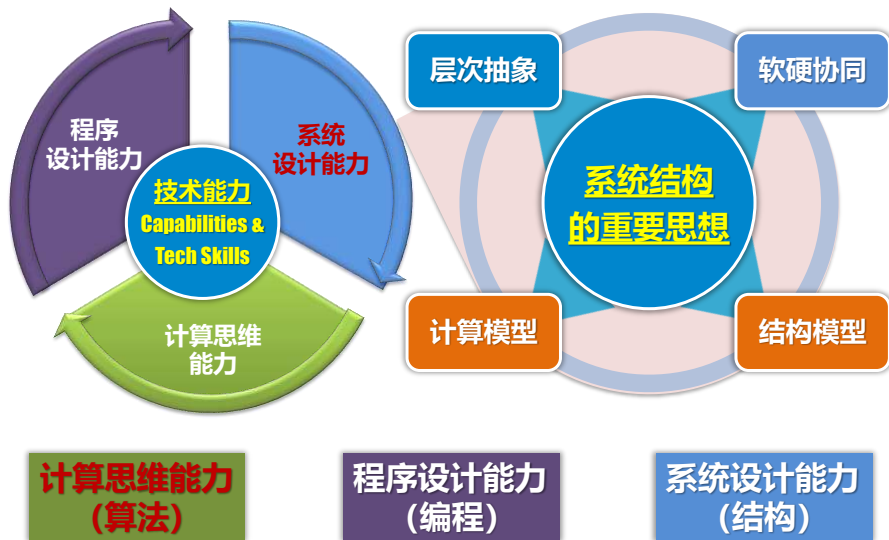
Chapter 1-2 Great Ideas in Computing Systems

计算机科学与技术学院
School of Computer Science and Technology

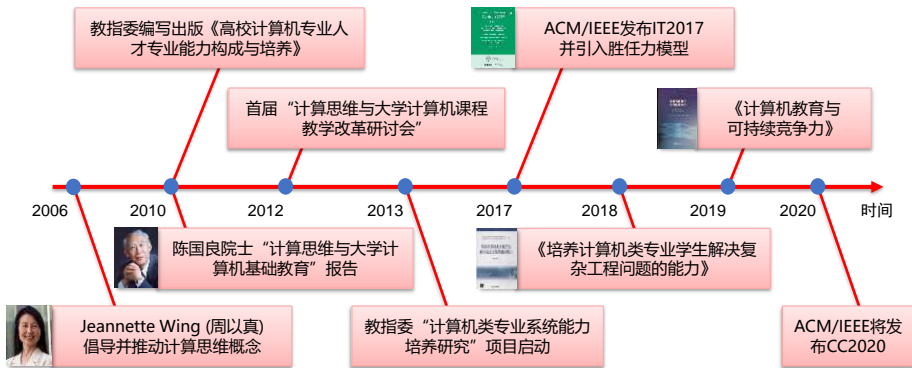


- 1 Course and Crew
- 2 What's the difference between "Big" and "Small"?
- 3 Why Take This Course?
- 4 **Great Ideas in Computing Systems**
- 5 What' s This Course All About ?
- 6 Summary

课程目标：理解计算机系统结构的重要思想



教育理念：计算思维的发展历程(内涵与外延)



“在异构计算的时代程序员必须对于算法和硬件模型融汇贯通，才能写出高质量的代码。因此，未来的程序员也必须懂硬件！”

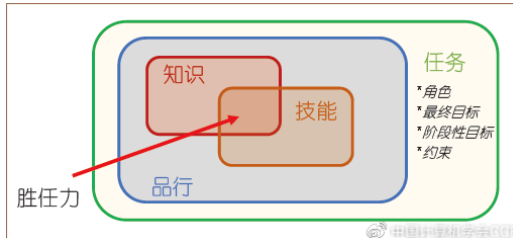
——图灵奖得主David Patterson

Computational thinking -Jeannette M. Wing

- Computational thinking will be a fundamental skill used by everyone in the world by the middle of the 21st Century.
 - Just like reading, writing, and arithmetic.
 - Incestuous: Computing and computers will enable the spread of computational thinking.
 - In research: scientists, engineers, ..., historians, artists
 - In education: K-12 students and teachers, undergrads, ...
- Computational Thinking is the thought processes involved in formulating a problem and expressing its solution in a way that a computer—human or machine—can effectively carry out.
- Computational Thinking is what comes before any computing technology—thought of by a human, knowing full well the power of automation.

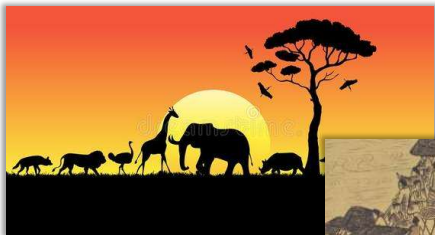
ACM和IEEE-CS计算机类专业课程体系规范CC2020

- CC2020采用胜任力模型，融合知识、技能、品行三个方面的综合能力培养。CC2020将对中国计算机专业设置带来积极而深远的影响。
- CC2020采用“计算”(computing)一词作为计算机工程、计算机科学和信息技术等所有计算机领域的统一术语；同时采用“胜任力”(competency)一词来代表所有计算教育项目的基本主导思想。其目标就是从知识(knowledge)、技能(skills)和品行(dispositions)三方面培养，使学生胜任未来计算相关工作内容。
 - 知识对应胜任力的“了解”(know-what)维度
 - 技能表达了知识的应用，是胜任力的“诀窍”(know-how)维度
 - 品行构成胜任力的“知道为什么”(know-why)维度



什么是计算思维能力?

**计算思维：利用包括网络在内的计算系统进行问题求解
(自然问题、社会问题、技术问题) 的思维方式**



自然



社会



技术

什么是计算思维能力？

不同的历史时期，人类关注的**重大问题**不同

- **20世纪**：**战争、人口增长、饥饿**等
- **21世纪**：自然资源消耗过快、**环境污染**、气候异常、**健康医疗**、人口老龄化、贫富差距过大、城市交通、非传统**安全问题**（非典型传染病爆发、金融危机、恐怖主义、网络攻击等）

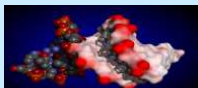
科学与工程计算



量子计算模拟



可控核聚变模拟



蛋白质与药物设计



大气环境监测

大数据，人工智能和云计算



健康大数据



人工智能



基因分析



金融分析

How to use the Sunway Taihulight to solve complex problems?

- ## ■ 神威太湖之光荣获戈登贝尔奖，中国超算实现零的突破

Great Ideas in Computing Systems in this courses

- **Great Idea #0: Great Idea from Ancient Chinese Philosophy(Bits and Bytes)**
- **Great Idea #1: Computer is an Universal Computing Device(Turing Machine Model)**
- **Great Idea #2: Stored program computer(Von Neumann Model)**
- **Great Idea #3: Abstraction Helps Us Manage Complexity(Layers of Representation/Interpretation)**
- **Great Idea #4: Software and Hardware Co-design**

Great Ideas in Computing Systems in other courses

- **Great Idea #5: Computer Family (IBM 360)**
- **Great Idea #6: Principle of Locality (Memory Hierarchy)**
- **Great Idea #7: Make the Common Case Fast**
- **Great Idea #8: RISC vs. CISC**
- **Great Idea #9: Moore' s Law (Designing through trends)**
- **Great Idea #10: Parallelism & Amdahl's law (which limits it)**
- **Great Idea #11: Dependability via Redundancy**

Great Idea from Ancient Chinese Philosophy

Everything comes from being and being comes from nothing.

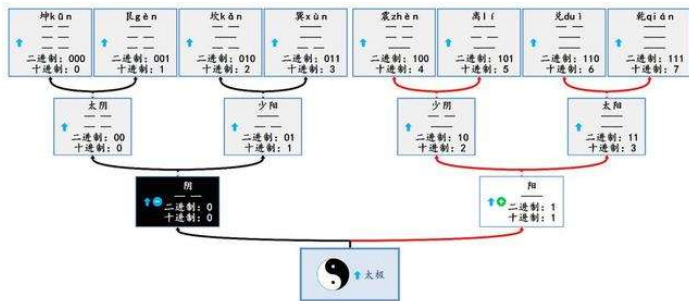
天下万物生于有, 有生于无

《老子·四十章》



《易经》

太极生两仪,
两仪生四象,
四象生八卦,
八卦演万物。



Great Idea from Ancient Chinese Philosophy

■ Binary Number

- The modern binary number system was studied in Europe in the 16th and 17th centuries by Thomas Harriot, Juan Caramuel y Lobkowitz, and Gottfried **Leibniz**.
- **Leibniz** was specifically inspired by the Chinese "I Ching" .

28 MEMOIRES DE L'ACADEMIE ROYALE

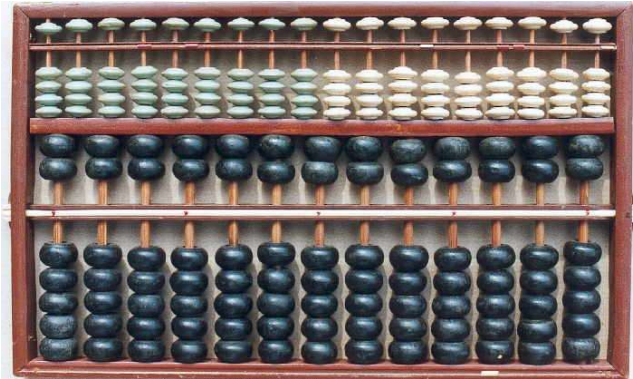
res Lincéaires qu'on lui attribue. Elles reviennent toutes à cette Arithmétique; mais il suffit de mettre ici la *Figure de huit Ceros* comme on l'appelle, qui passe pour fondamentale, & d'y joindre l'explication qui est manifeste, pourvu qu'on remarque premierement qu'une ligne entiere — signifie l'unité ou 1, & secondement qu'une ligne brisée — — signifie le zero ou 0.

| | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|
| 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 |
| 0 | 1 | 10 | 11 | 100 | 101 | 110 | 111 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Les Chinois ont perdu la signification des *Ceros* ou Lincéaires de Fohy, peut-être depuis plus d'un millénaire d'année; & ils ont fait des Commentaires là-dessus, où ils ont cherché je ne sçai quels sens éloignés. De sorte qu'il a fallu que la vraie explication leur vint maintenant des Européens: voici comment. Il n'y a gueres plus de deux ans que j'envoyai au R. P. Bouvet Jésuite, François célèbre, qui demeure à Pekin, ma manière de compter par 0 & 1; & il n'en fallut pas davantage pour lui faire reconnoître que c'est la clef des Figures de Fohy. Ainsi m'écrivant le 14 Novembre 1701, il m'a envoyé la grande Figure de ce Prince Philosophe qui va à 64, & ne laisse plus lieu de douter de la vérité de notre interprétation; de sorte qu'on peut dire que ce Pere a décelé l'Enigme de Fohy à l'aide de ce que je lui avois communiqué. Et comme ces Figures sont peut-être le plus ancien monument de science qui soit au monde, cette restitution de leur sens, après un si grand intervalle de tems, paroitra d'autant plus curieuse.

Le consentement des Figures de Fohy & de ma Table des Nombres, se fait mieux voir lorsque dans la Table on supplée les zeros initiaux, qui paroissent superflus, mais qui servent à mieux marquer la période de la colonne,

Great Idea from Ancient Chinese Philosophy



Is an abacus a computer?



5

Computer Organization and Design: The Hardware/Software Interface,

David A. Patterson,
John L. Hennessy,
5th edition. Morgan
Kaufmann
Publishers, Inc.,
2017

A computing tool that does not use electricity

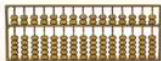
■ Abacus (公元前500年, 中国)

Abacus

China

c. 1970

**Loan of Gwen and Gordon
Bell, B1643.01**



**Table abacus
(reproduction) and
jetons**

Germany

17th century

**Loan of Michael R.
Williams, L2003.3.2**



Soroban

Japan

c. 1960

**Loan of Gw
Bell, B1659.01**



Counting Frame

Early 20th century

**Gift of Gwen and
Gordon Bell, B141.80**



Schoty

Russia

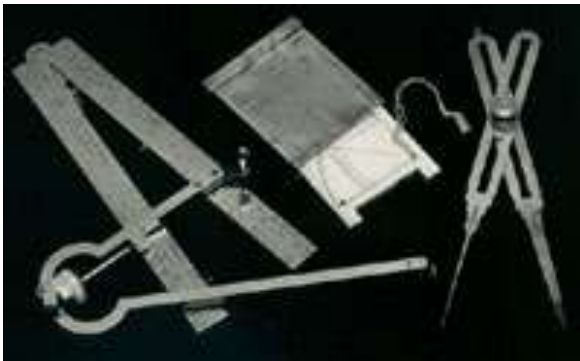
Early 20th century

Gift of Warren Yogi, 102



A computing tool that does not use electricity

■ Sectors



A computing tool that does not use electricity

■ Slide Rules

Slide rule

US

c. 1956

Gift of Lynn Yarbrough, X121.82



Pickett circular slide rule

Japan

c. 1955

Loan of Gwen and Gordon Bell, B1657.01



Mannheim slide rule

France

c. 1860

Loan of Gwen and Gordon Bell, B203.82



Fuller's Rule

US

1921

Gift of University of Illinois, X250.83A



Lord's calculator

England

c. 1900

Loan of Gwen and Gordon Bell, B123.80



Automatic computing equipment: from mechanical computer to electronic computer



**Charles Babbage,
1791 – 1871, England**



**Alan
Turing(24)**



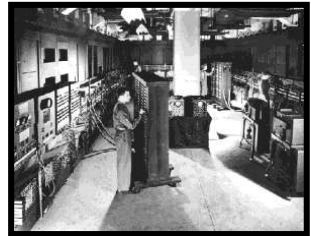
Eckert(24) and Mauchly(36)



**1832, 2002, 2008
The Babbage Difference
Engine, 17 years, 25,000
parts, 5ton, cost: £17,470**



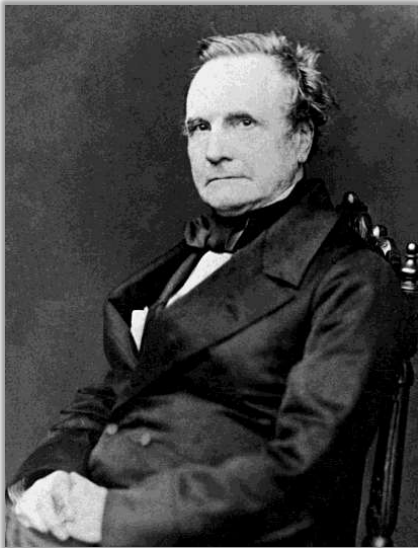
**Turing Machine,
1936**



ENIAC

1946

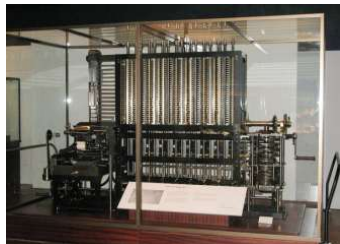
Charles Babbage (1791-1871): **A Fallen Hero!**



*[Copyright expired and in public domain.
Image obtained from Wikimedia Commons.]*

- **Lucasian Professor of Mathematics, Cambridge University, 1828-1839**
- **A true “polymath” with interests in many areas**
- **Frustrated by errors in printed tables, wanted to build machines to evaluate and print accurate tables**
- **Inspired by earlier work organizing human “computers” to methodically calculate tables by hand**

Babbage difference engine: the first mechanical computer (1832)



- 2002, 2008
- The Babbage
- Difference Engine
- 17 years,
- 25,000 parts, 5ton
- cost: £17,470

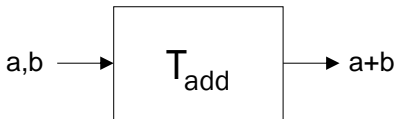


Turing Machine

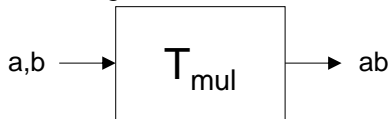
■ Mathematical model of a device that can perform any computation – Alan Turing (1937)

- ability to read/write symbols on an infinite "tape"
- state transitions, based on current state and symbol

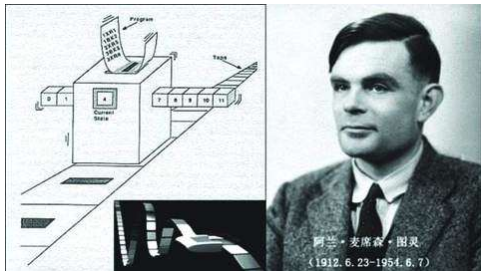
■ Every computation can be performed by some Turing machine. (*Turing's thesis*)



Turing machine that adds



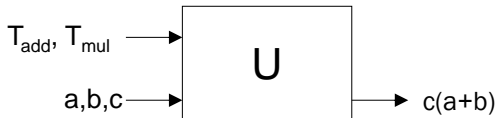
Turing machine that multiplies



Universal Turing Machine

■ Turing described a Turing machine that could implement all other Turing machines.

- inputs: data, plus a description of computation (Turing machine)



Universal Turing Machine

■ U is programmable – so is a computer!

- instructions are part of the input data
- a computer can emulate a Universal Turing Machine, and vice versa

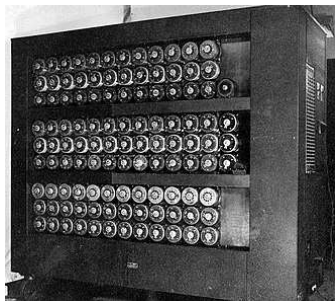
■ *Therefore, a computer is a universal computing device!*

Bombe (1939)

- The **bombe** is an electro-mechanical device used by the British cryptologists to help decipher German **Enigma-machine**-encrypted secret messages during World War II.
- The initial design of the British **bombe** was produced in 1939 at the UK Government Code and Cypher School (GC&CS) at Bletchley Park by **Alan Turing**.



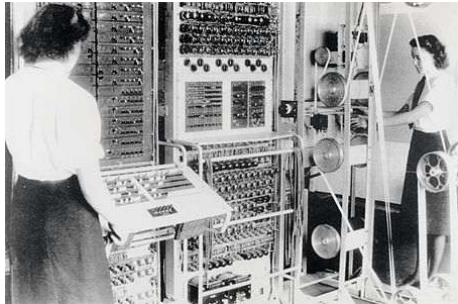
Enigma machine



A wartime picture of a [Bletchley Park Bombe](#)

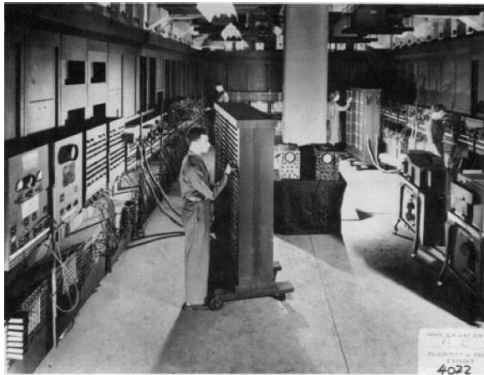
Colossus- The world's first programmable, electronic, digital computer (1943-1945)

- **Colossus** was a set of computers developed by British codebreakers in the years 1943–1945[1] to help in the cryptanalysis of the Lorenz cipher.
- **Colossus** used thermionic valves (vacuum tubes) to perform Boolean and counting operations.
- **Colossus** is regarded as the world's **first programmable, electronic, digital computer**, although it was programmed by switches and plugs and **not by a stored program**.
- **Colossus** was designed by General Post Office (GPO) research telephone engineer Tommy Flowers at Bletchley Park. **Alan Turing's** use of probability in cryptanalysis contributed to its design.
- **Turing's** machine that helped decode Enigma was the electromechanical **Bombe**, not **Colossus**.



A Colossus Mark 2 computer being operated by Wrens.

ENIAC - The first electronic computer ,1946



1904, The world's first electron tube was born at the hands of the British physicist Fleming

ENIAC(Electrical Numerical Integrator And Calculator)

- 17,468 vacuum tubes
- Power 150kW
- Weighed 30 tons
- Occupied 1800 sq ft
 - 80 feet long
 - 8.5 feet high
- Clock: 100kHz, About 5000 additions per second
- RAM: ~230bytes, Could store 20 numbers Could store 20 numbers in main memory
- IO: punched card
- Cost about \$500,000

ENIAC (1946)

- **First electronic general-purpose computer**

- Construction started in secret at UPenn Moore School of Electrical Engineering during WWII to calculate firing tables for US Army, designed by Eckert and Mauchly
- Twelve 10-decimal-digit accumulators
- Had a conditional branch!

- Programmed by plugboard and switches, time consuming!

- Purely electronic instruction fetch and execution, so fast

- 10-digit x 10-digit multiply in 2.8ms (2000x faster than Mark-1)

- As a result of speed, it was almost entirely I/O bound

- As a result of large number of tubes, it was often broken (5 days was longest time between failures)

Great Idea #1:

Computer is an **Universal Computing Device**

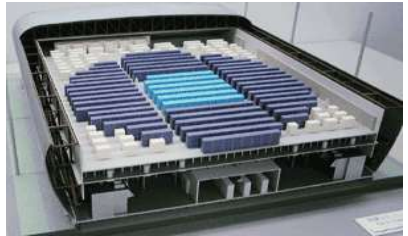
- All computers, **given enough time and memory**, are capable of computing exactly the same things.



=



=



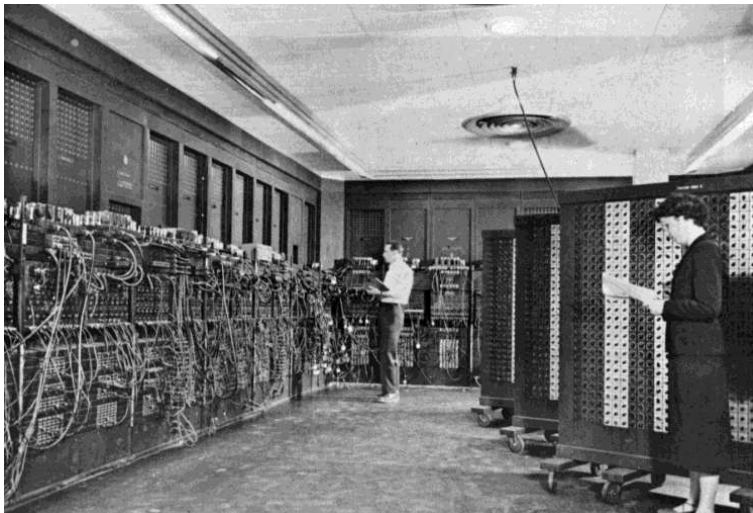
Supercomputers

How is ubiquitous computing done?

During the four years of undergraduate study,
which computer courses are relatively more basic and
must be mastered?

From Theory to Practice

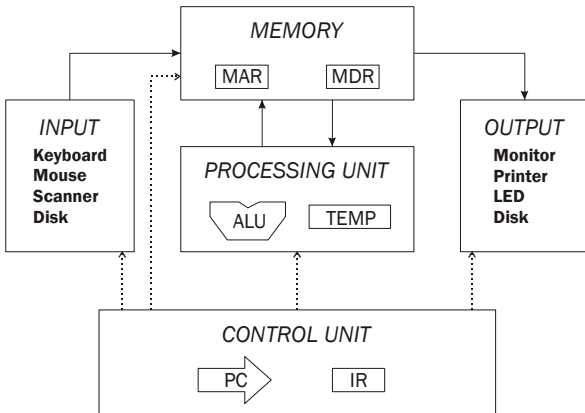
- In theory, computer can *compute* anything that's possible to compute
 - given *enough* *memory* and *time*
- In practice, *solving problems* involves computing under constraints.
 - *time*
 - weather forecast, next frame of animation, ...
 - *power*
 - cell phone, handheld video game, ...
 - *cost*
 - cell phone, automotive engine controller, ...
 -



Changing the program could take days or weeks!

Great Idea #2

Von Neumann Model(存储程序计算机)

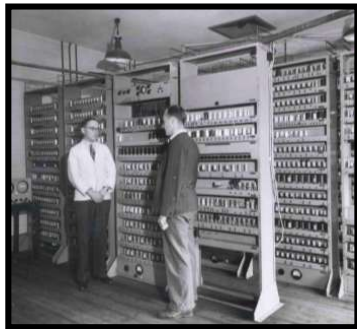


Von Neumann



Maurice
Vincent Wilkes

Electronic Delay Storage Automatic Calculator (EDSAC), University of Cambridge, UK, 1949



我们的祖上“挺阔的”

- 中国科大计算机专业创建于1958年, 当时隶属于应用数学和计算技术系
- 应用数学和计算技术系首任系主任: 华罗庚
- 计算机专业首任教研室主任: 夏培肃
- 中国第一个计算机三人小组
 - 1952年, 华罗庚教授会见了三位年轻科学家, 讨论的是一个前沿话题: 研制中国的计算机。由此中国第一个计算机三人小组成立。他们是夏培肃、闵乃大, 王传英。



我校首台计算机：107计算机

- 夏培肃先生主持研制
- 中国第一台自主设计的通用电子计算机
- 中国第一台自主设计的冯·诺依曼结构计算机
 - EDVAC, 美国, 1945-1952
 - EDSAC, 英国, 1945-1949
 - 107机, 中国, 1953-1959
- 1960年在中国科大投入使用 (命名为KD-1)
- 1970年随科大下迁至合肥
- 1974年被拆除



首批教师：参与107机研制的教师



夏培肃院士（左六）与我校计算机学科早期建设者、107机研制者、原计算机系统结构教研室主任郑世荣教授、钟津立、周行仁、赵鼎文、杨学良、王武良等在玉泉路校园合影。



我校首套计算机教材

- 夏培肃先生主持编写的我校第一套《计算机原理》教材。也是国内最早的《计算机原理》教材。



The USTCers' outstanding contributions to the Chinese computer

- 1959年, 107计算机
- 中国第一台自主设计的通用计算机
- 主设计师: **夏培肃**



- 2002年, 龙芯1号
- 中国第一颗自主设计的通用微处理器芯片
- 主设计师: **胡伟武**(86本)

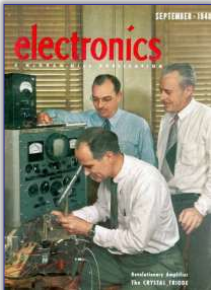


The invention of the transistor



Bell Labs lays the groundwork:

- 1945: Bell sets up lab in the hopes of developing “solid state” components to replace existing electromechanical systems. William Shockley, John Bardeen, Walter Brattain: all solid-state physicists. Focus on Si and Ge.
- 1951: Shockley develops junction transistor which can be manufactured in quantity.
- 1954: The first transistor radio! Also, TI makes first silicon transistor (price \$2.50)
- 1956: Bardeen, Shockley, Brattain receive Nobel Prize.



Two major inventions of the microprocessor chip

Stored program + Transistor technology



Change the program
so that you can do all
kinds of tasks **on the**
same hardware

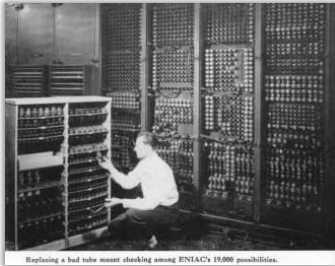


The device is
smaller and **faster**
than a vacuum
tube

First computer vs. First microprocessor chip

1946, ENIAC(Electrical Numerical Integrator And Calculator)

- 18000 vacuum tubes
- 1500 relays
- 174 KW
- 30 tons
- 1800 sq. ft. footprint
- Clock: 100kHz
- RAM: ~230bytes
- IO: punched card



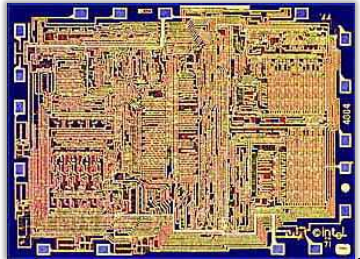
Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

After 25 years



1971, Intel 4004

- 10 micron process, NMOS-Only Logic
- 2,300 transistors
- 3x4 mm die
- 4-bit bus
- Performance < 0.1 MIPS
- 640 bytes of addressable Memory
- 750 KHz

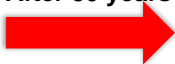


Thirty years after the first microprocessor chip was born

1971, Intel 4004

- 10 micron process
- 2,300 transistors
- 3x4 mm die
- 4-bit bus
- 640 bytes of addressable Memory
- 750 KHz

After 30 years



2000, Intel Pentium IV

- Issues up to 5 uOPs per cycle
- MMX, SSE, and SSE2
- 0.18 micron process
- 42 million transistors
- 217 mm die
- 64-bit bus
- 8KB D-cache, 12KB op trace cache (I-cache), 256KB L2 cache
- 1.4 GHz

Performance improved 5000x:
smaller, faster, cheaper



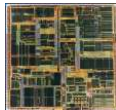
386
(275K)



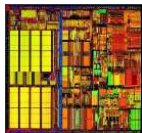
486
(1180K)



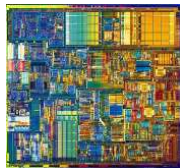
Pentium
(3100K)



Pentium II
(7500K)



Pentium III
(24000K)



Pentium IV
(42000K)

人类如何实现从物理设备到问题求解的?

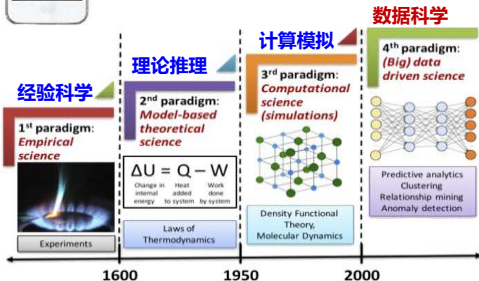
Application



差距太大，
一步无法跨越！

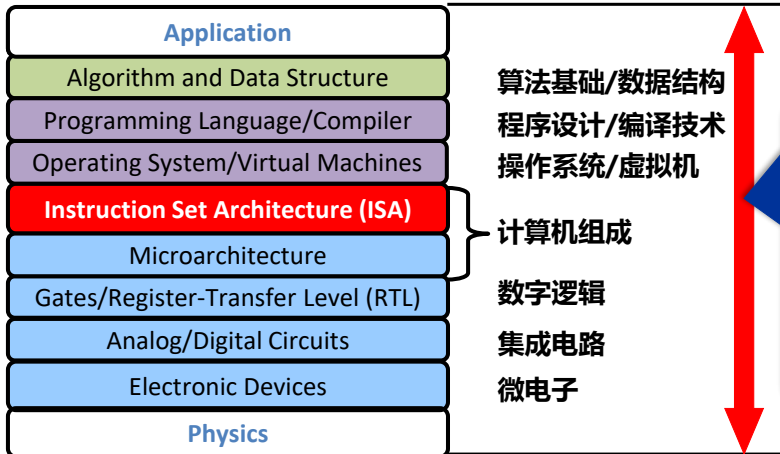
也有例外，
例如罗盘

Physics



Great Idea #3: **Abstraction** helps us Manage Complexity

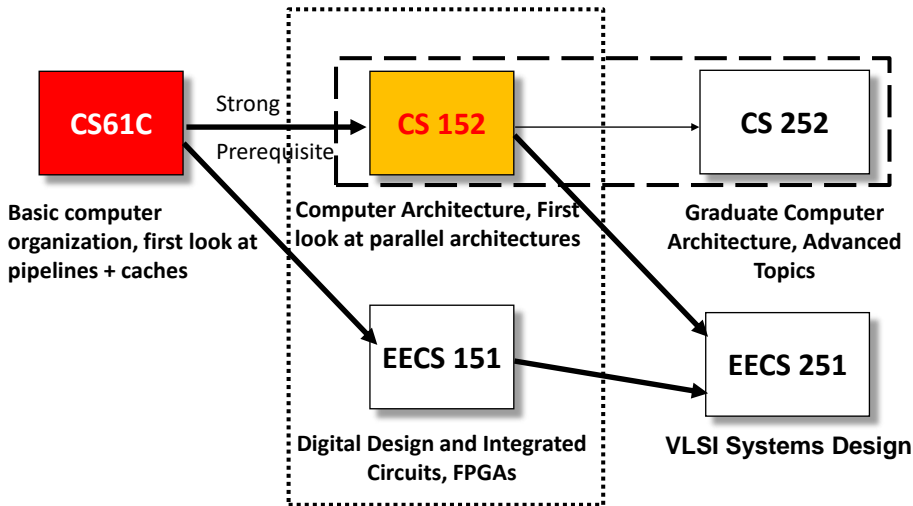
USTC Courses



需要一门**贯通课程**, 帮助学生从底层物理到高层应用, 整体上理解计算机系统。

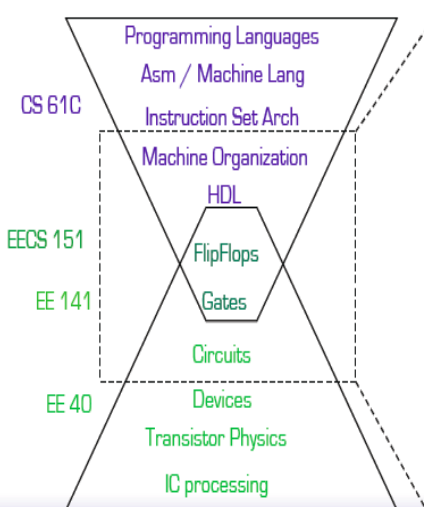
从广义上讲, **计算机系统结构**是抽象层次的设计, 它允许我们使用可用的制造技术有效地实现信息处理**应用程序**。

Related Courses in UC Berkeley



EECS151 Courses in UC Berkeley

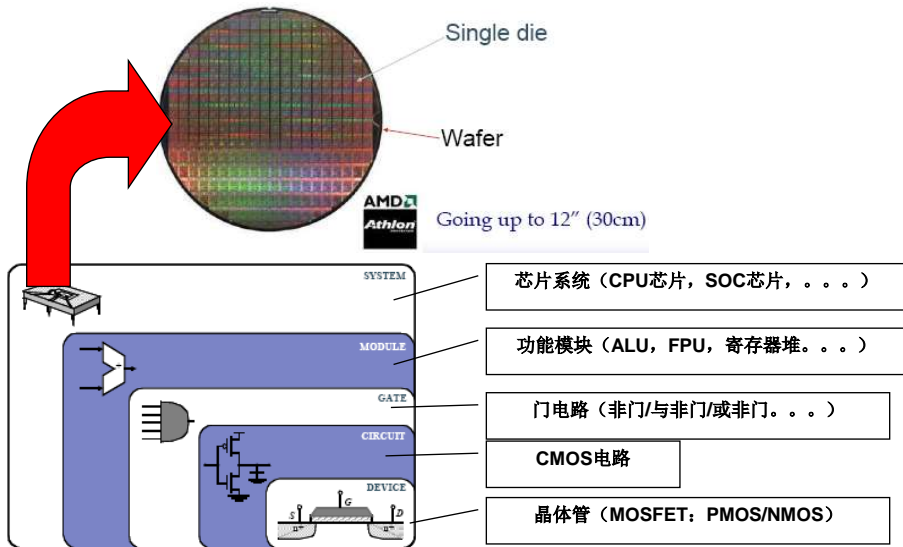
Digital design is not a spectator sport! Learn by doing.



Deep Digital Design Experience

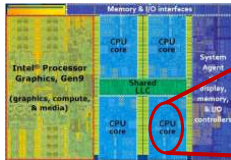
- Fundamentals of Boolean Logic
- Synchronous Circuits
- Finite State Machines
- Timing & Clocking
- Device Technology & Implications
- Controller Design
- Arithmetic Units
- Memories
- Testing, Debugging
- Hardware Architecture
- Hardware Design Language (HDL)
- Design Flow (CAD)

Abstraction to Simplify Hardware Design

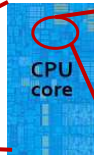


How do we put the devices into system?

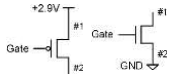
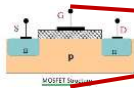
Abstraction to Simplify System Design



Integrated Circuit Design
100 Modules/ IC
0.25M~20G Devices



Register Transfer Level (RTL) Design
1K~10K Cells/Module
(100K Devices)

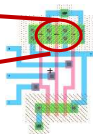


Transistor Physical Layout

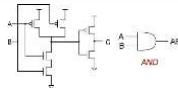


Scheme for Representing Information

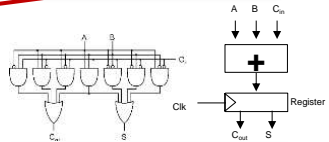
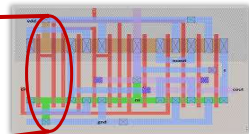
2025/2/24



Gate Level Design



Circuit Level Design
(Transistor Level Design)
(2~8 Devices/Gate)



Register Transfer Level (RTL) Design
2~16 Gates/Cell
(16~64 Devices)

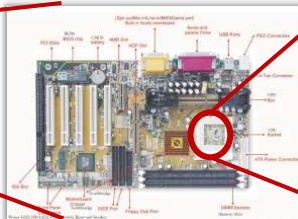


Abstraction to Simplify System Design

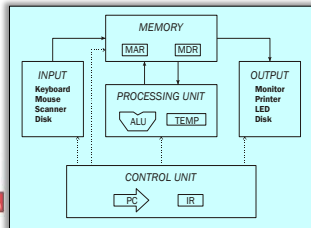
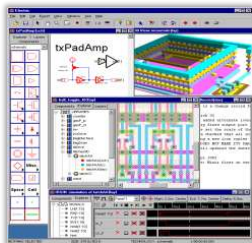
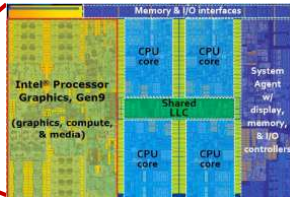
**Personal Computer:
Hardware & Software Design**
1~10PCBs/System



Motherboard Circuit Design
10 ICs/PCB
1~50G Devices

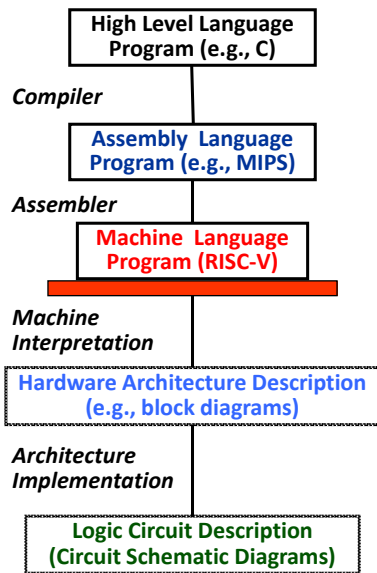


Integrated Circuit Design
100 Modules/IC
0.25M~20G Devices



Electronic System Level (ESL) Design

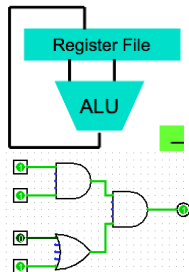
How do we get the electrons to do the work?



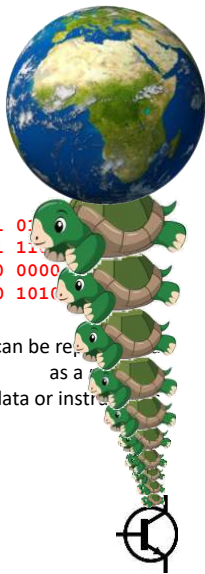
```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

```
lw    $t0, 0($2)  
lw    $t1, 4($2)  
sw    $t1, 0($2)  
sw    $t0, 4($2)
```

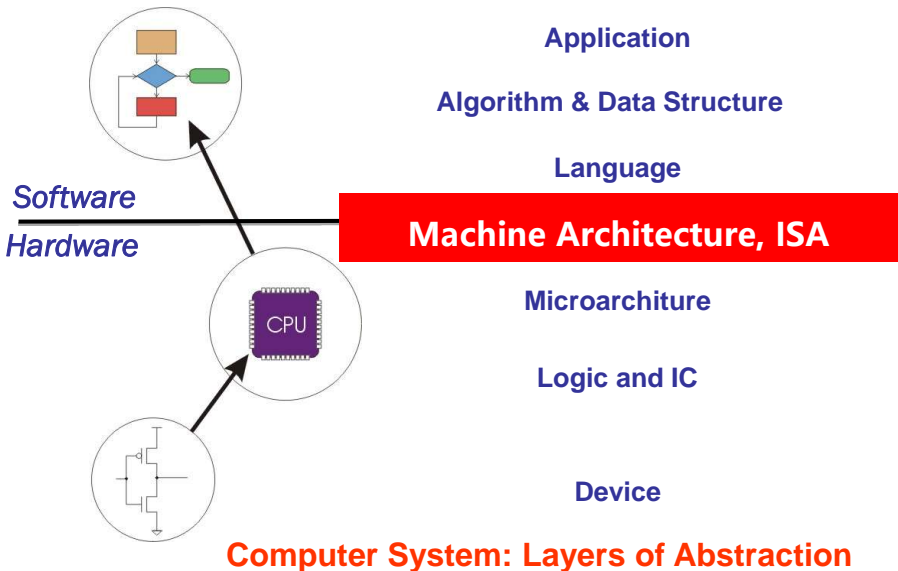
```
0000 1001 1100 0110 1010 1111 0101 1001  
1010 1111 0101 1000 0000 1001 1111 0101  
1100 0110 1010 1111 0101 1000 0000 1111  
0101 1000 0000 1001 1100 0110 1010 1111
```



Anything can be represented
as a sequence of bits
i.e., data or instructions



Great Idea #4: Software and Hardware Co-design

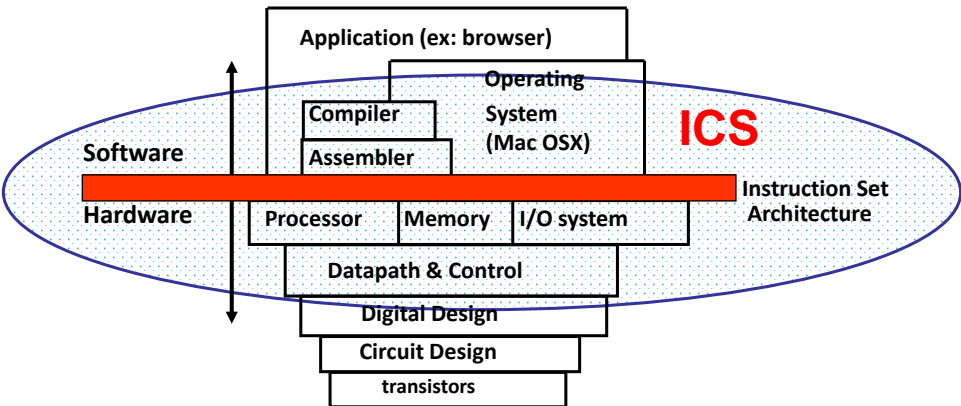


Old Machine Structures

■ Mainframe: IBM System/360

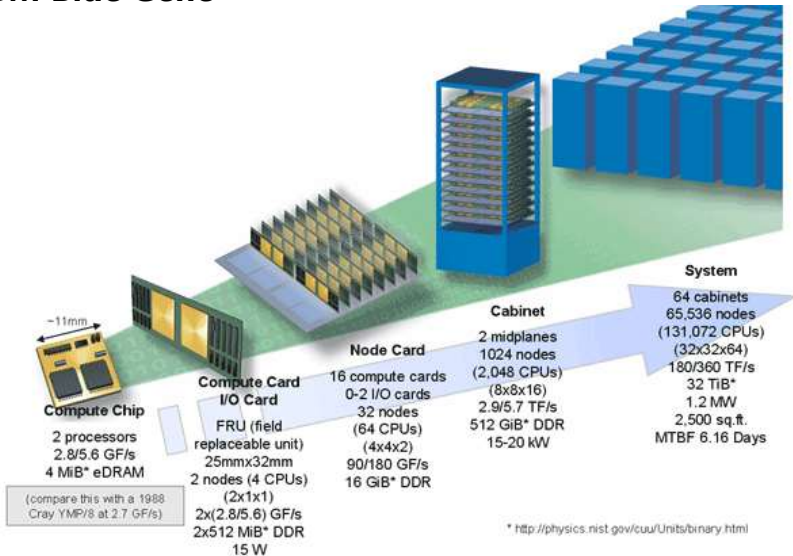


Old Machine Structures



New Machine Structures

■ IBM Blue Gene



New Machine Structures: From the Gate to the Cloud

Software

■ Parallel Requests

Assigned to computer
e.g., Search "Katz"

■ Parallel Threads

Assigned to core
e.g., Lookup, Ads

*Leverage Parallelism &
Achieve High Performance*

■ Parallel Instructions

>1 instruction @ one time
e.g., 5 pipelined instructions

■ Parallel Data

>1 data item @ one time
e.g., Add of 4 pairs of words

■ Hardware Descriptions

All gates functioning in parallel at same time

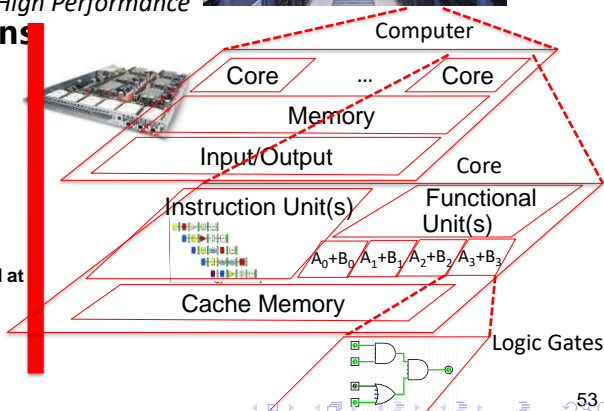
■ Programming Languages

Hardware

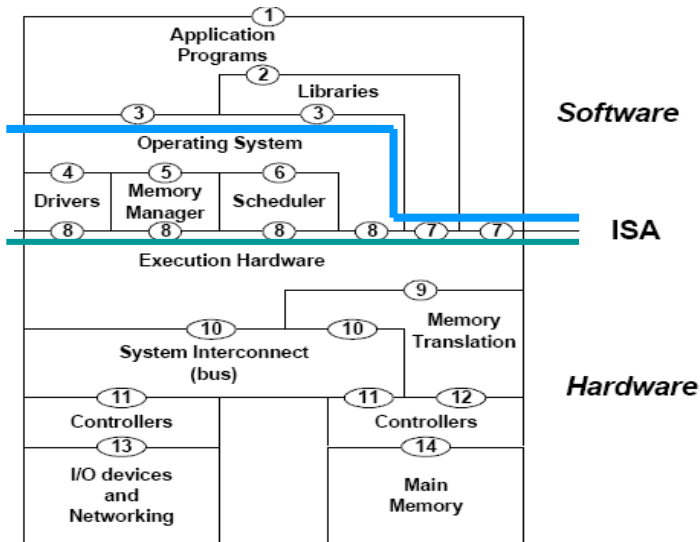
Warehouse
Scale
Computer



Smart
Phone



A Computer Architecture in a broad sense



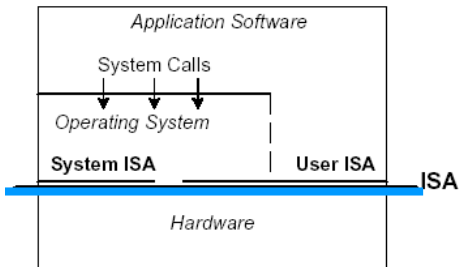
ISA and ABI

■ Interface at the top

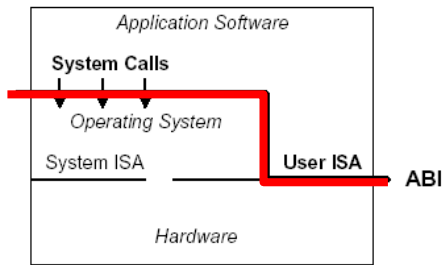
- **ABI** (Application Binary Interface, 应用二进制代码接口)
- **ISA** (Instruction Set Architecture, 指令集体系结构)
- C.P. API (Application Program Interface, 应用程序接口)

■ ISA separates hardware from rest

■ ABI separates processes from rest



(a)



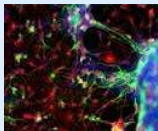
(b)

人工智能无处不在

Supercomputers



Business analytics AlphaGo



Drug design

Data Centers



Ad prediction



Automatic translation

Smartphones



Audio recognition



Image analysis

Embedded Devices



Robotics



Consumer electronics

新应用如何影响软硬件层次的设计？

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

Electronic Devices

Physics



But do we really need to charge them?
Can they work without battery?



如何设计新结构去应对层出不穷的新应用？

Neuromorphic



Neural Network



Large Graph Processing



IoT



Application-Driven Innovations

Computer Architecture Innovations

Heterogeneous Computing

Technology-Driven Innovations

Emerging Technology



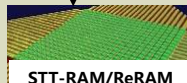
TESLA P100
GPU



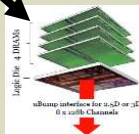
Cambricon 寒武纪
ASIC



Intel HARP
FPGA



**STT-RAM/ReRAM
PCM/Memristor**
HP labs, 2012



3D Stacking



中国科学技术大学
University of Science and Technology of China

计算机系统概论

Introduction to Computing Systems
(CS1002A.02)

Chapter 1-3 Course Implementation

计算机科学与技术学院
School of Computer Science and Technology



- 1 Course and Crew
- 2 What's the difference between "Big" and "Small"?
- 3 Why Take This Course?
- 4 Great Ideas in Computing Systems
- 5 What' s This Course All About ?
- 6 Summary

■ Introduction to computer architecture(ISA)

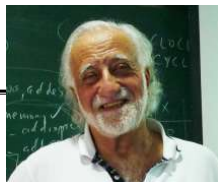
- How is data represented?
- What are the pieces of a computer?
- How do computers work?

■ Programming

- How do I "talk" directly to the machine?-Assembly language
- How do I program in "C"?-high level language(HLL)

■ Computer systems and computation

- How do simple HW/SW elements come together to realize complex computations?



Introduction to Computing Systems: from bits and gates to C/C++ and beyond(3rd edition),

**Yale N. Patt and Sanjay J. Patel
, September 2019, McGraw-Hill Higher Education**

Text Book Components

■ Part 1: Hardware(Chapter 1-4)

- Representing data, transistors, gates, digital logic structures
- von Neumann machine model

■ Part 2: Software: Assembly language(Chapter 5-10)

- Instructions, (structured) programming, input/output, *relationship to hardware*

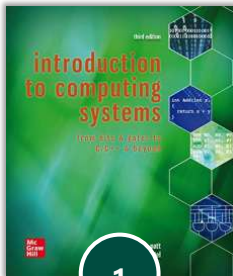
■ Part 3: Software: C programming(Chapter11-19), selected

- Syntax, operators, control structures, functions, *pointers*, recursion, data structures, *relationship to assembly language*
- Assume already familiar with programming (C)

This Course Focus on

- **Chapt 2 Bits, Data Types, and Operations**
 - How do we represent information using electrical signals?
- **Chapt 3 Digital Logic Structures**
 - How do we build circuits to process information?
- **Chapt 4, 5,6 Computer Machine Model, Processor and Instruction Set**
 - How do we build a processor out of logic elements?
 - What operations (instructions) will we implement?
- **Chapt 7 Assembly Language Programming**
 - How do we use processor instructions to implement algorithms?
 - How do we write modular, reusable code? (subroutines)
- **Chapt 8 Data Structures**
- **Chapt 9 I/O, Traps, and Interrupts**
 - How does a processor communicate with outside world?
- **Chapt 11, C Implementation related to hardware**

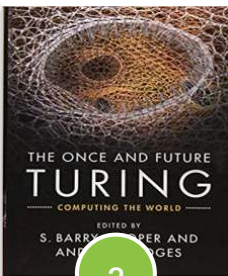
Other Reference Text Books



1

Introduction to Computing Systems: from bits and gates to C/C++ and beyond(3rd edition),

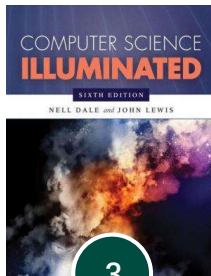
Yale N. Patt and Sanjay J. Patel, June 2019, McGraw-Hill Higher Education



2

The Once and Future Turing: Computing the World (1st Edition) ,

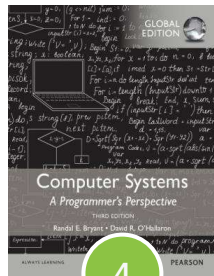
S. Barry Cooper, Andrew Hodges, Cambridge University Press, 2016



3

Computer Science Illuminated(6th Edition) ,

D. M. Harris, S. L. Harris, Morgan Kaufmann, San Francisco, 2016

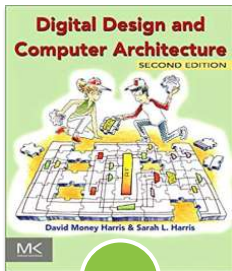


4

Computer Systems: A Programmer 's Perspective (3rd Edition) ,

Randal E. Bryant, David R. O' Hallaron, Pearson Education Inc. , 2016

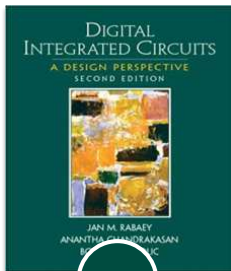
Other Reference Text Books



5

Digital Design and Computer Architecture (2nd Edition) ,

David Harris Sarah Harris, p712, Morgan Kaufmann, 24th July 2012



6

Digital Integrated Circuits: A Design Perspective (2nd Edition),

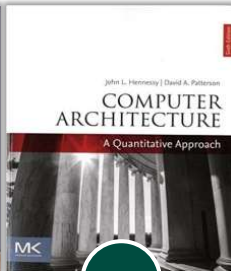
Jan M. Rabaey, Anantha Chandrakasan, Borivoje, Prentice-Hall, Inc, Nikolic, Jan 3, 2003



7

Computer Organization and Design: The Hardware/Software Interface,

David A Patterson, John L. Hennessy, 5th edition. Morgan Kaufmann Publishers, Inc., 2017



8

Computer Architecture: A Quantitative Approach,

John L. Hennessy and David A. Patterson, The Morgan Kaufmann , Dec 7, 2017

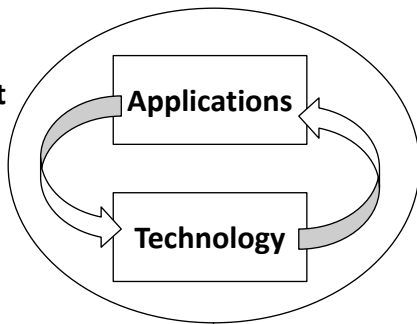
Background/Prerequisites

■ Requirement

- background in programming(C)
- Assume you can program/debug in C

Architecture continually changing

Applications suggest how to improve technology, provide revenue to fund development

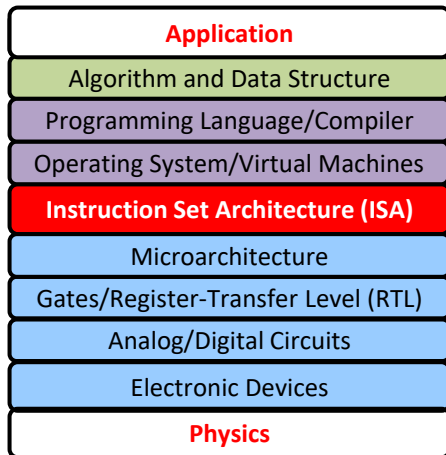


Improved **technologies** make new applications possible

Compatibility

Cost of software development makes **compatibility** a major force in market

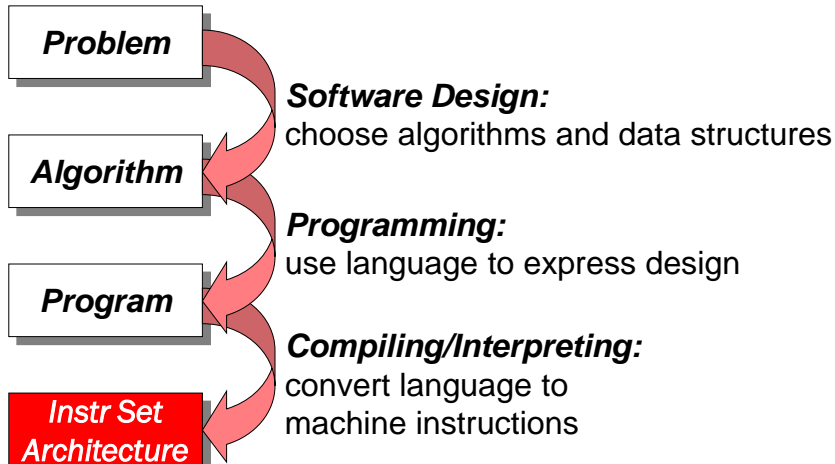
Abstraction Layers in Modern Systems



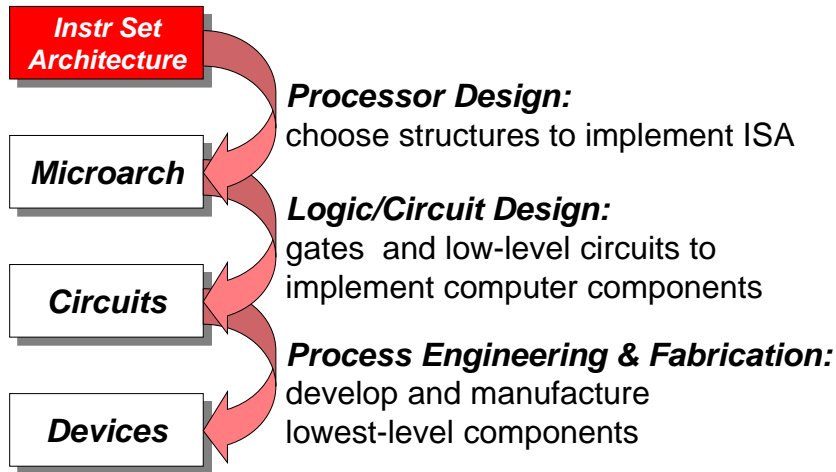
Transformations Between Layers

How do we solve a problem using a computer?

A systematic sequence of transformations between layers of abstraction.



Deeper and Deeper...



Descriptions of Each Level

■ Problem Statement

- stated using "natural language"
- may be ambiguous, imprecise

■ Algorithm

- step-by-step procedure, guaranteed to finish
- definiteness, effective computability, finiteness

■ Program

- express the algorithm using a computer language
- high-level language, low-level language

■ Instruction Set Architecture (ISA)

- specifies the set of instructions the computer can perform
- data types, addressing mode

Descriptions of Each Level (cont.)

■ Microarchitecture

- detailed organization of a processor implementation
- different implementations of a single ISA

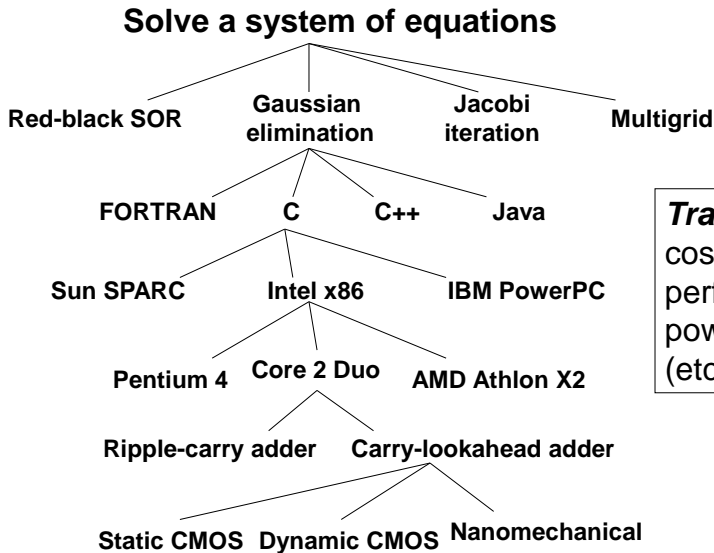
■ Logic Circuits

- combine basic operations to realize microarchitecture
- many different ways to implement a single function (e.g., addition)

■ Devices

- properties of materials, manufacturability

Many Choices at Each Level



Tradeoffs:

cost
performance
power
(etc.)

Course Objectives

- Understand role & relationship of hardware and software
 - Exposure to. . .
 - Machine organization
 - Assembly language programming
 - C programming
 - Understand how to build entire (slow) computing system
 - Hardware and software
 - You' ll get a chance in complementary courses
 - Be distinguished from mere programmers
- 计算机系统（晶体管器件、数字逻辑、组成原理、高级语言的编译与汇编、高级语言的硬件实现、操作系统）核心概念和思想的最小集

Course Objectives

“Any sufficiently advanced technology is indistinguishable from magic.”

Arthur C. Clarke, "Profiles of The Future" (Clarke's 3rd law)

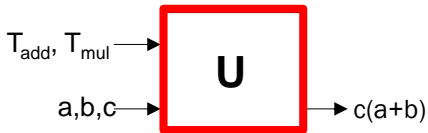
- **No magic:** Computers should **not** be magic to computer scientists!
- **Bottom UP:** Start with what they “know”
 - Computing systems from **transistors** on up
 - The transistor as light switch
 - Not quantum mechanics
- **Choose a computer model that is simple**
 - **Not about “design”, but about “insight” into all computers**
 - As the genius said: simple, but still rich
 - Continually build on what you know
 - Continually raising the level of abstraction
 - Memorizing as little as absolutely necessary
 - Trying very hard to not introduce magic

You take, You enjoy!!!

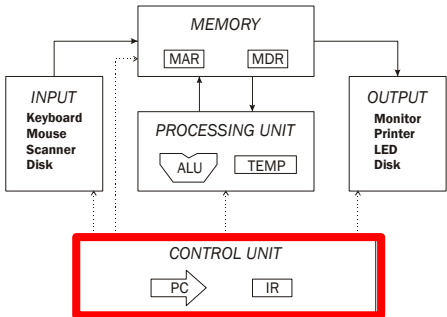
Courses Contents

1. *Overview*
12. *Transition to C*
13. *Programming in C*
8. *Programming and Debugging*
11. *Subroutines, Calls, traps, interrupts*
10. *Physical I/O*
9. *Assembly Language programming*
2. *Operations on bits, bytes (arithmetic, logical)*
7. *The LC-3 Instruction set architecture*
6. *The Von Neumann model*
5. *The finite state machine*
4. *Digital Logic*
3. *The transistor*

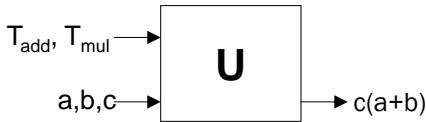
Computational model (Turing machine) vs. Structural model (Von structure)



Universal Turing Machine



Computational model (Turing machine) vs. Structural model (Von structure)



Universal Turing Machine

568

appendix c The Microarchitecture of the LC-3

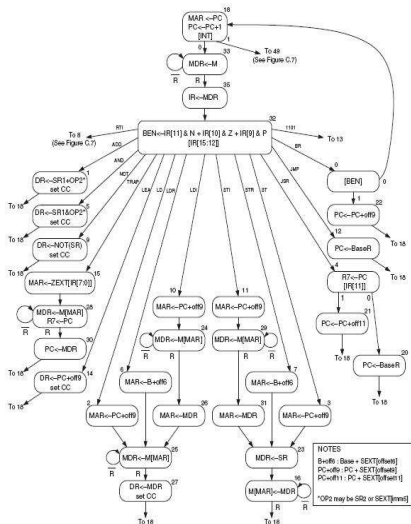


Figure C.2 A state machine for the LC-3

Instruction Set Architecture (ISA) vs. Finite State Machine

A.3 The Instruction Set

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------------------|------|----|----|----|-----|----|---|------|-------|---|---|---|---|---|------|------------|
| ADD ⁺ | 0001 | | | | DR | | | SR1 | | 0 | 0 | 0 | | | SR2 | |
| ADD ⁺ | 0001 | | | | DR | | | SR1 | | 1 | | | | | imm5 | |
| AND ⁺ | 0101 | | | | DR | | | SR1 | | 0 | 0 | 0 | | | SR2 | |
| AND ⁺ | 0101 | | | | DR | | | SR1 | | 1 | | | | | imm5 | |
| BR | 0000 | | n | | z | | p | | | | | | | | | PCoffset9 |
| JMP | 1100 | | | | 000 | | | | BaseR | | | | | | | 000000 |
| JSR | 0100 | | | 1 | | | | | | | | | | | | PCoffset11 |
| JSRR | 0100 | | 0 | | 0 | 0 | | | BaseR | | | | | | | 000000 |
| LD ⁺ | 0010 | | | | DR | | | | | | | | | | | PCoffset9 |
| LDI ⁺ | 1010 | | | | DR | | | | | | | | | | | PCoffset9 |
| LDR ⁺ | 0110 | | | | DR | | | | BaseR | | | | | | | offset8 |
| LEA ⁺ | 1110 | | | | DR | | | | | | | | | | | PCoffset9 |
| NOT ⁺ | 1001 | | | | DR | | | SR | | | | | | | | 111111 |
| RET | 1100 | | | | 000 | | | 111 | | | | | | | | 000000 |
| RTI | 1000 | | | | | | | | | | | | | | | 0000000000 |
| ST | 0011 | | | | SR | | | | | | | | | | | PCoffset9 |
| STI | 1011 | | | | SR | | | | | | | | | | | PCoffset9 |
| STR | 0111 | | | | SR | | | | BaseR | | | | | | | offset8 |
| TRAP | 1111 | | | | | | | 0000 | | | | | | | | trapvec8 |
| reserved | 1101 | | | | | | | | | | | | | | | |

Figure A.2 Format of the entire LC-3 instruction set. Note: + indicates instructions that modify condition codes

568

Appendix C The Microarchitecture of the LC-3

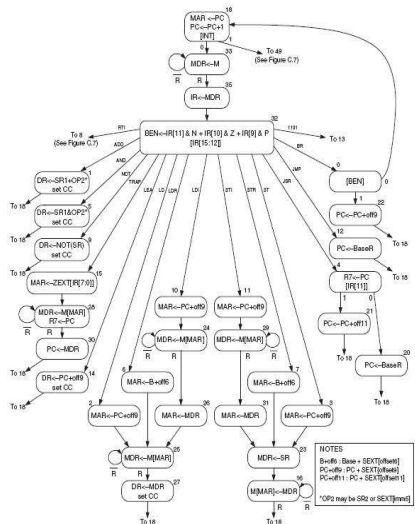


Figure C.2 A state machine for the LC-3

Instruction Set Architecture (ISA) vs. Finite State Machine

A.3 The Instruction Set

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------------------|------|----|----|----|--------------|----|---|-------------|---|-----------|---------|------|---|-----|---|---|
| ADD ⁺ | 0001 | | | | DR | | | SR1 | | | 0 | 00 | | SR2 | | |
| ADD ⁺ | 0001 | | | | DR | | | SR1 | | | 1 | imm5 | | | | |
| AND ⁺ | 0101 | | | | DR | | | SR1 | | | 0 | 00 | | SR2 | | |
| AND ⁺ | 0101 | | | | DR | | | SR1 | | | 1 | imm5 | | | | |
| BR | 0000 | | | | n | z | | p | | PCoffset9 | | | | | | |
| JMP | 1100 | | | | 000 | | | BaseR | | | 000000 | | | | | |
| JSR | 0100 | | | | 1 | | | PCoffset11 | | | | | | | | |
| JSRR | 0100 | | | | 0 00 | | | BaseR | | | 000000 | | | | | |
| LD ⁺ | 0010 | | | | DR | | | PCoffset9 | | | | | | | | |
| LDI ⁺ | 1010 | | | | DR | | | i PCoffset9 | | | | | | | | |
| LDR ⁺ | 0110 | | | | DR | | | BaseR | | | offset6 | | | | | |
| LEA ⁺ | 1110 | | | | DR | | | PCoffset9 | | | | | | | | |
| NOT ⁺ | 1001 | | | | DR | | | SR | | | 111111 | | | | | |
| RET | 1100 | | | | 000 | | | 111 | | | 000000 | | | | | |
| RTI | 1000 | | | | 000000000000 | | | | | | | | | | | |
| ST | 0011 | | | | SR | | | PCoffset9 | | | | | | | | |
| STI | 1011 | | | | SR | | | PCoffset9 | | | | | | | | |
| STR | 0111 | | | | SR | | | BaseR | | | offset6 | | | | | |
| TRAP | 1111 | | | | 0000 | | | trapvect8 | | | | | | | | |
| reserved | 1101 | | | | | | | | | | | | | | | |

Figure A.2 Format of the entire LC-3 instruction set. Note: + indicates instructions that modify condition codes

568

appendix c The Microarchitecture of the LC-3

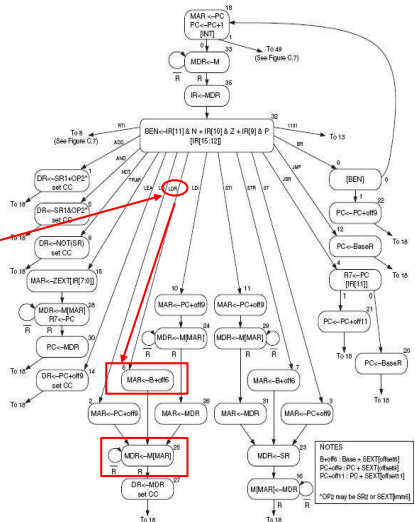


Figure C.2 A state machine for the LC-3

Computer microarchitecture vs. Finite State Machine

570

Appendix C The Microarchitecture of the LC-3

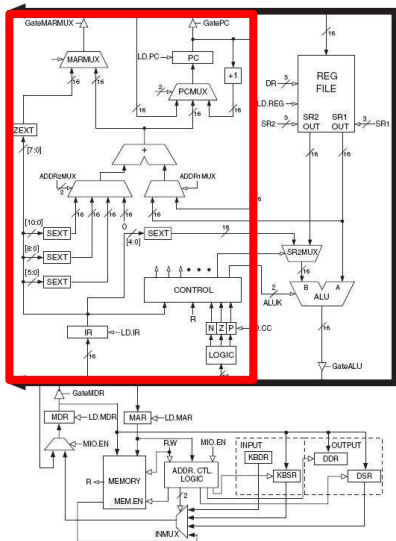


Figure C.3 The LC-3 data path

568

Appendix C The Microarchitecture of the LC-3

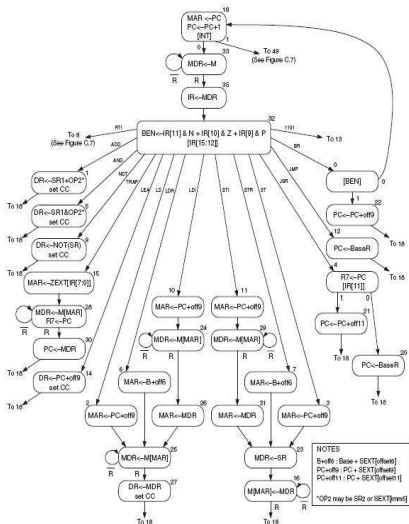


Figure C.2 A state machine for the LC-3

Levels of Abstraction

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

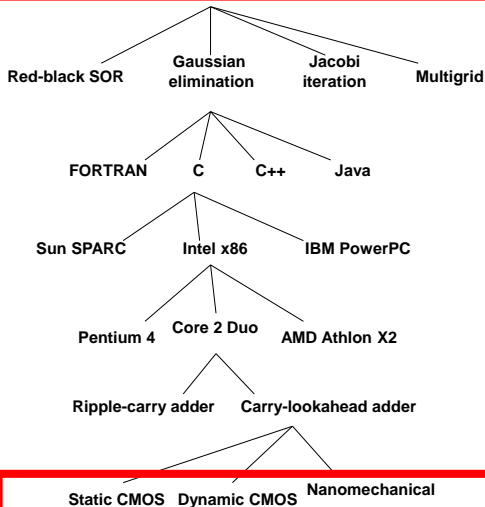
Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

Electronic Devices

Physics

Solve a system of equations



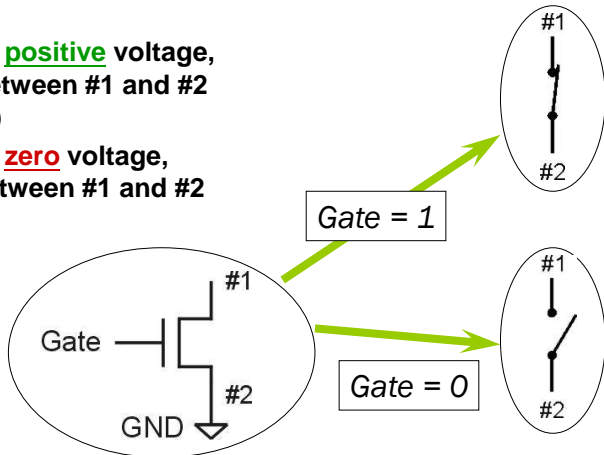
N-type MOS Transistor

■ MOS = Metal Oxide Semiconductor

- two types: N-type and P-type

■ N-type

- when Gate has positive voltage, short circuit between #1 and #2 (switch closed)
- when Gate has zero voltage, open circuit between #1 and #2 (switch open)

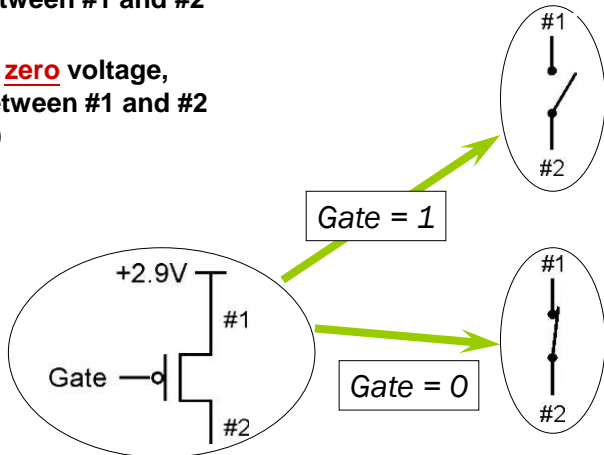


Terminal #2 must be connected to GND (0V).

P-type MOS Transistor

■ P-type is *complementary* to N-type

- when Gate has positive voltage, open circuit between #1 and #2 (switch open)
- when Gate has zero voltage, short circuit between #1 and #2 (switch closed)



Terminal #1 must be connected to +2.9V.

Levels of Abstraction

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

Electronic Devices

Physics

Solve a system of equations

Red-black SOR Gaussian elimination Jacobi iteration Multigrid

FORTTRAN C C++ Java

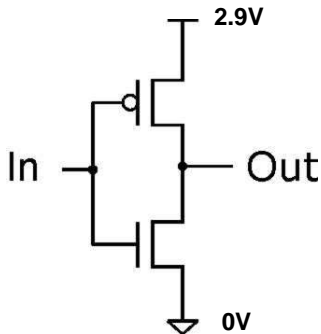
Sun SPARC Intel x86 IBM PowerPC

Pentium 4 Core 2 Duo AMD Athlon X2

Ripple-carry adder Carry-lookahead adder

Static CMOS Dynamic CMOS Nanomechanical

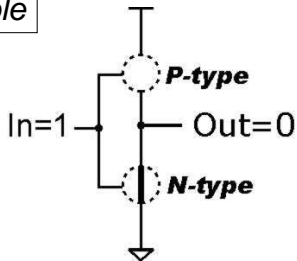
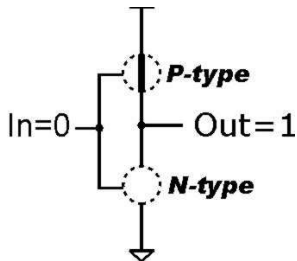
Inverter (NOT Gate)



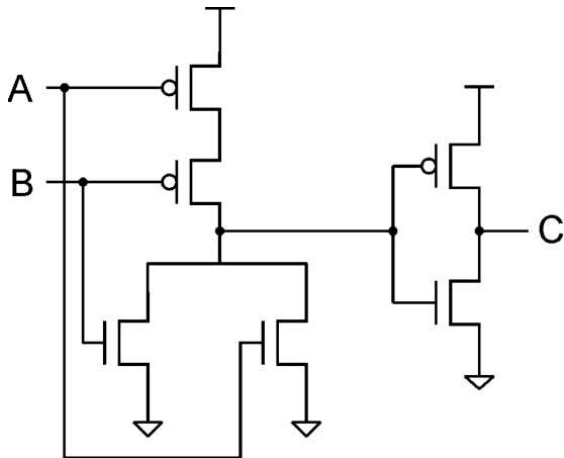
Truth table

| In | Out |
|-------|-------|
| 0 V | 2.9 V |
| 2.9 V | 0 V |

| In | Out |
|----|-----|
| 0 | 1 |
| 1 | 0 |



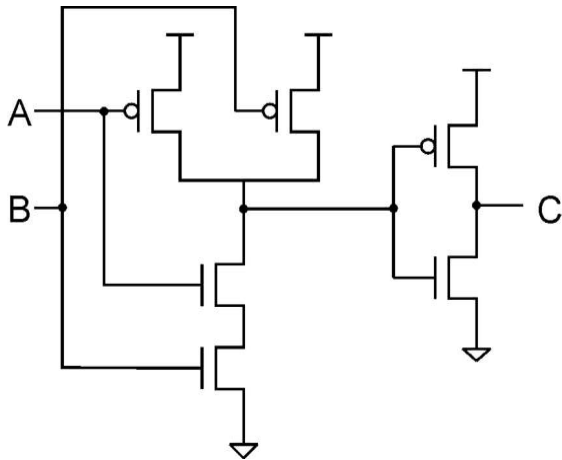
OR Gate



| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Add inverter to NOR.

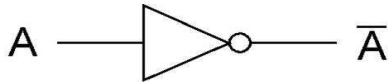
AND Gate



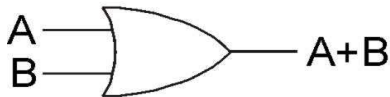
| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Add inverter to NAND.

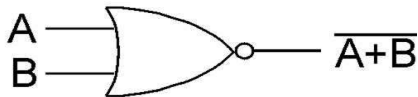
Gates



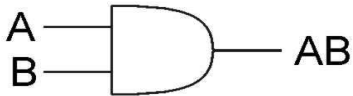
NOT



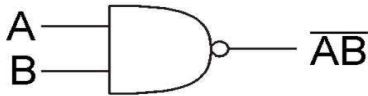
OR



NOR



AND



NAND

Levels of Abstraction

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

Electronic Devices

Physics

Solve a system of equations

Red-black SOR Gaussian elimination Jacobi iteration Multigrid

FORTTRAN C C++ Java

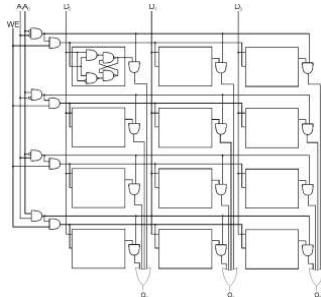
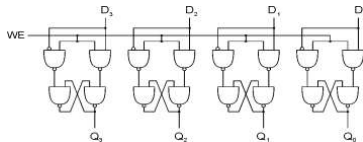
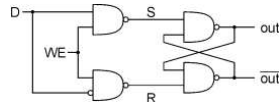
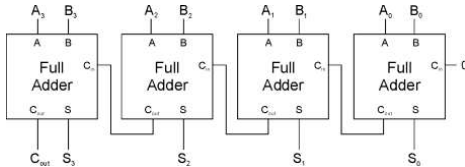
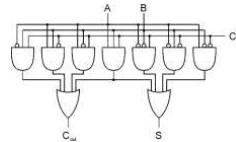
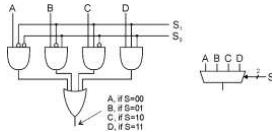
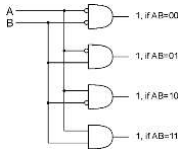
Sun SPARC Intel x86 IBM PowerPC

Pentium 4 Core 2 Duo AMD Athlon X2

Ripple-carry adder Carry-lookahead adder

Static CMOS Dynamic CMOS Nanomechanical

Basic Logical Structure



Levels of Abstraction

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

Electronic Devices

Physics

Solve a system of equations

Red-black SOR Gaussian elimination Jacobi iteration Multigrid

FORTTRAN C C++ Java

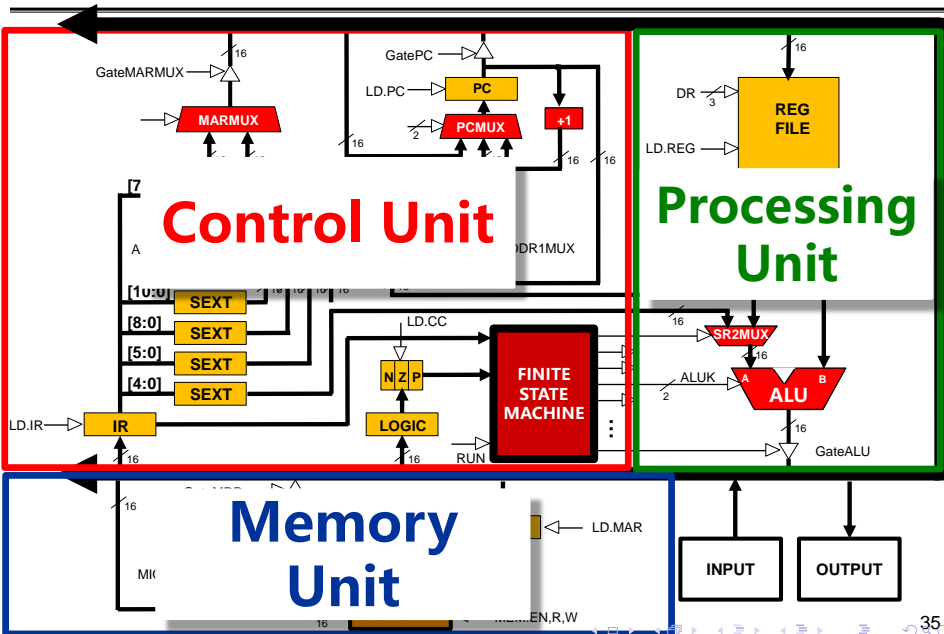
Sun SPARC Intel x86 IBM PowerPC

Pentium 4 Core 2 Duo AMD Athlon X2

Ripple-carry adder Carry-lookahead adder

Static CMOS Dynamic CMOS Nanomechanical

LC-3 Data Path(Microarchitecture)



Levels of Abstraction

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

Electronic Devices

Physics

Solve a system of equations

Red-black SOR Gaussian elimination Jacobi iteration Multigrid

FORTTRAN C C++ Java

Sun SPARC Intel x86 IBM PowerPC

Pentium 4 Core 2 Duo AMD Athlon X2

Ripple-carry adder Carry-lookahead adder

Static CMOS Dynamic CMOS Nanomechanical

LC-3 ISA Overview

运算指令(Operate Instructions)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|----|----|----|----|----|----|---|-----|---|---|---|------|---|-----|---|---|
| ADD | 0 | 0 | 0 | 1 | DR | | | SR1 | | | 0 | 0 | 0 | SR2 | | |
| ADD | 0 | 0 | 0 | 1 | DR | | | SR1 | | | 1 | Imm5 | | | | |
| AND | 0 | 1 | 0 | 1 | DR | | | SR1 | | | 0 | 0 | 0 | SR2 | | |
| AND | 0 | 1 | 0 | 1 | DR | | | SR1 | | | 1 | Imm5 | | | | |
| NOT | 1 | 0 | 0 | 1 | DR | | | SR1 | | | 1 | 1 | 1 | 1 | 1 | 1 |
| Reserved | 1 | 1 | 0 | 1 | | | | | | | | | | | | |

控制指令(Control Instructions)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|------------|---|-----------|-------------|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | n | z | p | PCoffset9 | | | | | | | | |
| JSR | 0 | 1 | 0 | 0 | 1 | PCoffset11 | | | | | | | | | | |
| JSRR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RET | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| TRAP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | TrapVector8 | | | | | | | |

数据移动指令 (Data Movement Instructions)

取数指令(Load)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|-----------|---|---|---|---|---|---|
| LD | 0 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LDR | 0 | 1 | 1 | 0 | DR | | | BaseR | | PCOffset6 | | | | | | |
| LDI | 1 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LEA | 1 | 1 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |

存数指令(Store)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|---|-----------|---|---|---|---|---|
| ST | 0 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |
| STR | 0 | 1 | 1 | 1 | SR | | | BaseR | | | PCOffset6 | | | | | |
| STI | 1 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |

Class Organization

■ Lectures

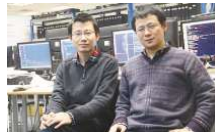
- Will not simply “cover” the material
- Will focus on the “hard stuff”
- Will not stand alone, instead build on reading

■ Guest Lectures

- Turing Machine
- RISC-V
- Cambricon

■ Discussion sessions

- Encouraged! (4 TAs, 4 discussion group, every week!)
- Okay: discuss meaning of problem, discuss approaches
- Not okay: comparing answers, solving questions together



Class Organization



Class Organization



Homework Assignments

■ Problem Sets: 6 sets

- Problem solving
- Complete *before* each due date
- Can work ahead
- Great exam preparation!

EPA vs. GPA

■ Effort

- Attending prof and TA office hours, completing all assignments, turning in HW, doing reading quizzes

■ Participation

- Attending lecture
- You have 2 slip day tokens (NOT hour or min)
- Asking great questions in discussion and lecture and making it more interactive

■ Altruism

- Helping others in lab or on Piazza: Be Excellent to Each Other

■ EPA! points have the potential to bump students up to the next grade level!

Tips on How to Get a Good Grade

- The lecture material is **not** the most challenging part of the course. You should be able to understand everything as we go along.
- DO NOT fall behind in lecture and tell yourself you “will figure it out later from the notes or books” .
- Notes will be **online** after the lecture (usually the night). Do assigned reading before the lecture.
- Ask questions in class and stay involved in the class - that will help you understand.
- Discuss with TAs to check your understanding or to ask questions.
- Complete all the homework problems - even the difficult ones. The exams will test your depth of knowledge.

Do not post your work on public repositories like github (private o.k.) --negative points

■ The rule is simple

- Claiming another's work as your own *will ruin your life*
- See syllabus for details and examples

■ Who will know?

- We will (inspection, similarity detectors, exams)
- Your friends will... your parents will...
- You will

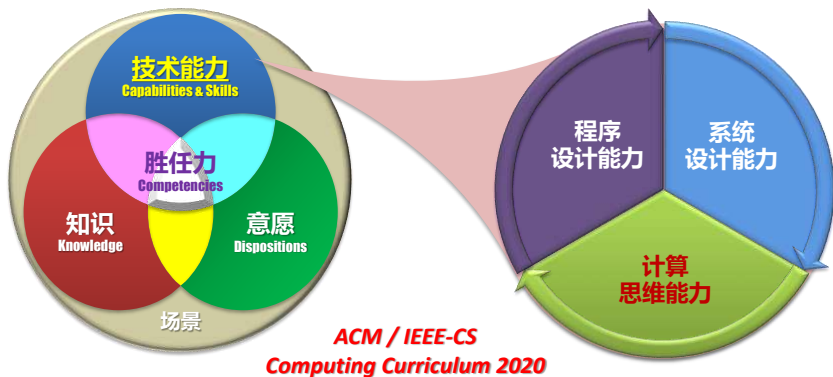
■ Remember

- If you need to cheat now, you've got much bigger problems
- **Cheating is like going 150 MPH over speed limit while drunk!**

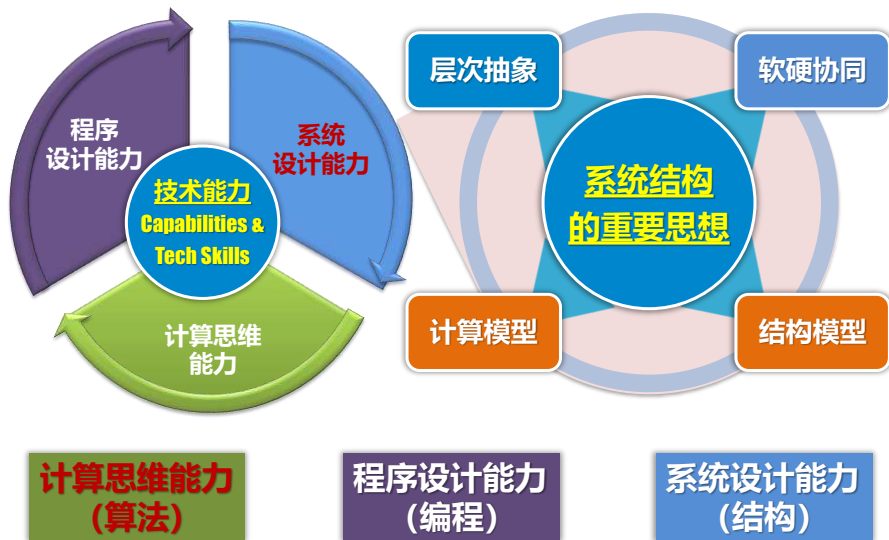


- 1 Course and Crew
- 2 What's the difference between "Big" and "Small"?
- 3 Why Take This Course?
- 4 Great Ideas in Computing Systems
- 5 What' s This Course All About ?
- 6 Summary

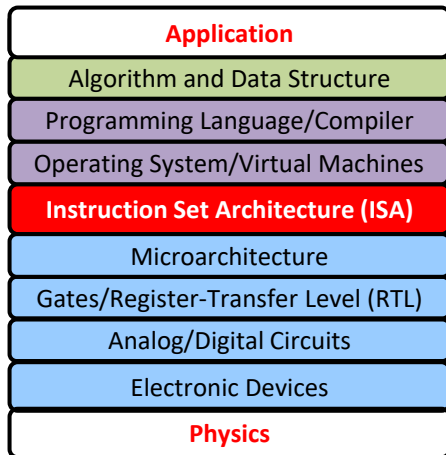
面向世界，面向未来，培养一流人才



课程目标：理解计算机系统结构的重要思想



Abstraction Layers in Modern Systems



Acknowledgements

- **Special thanks to Prof. An**
- **This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:**
 - Arvind (MIT)
 - Joel Emer (Intel/MIT)
 - James Hoe (CMU)
 - John Kubiatowicz (UCB)
 - David Patterson (UCB)
 - Krste Asanovic (UCB)



中国科学技术大学

University of Science and Technology of China

计算机系统概论

Introduction to Computing Systems

(CS1002A.02)

Chapter 2-1

Bits, Bytes and Data Types

计算机科学与技术学院

School of Computer Science and Technology

- 1 How do we represent data in a computer?
- 2 Integer Data Types
- 3 2' Complement Integers
- 4 Binary-Decimal Conversion

1 How do we represent data in a computer?

2 Integer Data Types

3 2' Complement Integers

4 Binary-Decimal Conversion

How do we represent data in a computer?

How do we represent data in a computer?

Great Idea from Ancient Chinese Philosophy

All things come into being, all things come into nothing

天下万物生于有, 有生于无

《老子·四十章》



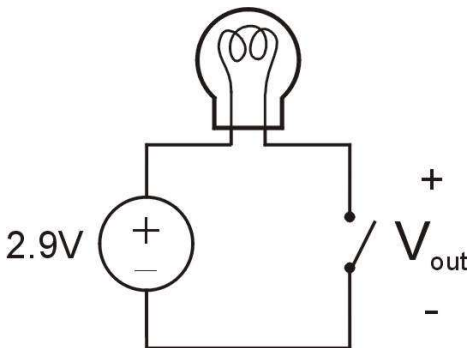
《易经》

太极生两仪,
两仪生四象,
四象生八卦,
八卦演万物。

How do we represent data in a computer?

- At the lowest level, a computer is an electronic machine.
 - works by controlling the flow of electrons
- Easy to recognize two conditions:
 - presence of a voltage – we'll call this state "1"
 - absence of a voltage – we'll call this state "0"
- Could base state on *value* of voltage, but control and detection circuits more complex.
 - compare turning on a light switch to measuring or regulating voltage
- We'll see examples of these circuits in the next chapter.

Simple Switch Circuit



Switch **open**:

- No current through circuit
- Light is **off**
- V_{out} is **+2.9V**

Switch **closed**:

- Short circuit across switch
- Current flows
- Light is **on**
- V_{out} is **0V**

Switch-based circuits can easily represent two states:
on/off, open/closed, voltage/no voltage.

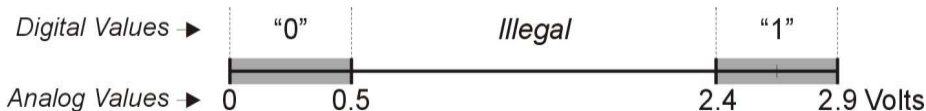
Computer is a binary digital system.

Digital system:

- finite number of symbols

Binary (base two) system:

- has two states: 0 and 1



■ Basic unit of information is the *binary digit*, or **bit**.
Values with more than two states require multiple bits.

- A collection of **two** bits has **four** possible states:
00, 01, 10, 11
- A collection of **three** bits has **eight** possible states:
000, 001, 010, 011, 100, 101, 110, 111
- A collection of **n** bits has **2^n** possible states.

Data input: Analog → Digital

■ Real world is analog!

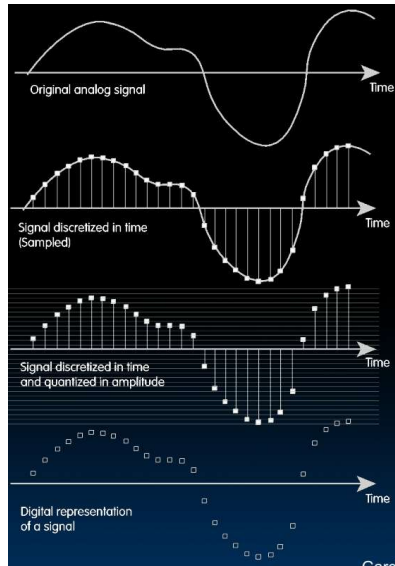
■ To import analog information, we must do two things

● Sample

- E.g., for a CD, every 44,100 ths of a second, we ask a music signal how loud it is.

● Quantize

- For every one of these samples, we figure out where, on a 16-bit (65,536 tic-mark) “y



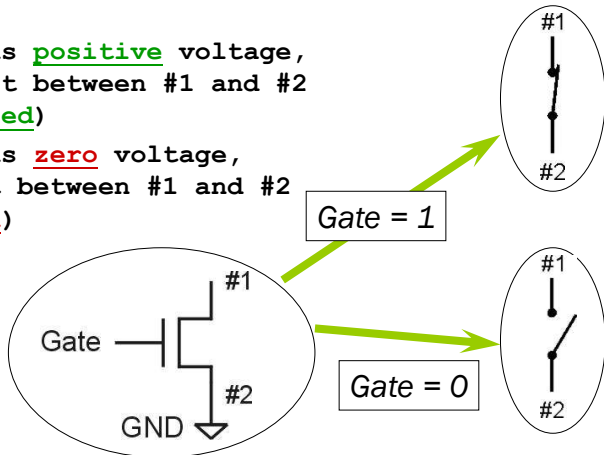
N-type MOS Transistor

■ MOS = Metal Oxide Semiconductor

- two types: N-type and P-type

■ N-type

- when Gate has positive voltage, short circuit between #1 and #2 (switch closed)
- when Gate has zero voltage, open circuit between #1 and #2 (switch open)

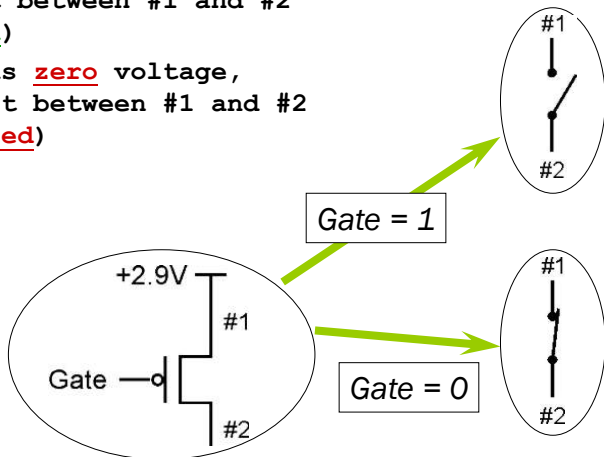


Terminal #2 must be connected to GND (0V).

P-type MOS Transistor

■ P-type is *complementary* to N-type

- when Gate has positive voltage, open circuit between #1 and #2 (switch open)
- when Gate has zero voltage, short circuit between #1 and #2 (switch closed)



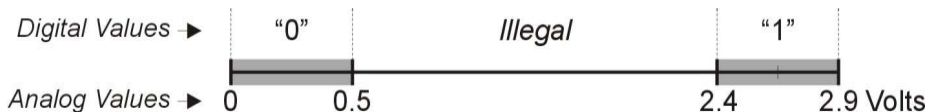
Terminal #1 must be connected to +2.9V.

Logic Gates

- Use switch behavior of MOS transistors to implement logical functions: AND, OR, NOT.

- Digital symbols:

- recall that we assign a range of analog voltages to each digital (logic) symbol



- assignment of voltage ranges depends on electrical properties of transistors being used
 - typical values for "1": +5V, +3.3V, +2.9V, +1.1V
 - for purposes of illustration, we'll use +2.9V

Within the Computer: Everything is a Number.

■ Numbers within the Computer

- Base 10 #s:

Dec(imal)

– Digits:

0,1,2,3,4,5,6,7,8,9

- Base 2 #s:

Bin(ary)

– Digits: 0,1

- Base 8 #s: Oct(al)

– Digits: 0,1,2,3,4,5,6,7

- Base 16 #s:

Hex(adecimal)

– Digits:

0,1,2,3,4,5,6,7,8,9,A,B,
C,D,E,F

| Dec(imal) | Hex(adecimal) | Oct(al) | Bin(ary) |
|------------|---------------|---------|----------|
| 00 | 0 | 00 | 0000 |
| 01 | 1 | 01 | 0001 |
| 02 | 2 | 02 | 0010 |
| 03 | 3 | 03 | 0011 |
| 04 | 4 | 04 | 0100 |
| 05 | 5 | 05 | 0101 |
| 06 | 6 | 06 | 0110 |
| 07 | 7 | 07 | 0111 |
| 08 | 8 | 10 | 1000 |
| 09 | 9 | 11 | 1001 |
| 10 | A | 12 | 1010 |
| 11 | B | 13 | 1011 |
| 12 | C | 14 | 1100 |
| 13 | D | 15 | 1101 |
| 14 | E | 16 | 1110 |
| 15 | F | 17 | 1111 |

Hexadecimal Notation

■ It is often convenient to write binary (base-2) numbers as hexadecimal (base-16) numbers instead.

- fewer digits -- four bits per hex digit
- less error prone -- easy to corrupt long string of 1's and 0's

| Binary | Hex | Decimal |
|--------|-----|---------|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |

| Binary | Hex | Decimal |
|--------|-----|---------|
| 1000 | 8 | 8 |
| 1001 | 9 | 9 |
| 1010 | A | 10 |
| 1011 | B | 11 |
| 1100 | C | 12 |
| 1101 | D | 13 |
| 1110 | E | 14 |
| 1111 | F | 15 |

011101010001111010011010111

Converting from Binary to Hexadecimal

■ Every four bits is a hex digit.

- start grouping from right-hand side

| | | | | | | |
|------|------|------|------|------|------|------|
| 0111 | 0101 | 0100 | 0111 | 1010 | 0110 | 1011 |
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| 3 | A | 8 | F | 4 | D | 7 |

Converting from Binary to Hexadecimal

- Every four bits is a hex digit.
 - start grouping from right-hand side

3 A 8 F 4 D 7

***This is not a new machine representation,
just a convenient way to write the number.***

Within the Computer: Everything is a Number.

■ Which base do we use?

```
#include <stdio.h>

int main() {
    const int N = 1234;

    printf("Decimal: %d\n", N);
    printf("Hex:      %x\n", N);
    printf("Octal:     %o\n", N);

    printf("Literals (not supported by all compilers):\n");
    printf("0x4d2      = %d (hex)\n", 0x4d2);
    printf("0b10011010010 = %d (binary)\n", 0b10011010010);
    printf("02322      = %d (octal, prefix 0 - zero)\n", 02322);
}
```

Output

```
Decimal: 1234
Hex:      4d2
Octal:     2322
Literals (not supported by all compilers):
0x4d2      = 1234 (hex)
0b10011010010 = 1234 (binary)
02322      = 1234 (octal, prefix 0 - zero)
```


Within the Computer: Everything is a Number.

■ Bit(BIrary digiT)

- 1Bits=2things;
- 2Bits=4things;
- 4Bits=16things;
- 8Bits=256things
-

■ Byte

- 1Byte=8Bits
- A byte is 8 bits

■ But numbers usually stored with a fixed size

- 8-bit bytes;
- 16-bit **half words**;
- 32-bit words;
- 64-bit double words, ...
- And there are really only two primitive "numbers":
0 and 1 is a "bit"

BIG IDEA: Bits can represent anything!!!

■ Characters?

- 26 letters \Rightarrow 5 bits ($2^5 = 32$)
- upper/lower case + punctuation \Rightarrow 7 bits (in 8) ("ASCII")
- standard code to cover all the world's languages \Rightarrow 8,16,32 bits ("Unicode") www.unicode.com

■ Logical values?

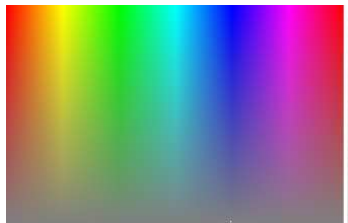
- 0 \rightarrow False, 1 \rightarrow True

■ colors ?

- Ex: Red(00) , Green(01) , Blue(11)

■ locations / addresses?

■ commands?



MEMORIZE: N bits \Leftrightarrow at most 2^N things

What kinds of data do we need to represent?

■ Kinds of data

- **Numbers** - signed, unsigned, integers, floating point, complex, rational, irrational, ...
- **Text** - characters, strings, ...
- **Logical** - true, false
- **Images** - pixels, colors, shapes, ...
- **Sound**
- **Video** - a series of images
- **Instructions**
- ...

- **Data type:** *representation* and *operations* within the computer

We' ll start with numbers...

Outline

1 How do we represent data in a computer?

2 Integer Data Types

3 2' Complement Integers

4 Binary-Decimal Conversion

结绳记数

- 名称：(Quipu) 基普结绳
- 产地：南美印加部落
- 时间：16世纪



■ Non-positional notation

- Could represent a number ("5") with a string of ones ("11111") problems?



Unsigned Integers

■ Weighted positional notation

- like decimal numbers: "329"
- "3" is worth 300, because of its position, while "9" is only worth 9

$$\begin{array}{ccc} & 329 & \\ / & | & \backslash \\ 10^2 & 10^1 & 10^0 \end{array}$$

$$3 \times 100 + 2 \times 10 + 9 \times 1 = 329$$

$$\begin{array}{ccc} \text{most} & & \text{least} \\ \text{significant} & \xrightarrow{\quad} & \text{significant} \\ & 101 & \\ / & | & \backslash \\ 2^2 & 2^1 & 2^0 \end{array}$$

$$1 \times 4 + 0 \times 2 + 1 \times 1 = 5$$

Unsigned Integers (cont.)

■ An n -bit unsigned integer represents 2^n values: from 0 to $2^n - 1$.

| 2^2 | 2^1 | 2^0 | |
|-------|-------|-------|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 2 |
| 0 | 1 | 1 | 3 |
| 1 | 0 | 0 | 4 |
| 1 | 0 | 1 | 5 |
| 1 | 1 | 0 | 6 |
| 1 | 1 | 1 | 7 |

Unsigned Binary Arithmetic

■ Base-2 addition – just like base-10!

- add from right to left, propagating carry

$$\begin{array}{r} 10010 \\ + \underline{1001} \\ 11011 \end{array}$$
$$\begin{array}{r} \text{carry} \downarrow \\ 10010 \\ + \underline{1011} \\ 11101 \end{array}$$
$$\begin{array}{r} \text{carry} \downarrow \downarrow \downarrow \downarrow \\ 1111 \\ + \underline{1} \\ 10000 \end{array}$$
$$\begin{array}{r} 10111 \\ + \underline{111} \end{array}$$

Subtraction, multiplication, division,...

Unsigned Integers (cont.)

0000 0000 0000 0000 0000 0000 0000 0000_{two} = 0_{ten}

0000 0000 0000 0000 0000 0000 0000 0001_{two} = 1_{ten}

0000 0000 0000 0000 0000 0000 0000 0010_{two} = 2_{ten}

...

...

0111 1111 1111 1111 1111 1111 1111 1101_{two} = 2,147,483,645_{ten}

0111 1111 1111 1111 1111 1111 1111 1110_{two} = 2,147,483,646_{ten}

0111 1111 1111 1111 1111 1111 1111 1111_{two} = 2,147,483,647_{ten}

1000 0000 0000 0000 0000 0000 0000 0000_{two} = 2,147,483,648_{ten}

1000 0000 0000 0000 0000 0000 0000 0001_{two} = 2,147,483,649_{ten}

1000 0000 0000 0000 0000 0000 0000 0010_{two} = 2,147,483,650_{ten}

...

...

1111 1111 1111 1111 1111 1111 1111 1101_{two} = 4,294,967,293_{ten}

1111 1111 1111 1111 1111 1111 1111 1110_{two} = 4,294,967,294_{ten}

1111 1111 1111 1111 1111 1111 1111 1111_{two} = 4,294,967,295_{ten}

Signed Integers

■ With n bits, we have 2^n distinct values.

- assign about half to positive integers
(1 through $2^{n-1}-1$)
and about half to negative ($-2^{n-1}+1$ through -1)
- that leaves two values: one for 0, and one extra

■ Positive integers

- just like unsigned - zero in Most Significant (MS) bit

00101 = 5

■ Negative integers

- signed-magnitude - set MS bit to show negative, other bits are the same as unsigned

10101 = -5

- 1's complement - flip every bit to represent negative

11010 = -5

- in either case, MS bit indicates sign: 0=positive, 1=negative

Three representations of signed integers

| Representation | Value Represented | | | Representation | Value Represented | | |
|----------------|-------------------|----------------|----------------|----------------|-------------------|----------------|----------------|
| | Signed Magnitude | 1's Complement | 2's Complement | | Signed Magnitude | 1's Complement | 2's Complement |
| 0 0 0 0 0 | 0 | 0 | 0 | 1 0 0 0 0 | -0 | -15 | -16 |
| 0 0 0 0 1 | 1 | 1 | 1 | 1 0 0 0 1 | -1 | -14 | -15 |
| 0 0 0 1 0 | 2 | 2 | 2 | 1 0 0 1 0 | -2 | -13 | -14 |
| 0 0 0 1 1 | 3 | 3 | 3 | 1 0 0 1 1 | -3 | -12 | -13 |
| 0 0 1 0 0 | 4 | 4 | 4 | 1 0 1 0 0 | -4 | -11 | -12 |
| 0 0 1 0 1 | 5 | 5 | 5 | 1 0 1 0 1 | -5 | -10 | -11 |
| 0 0 1 1 0 | 6 | 6 | 6 | 1 0 1 1 0 | -6 | -9 | -10 |
| 0 0 1 1 1 | 7 | 7 | 7 | 1 0 1 1 1 | -7 | -8 | -9 |
| 0 1 0 0 0 | 8 | 8 | 8 | 1 1 0 0 0 | -8 | -7 | -8 |
| 0 1 0 0 1 | 9 | 9 | 9 | 1 1 0 0 1 | -9 | -6 | -7 |
| 0 1 0 1 0 | 10 | 10 | 10 | 1 1 0 1 0 | -10 | -5 | -6 |
| 0 1 0 1 1 | 11 | 11 | 11 | 1 1 0 1 1 | -11 | -4 | -5 |
| 0 1 1 0 0 | 12 | 12 | 12 | 1 1 1 0 0 | -12 | -3 | -4 |
| 0 1 1 0 1 | 13 | 13 | 13 | 1 1 1 0 1 | -13 | -2 | -3 |
| 0 1 1 1 0 | 14 | 14 | 14 | 1 1 1 1 0 | -14 | -1 | -2 |
| 0 1 1 1 1 | 15 | 15 | 15 | 1 1 1 1 1 | -15 | -0 | -1 |

Signed Magnitude:

$$5 - 5 = -10$$

1's Complement:

$$5 - 5 = -0$$

$$\begin{array}{rcl}
 00101 & (5) \\
 + \underline{10101} & (-5) \\
 \hline
 11010 & (-10)
 \end{array}$$

$$\begin{array}{rcl}
 00101 & (5) \\
 + \underline{11010} & (-5) \\
 \hline
 11111 & (-0)
 \end{array}$$

Outline

1 How do we represent data in a computer?

2 Integer Data Types

3 **2' Complement Integers**

4 Binary-Decimal Conversion

2's Complement Representation




■ If number is positive or zero,

- normal binary representation, zeroes in upper bit(s)

■ If number is negative,

- start with positive number
- flip every bit (i.e., take the 1's complement)
- then add one

■ This representation makes the hardware simple!

| | | | |
|---|------------------|---|------------|
|  | 00101 (5) |  | 01001 (9) |
|  | 11010 (1's comp) | | (1's comp) |
| + | <u>1</u> | + | <u>1</u> |
| | 11011 (-5) | | (-9) |

2's Complement

■ Problems with signed-magnitude and 1's complement

- two representations of zero (+0 and -0)
- arithmetic circuits are complex
 - How to add two signed-magnitude numbers?
 - e.g., try $2 + (-3)$
 - How to add two 1's complement numbers?
 - e.g., try $4 + (-3)$

■ 2's complement representation developed to make circuits easy for arithmetic.


- for each positive number (X), assign value to its negative (-X), such that $X + (-X) = 0$ with "normal" addition, ignoring carry out

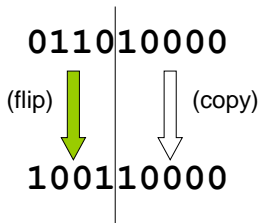
| | |
|---------------------|--------------|
| 00101 (5) | 01001 (9) |
| + <u>11011</u> (-5) | + _____ (-9) |
| 00000 (0) | 00000 (0) |

2's Complement Shortcut

■ To take the 2's complement of a number:

- copy bits from right to left until (and including) the first "1"
- flip remaining bits to the left


$$\begin{array}{r} 011010000 \\ 100101111 \text{ (1's comp)} \\ + \quad \quad \quad 1 \\ \hline 100110000 \end{array}$$


$$\begin{array}{c} 0110 \mid 10000 \\ \text{(flip)} \downarrow \quad \downarrow \text{(copy)} \\ 1001 \mid 10000 \end{array}$$

2's Complement Signed Integers

- MS bit is sign bit – it has weight -2^{n-1} .
- Range of an n-bit number: -2^{n-1} through $2^{n-1} - 1$.
 - The most negative number (-2^{n-1}) has no positive counterpart.

| -2^3 | 2^2 | 2^1 | 2^0 | | -2^3 | 2^2 | 2^1 | 2^0 | |
|--------|-------|-------|-------|---|--------|-------|-------|-------|----|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | -8 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | -7 |
| 0 | 0 | 1 | 0 | 2 | 1 | 0 | 1 | 0 | -6 |
| 0 | 0 | 1 | 1 | 3 | 1 | 0 | 1 | 1 | -5 |
| 0 | 1 | 0 | 0 | 4 | 1 | 1 | 0 | 0 | -4 |
| 0 | 1 | 0 | 1 | 5 | 1 | 1 | 0 | 1 | -3 |
| 0 | 1 | 1 | 0 | 6 | 1 | 1 | 1 | 0 | -2 |
| 0 | 1 | 1 | 1 | 7 | 1 | 1 | 1 | 1 | -1 |

Three representations of signed integers

| Representation | Value Represented | | | Representation | Value Represented | | |
|----------------|-------------------|----------------|----------------|----------------|-------------------|----------------|----------------|
| | Signed Magnitude | 1's Complement | 2's Complement | | Signed Magnitude | 1's Complement | 2's Complement |
| 0 0 0 0 0 | 0 | 0 | 0 | 1 0 0 0 0 | -0 | -15 | -16 |
| 0 0 0 0 1 | 1 | 1 | 1 | 1 0 0 0 1 | -1 | -14 | -15 |
| 0 0 0 1 0 | 2 | 2 | 2 | 1 0 0 1 0 | -2 | -13 | -14 |
| 0 0 0 1 1 | 3 | 3 | 3 | 1 0 0 1 1 | -3 | -12 | -13 |
| 0 0 1 0 0 | 4 | 4 | 4 | 1 0 1 0 0 | -4 | -11 | -12 |
| 0 0 1 0 1 | 5 | 5 | 5 | 1 0 1 0 1 | -5 | -10 | -11 |
| 0 0 1 1 0 | 6 | 6 | 6 | 1 0 1 1 0 | -6 | -9 | -10 |
| 0 0 1 1 1 | 7 | 7 | 7 | 1 0 1 1 1 | -7 | -8 | -9 |
| 0 1 0 0 0 | 8 | 8 | 8 | 1 1 0 0 0 | -8 | -7 | -8 |
| 0 1 0 0 1 | 9 | 9 | 9 | 1 1 0 0 1 | -9 | -6 | -7 |
| 0 1 0 1 0 | 10 | 10 | 10 | 1 1 0 1 0 | -10 | -5 | -6 |
| 0 1 0 1 1 | 11 | 11 | 11 | 1 1 0 1 1 | -11 | -4 | -5 |
| 0 1 1 0 0 | 12 | 12 | 12 | 1 1 1 0 0 | -12 | -3 | -4 |
| 0 1 1 0 1 | 13 | 13 | 13 | 1 1 1 0 1 | -13 | -2 | -3 |
| 0 1 1 1 0 | 14 | 14 | 14 | 1 1 1 1 0 | -14 | -1 | -2 |
| 0 1 1 1 1 | 15 | 15 | 15 | 1 1 1 1 1 | -15 | -0 | -1 |

Signed Magnitude:

$$5 - 5 = -10$$

1's Complement:

$$5 - 5 = -0$$

2's Complement:

$$5 - 5 = 0$$

$$\begin{array}{rcl}
 00101 & (5) \\
 + \underline{10101} & (-5) \\
 \hline
 11010 & (-10)
 \end{array}$$

$$\begin{array}{rcl}
 00101 & (5) \\
 + \underline{11010} & (-5) \\
 \hline
 11111 & (-0)
 \end{array}$$

$$\begin{array}{rcl}
 00101 & (5) \\
 + \underline{11011} & (-5) \\
 \hline
 00000 & (0)
 \end{array}$$

2's Complement

Sign Bit

0000 0000 0000 0000 0000 0000 0000 0000_{two} = 0_{ten}

0000 0000 0000 0000 0000 0000 0000 0001_{two} = 1_{ten}

0000 0000 0000 0000 0000 0000 0000 0010_{two} = 2_{ten}

...

0111 1111 1111 1111 1111 1111 1111 1101_{two} = 2,147,483,645_{ten}

0111 1111 1111 1111 1111 1111 1111 1110_{two} = 2,147,483,646_{ten}

0111 1111 1111 1111 1111 1111 1111 1111_{two} = 2,147,483,647_{ten}

1000 0000 0000 0000 0000 0000 0000 0000_{two} = -2,147,483,648_{ten}

1000 0000 0000 0000 0000 0000 0000 0001_{two} = -2,147,483,647_{ten}

1000 0000 0000 0000 0000 0000 0000 0010_{two} = -2,147,483,646_{ten}

...

1111 1111 1111 1111 1111 1111 1111 1101_{two} = -3_{ten}

1111 1111 1111 1111 1111 1111 1111 1110_{two} = -2_{ten}

1111 1111 1111 1111 1111 1111 1111 1111_{two} = -1_{ten}

Q&A

- Suppose we had a 5-bit word. What integers can be represented in 2's complement?
 - A. $-32 \sim +31$
 - B. $0 \sim +31$
 - C. $-16 \sim +15$
 - D. $-15 \sim +16$
- Suppose we had a 8-bit word. What integers can be represented in 2's complement?
- Suppose we had a 16-bit word. What integers can be represented in 2's complement?
- Suppose we had a 32-bit word. What integers can be represented in 2's complement?

Q&A

- Suppose we had a 5-bit word. What integers can be represented in 2's complement?
 - A. $-32 \sim +31$
 - B. $0 \sim +31$
 - C. $-16 \sim +15$
 - D. $-15 \sim +16$
- Suppose we had a 8-bit word. What integers can be represented in 2's complement?
- Suppose we had a 16-bit word. What integers can be represented in 2's complement?
- Suppose we had a 32-bit word. What integers can be represented in 2's complement?

Outline

- 1 How do we represent data in a computer?
- 2 Integer Data Types
- 3 2' Complement Integers
- 4 **Binary-Decimal Conversion**

Converting Binary (2's complement) to Decimal

1. If leading bit is one, take 2's complement to get a positive number.
2. Add powers of 2 that have "1" in the corresponding bit positions.
3. If the original number was negative, add a minus sign.

$$\begin{aligned} X &= 01101000_{\text{two}} \\ &= 2^6 + 2^5 + 2^3 = 64 + 32 + 8 \\ &= 104_{\text{ten}} \end{aligned}$$

| n | 2^n |
|-----|-------|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 10 | 1024 |

Assuming 8-bit 2's complement numbers.

More Examples

$$\begin{aligned} X &= 00100111_{\text{two}} \\ &= 2^5 + 2^2 + 2^1 + 2^0 = 32 + 4 + 2 + 1 \\ &= 39_{\text{ten}} \end{aligned}$$

$$\begin{aligned} X &= 11100110_{\text{two}} \\ -X &= 00011010 \\ &= 2^4 + 2^3 + 2^1 = 16 + 8 + 2 \\ &= 26_{\text{ten}} \\ X &= -26_{\text{ten}} \end{aligned}$$

| n | 2^n |
|-----|-------|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 10 | 1024 |

Assuming 8-bit 2's complement numbers.

Converting Decimal to Binary (2's C)

First Method: *Division*

1. Divide by 2 – the remainder is the least significant bit.
2. Keep dividing by 2 until answer is zero, writing remainders from right to left.
3. Append a zero as the MS bit; if the original number is negative, take 2's complement.

$$X = 104_{\text{ten}}$$

$$104/2 = 52 \text{ r}0 \text{ bit } 0$$

$$52/2 = 26 \text{ r}0 \text{ bit } 1$$

$$26/2 = 13 \text{ r}0 \text{ bit } 2$$

$$13/2 = 6 \text{ r}1 \text{ bit } 3$$

$$6/2 = 3 \text{ r}0 \text{ bit } 4$$

$$3/2 = 1 \text{ r}1 \text{ bit } 5$$

$$X = 01101000_{\text{two}}$$

$$1/2 = 0 \text{ r}1 \text{ bit } 6$$

Converting Decimal to Binary (2's C)

Second Method: *Subtract Powers of Two*

1. Change to positive decimal number.
2. Subtract the largest power of two less than or equal to number.
3. Put an 1 in the corresponding bit position.
4. Keep subtracting until result is 0.
5. Append a 0 as MS bit;
if original was negative, take 2's complement

| n | 2^n |
|-----|-------|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 10 | 1024 |

$$X = 104_{\text{ten}}$$

$$104 - 64 = 40 \quad \text{bit 6}$$

$$40 - 32 = 8 \quad \text{bit 5}$$

$$8 - 8 = 0 \quad \text{bit 3}$$

$$X = 01101000_{\text{two}}$$

Signed and Unsigned Integers

- C, C++, and Java have signed integers, e.g., 7, -255
 - `int x, y, z;`
- C, C++ also have unsigned integers, which are used for addresses
- 32-bit word can represent 2^{32} binary numbers
- Unsigned integers in 32 bit word represent 0 to $2^{32}-1$ (4,294,967,295)



中国科学技术大学
University of Science and Technology of China

计算机系统概论

Introduction to Computing Systems
(CS1002A.02)

Chapter 2-2 Operations and Other Data Types

计算机科学与技术学院
School of Computer Science and Technology



1 Operations on Bits: Arithmetic and Logical

2 Other Representation

3 Summary



1 Operations on Bits: Arithmetic and Logical

2 Other Representation

3 Summary

Operations: Arithmetic and Logical

- **Recall: a data type includes representation and operations. We now have a good representation for signed integers, so let's look at some arithmetic operations:**
 - Addition
 - Subtraction
 - Sign Extension
- **We'll also look at overflow conditions for addition.**
- **Multiplication, division, etc., can be built from these basic operations.**
- **Logical operations are also useful:**
 - AND
 - OR
 - NOT

2's Complement Signed Integers

- MS bit is sign bit – it has weight -2^{n-1} .
- Range of an n-bit number: -2^{n-1} through $2^{n-1} - 1$.
 - The most negative number (-2^{n-1}) has no positive counterpart.

| -2^3 | 2^2 | 2^1 | 2^0 | | -2^3 | 2^2 | 2^1 | 2^0 | |
|--------|-------|-------|-------|---|--------|-------|-------|-------|----|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | -8 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | -7 |
| 0 | 0 | 1 | 0 | 2 | 1 | 0 | 1 | 0 | -6 |
| 0 | 0 | 1 | 1 | 3 | 1 | 0 | 1 | 1 | -5 |
| 0 | 1 | 0 | 0 | 4 | 1 | 1 | 0 | 0 | -4 |
| 0 | 1 | 0 | 1 | 5 | 1 | 1 | 0 | 1 | -3 |
| 0 | 1 | 1 | 0 | 6 | 1 | 1 | 1 | 0 | -2 |
| 0 | 1 | 1 | 1 | 7 | 1 | 1 | 1 | 1 | -1 |

Addition

As we've discussed, 2's complement addition is just binary addition.

- assume all integers have the same number of bits
- ignore carry out
- for now, assume that sum fits in n-bit 2's complement representation

| | |
|-------------------------|--------------------------|
| 01101000 (104) | 11110110 (-10) |
| + <u>11110000</u> (-16) | + <u> </u> (-9) |
| 01011000 (88) | (-19) |

Assuming 8-bit 2's complement numbers.

Subtraction

Negate subtrahend (2nd number) and add.

- assume all integers have the same number of bits
- ignore carry out
- for now, assume that the difference fits in n-bit 2's complement representation

$$\begin{array}{rcl} & 01101000 & (104) \\ - & \underline{00010000} & (16) \\ \hline \end{array} \qquad \begin{array}{rcl} & 11110110 & (-10) \\ + & \underline{\hspace{2cm}} & (-9) \\ \hline \end{array}$$

$$\begin{array}{rcl} & 01101000 & (104) \\ + & \underline{11110000} & (-16) \\ \hline & 01011000 & (88) \end{array} \qquad \begin{array}{rcl} & 11110110 & (-10) \\ + & \underline{\hspace{2cm}} & (9) \\ \hline & & (-1) \end{array}$$

Assuming 8-bit 2's complement numbers.

Sign Extension

To add two numbers, we must represent them with the same number of bits.

If we just pad with zeroes on the left:

4-bit

0100 (4)

1100 (-4)

8-bit

00000100 (still 4)

00001100 (12, not -4)

Instead, replicate the most significant bit -- the sign bit:

4-bit

0100 (4)

1100 (-4)

8-bit

00000100 (still 4)

11111100 (still -4)

Overflow

- Recall the represent range of n-bit 2' complement Signed Integers
- For an n-bit number:

$$-2^{n-1} \sim 2^{n-1} - 1$$

- Can we use n-bit 2' complement to represent a value larger than $2^{n-1}-1$? Or a value smaller than -2^{n-1} ?

Overflow

■ If operands are too big, then sum cannot be represented as an n -bit 2's complement number.

| | |
|---------------------|---------------------|
| 01000 (8) | 11000 (-8) |
| + <u>01001</u> (9) | + <u>10111</u> (-9) |
| 1 0001 (-15) | 0 1111 (+15) |

■ We have overflow if:

- signs of both operands are the same, and
- the sign of sum is different.

Overflow

■ Another test -- easy for hardware:

- The carry into most significant bit is not equal to the carry out

$$\begin{array}{r} 01000 \quad (8) \\ + \underline{01001} \quad (9) \\ \hline \textcolor{red}{1}0001 \quad (-15) \\ \swarrow \searrow \\ 01 \end{array}$$

$$\begin{array}{r} 11000 \quad (-8) \\ + \underline{10111} \quad (-9) \\ \hline \textcolor{red}{0}1111 \quad (+15) \\ \swarrow \searrow \\ 10 \end{array}$$

Logical Operations

■ Operations on logical **TRUE** or **FALSE**

- two states -- takes one bit to represent:
- TRUE=1, FALSE=0

■ View n -bit number as a collection of n logical values

- operation applied to each bit independently
- Bitwise operation

| A | B | A AND B | A | B | A OR B | A | NOT A |
|---|---|---------|---|---|--------|---|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | | |
| 1 | 1 | 1 | 1 | 1 | 1 | | |

Examples of Logical Operations

AND

- useful for **clearing** bits, bitmask
 - AND with zero = 0
 - AND with one = no change

$$\begin{array}{r} 11000101 \\ \text{AND } \underline{00001111} \\ 00000101 \end{array}$$

Inclusive OR

- useful for **setting** bits
 - OR with zero = no change
 - OR with one = 1

$$\begin{array}{r} 11000101 \\ \text{OR } \underline{00001111} \\ 11001111 \end{array}$$

NOT

- unary operation -- one argument
- flips every bit

$$\begin{array}{r} \text{NOT } \underline{11000101} \\ 00111010 \end{array}$$

Examples of Logical Operations

Exclusive-OR (XOR)

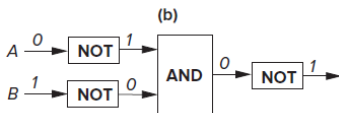
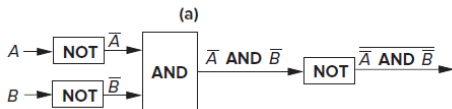
- The output of XOR is 1 if one (but not both) of the two sources is 1. The output of XOR is 0 if both sources are 1 or if neither source is 1.

| A | B | XOR |
|---|---|-----|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

$$\begin{array}{r} 11000101 \\ \text{XOR } 00001111 \\ \hline 11001010 \end{array}$$

DeMorgan's Laws

- There are two well-known relationships between AND functions and OR functions, known as DeMorgan's Laws.



$$\overline{\overline{A} \text{ AND } \overline{B}} = A \text{ OR } B$$

(c)

| A | B | \overline{A} | \overline{B} | $\overline{A} \text{ AND } \overline{B}$ | $\overline{\overline{A} \text{ AND } \overline{B}}$ |
|---|---|----------------|----------------|--|---|
| 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 |



1 Operations on Bits: Arithmetic and Logical

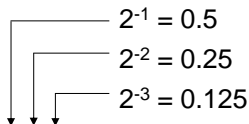
2 Other Representation

3 Summary

Fractions: Fixed-Point

■ How can we represent fractions?

- Use a “binary point” to separate positive from negative powers of two -- just like “decimal point.”
- 2's complement addition and subtraction still work.
 - if binary points are aligned


$$\begin{array}{r} 00101000.101 \quad (40.625) \\ + \underline{11111110.110} \quad (-1.25) \\ \hline 00100111.011 \quad (39.375) \end{array}$$

| n | 2^n |
|-----|-------|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 10 | 1024 |

No new operations -- same as integer arithmetic.

Very Large and Very Small Data

The LC-3 use the 16-bit 2' s complement data type, One bit to identify positive or negative, 15bits to represent the magnitude of the value. We can express values:

- 2^{15} through $2^{15} - 1$
(- 32768 through 32767)

**How can we represent
very large and very small data?**

Very Large and Very Small Data

Large values: 6.023×10^{23} — requires **79 bits**

Small values: 6.626×10^{-34} — requires **>110 bits**

**How can we represent
very large and very small data?**

Very Large and Very Small: Floating-Point

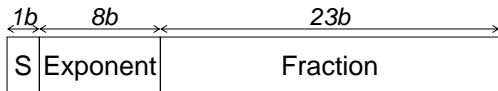
Large values: 6.023×10^{23} — requires **79 bits**

Small values: 6.626×10^{-34} — requires **>110 bits**

Use equivalent of “**scientific notation**” : $F \times 2^E$

Need to represent F (*fraction*), E (*exponent*), and sign.

IEEE 754 Floating-Point Standard (32-bits):



Normalized Form

$$N = (-1)^s \times \text{1.fraction} \times 2^{\text{exponent}-127}, \quad 1 \leq \text{exponent} \leq 254$$

exponent : 0b0000 0000 < exponent < 0b1111 1111

Single-precision IEEE floating point number:

1 01111110 10000000000000000000000000000000



- Sign is 1 – number is negative.
- Exponent field, unsigned integer, excess code, biased representations: 01111110 = 126 (decimal).
- Fraction is .100000000000... = .5 (decimal).

Value = $-1.5 \times 2^{(126-127)} = -1.5 \times 2^{-1} = -0.75$.

Floating Point Example

■ Example 2.13

- $6\frac{5}{8}$ represented in the floating point data type

■ Example 2.14

■ Example 2.15

Very Small: Floating-Point

$$N = (-1)^S \times 1.\text{fraction} \times 2^{\text{exponent}-127}, \quad 1 \leq \text{exponent} \leq 254$$

exponent : 0b0000_0000 < exponent < 0b1111_1111

- The smallest number that can be represented in normalized form is

$$N = 1.000000000000000000000000 \times 2^{-126}$$

Very Small: subnormal numbers

$$N = (-1)^S \times 0.\text{fraction} \times 2^{-126}, \quad \text{exponent} = 0$$

■ The largest subnormal number is

$$N = 0.111111111111111111111111111111 \times 2^{-126}$$

■ The smallest subnormal number is

$$\begin{aligned} N &= 0.000000000000000000000000000001 \times 2^{-126} \\ &= 2^{-23} \times 2^{-126} = 2^{-149} \end{aligned}$$

■ Example

$$\begin{aligned} &0 \quad \underline{00000000} \quad 000010000000000000000000 \\ &2^{-5} \times 2^{-126} = 2^{-131} \end{aligned}$$

Infinites

Normalized From :

$$N = (-1)^S \times 1.\text{fraction} \times 2^{\text{exponent}-127}, \quad 1 \leq \text{exponent} \leq 254$$

Subnormal numbers :

$$N = (-1)^S \times 0.\text{fraction} \times 2^{-126}, \quad \text{exponent} = 0$$

■ So, what if the exponent is equal to 1111_1111?

- If the exponent field contains **1111_1111**, we use the floating point data type to represent **various things**, among them the notion of infinity.
- Infinity is represented by the exponent field containing all 1s and the fraction field containing all 0s.
- We represent positive infinity if the sign bit is 0 and negative infinity if the sign bit is 1

Floating-Point Operations

■ Question

- Will the regular 2's complement arithmetic work for Floating Point numbers?
- (Hint: In decimal, how do we compute $3.07 \times 10^{12} + 9.11 \times 10^8$?)

Text: ASCII Characters

ASCII: Maps 128 characters to 7-bit code.

- both printable and non-printable (ESC, DEL, ...) characters

| | | | | | | | |
|--------|--------|-------|------|------|------|------|--------|
| 00 nul | 10 dle | 20 sp | 30 0 | 40 @ | 50 P | 60 ` | 70 p |
| 01 soh | 11 dc1 | 21 ! | 31 1 | 41 A | 51 Q | 61 a | 71 q |
| 02 stx | 12 dc2 | 22 " | 32 2 | 42 B | 52 R | 62 b | 72 r |
| 03 etx | 13 dc3 | 23 # | 33 3 | 43 C | 53 S | 63 c | 73 s |
| 04 eot | 14 dc4 | 24 \$ | 34 4 | 44 D | 54 T | 64 d | 74 t |
| 05 enq | 15 nak | 25 % | 35 5 | 45 E | 55 U | 65 e | 75 u |
| 06 ack | 16 syn | 26 & | 36 6 | 46 F | 56 V | 66 f | 76 v |
| 07 bel | 17 etb | 27 ' | 37 7 | 47 G | 57 W | 67 g | 77 w |
| 08 bs | 18 can | 28 (| 38 8 | 48 H | 58 X | 68 h | 78 x |
| 09 ht | 19 em | 29) | 39 9 | 49 I | 59 Y | 69 i | 79 y |
| 0a nl | 1a sub | 2a * | 3a : | 4a J | 5a Z | 6a j | 7a z |
| 0b vt | 1b esc | 2b + | 3b ; | 4b K | 5b [| 6b k | 7b { |
| 0c np | 1c fs | 2c , | 3c < | 4c L | 5c \ | 6c l | 7c |
| 0d cr | 1d gs | 2d - | 3d = | 4d M | 5d] | 6d m | 7d } |
| 0e so | 1e rs | 2e . | 3e > | 4e N | 5e ^ | 6e n | 7e ~ |
| 0f si | 1f us | 2f / | 3f ? | 4f O | 5f _ | 6f o | 7f del |

ASCII (American Standard Code for Information Interchange)

| Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|-----|-----|------|-----|-----|---------|-----|-----|------|-----|-----|------|
| 0 | 0 | NUL | 32 | 20 | (space) | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | SOH | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | STX | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | ETX | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | EOT | 36 | 24 | \$ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | ENQ | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | ACK | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | BEL | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | BS | 40 | 28 | (| 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | HT | 41 | 29 |) | 73 | 49 | I | 105 | 69 | i |
| 10 | 0A | LF | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | 0B | VT | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | 0C | FF | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | 0D | CR | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | 0E | SO | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | 0F | SI | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | SLE | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | CS1 | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | DC2 | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | DC3 | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | DC4 | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | NAK | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | SYN | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | ETB | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | CAN | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | EM | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | SIB | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | ESC | 59 | 3B | ; | 91 | 5B | [| 123 | 7B | { |
| 28 | 1C | FS | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | |
| 29 | 1D | GS | 61 | 3D | = | 93 | 5D |] | 125 | 7D | } |
| 30 | 1E | RS | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | US | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | DEL |

ASCII 码

ASCII表

(American Standard Code for Information Interchange 美国标准信息交换代码)

| 高四位 | | ASCII控制字符 | | | | | | | | | | | | ASCII打印字符 | | | | | | | | | | | | | |
|------|-----|-----------|------|----|-----|----|------|------|------|-----|-----|----|--------|-----------|------|------|------|------|------|----|-----|-----|-----|-----|-----|----|-----------------------|
| | | 0000 | | | | | | 0001 | | | | | | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | | | | | | | | |
| | | 0 | | | | | | 1 | | | | | | 2 | 3 | 4 | 5 | 6 | 7 | | | | | | | | |
| 低四位 | 十进制 | 字符 | Ctrl | 代码 | 转义 | 字符 | 十进制 | 字符 | Ctrl | 代码 | 转义 | 字符 | 十进制 | 字符 | 十进制 | 字符 | 十进制 | 字符 | 十进制 | 字符 | 十进制 | 字符 | 十进制 | 字符 | 十进制 | 字符 | Ctrl |
| 0000 | 0 | 0 | | ^@ | NUL | \0 | 空字符 | 16 | ▶ | ^P | DLE | | 数据链路转义 | 32 | | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p | | |
| 0001 | 1 | 1 | ☺ | ^A | SOH | | 标题开始 | 17 | ◀ | ^Q | DC1 | | 设备控制 1 | 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q | | |
| 0010 | 2 | 2 | ☹ | ^B | STX | | 正文开始 | 18 | ↕ | ^R | DC2 | | 设备控制 2 | 34 | " | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r | | |
| 0011 | 3 | 3 | ♥ | ^C | ETX | | 正文结束 | 19 | !! | ^S | DC3 | | 设备控制 3 | 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s | | |
| 0100 | 4 | 4 | ♦ | ^D | EOT | | 传输结束 | 20 | ¶ | ^T | DC4 | | 设备控制 4 | 36 | \$ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t | | |
| 0101 | 5 | 5 | ♣ | ^E | ENQ | | 查询 | 21 | § | ^U | NAK | | 否定应答 | 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u | | |
| 0110 | 6 | 6 | ♠ | ^F | ACK | | 肯定应答 | 22 | — | ^V | SYN | | 同步空闲 | 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v | | |
| 0111 | 7 | 7 | ● | ^G | BEL | \a | 响铃 | 23 | ↕ | ^W | ETB | | 传输块结束 | 39 | ' | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w | | |
| 1000 | 8 | 8 | ◻ | ^H | BS | \b | 退格 | 24 | ↑ | ^X | CAN | | 取消 | 40 | (| 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x | | |
| 1001 | 9 | 9 | ◯ | ^I | HT | \t | 横向制表 | 25 | ↓ | ^Y | EM | | 介质结束 | 41 |) | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y | | |
| 1010 | A | 10 | ◐ | ^J | LF | \n | 换行 | 26 | → | ^Z | SUB | | 替代 | 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z | | |
| 1011 | B | 11 | ♂ | ^K | VT | \v | 纵向制表 | 27 | ← | ^[| ESC | \e | 溢出 | 43 | + | 59 | ; | 75 | K | 91 | [| 107 | k | 123 | { | | |
| 1100 | C | 12 | ♀ | ^L | FF | \f | 换页 | 28 | └ | ^[_ | FS | | 文件分隔符 | 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | | | |
| 1101 | D | 13 | ♪ | ^M | CR | \r | 回车 | 29 | ↔ | ^_] | GS | | 组分分隔符 | 45 | - | 61 | = | 77 | M | 93 |] | 109 | m | 125 | } | | |
| 1110 | E | 14 | 🎵 | ^N | SO | | 移出 | 30 | ▲ | ^^ | RS | | 记录分隔符 | 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ | | |
| 1111 | F | 15 | 🎵 | ^O | SI | | 移入 | 31 | ▼ | ^_ | US | | 单元分隔符 | 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | ␣ | | ^Backspace 代码: DEL |

注：表中的ASCII字符可以用“Alt + 小键盘上的数字键”方法输入。

<http://blog.csdn.net/2011/08/20/7851417> 云教程中心

2011/08/20 09:00:00

Interesting Properties of ASCII Code

- What is the relationship between a decimal digit ('0', '1', ...) and its ASCII code?
 - "30h"
- What is the difference between an upper-case letter ('A', 'B', ...) and its lower-case equivalent ('a', 'b', ...)?
 - "20h"
- Given two ASCII characters, how do we tell which comes first in alphabetical order?
- Are 128 characters enough?
(<http://www.unicode.org/>)

No new operations -- integer arithmetic and logic.

Other Data Types

■ Text strings

- sequence of characters, terminated with NULL (0)
- typically, no hardware support

■ Image

- array of pixels
 - monochrome: one bit (1/0 = black/white)
 - color: red, green, blue (RGB) components (e.g., 8 bits each)
 - other properties: transparency
- hardware support:
 - typically none, in general-purpose processors
 - MMX -- multiple 8-bit operations on 32-bit word

■ Sound

- sequence of fixed-point numbers

Within the Computer: Everything is a Number.

LC-3 Data Types

- Some data types are supported directly by the instruction set architecture.
- For LC-3, there is only one supported data type:
 - 16-bit 2's complement signed integer
 - Operations: ADD, AND, NOT
- Other data types are supported by interpreting 16-bit values as logical, text, fixed-point, etc., in the software that we write.



1 Operations on Bits: Arithmetic and Logical

2 Other Representation

3 Summary

Summary

- Everything in a computer is a number, in fact only 0 and 1.
- Integers are interpreted by adhering to fixed length
- Negative numbers are represented with 2' s complement
- Overflows can be detected utilizing the carry bit
- We will get into some more representations later when we talk about floating point
- Signed Magnitude & Biased representations are needed in floating point for specific uses
- Not going to talk about '1' s complement' , its a joke that nobody uses



中国科学技术大学
University of Science and Technology of China

计算机系统概论

Introduction to Computing Systems
(CS1002A.02)

Chapter 3-1 Transistors & Logic Gates

计算机科学与技术学院
School of Computer Science and Technology



1 Review

2 The Transistor

3 Logic Gates

4 Summary



1 Review

2 The Transistor

3 Logic Gates

4 Summary

Review

■ Great Idea #0: Great Idea from Ancient Chinese Philosophy(Bits and Bytes)

■ How do we represent data in a computer?

- Bits: 0/1
- Data type: *representation* and *operations* within the computer
 - Integer Data Types
 - Unsigned Integers
 - 2' Complement Integers
 - Fixed-Point Data Types
 - Floating-Point Data Types
 - Text – characters, strings, ...
 - Images – pixels, colors, shapes, ...
 - Sound
 - Instructions
 -
- Arithmetic and Logical Operations
- Binary-Decimal Conversion

Today

■ Microprocessors contain millions of transistors

- Intel Core 2 Duo: **291 million**
- AMD Barcelona: **463 million**
- IBM Power6: **790 million**

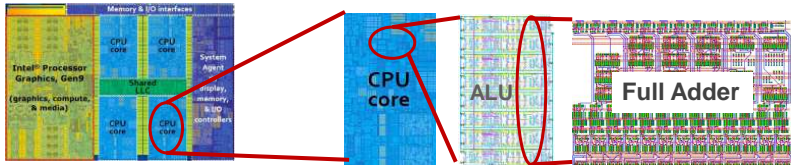
■ Transistor: Building Block of Computers

- Logically, each transistor acts as a **switch**

■ Combined to **implement** logic functions

- AND, OR, NOT

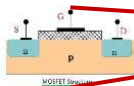
Approach: Bottom Up



Integrated Circuit Design
100 Modules/ IC
0.25M~20G Devices

Register Transfer Level (RTL) Design
1K~10K Cells/Module
(100K Devices)

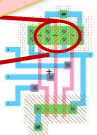
Now, You are Here.



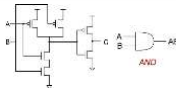
Transistor Physical Layout



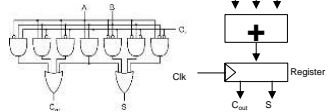
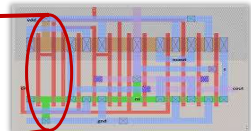
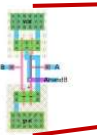
**Scheme for
Representing Information**



Gate Level Design



Circuit Level Design
(Transistor Level Design)
(2~8 Devices/Gate)



Register Transfer Level (RTL) Design
2~16 Gates/Cell
(16~64 Devices)

Great Idea #3: Abstraction Helps Us Manage Complexity

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

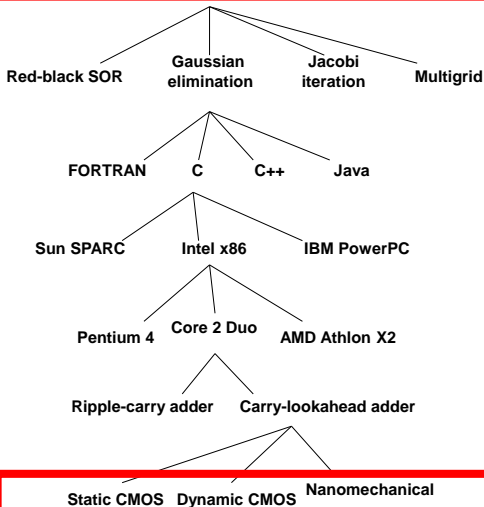
Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

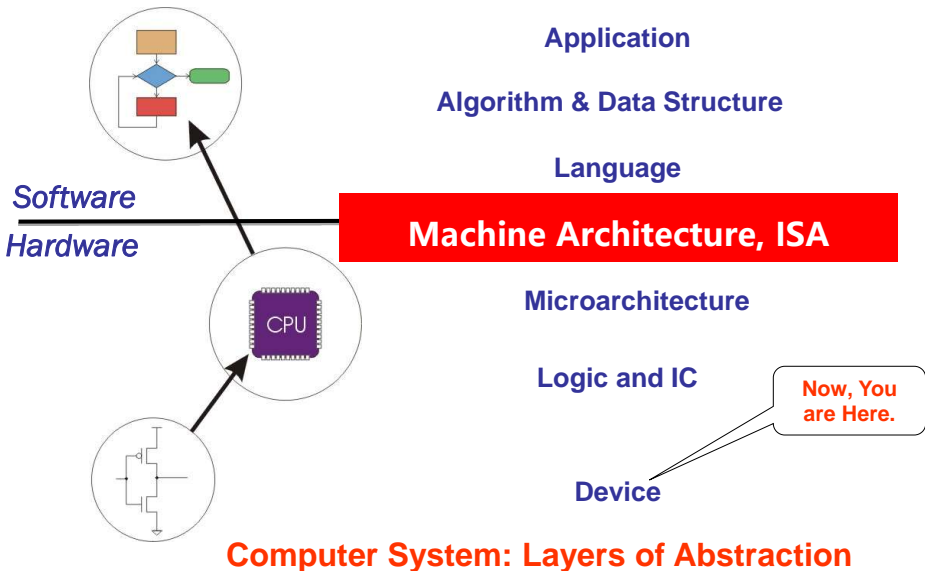
Electronic Devices

Physics

Solve a system of equations



Great Idea #4: Software and Hardware Co-design





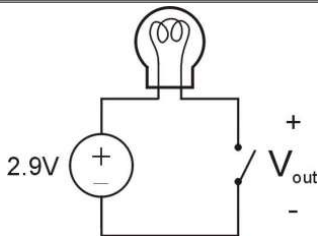
1 Review

2 **The Transistor**

3 Logic Gates

4 Summary

Vacuum tubes used as switches



Switch **open**:

- No current through circuit
- Light is **off**
- V_{out} is **+2.9V**



Switch **closed**:

- Short circuit across switch
- Current flows
- Light is **on**
- V_{out} is **0V**

Switch-based circuits can easily represent two states:
on/off, open/closed, voltage/no voltage.

Vacuum tubes used as switches made electronic computing



Electronic computing was made possible for the first time by the short [mean time to failure](#) of tubes. The main problem was that **valves**—which, like light bulbs, have a short filament—could never be used if they were unreliable, and in a large scale installation, "for a short time". [Tommy Flowers](#), who designed the Colossus, said that, so long as **valves** were reliable for very long periods, they could be used on a reduced current". In 1934, the first installation using over 3,000 tubes was at Bletchley Park. If a tube failed, it was possible to replace it with others going, thereby reducing the cost (the cost of the exchanges). Flowers was also a pioneer (compared to electromechanical computers). It was confirmed that tube unreliability was a problem; the 1946 [ENIAC](#), with 17,468 tubes (which took 15 minutes to locate) on the other hand, the failure of the tubes was a factor, and the war Colossus was instrumental in the development continued with tube-computers [ENIAC](#) and [Whirlwind](#), the first commercially available electronic computer.

The first commercially available electronic computer included the Jaincomp series of computers. The Instrument Company of Bethesda, Maryland, the Jain-B employed just 300 such tubes and its performance to rival many of the then

The invention of the transistor



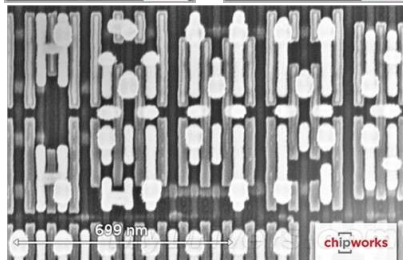
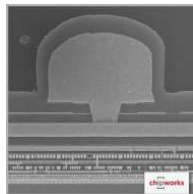
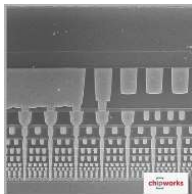
Bell Labs lays the groundwork

- 1945: Bell sets up lab in the hopes of developing “solid state” components to replace existing electromechanical systems. William Shockley, John Bardeen, Walter Brattain: all solid-state physicists. Focus on Si and Ge.
- 1947: The Invention of the First Transistor- the point-contact transistor
- 1951: Shockley develops junction transistor which can be manufactured in quantity.
- 1954: The first transistor radio! Also, TI makes first silicon transistor (price \$2.50)
- 1956: Bardeen, Shockley, Brattain receive Nobel Prize.

A transistor under a microscope

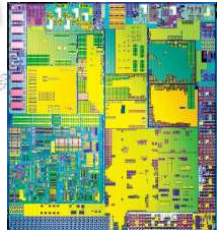
■ <http://www.zhihu.com/question/26998618>

- How is it possible to have tens of millions of transistors in a chip?



显微镜下的intel 14纳米晶体管

Microprocessors contain millions of transistors



Microprocessors contain millions of transistors

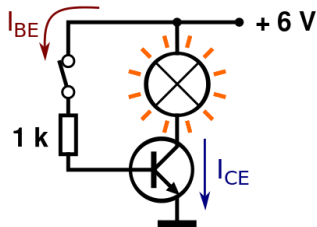
| CPU | Manufacturing Process | Cores | GPU | Transistor Count (Schematic) | Die Size |
|-----------------------|-----------------------|-------|-----|------------------------------|--------------------------|
| Haswell GT3 4C | 22nm | 4 | GT3 | ? | 264mm ² (est) |
| Haswell GT2 4C | 22nm | 4 | GT2 | 1.4B | 177mm ² |
| AMD Vishera 8C | 32nm | 8 | N/A | 1.2B | 315mm ² |
| Intel Sandy Bridge 4C | 32nm | 4 | GT2 | 995M | 216mm ² |
| Intel Lynnfield 4C | 45nm | 4 | N/A | 774M | 296mm ² |

Our World
in Data

The graph illustrates the exponential growth of transistor counts in integrated circuits over time. The y-axis represents the transistor count on a logarithmic scale, ranging from 1,000 to 50,000,000,000. The x-axis represents the year, from 1970 to 2018. The data points show a clear upward trend, with transistor counts increasing by orders of magnitude every few years. Key milestones include the Intel 4004 (1971, 2,300 transistors), Intel 8086 (1982, 290,000 transistors), Intel Pentium (1992, 3.1 million transistors), and the ARM Cortex-A9 (2008, 1.5 billion transistors). The plot also shows the emergence of mobile processors like the ARM Cortex-A9 and the Intel Atom, as well as the continued growth of desktop processors like the Intel Core i7 and the AMD K10.

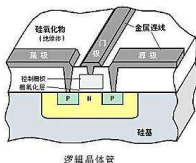
Licensed under [CC-BY-SA](#) by the author Max Roser.

Transistor as a switch



Switch **open**:

- No current through circuit
- Light is **off**
- V_{out} is **+2.9V**



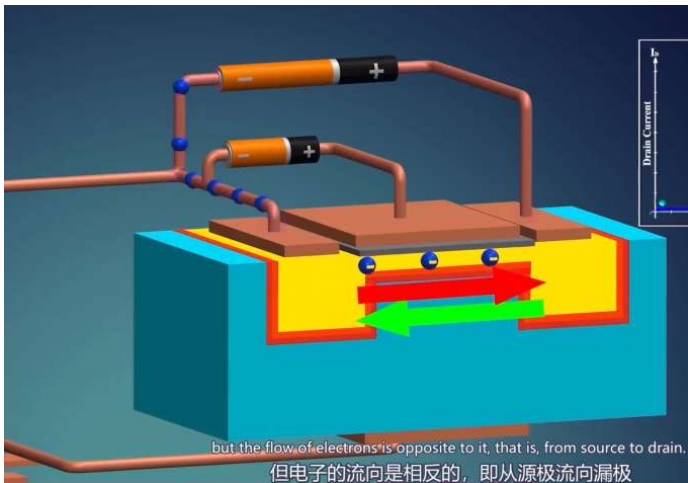
Switch **closed**:

- Short circuit across switch
- Current flows
- Light is **on**
- V_{out} is **0V**

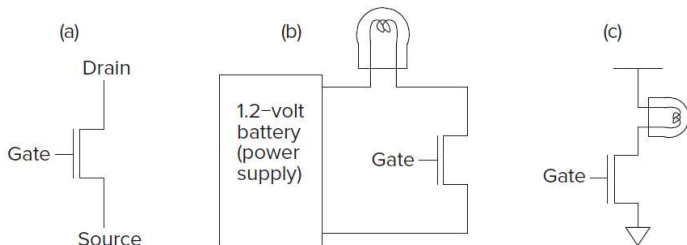
Switch-based circuits can easily represent two states: on/off, open/closed, voltage/no voltage.

How does MOSFET work?

- 【探索】MOSFET是如何工作的,科学,科普,好看视频(baidu.com)



N-type MOS Transistor



- When the gate is supplied with 1.29 volts, the transistor acts like a piece of wire, completing the circuit and causing the bulb to glow.
- When the gate is supplied with 0 volts, the transistor acts like an open circuit, breaking the circuit, and causing the bulb to not glow.

N-type MOS Transistor

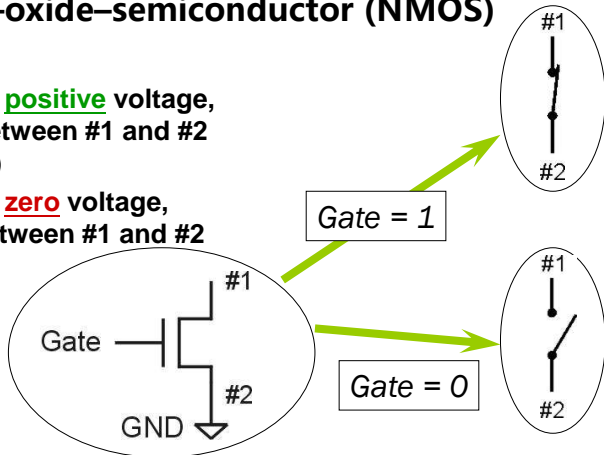
■ MOS = Metal Oxide Semiconductor

- two types: N-type and P-type

■ N-type metal–oxide–semiconductor (NMOS)

■ N-type

- when Gate has positive voltage, short circuit between #1 and #2 (switch closed)
- when Gate has zero voltage, open circuit between #1 and #2 (switch open)



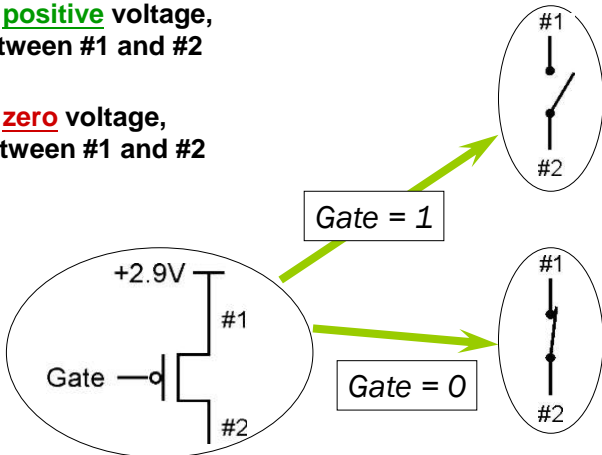
Terminal #2 must be connected to GND (0V).

P-type MOS Transistor

■ P-type metal–oxide–semiconductor (PMOS)

■ P-type is *complementary* to N-type

- when Gate has positive voltage, open circuit between #1 and #2 (switch open)
- when Gate has zero voltage, short circuit between #1 and #2 (switch closed)



Terminal #1 must be connected to +2.9V.



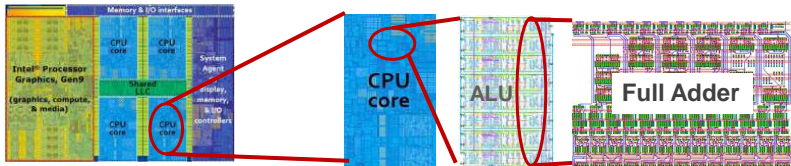
1 Review

2 The Transistor

3 Logic Gates

4 Summary

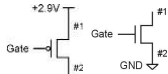
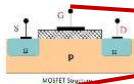
Approach: Bottom Up



Integrated Circuit Design
100 Modules/ IC
0.25M~20G Devices

Register Transfer Level (RTL) Design
1K~10K Cells/Module
(100K Devices)

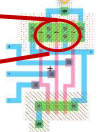
Now, You are Here.



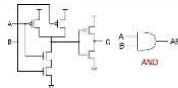
Transistor Physical Layout



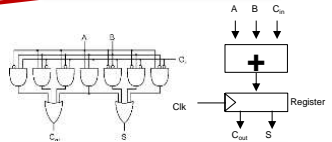
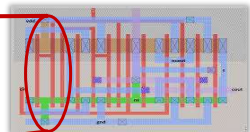
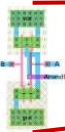
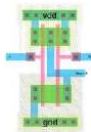
Scheme for Representing Information



Gate Level Design



Circuit Level Design
(Transistor Level Design)
(2~8 Devices/Gate)



Register Transfer Level (RTL) Design
2~16 Gates/Cell
(16~64 Devices)

Great Idea #3: Abstraction Helps Us Manage Complexity

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

Electronic Devices

Physics

Solve a system of equations

Red-black SOR Gaussian elimination Jacobi iteration Multigrid

FORTTRAN C C++ Java

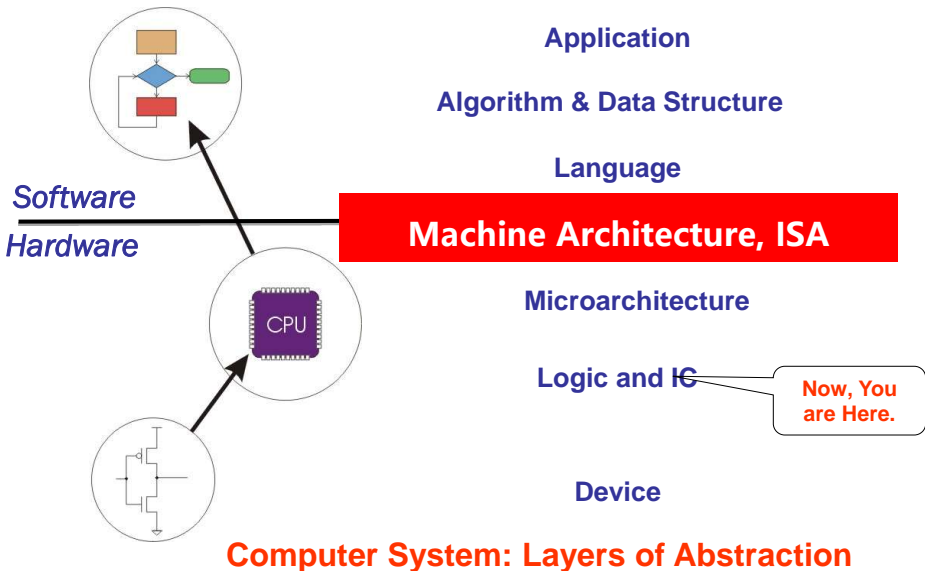
Sun SPARC Intel x86 IBM PowerPC

Pentium 4 Core 2 Duo AMD Athlon X2

Ripple-carry adder Carry-lookahead adder

Static CMOS Dynamic CMOS Nanomechanical

Great Idea #4: Software and Hardware Co-design

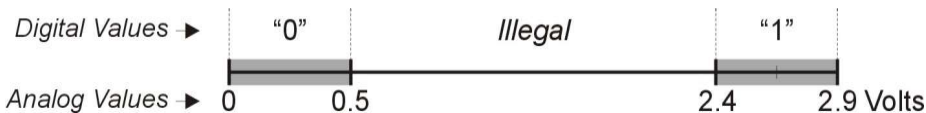


Logic Gates

- Use switch behavior of MOS transistors to implement logical functions: AND, OR, NOT.

- Digital symbols:

- recall that we assign a range of analog voltages to each digital (logic) symbol



- assignment of voltage ranges depends on electrical properties of transistors being used

CMOS Circuit

■ CMOS: Complementary MOS

■ Uses both N-type and P-type MOS transistors

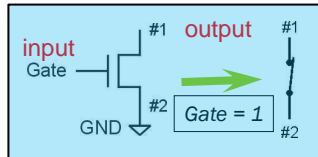
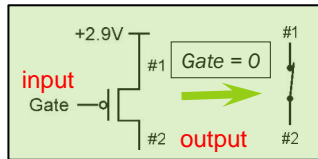
● P-type

- Attached to + voltage
- Pulls output voltage **UP** when input is **zero**

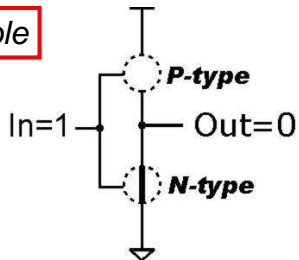
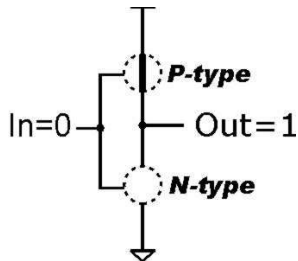
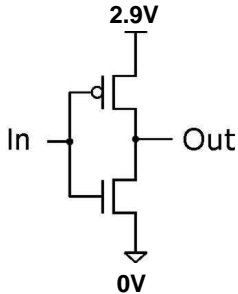
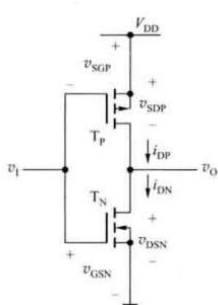
● N-type

- Attached to GND
- Pulls output voltage **DOWN** when input is **one**

- For all inputs, make sure that output is either connected to GND or to +, but **not both!**



Inverter (NOT Gate)



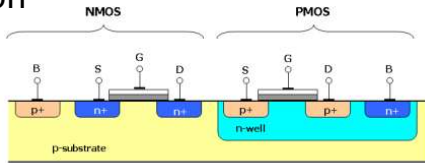
Truth table

| In | Out |
|-------|-------|
| 0 V | 2.9 V |
| 2.9 V | 0 V |

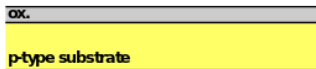
| In | Out |
|----|-----|
| 0 | 1 |
| 1 | 0 |

The process of fabrication of a CMOS inverter

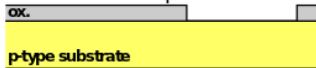
- Simplified process of fabrication of a CMOS inverter on p-type substrate in semiconductor microfabrication.



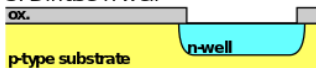
1. Grow field oxide



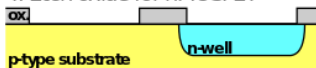
2. Etch oxide for pMOSFET



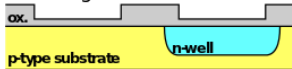
3. Diffuse n-well



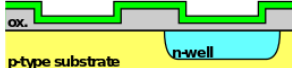
4. Etch oxide for nMOSFET



5. Grow gate oxide



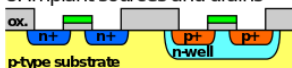
6. Deposit polysilicon



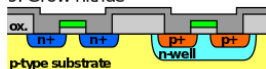
7. Etch polysilicon and oxide



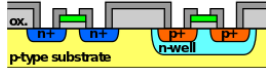
8. Implant sources and drains



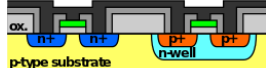
9. Grow nitride



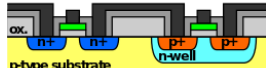
10. Etch nitride



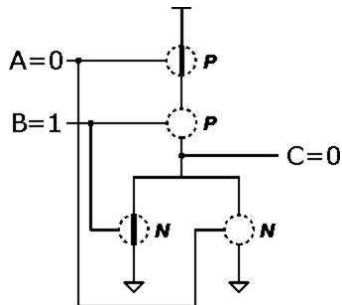
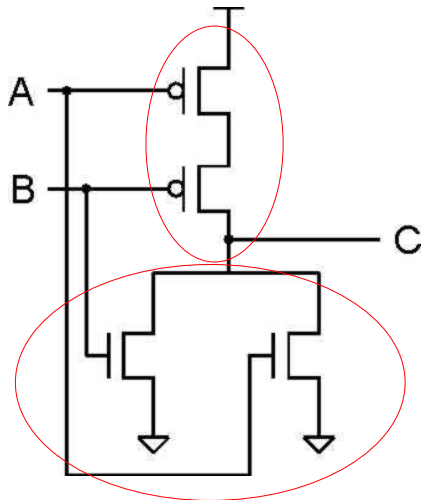
11. Deposit metal



12. Etch metal



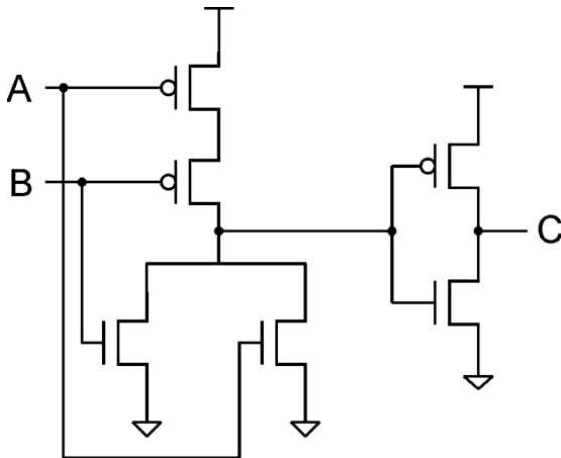
NOR Gate



| INPUT | | OUTPUT |
|-------|---|--------|
| A | B | C |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

Note: Serial structure on top, parallel on bottom.
2025/2/24

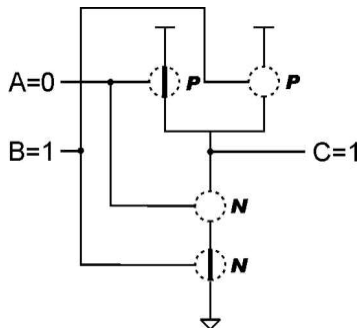
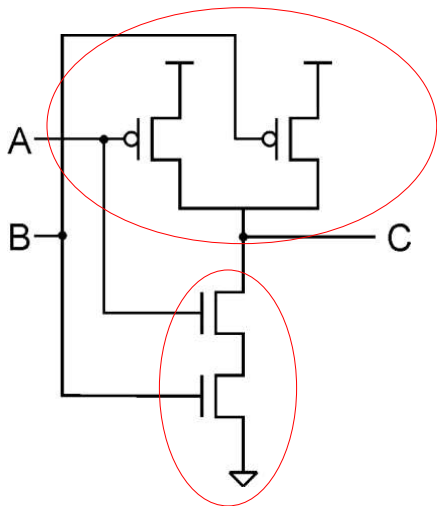
OR Gate



| INPUT | | OUTPUT |
|-------|---|--------|
| A | B | C |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Add inverter to NOR.

NAND Gate (AND-NOT)

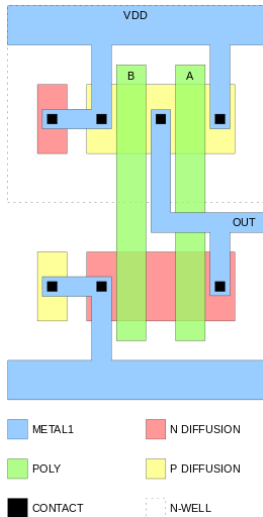
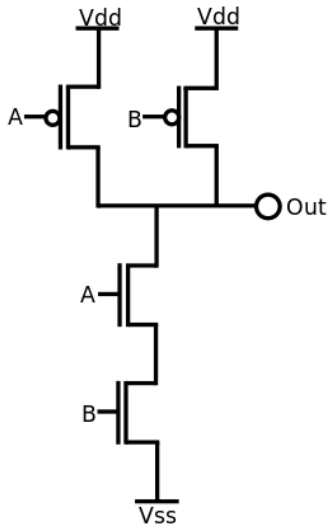


| INPUT | | OUTPUT |
|-------|---|--------|
| A | B | C |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Note: Parallel structure on top, serial on bottom.
2025/2/24

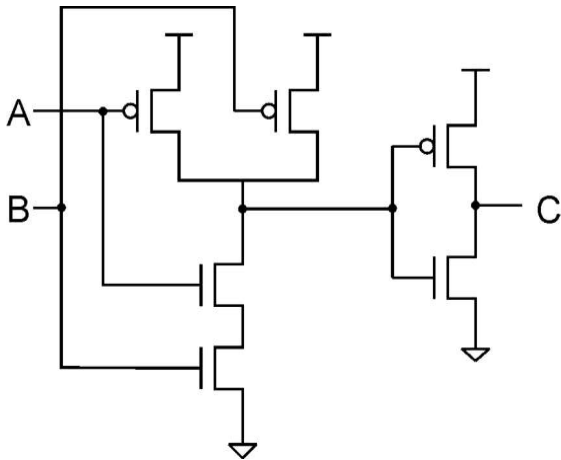
Example: NAND gate in physical layout

NAND gate in CMOS logic



The [physical layout](#) of a NAND circuit. The larger regions of N-type diffusion and P-type diffusion are part of the transistors. The two smaller regions on the left are taps to prevent [latchup](#).

AND Gate



| INPUT | | OUTPUT |
|-------|---|--------|
| A | B | C |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Add inverter to NAND.

Practice 1

Implement a 3-input NOR gate with CMOS.

Basic Gates

■ From Now on.....Gates

- Covered transistors mostly so that you know they exist
- Note: "Logic Gate" not related to "Gate" of MOSFET transistors

■ Will study implementation in terms of gates

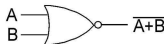
- Circuits that implement Boolean functions



NOT



OR



NOR



AND



NAND

■ More complicated gates from transistors possible

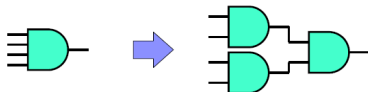
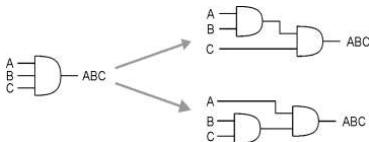
- XOR, Multiple-input AND-OR-Invert (AOI) gates

More than 2 Inputs?

■ AND/OR can take any number of inputs.

- AND = 1 if all inputs are 1.
- OR = 1 if any input is 1.
- Similar for NAND/NOR.

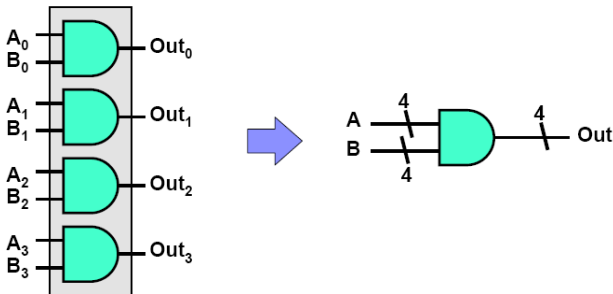
■ Can implement with multiple two-input gates, or with single CMOS circuit.



Visual Shorthand for Multi-bit Gates

■ Use a cross-hatch mark to group wires

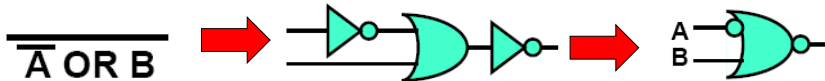
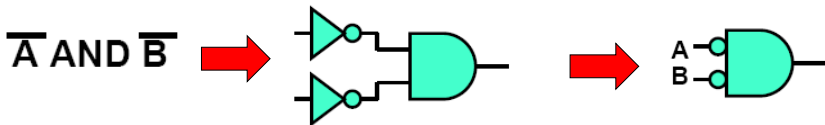
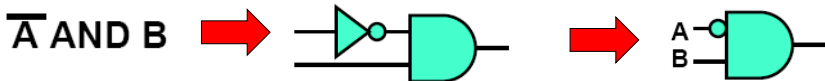
- Example: calculate the AND of a pair of 4-bit numbers
- A3 is "high-order" or "most-significant" bit
- If "A" is 1000, then $A_3 = 1$, $A_2 = 0$, $A_1 = 0$, $A_0 = 0$



Shorthand for Inverting Signals

■ Invert a signal by adding either

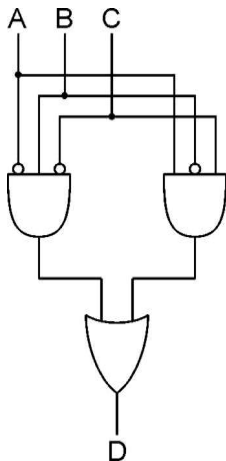
- A circle or an "inversion bubble" before/after a gate
- A "bar" over the letter



Logical Completeness

■ AND, OR, NOT can implement **ANY** truth table

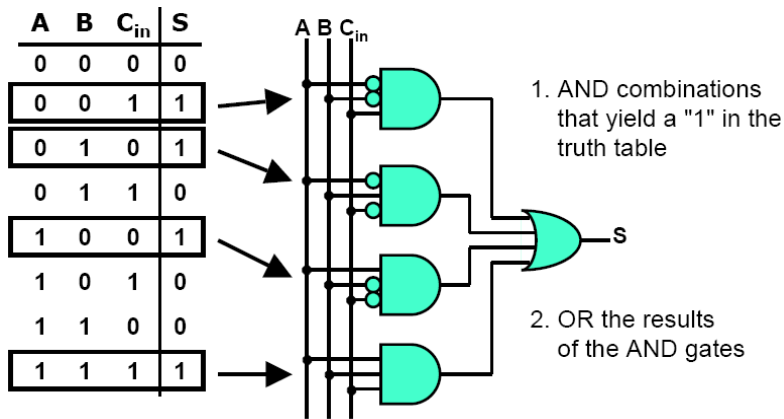
| A | B | C | D |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |



1. **AND** combinations that yield a "1" in the truth table.
2. **OR** the results of the AND gates.

Logical Completeness

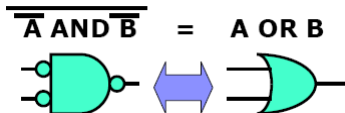
■ AND, OR, NOT can implement ANY truth table



DeMorgan's Law

Converting AND to OR (with some help from NOT)

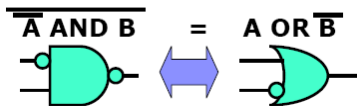
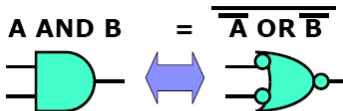
Consider the following gate:



| A | B | \overline{A} | \overline{B} | $\overline{A} \cdot \overline{B}$ | $\overline{\overline{A} \cdot \overline{B}}$ |
|---|---|----------------|----------------|-----------------------------------|--|
| 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 |

Same as **A OR B**!

*To convert AND to OR
(or vice versa),
invert inputs and output.*



Why might this be useful?

Practice 2

| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



1 Review

2 The Transistor

3 Logic Gates

4 Summary

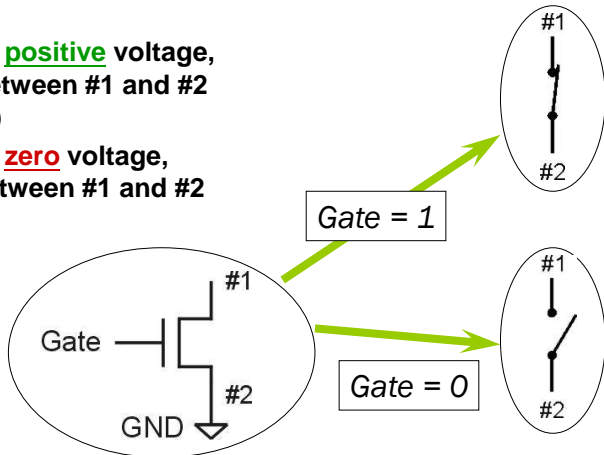
N-type MOS Transistor

■ MOS = Metal Oxide Semiconductor

- two types: N-type and P-type

■ N-type

- when Gate has **positive** voltage, short circuit between #1 and #2 (switch **closed**)
- when Gate has **zero** voltage, open circuit between #1 and #2 (switch **open**)

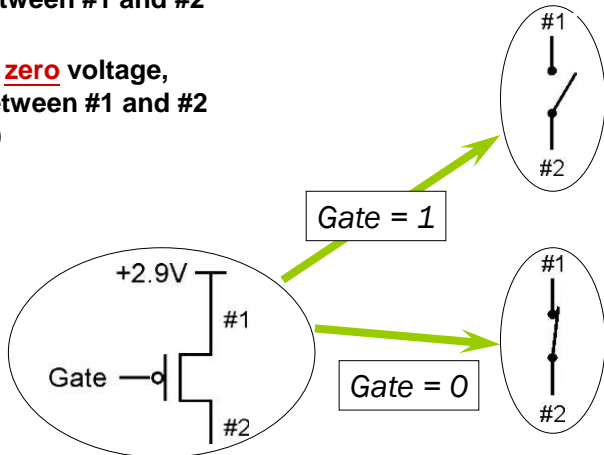


Terminal #2 must be connected to GND (0V).

P-type MOS Transistor

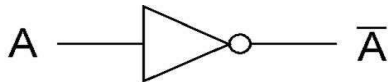
■ P-type is *complementary* to N-type

- when Gate has positive voltage, open circuit between #1 and #2 (switch open)
- when Gate has zero voltage, short circuit between #1 and #2 (switch closed)

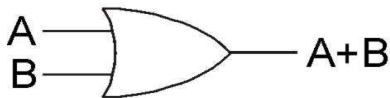


Terminal #1 must be connected to +2.9V.

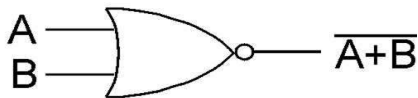
Gates



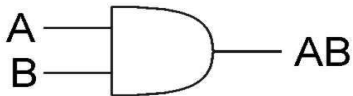
NOT



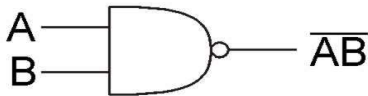
OR



NOR



AND



NAND



中国科学技术大学
University of Science and Technology of China

计算机系统概论

Introduction to Computing Systems
(CS1002A.02)

Chapter 3-2 Combinational Logic Circuits & Basic Storage Elements

计算机科学与技术学院
School of Computer Science and Technology



- 1 Review
- 2 Combinational Logic Circuits
- 3 Basic Storage Elements
- 4 Summary



1 Review

2 Combinational Logic Circuits

3 Basic Storage Elements

4 Summary

Review

■ Transistor: Building Block of Computers

- Logically, each transistor acts as a switch

■ Combined to implement logic functions

- AND, OR, NOT

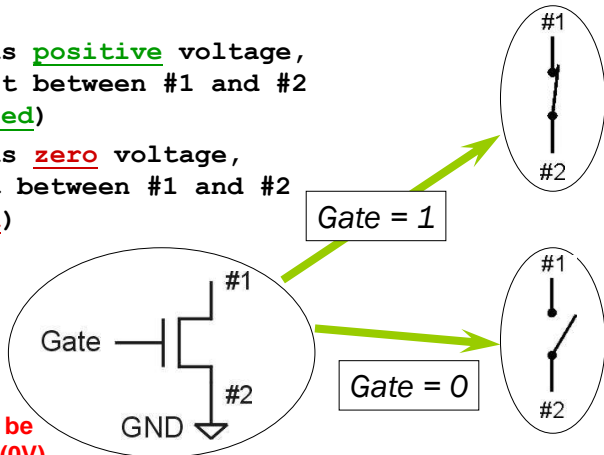
N-type MOS Transistor

■ MOS = Metal Oxide Semiconductor

- two types: N-type and P-type

■ N-type

- when Gate has positive voltage, short circuit between #1 and #2 (switch closed)
- when Gate has zero voltage, open circuit between #1 and #2 (switch open)



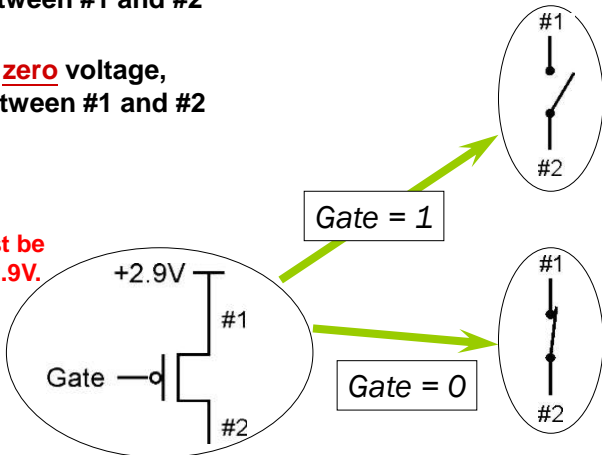
Terminal #2 must be connected to GND (0V).

P-type MOS Transistor

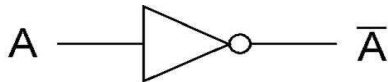
■ P-type is *complementary* to N-type

- when Gate has positive voltage, open circuit between #1 and #2 (switch open)
- when Gate has zero voltage, short circuit between #1 and #2 (switch closed)

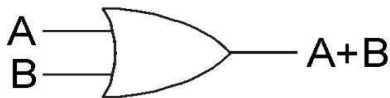
Terminal #1 must be connected to +2.9V.



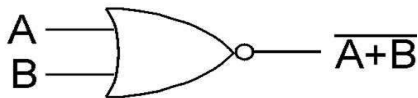
Gates



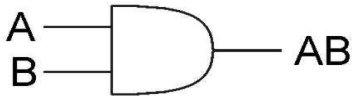
NOT



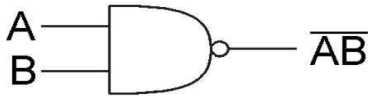
OR



NOR



AND

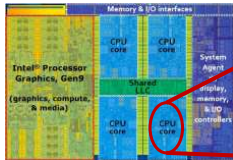


NAND

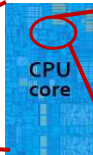
Today: Building Functions from Logic Gates

- We've already seen how to implement truth tables using AND, OR, and NOT -- an example of *combinational logic*.
- **Combinational Logic Circuit**
 - output depends only on the current inputs
 - stateless
- **Sequential Logic Circuit**
 - output depends on the sequence of inputs (past and present)
 - stores information (state) from past inputs
- We'll first look at some useful combinational circuits, then show how to use sequential circuits to store information.

Approach: Bottom Up



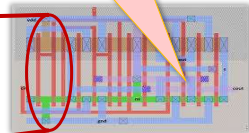
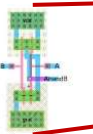
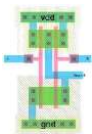
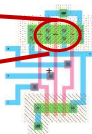
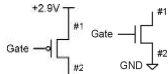
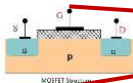
Integrated Circuit Design
100 Modules/ IC
0.25M-20G Devices



Register Transfer Level (RTL) Design

Now, You are Here.

And Here.

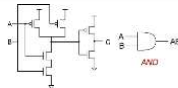


Gate Level Design

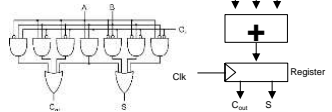
Transistor Physical Layout



Scheme for Representing Information



Circuit Level Design
(Transistor Level Design)
(2-8 Devices/Gate)



Register Transfer Level (RTL) Design
2-16 Gates/Cell
(16-64 Devices)

Great Idea #3: Abstraction Helps Us Manage Complexity

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

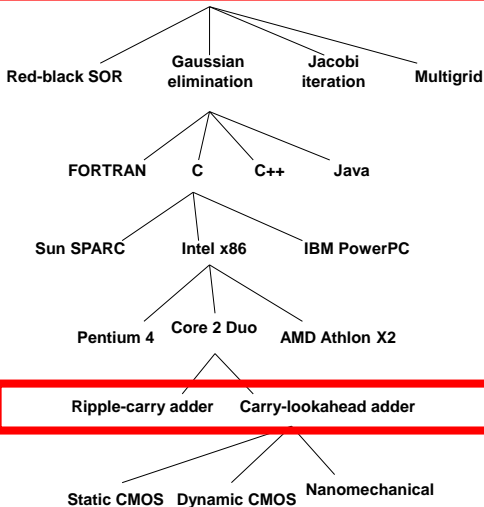
Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

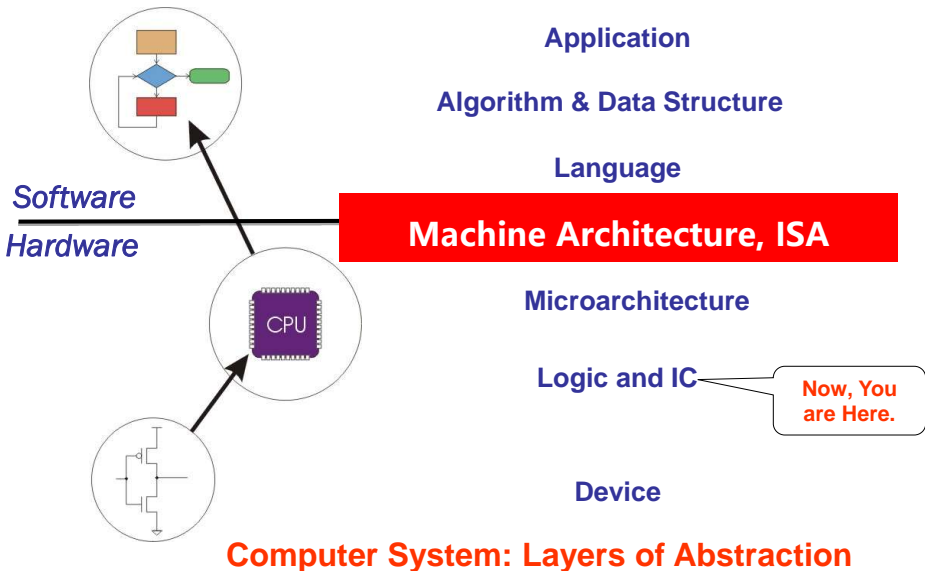
Electronic Devices

Physics

Solve a system of equations



Great Idea #4: Software and Hardware Co-design





1 Review

2 **Combinational Logic Circuits**

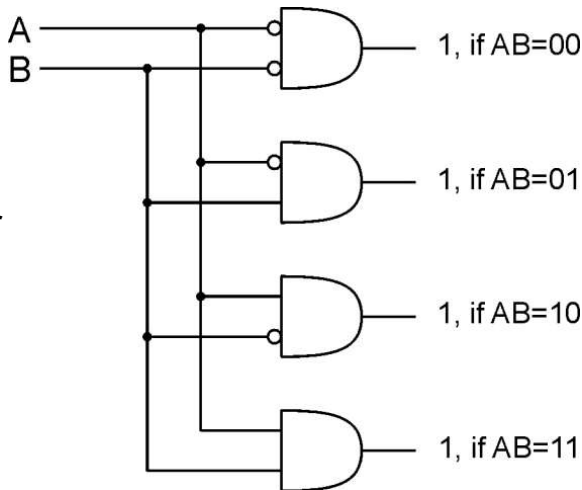
3 Basic Storage Elements

4 Summary

■ n inputs, 2^n outputs

- exactly one output is 1 for each possible input pattern

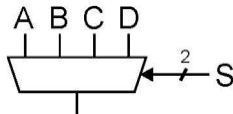
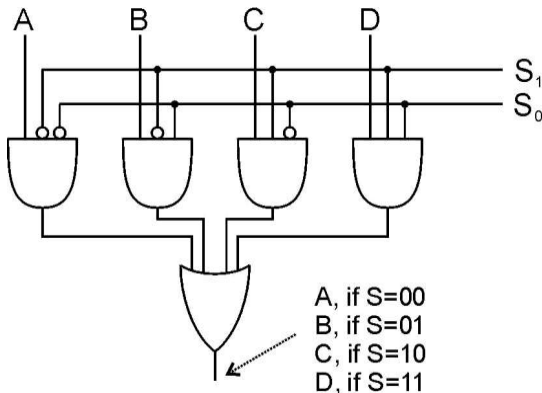
*2-bit
decoder*



Multiplexer (MUX)

■ n -bit selector and 2^n inputs, one output

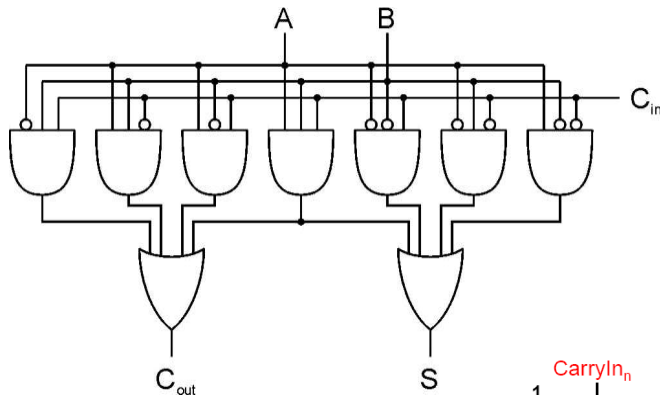
- output equals one of the inputs, depending on selector



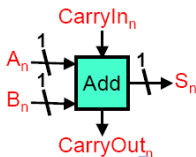
4-to-1 MUX

Full Adder

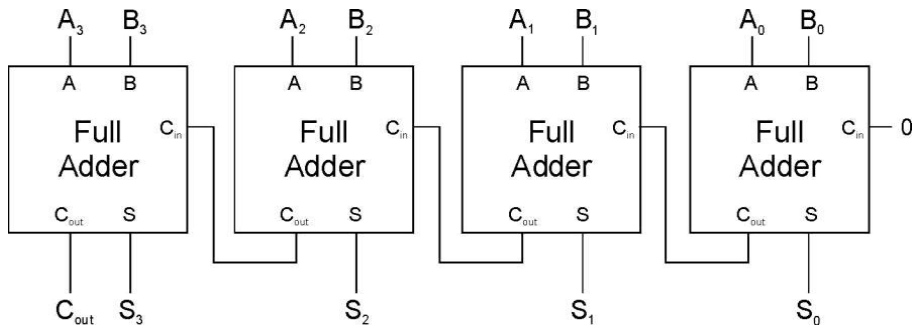
Add two bits and carry-in,
produce one-bit sum and carry-out.



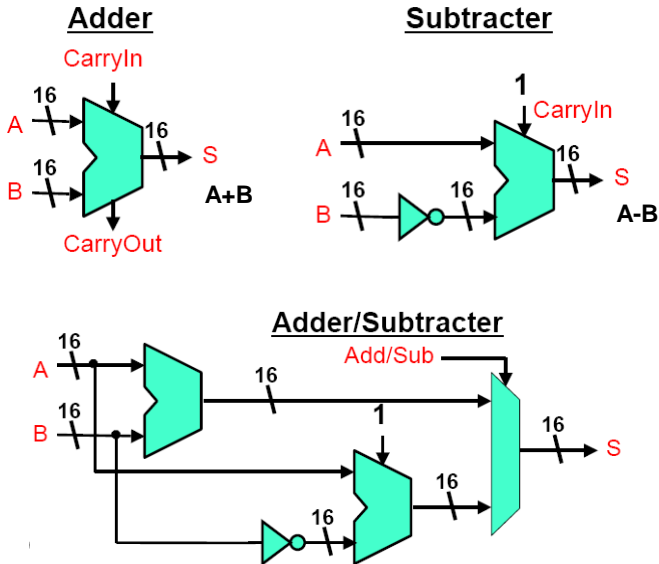
| A | B | C_{in} | S | C_{out} |
|---|---|----------|---|-----------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |



Four-bit Adder

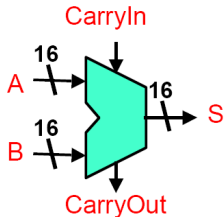


Adder/Subtractor - Approach #1

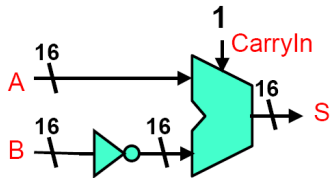


Adder/Subtractor - Approach #2

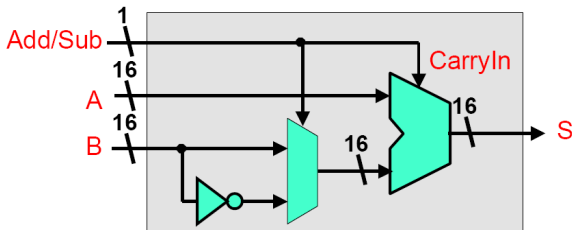
Adder



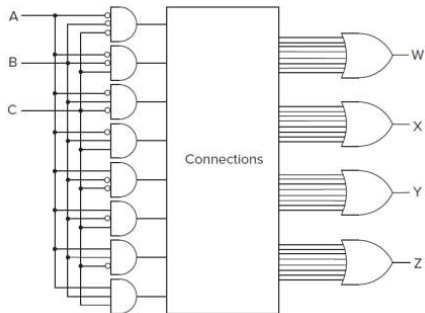
Subtractor



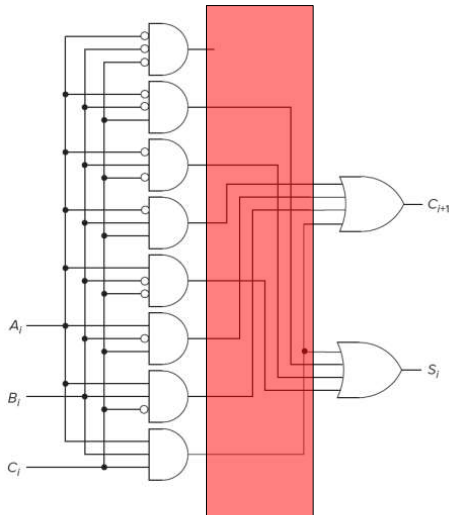
Adder/Subtractor



The Programmable Logic Array (PLA)



| A_i | B_i | C_i | C_{i+1} | S_i |
|-------|-------|-------|-----------|-------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |



The truth table for a one-bit adder.

Incrementer

■ Let' s create an incrementer

- Input: A (as a 16-bit 2's complement integer)
- Output: A+1 (also as a 16-bit 2's complement integer)

■ Approach #1 (impractical):

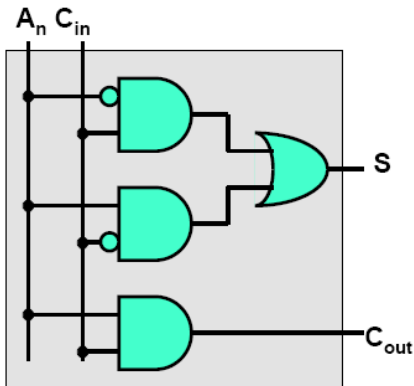
- Use PLA-like techniques to implement circuit
- Problem: 2^{16} or 65536 rows, 16 output columns
- In theory, possible; in practice, intractable

■ Approach#2 (pragmatic):

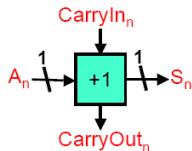
- Create an 1-bit incrementer circuit
- Replicate it 16 times

One-bit Incrementer

■ Implement a single-column of incrementer



| A | C_{in} | S | C_{out} |
|---|----------|---|-----------|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |





1 Review

2 Combinational Logic Circuits

3 Basic Storage Elements

4 Summary

Combinational vs. Sequential

■ Combinational Circuit

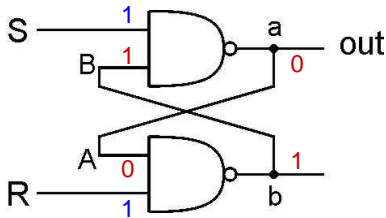
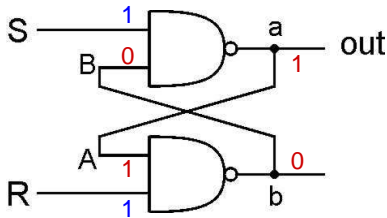
- always gives the same output for a given set of inputs
 - ex: adder always generates sum and carry, regardless of previous inputs

■ Sequential Circuit

- stores information
- output depends on stored information (state) plus input
 - so a given input might produce different outputs, depending on the stored information
- *example*: ticket counter
 - advances when you push the button
 - output depends on previous state
- useful for building "memory" elements and "state machines"

R-S Latch: Simple Storage Element

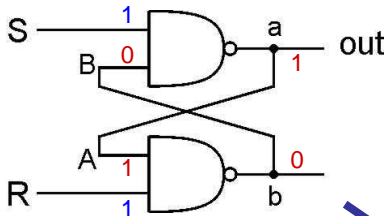
- R is used to “reset” or “clear” the element – set it to zero.
- S is used to “set” the element – set it to one.



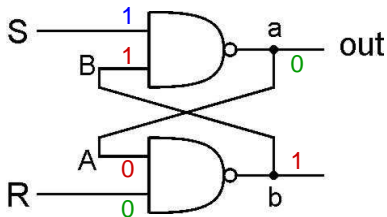
- If both R and S are one, out could be either zero or one.
 - “quiescent” state -- holds its previous value
 - note: if a is 1, b is 0, and vice versa

Clearing the R-S latch

■ Suppose we start with output = 1, then change R to zero.



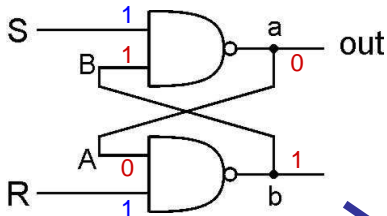
Output changes to zero.



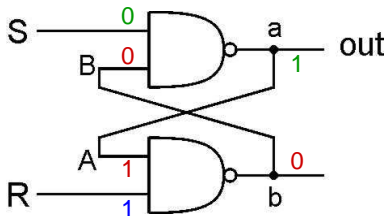
Then set $R=1$ to “store” value in quiescent state.

Setting the R-S Latch

■ Suppose we start with output = 0, then change S to zero.



Output changes to one.



Then set S=1 to “store” value in quiescent state.

R-S Latch Summary

$R = S = 1$

- hold current value in latch

$S = 1$ and $R = 0$,

- set value to 0

$R = 1$ and $S = 0$

- set value to 1

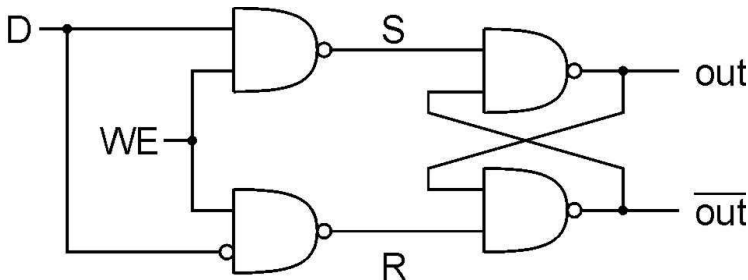
$R = S = 0$

- both outputs equal one
- final state determined by electrical properties of gates
- *Don't do it!*

Gated D-Latch

■ Two inputs: D (data) and WE (write enable)

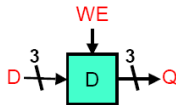
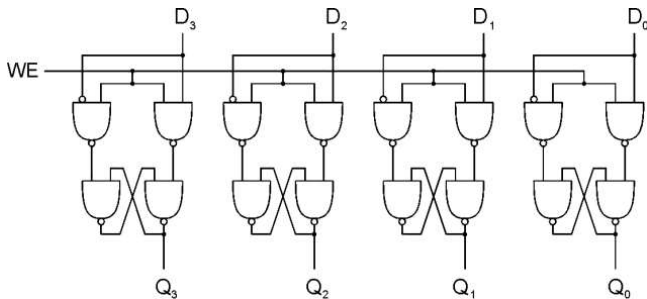
- when **WE = 1**, latch is set to **value of D**
 - $S = \text{NOT}(D)$, $R = D$
- when **WE = 0**, latch holds **previous value**
 - $S = R = 1$



Register

■ A register stores a multi-bit value.

- We use a collection of D-latches, all controlled by a common WE.
- When $WE=1$, n-bit value D is written to register.



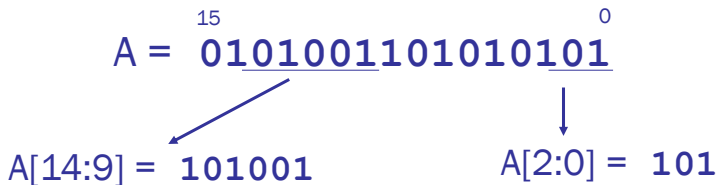
Representing Multi-bit Values

■ Number bits from right (0) to left (n-1)

- just a convention -- could be left to right, but must be consistent

■ Use brackets to denote range:

$D[l:r]$ denotes bit l to bit r , from *left to right*



■ May also see $A\langle 14:9 \rangle$, especially in hardware block diagrams.

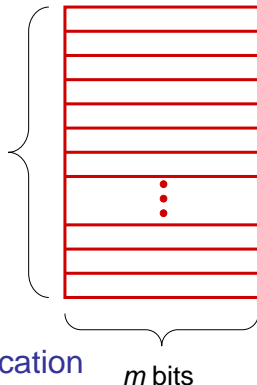
Memory

■ Now that we know how to store bits, we can build a memory – a logical $k \times m$ array of stored bits.

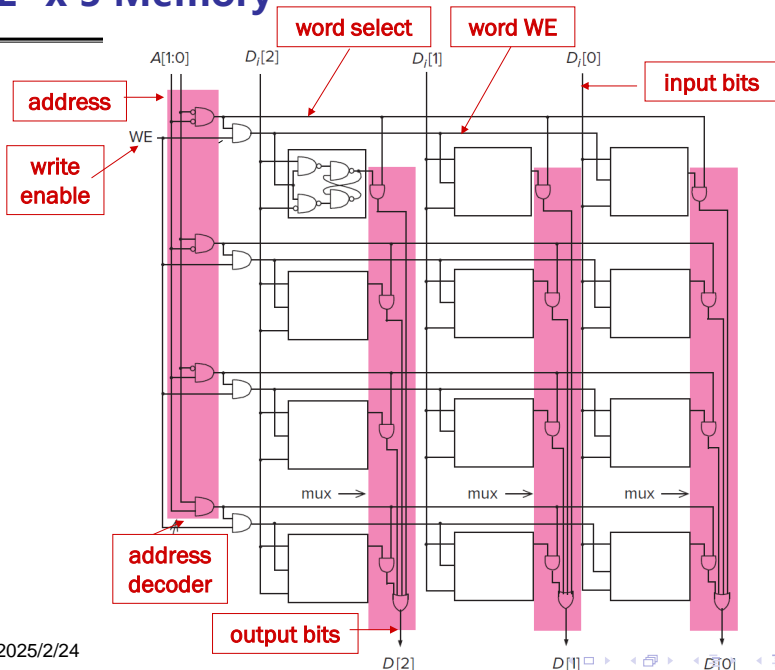
Address Space:
number of locations
(usually a power of 2)

$k = 2^n$
locations

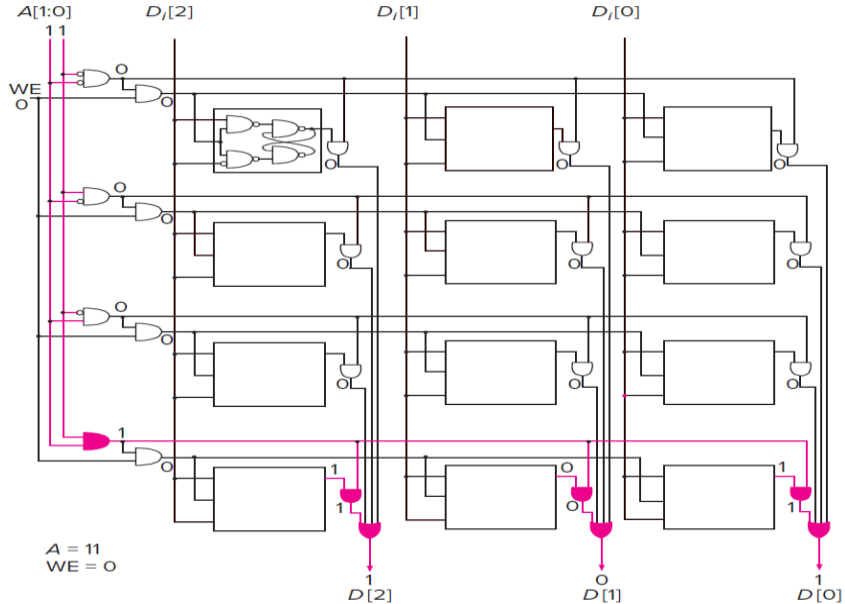
Addressability:
number of bits per location
(e.g., byte-addressable)



$2^2 \times 3$ Memory



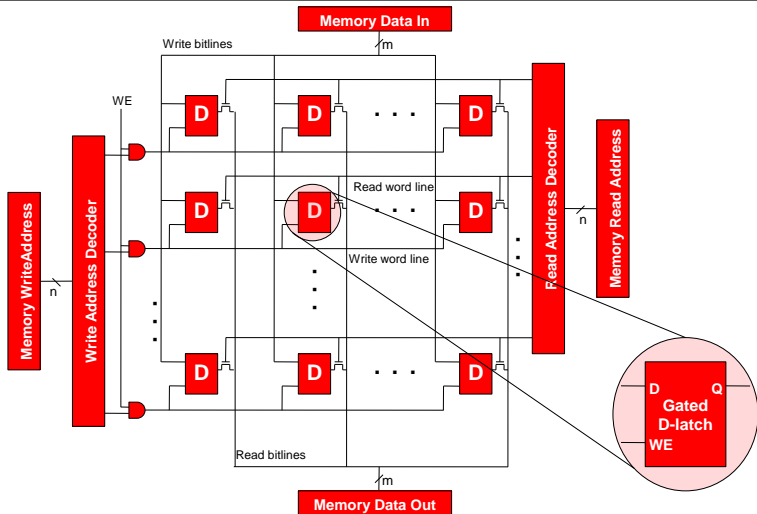
Reading location 3 in our 2^2 -by-3-bit memory.



More Memory Details

- This is not the way actual memory is implemented.
 - fewer transistors, much more dense, relies on electrical properties
- But the logical structure is very similar.
 - address decoder
 - word select line
 - word write enable
- Two basic kinds of **RAM** (Random Access Memory)
 - **Static RAM** (SRAM)
 - fast, not very dense (bitcell is a latch)
 - **Dynamic RAM** (DRAM)
 - slower but denser, bit storage must be periodically refreshed
 - each bitcell is a capacitor (like a leaky bucket) that decays

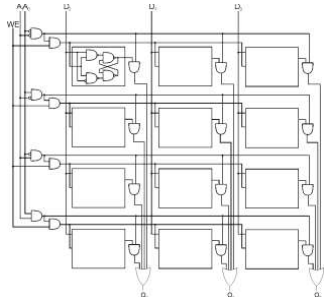
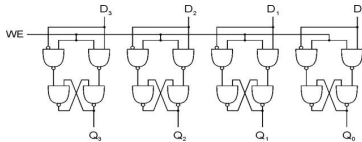
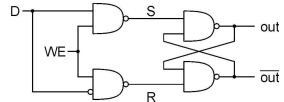
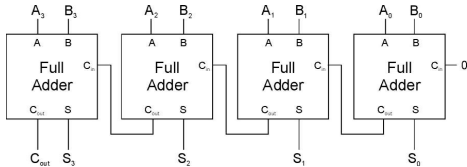
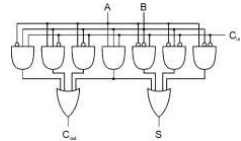
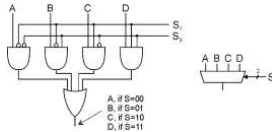
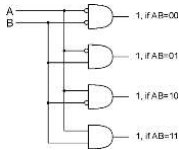
SRAM Memory





- 1 Review
- 2 Combinational Logic Circuits
- 3 Basic Storage Elements
- 4 Summary

Basic Logical Structure





中国科学技术大学
University of Science and Technology of China

计算机系统概论

Introduction to Computing Systems
(CS1002A.02)

Chapter 3-3 Sequential Logic Circuits

计算机科学与技术学院
School of Computer Science and Technology



- 1 Review**
- 2 Sequential Logic Circuits**
- 3 From Logic to Data Path**
- 4 Summary**



1 Review

2 Sequential Logic Circuits

3 From Logic to Data Path

4 Summary

Review

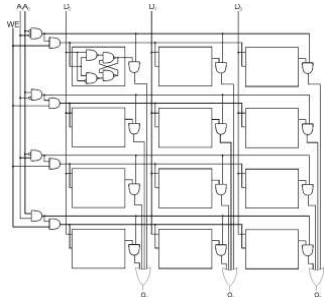
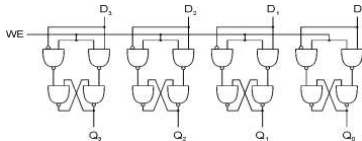
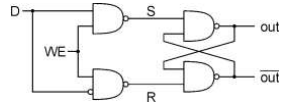
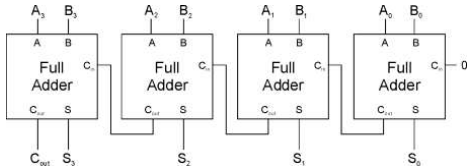
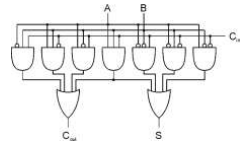
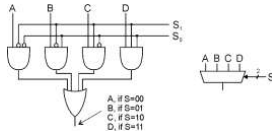
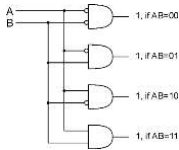
■ We' ve touched on basic digital logic

- Transistors
- Gates
- Storage (latches, flip-flops, register, memory)

■ Built some simple logical circuits

- adder, subtracter, adder/subtractor, Incrementer
- Counter (consisting of register and incrementer)

Basic Logical Structure



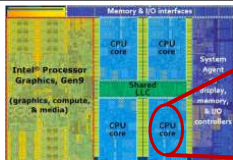
Today

■ A computer as a (simple?) state machine

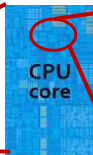
- State machines
- Hard-coded traffic sign state machine
- Programmable traffic sign state machine

Approach: Bottom Up

Now, You are Here.

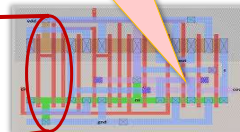
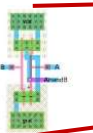
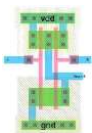
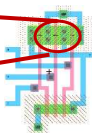
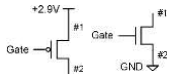
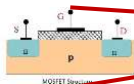


Integrated Circuit Design
100 Modules/ IC
0.25M~20G Devices



Register Transfer Level (RTL) Design

And Here.

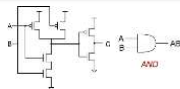


Gate Level Design

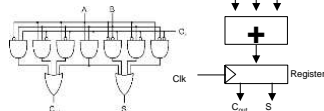
Transistor Physical Layout



Scheme for
Representing Information



Circuit Level Design
(Transistor Level Design)
(2~8 Devices/Gate)



Register Transfer Level (RTL) Design
2~16 Gates/Cell
(16~64 Devices)

Great Idea #3: Abstraction Helps Us Manage Complexity

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

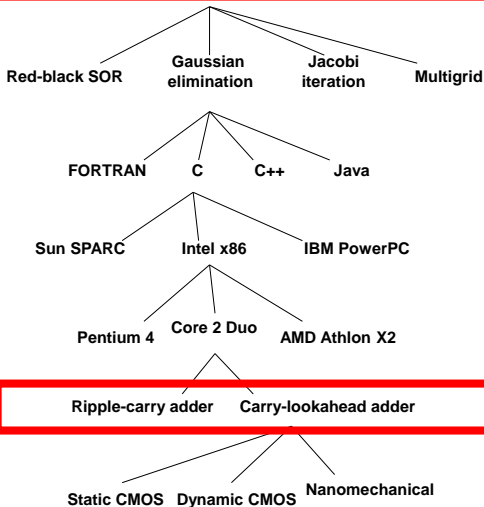
Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

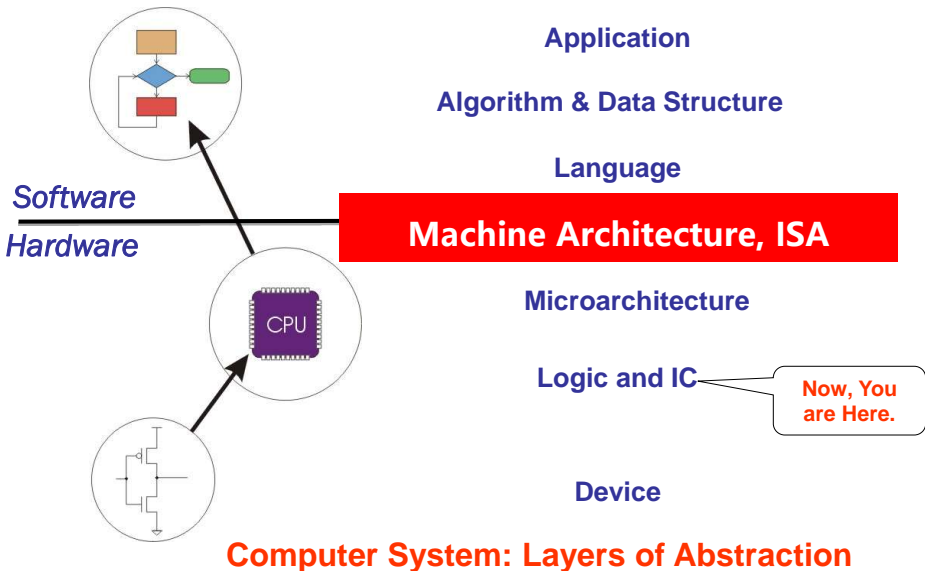
Electronic Devices

Physics

Solve a system of equations



Great Idea #4: Software and Hardware Co-design



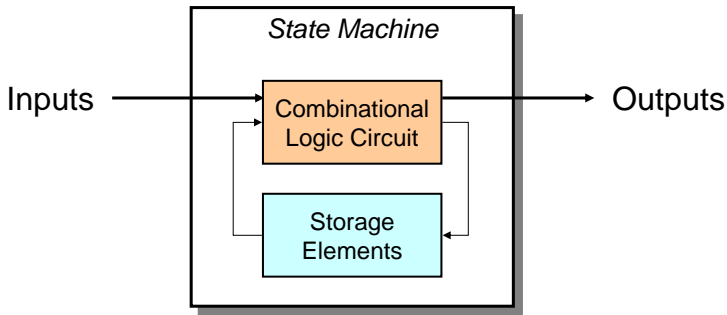


- 1 Review
- 2 Sequential Logic Circuits**
- 3 From Logic to Data Path
- 4 Summary

State Machine

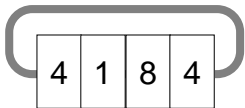
Another type of sequential circuit

- Combines combinational logic with storage
- “Remembers” state, and changes output (and state) based on **inputs** and **current state**



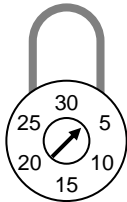
Combinational vs. Sequential

Two types of “combination” locks



Combinational

Success depends only on the **values**, not the order in which they are set.



Sequential

Success depends on the **sequence** of values (e.g., R-13, L-22, R-3).

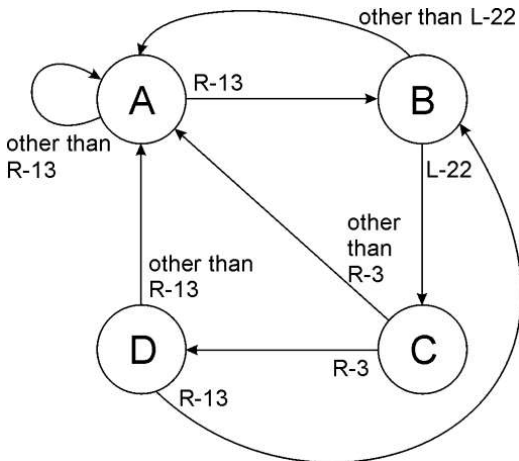
State of Sequential Lock

Our lock example has four different states, labelled A-D:

- A:** The lock is **not open**,
and no relevant operations have been performed.
- B:** The lock is **not open**,
and the user has completed the **R-13** operation.
- C:** The lock is **not open**,
and the user has completed **R-13**, followed by **L-22**.
- D:** The lock is **open**.

State Diagram

Shows **states** and **actions** that cause a **transition** between states.



Finite State Machine

A description of a system with the following components:

1. A finite number of **states**
2. A finite number of external **inputs**
3. A finite number of external **outputs**
4. An explicit specification of all **state transitions**
5. An explicit specification of what causes each external **output value**.

Often described by a state diagram.

- Inputs may cause state transitions.
- Outputs are associated with each state (or with each transition).

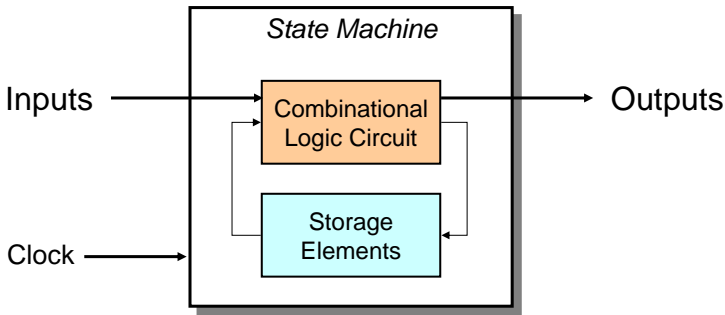
Implementing a Finite State Machine

Combinational logic

- Determine outputs and next state.

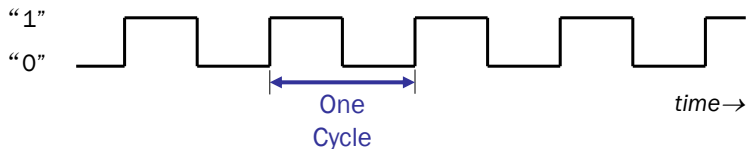
Storage elements

- Maintain state representation.



The Clock

Frequently, a **clock circuit** triggers transition from one state to the next.

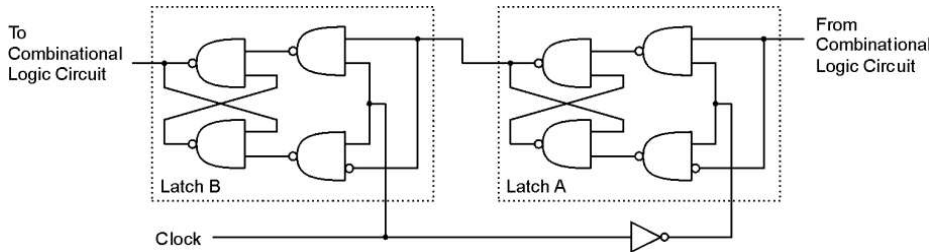


At the beginning of each clock cycle, state machine makes a transition, based on the current state and the external inputs.

- Not always required. In lock example, the input itself triggers a transition.

Storage: Master-Slave Flipflop

A pair of gated D-latches,
to isolate *next* state from *current* state.



During 1st phase (clock=1), previously-computed state becomes *current* state and is sent to the logic circuit.

During 2nd phase (clock=0), *next* state, computed by logic circuit, is stored in Latch A.

Storage

Each master-slave flipflop stores one state bit.

The number of storage elements (flipflops) needed is determined by the number of states (and the representation of each state).

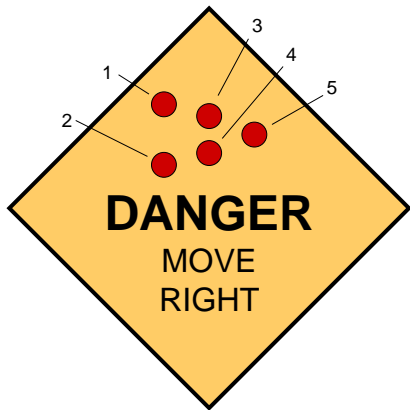
Examples:

- **Sequential lock**
 - **Four states - two bits**
- **Basketball scoreboard**
 - **7 bits for each score, 5 bits for minutes, 6 bits for seconds,**
 - 1 bit for possession arrow, 1 bit for half, ...**

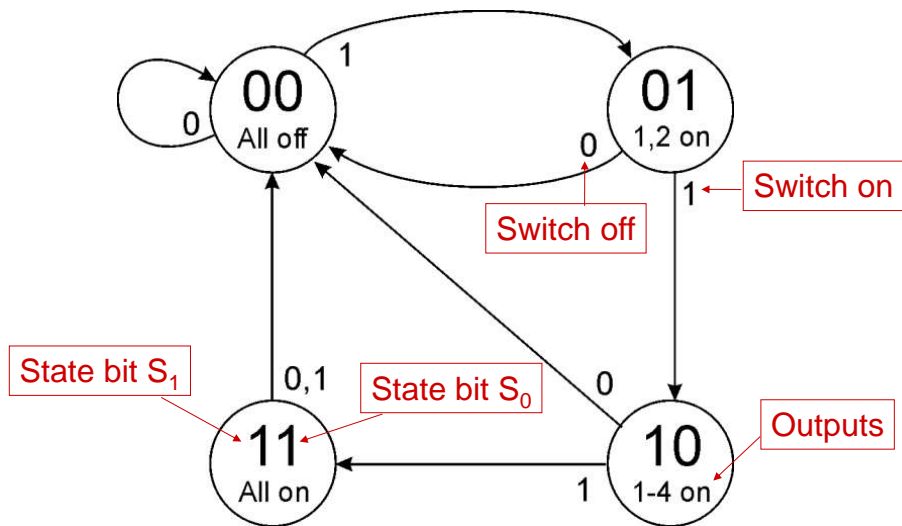
Complete Example

A blinking traffic sign

- No lights on
- 1 & 2 on
- 1, 2, 3, & 4 on
- 1, 2, 3, 4, & 5 on
- (repeat as long as switch is turned on)

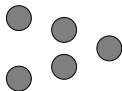
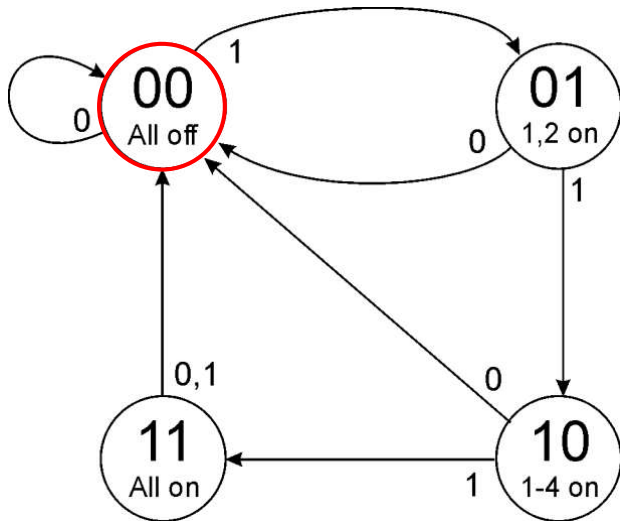


Traffic Sign State Diagram

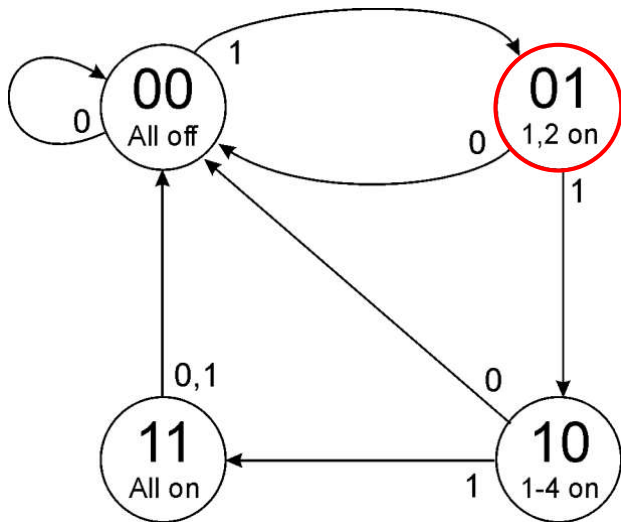


Transition on each clock cycle.

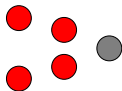
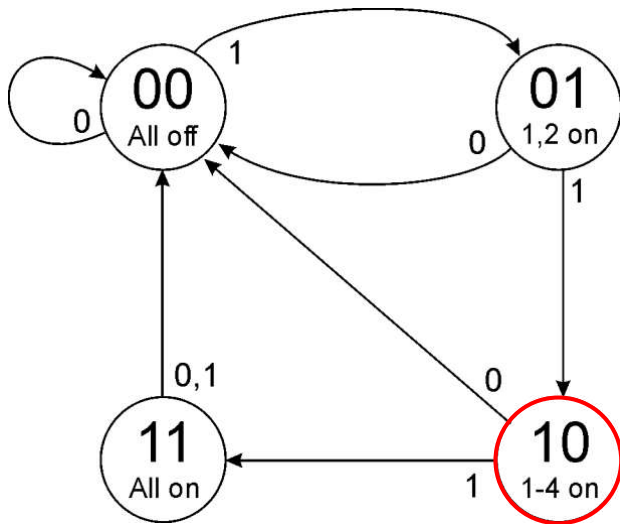
Traffic Sign State Diagram: State 00



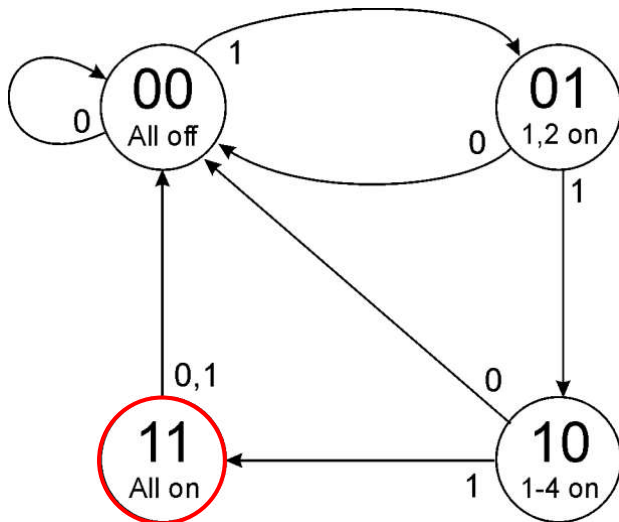
Traffic Sign State Diagram: State 01



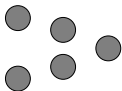
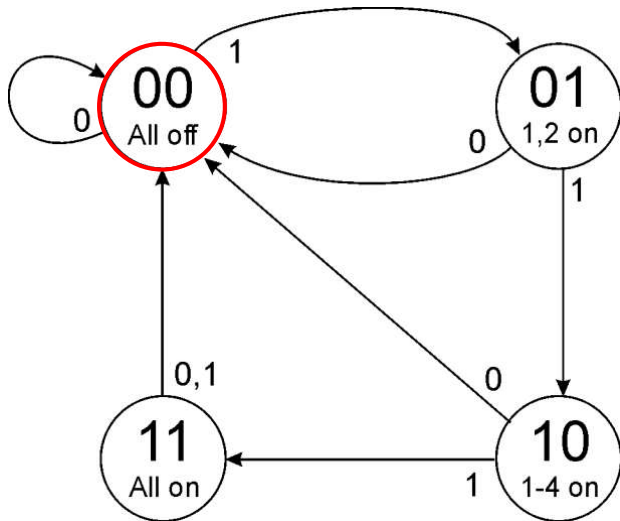
Traffic Sign State Diagram: State 10



Traffic Sign State Diagram: State 11



Traffic Sign State Diagram: State 00



Traffic Sign Truth Tables

Outputs
(depend only on state: $S_1 S_0$)

| S_1 | S_0 | Z | Y | X |
|-------|-------|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

Diagram showing output assignments:

- Lights 1 and 2: Z
- Lights 3 and 4: Y
- Light 5: X

Next State: $S_1' S_0'$
(depend on state and input)

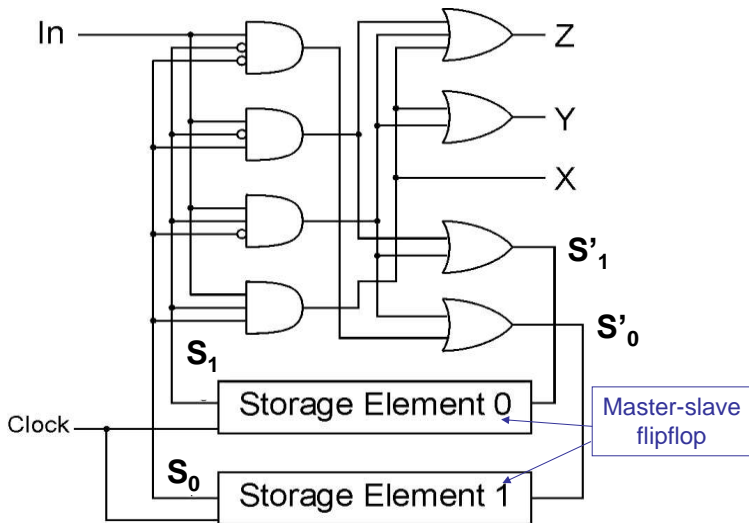
| In | S_1 | S_0 | S_1' | S_0' |
|----|-------|-------|--------|--------|
| 0 | X | X | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |

Diagram showing input assignment:

- Switch: In

Whenever In=0, next state is 00.

Traffic Sign Logic





- 1 Review
- 2 Sequential Logic Circuits
- 3 From Logic to Data Path
- 4 Summary

From Logic to Data Path

The data path of a computer is all the logic used to process information.

- See the data path of the LC-3 on next slide.

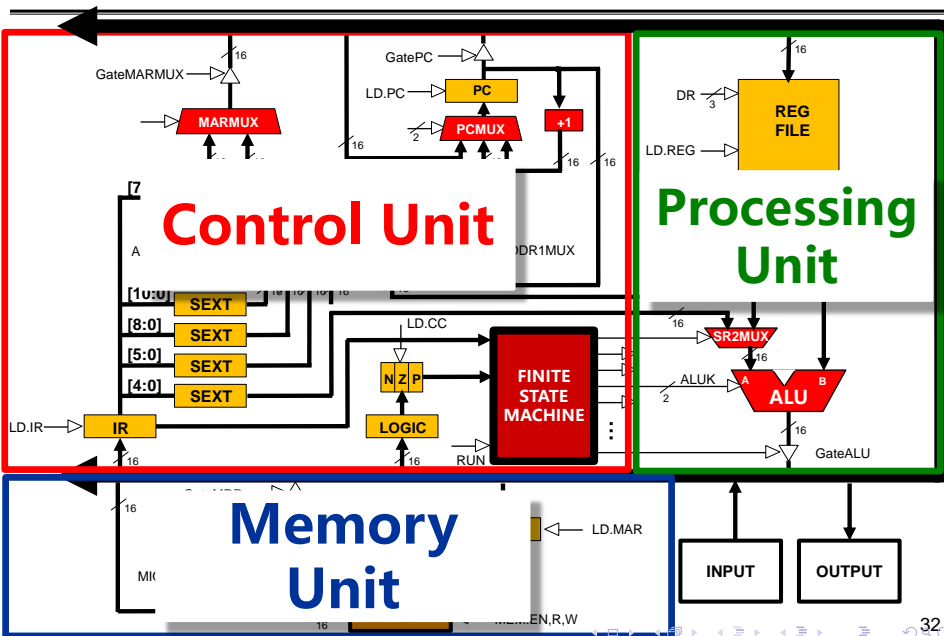
Combinational Logic

- **Decoders -- convert instructions into control signals**
- **Multiplexers -- select inputs and outputs**
- **ALU (Arithmetic and Logic Unit) -- operations on data**

Sequential Logic

- **State machine -- coordinate control signals and data movement**
- **Registers and latches -- storage elements**

LC-3 Data Path Overview (Microarchitecture)





- 1 Review
- 2 Sequential Logic Circuits
- 3 From Logic to Data Path
- 4 Summary

Summary

■ MOS transistors are used as switches to implement logic functions.

- N-type: connect to GND, turn on (with 1) to pull down to 0
- P-type: connect to +2.9V, turn on (with 0) to pull up to 1

■ Basic gates: NOT, NOR, NAND

- Logic functions are usually expressed with AND, OR, and NOT

■ Properties of logic gates

- Completeness
 - can implement any truth table with AND, OR, NOT
- DeMorgan's Law
 - convert AND to OR by inverting inputs and output

Summary

■ We' ve touched on basic digital logic

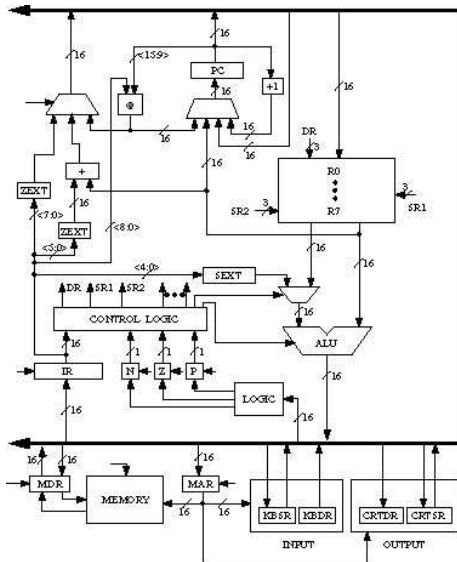
- Transistors
- Gates
- Storage (latches, flip-flops, memory)
- State machines

■ Built some simple circuits

- adder, subtracter, adder/subtracter, Incrementer
- Counter (consisting of register and incrementer)
- Hard-coded traffic sign state machine
- Programmable traffic sign state machine

■ Up next: a computer as a (simple?) state machine

LC-3 Data Path



Next Time

■ Topic

- The von Neumann Model

■ Readings

- Chapter 4.0 - 4.2



中国科学技术大学
University of Science and Technology of China

计算机系统概论

Introduction to Computing Systems
(CS1002A.02)

Chapter 4 The von Neumann Model

计算机科学与技术学院
School of Computer Science and Technology



- 1 **Review**
- 2 **From ENIAC to the Stored Program Computer**
- 3 **A Machine Structure: von Neumann Model**
- 4 **Summary**



1 Review

2 From ENIAC to the Stored Program Computer

3 A Machine Structure: von Neumann Model

4 Summary

Review

■ MOS transistors are used as switches to implement logic functions.

- N-type: connect to GND, turn on (with 1) to pull down to 0
- P-type: connect to +2.9V, turn on (with 0) to pull up to 1

■ Basic gates: NOT, NOR, NAND

- Logic functions are usually expressed with AND, OR, and NOT

■ Properties of logic gates

- Completeness
 - can implement any truth table with AND, OR, NOT
- DeMorgan's Law
 - convert AND to OR by inverting inputs and output

Review

■ We' ve touched on basic digital logic

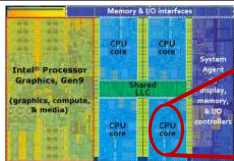
- Transistors
- Gates
- Storage (latches, flip-flops, memory)
- State machines

■ Built some simple circuits

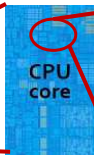
- adder, subtracter, adder/subtracter, Incrementer
- Counter (consisting of register and incrementer)
- Hard-coded traffic sign state machine
- Programmable traffic sign state machine

■ Up next: a computer as a state machine

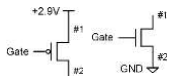
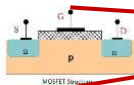
Bottom up approach



Integrated Circuit Design
100 Modules/ IC
0.25M~20G Devices



Register Transfer Level (RTL) Design
1K~10K Cells/Module
(100K Devices)

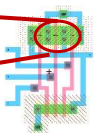


Transistor Physical Layout

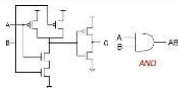
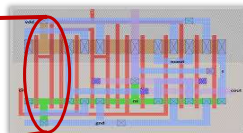
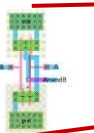
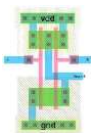


Scheme for Representing Information

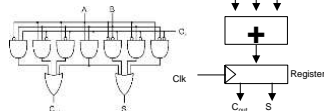
2025/2/24



Gate Level Design



Circuit Level Design
(Transistor Level Design)
(2~8 Devices/Gate)



Register Transfer Level (RTL) Design
2~16 Gates/Cell
(16~64 Devices)



■ Great Idea #2: Stored program computer(Von Neumann Model--A Machine Structure

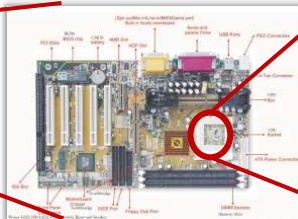
- Basic Components for a machine
- The LC-3: An Example of von Neumann Machine
- Instruction Processing

Bottom up approach

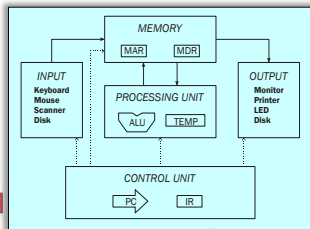
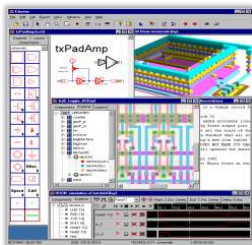
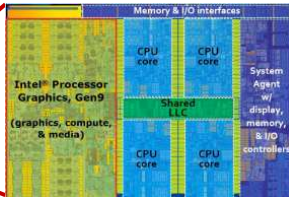
**Personal Computer:
Hardware & Software Design**
1~10PCBs/System



Motherboard Circuit Design
10 ICs/ PCB
1~50G Devices



Integrated Circuit Design
100 Modules/ IC
0.25M~20G Devices



Electronic System Level (ESL) Design

Now, You are Here.

Great Idea #3: Abstraction Helps Us Manage Complexity

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

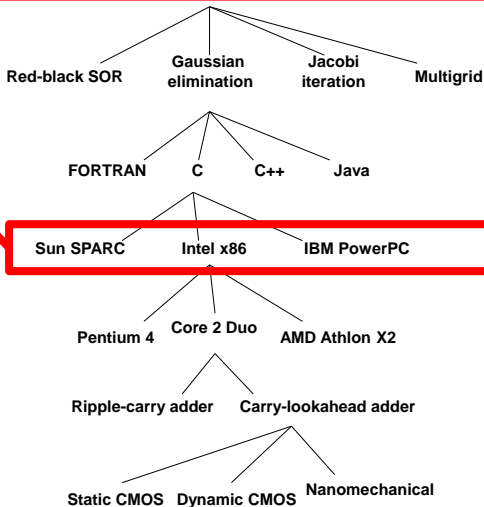
Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

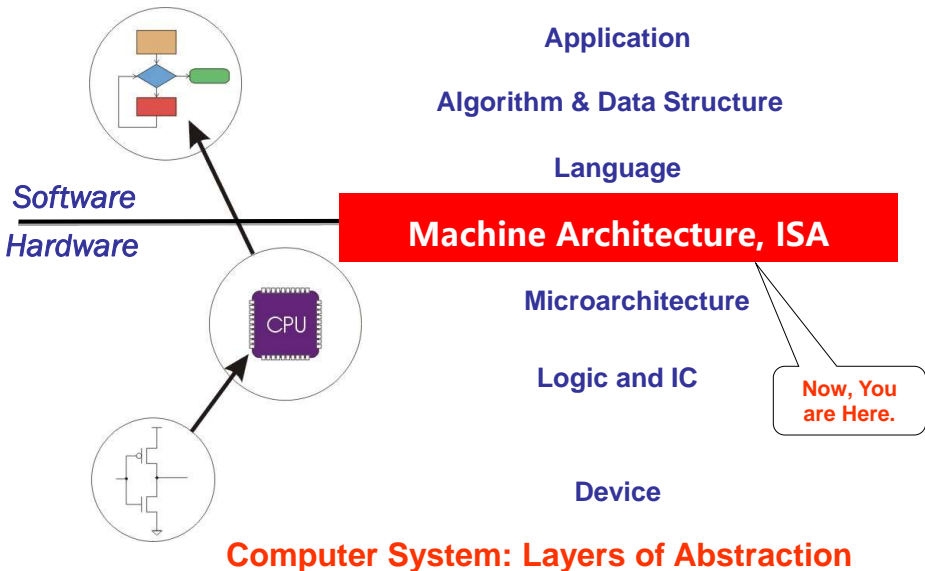
Electronic Devices

Physics

Solve a system of equations



Great Idea #4: Software and Hardware Co-design

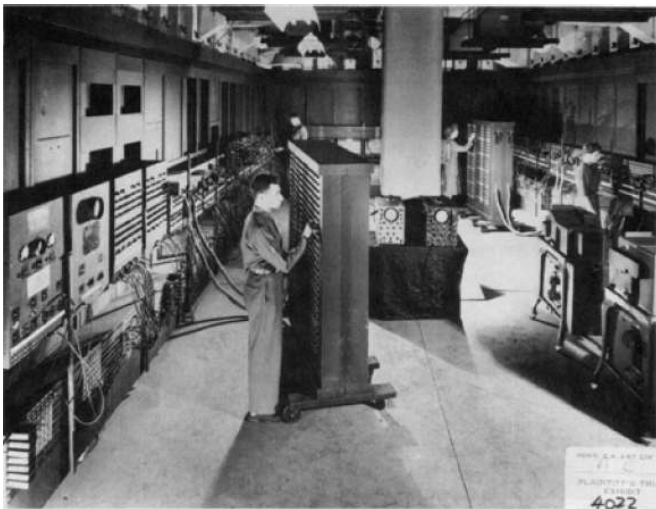




- 1 Review
- 2 From ENIAC to the Stored Program Computer
- 3 A Machine Structure: von Neumann Model
- 4 Summary

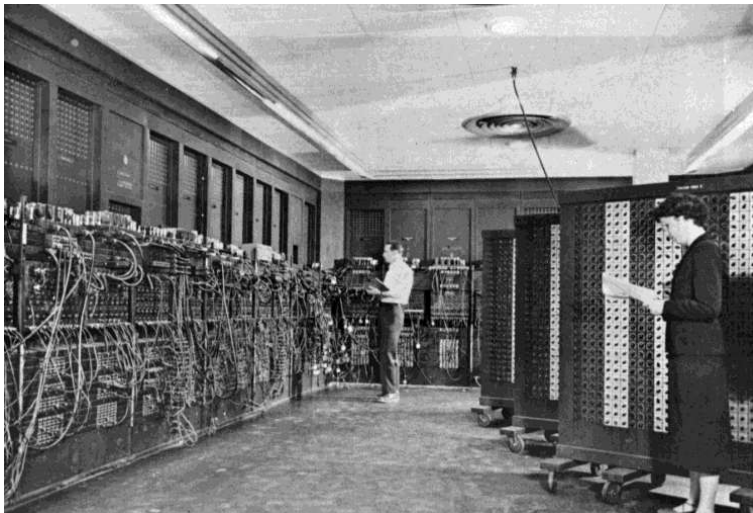


ENIAC - The first electronic computer ,1946年



Programmed by plugboard and switches, time consuming!

ENIAC - The first electronic computer ,1946年



Changing the program could take days!

The Origin of the Stored Program Computer



John von Neumann,
c. 1955
Credit: Computer
History Museum

1946: ENIAC

- Presper Eckert and John Mauchly -- first general electronic computer.
- Hard-wired program -- settings of dials and switches.

1944: Beginnings of EDVAC(Electronic Discrete Variable Automatic Computer)

- John von Neumann joined ENIAC team and proposed a stored program computer called EDVAC

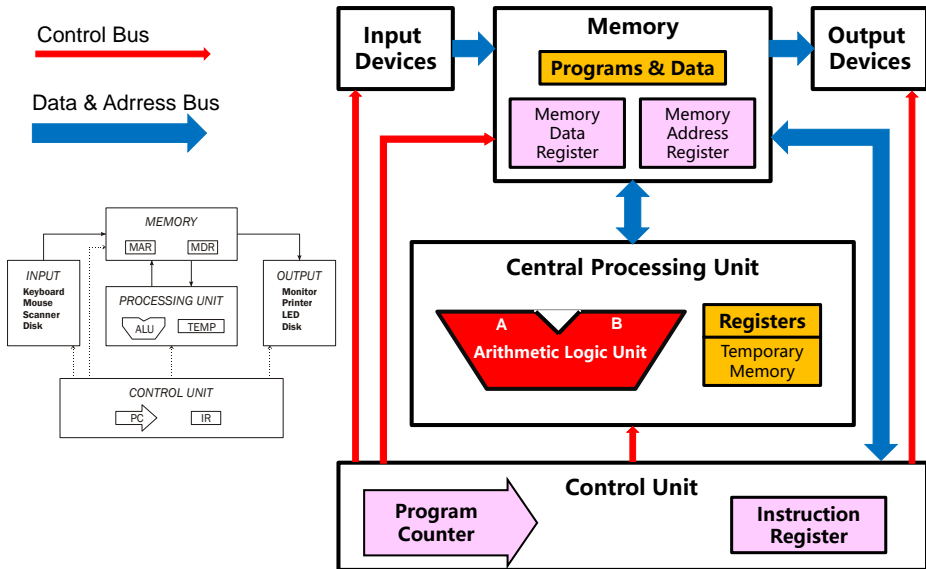
1945: John von Neumann

- John von Neumann wrote "First Draft of a Report on the EDVAC" in which he outlined the architecture of a stored-program computer.

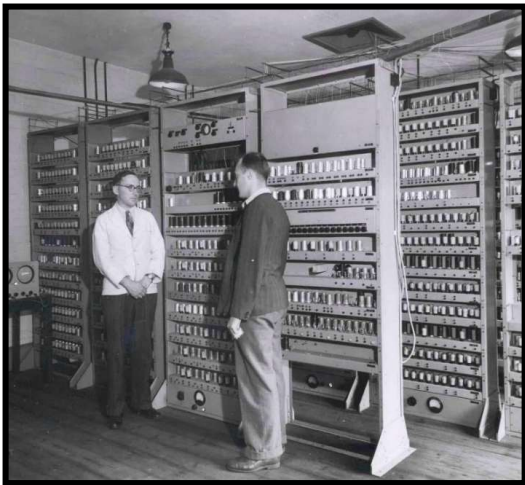
The basic structure proposed in the draft became known as the "von Neumann machine" (or model).

- a memory, containing instructions and data
- a processing unit, for performing arithmetic and logical operations
- a control unit, for interpreting instructions

The Stored Program Computer Architecture (von Neumann Machine Architecture or Model)



The Stored Program Computer



EDSAC
University of Cambridge
UK, 1949



Maurice Vincent
Wilkes

Electronic storage of programming information and data eliminated the need for the more clumsy methods of programming, such as punched paper tape — a concept that has characterized mainstream computer development since 1945.

Two major inventions of the microprocessor chip

Stored program + Transistor technology



Change the program
so that you can do all
kinds of tasks **on the**
same hardware

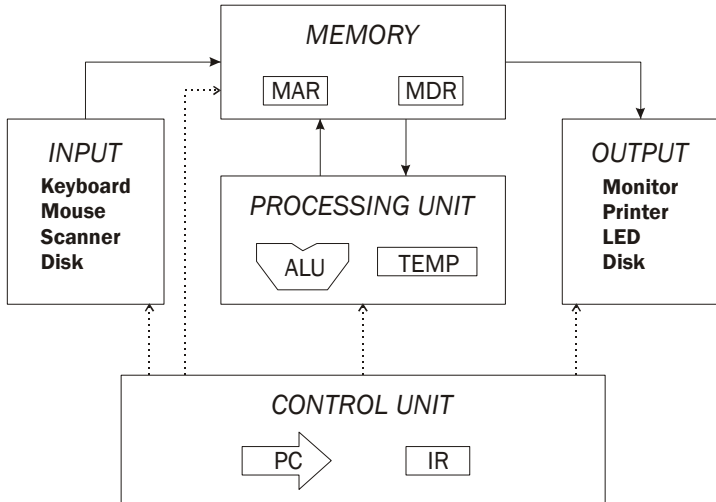


The device is
smaller and **faster**
than a vacuum
tube



- 1 Review
- 2 From ENIAC to the Stored Program Computer
- 3 A Machine Structure: von Neumann Model
- 4 Summary

von Neumann Model



LC-3 Data Path



Control Unit

Processing Unit

Memory Unit

INPUT

OUTPUT

Memory

$k \times m$ array of stored bits (k is usually 2^n)

Address

- unique (n -bit) identifier of location

Contents

- m -bit value stored in location

Basic Operations:

LOAD

- read a value from a memory location

STORE

- write a value to a memory location

| | |
|------|----------|
| 0000 | |
| 0001 | |
| 0010 | |
| 0011 | 00101101 |
| 0100 | |
| 0101 | |
| 0110 | |
| | ⋮ |
| 1101 | 10100010 |
| 1110 | |
| 1111 | |

Interface to Memory

How does processing unit get data to/from memory?

MAR: Memory Address Register

MDR: Memory Data Register



To read a location (A):

1. Write the address (A) into the MAR.
2. Send a "read" signal to the memory.
3. Read the data from MDR.

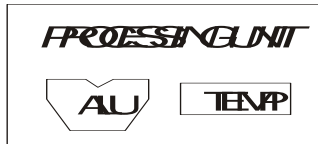
To write a value (X) to a location (A):

1. Write the data (X) to the MDR.
2. Write the address (A) into the MAR.
3. Send a "write" signal to the memory.

Processing Unit

Functional Units

- ALU = Arithmetic and Logic Unit
- could have many functional units.
some of them special-purpose
(multiply, square root, ...)
- LC-3 performs ADD, AND, NOT



Registers

- Small, temporary storage
- Operands and results of functional units
- LC-3 has eight register (R0, ..., R7)

Word Size

- number of bits normally processed by ALU in one instruction
- also width of registers
- LC-3 is 16 bits

Input and Output

- Devices for getting data into and out of computer memory
- Each device has its own interface, usually a set of registers like the memory's **MAR and MDR**

INPUT

Keyboard
Mouse
Scanner
Disk

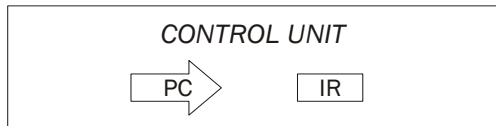
OUTPUT

Monitor
Printer
LED
Disk

- LC-3 supports keyboard (input) and console (output)
- keyboard: data register (KBDR) and status register (KBSR)
- console: data register (CRTDR) and status register (CRTSR)
- frame buffer: memory-mapped pixels
- Some devices provide both input and output
 - disk, network
- Program that controls access to a device is usually called **a driver**.

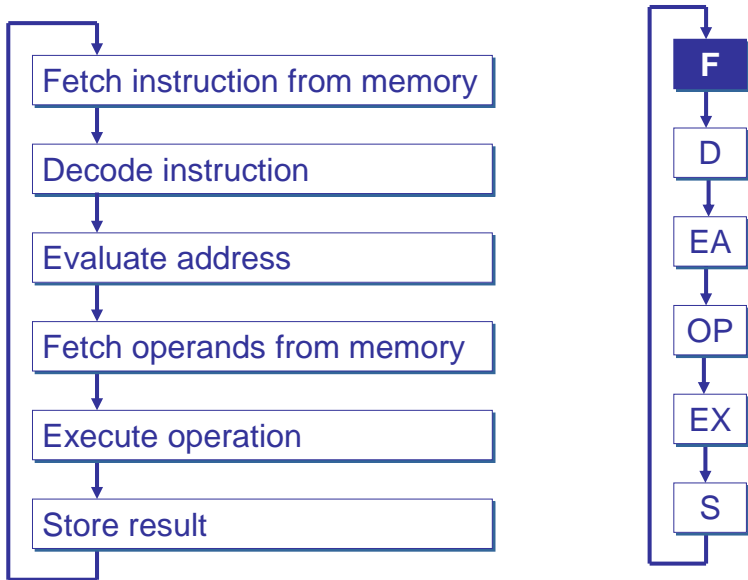
Control Unit

■ Orchestrates execution of the program



- **Instruction Register (IR)** contains the current instruction.
- **Program Counter (PC)** contains the address of the next instruction to be executed.
- **Control unit:**
 - reads an instruction from memory
 - the instruction's address is in the PC
 - interprets the instruction, generating signals that tell the other components what to do
 - an instruction may take many *machine cycles* to complete

Instruction Processing (State Transtion)



Instruction

- The instruction is the fundamental unit of work.
- Specifies two things:
 - opcode: operation to be performed
 - operands: data/locations to be used for operation
- An instruction is encoded as a sequence of bits.
(*Just like data!*)
 - Often, but not always, instructions have a fixed length, such as 16 or 32 bits.
 - Control unit interprets instruction:
generates sequence of control signals to carry out operation.
 - Operation is either executed completely, or not at all.
- A computer's instructions and their formats is known as its *Instruction Set Architecture (ISA)*.
 - Persistent ISA invented by UW grad Gene Amdahl (IBM 360)

Example: LC-3 ADD Instruction

LC-3 has 16-bit instructions.

- Each instruction has a four-bit opcode, bits [15:12].

LC-3 has eight *registers* (R0-R7) for temporary storage.

- Sources and destination of ADD are registers.

| | | | | | | | | | | | | | | | |
|-----|----|----|----|-----|----|---|------|---|---|---|---|---|------|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ADD | | | | Dst | | | Src1 | | | 0 | 0 | 0 | Src2 | | |

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

*“Add the contents of R2 to the contents of R6,
and store the result in R6.”*

Example: LC-3 LDR Instruction

Load instruction -- reads data from memory

Base + offset mode:

- add offset to base register -- result is memory address
- load from memory address into destination register

| | | | | | | | | | | | | | | | |
|-----|----|----|----|-----|----|---|------|---|---|--------|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| LDR | | | | Dst | | | Base | | | Offset | | | | | |

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |

“Add the value 6 to the contents of R3 to form a memory address. Load the contents stored in that address to R2.”

Instruction Processing: FETCH

■ Load next instruction (at address stored in PC) from memory into Instruction Register (IR).

- Load contents of PC into MAR.
- Send "read" signal to memory.
- Read contents of MDR, store in IR.

■ Then increment PC, so that it points to the next instruction in sequence.

- PC becomes PC+1.

| | | | | | | | | | | | | | | | |
|-----|----|----|----|-----|----|---|------|---|---|---|---|---|------|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ADD | | | | Dst | | | Src1 | | | 0 | 0 | 0 | Src2 | | |

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |



Instruction Processing: DECODE

■ First identify the opcode.

- In LC-3, this is always the first four bits of instruction.
- A 4-to-16 decoder asserts a control line corresponding to the desired opcode.

■ Depending on opcode, identify other operands from the remaining bits.

● Example:

- for ADD, last three bits is source operand #2
- for LDR, last six bits is offset

| | | | | | | | | | | | | | | | |
|-----|----|----|----|-----|----|---|------|---|---|---|---|---|------|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ADD | | | | Dst | | | Src1 | | | 0 | 0 | 0 | Src2 | | |

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

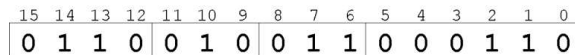
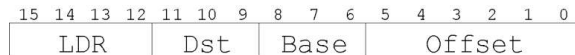
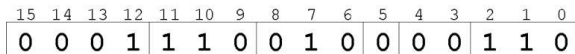
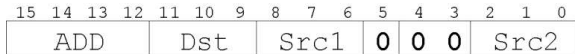


Instruction Processing: EVALUATE ADDRESS

- For instructions that require memory access, compute address used for access.

- Examples:

- add offset to base register (as in LDR)
- add offset to PC (or to part of PC)
- add offset to zero

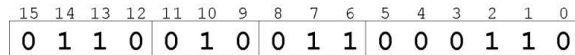
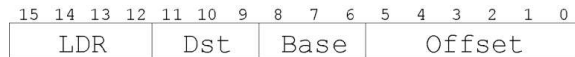
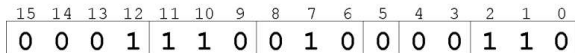
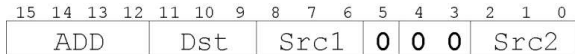


Instruction Processing: FETCH OPERANDS

- Obtain source operands needed to perform operation.

- Examples:

- read data from register file (ADD)
- load data from memory (LDR)

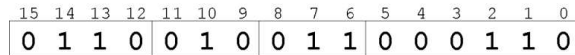
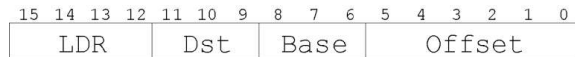
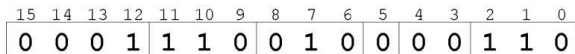
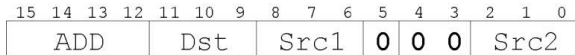


Instruction Processing: EXECUTE

Perform the operation,
using the source operands.

Examples:

- send operands to ALU and assert ADD signal
- do nothing (e.g., for loads and stores)



Instruction Processing: STORE

■ Write results to destination. (register or memory)

■ Examples:

- result of ADD is placed in destination register
- result of memory load is placed in destination register
- for store instruction, data is stored to memory
 - write address to MAR, data to MDR
 - assert WRITE signal to memory

| | | | | | | | | | | | | | | | |
|-----|----|----|----|-----|----|---|------|---|---|---|---|---|------|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ADD | | | | Dst | | | Src1 | | | 0 | 0 | 0 | Src2 | | |

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |



Example: LC-3 JMP Instruction

- Set the PC to the value contained in a register. This becomes the address of the next instruction to fetch.

| | | | | | | | | | | | | | | | |
|-----|----|----|----|----|----|---|------|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| JMP | | | | 0 | 0 | 0 | Base | | | 0 | 0 | 0 | 0 | 0 | 0 |

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

“Load the contents of R3 into the PC.”

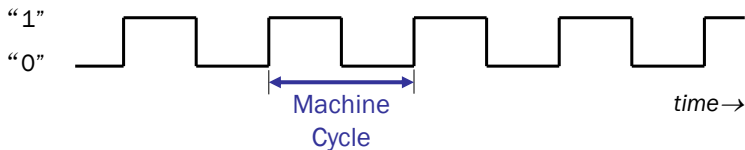
Driving Force: The Clock

The clock is a signal that keeps the control unit moving.

- At each clock "tick," control unit moves to the next machine cycle -- may be next instruction or next phase of current instruction.

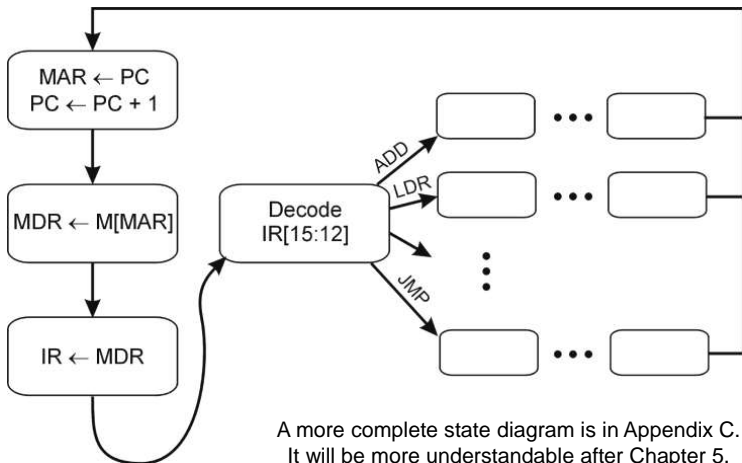
Clock generator circuit:

- Based on crystal oscillator
- Generates regular sequence of "0" and "1" logic levels
- Clock cycle (or machine cycle) -- rising edge to rising edge



Control Unit State Diagram

- The control unit is a state machine. Here is part of a simplified state diagram for the LC-3:



A more complete state diagram is in Appendix C.
It will be more understandable after Chapter 5.

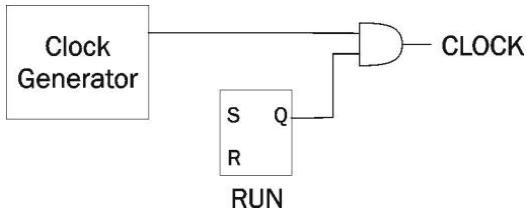
Stopping the Clock

Control unit will repeat instruction processing sequence as long as clock is running.

- If not processing instructions from your application, then it is processing instructions from the Operating System (OS).
- The OS is a special program that manages processor and other resources.

To stop the computer:

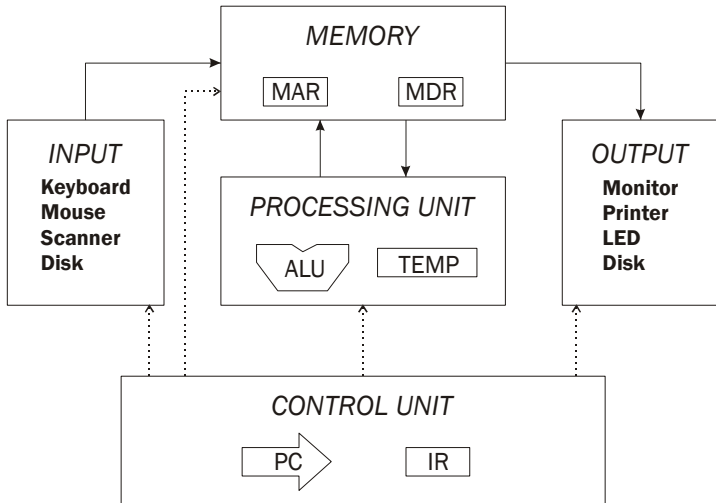
- AND the clock generator signal with ZERO
- when control unit stops seeing the CLOCK signal, it stops processing





- 1 Review
- 2 From ENIAC to the Stored Program Computer
- 3 A Machine Structure: von Neumann Model
- 4 **Summary**

Von Neumann Model



Instruction Processing Summary

■ Instructions look just like data -- it's all interpretation.

■ Three basic kinds of instructions:

- computational instructions (ADD, AND, ...)
- data movement instructions (LD, ST, ...)
- control instructions (JMP, BRnz, ...)

■ Six basic phases of instruction processing:

- not all phases are needed by every instruction
- phases may take variable number of machine cycles



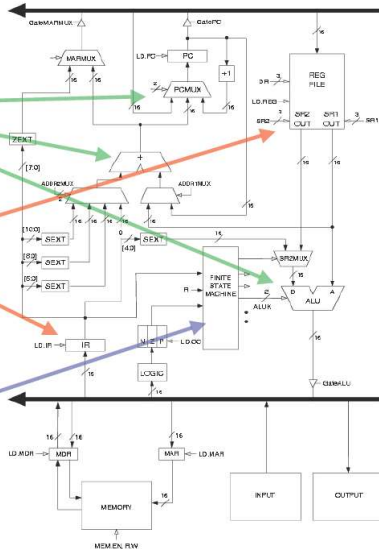
LC-3 Data Path

LC-3 Data Path

Combinational
Logic

Storage

State Machine



CSE 240



中国科学技术大学
University of Science and Technology of China

计算机系统概论

Introduction to Computing Systems
(CS1002A.02)

Chapter 5-1 The LC-3 Operate Instructions

计算机科学与技术学院
School of Computer Science and Technology



1 Review

2 LC-3 ISA Overview

3 LC-3 Operate Instructions and Data Path

4 Summary



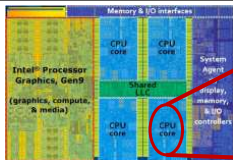
1 Review

2 LC-3 ISA Overview

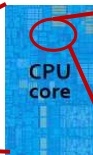
3 LC-3 Operate Instructions and Data Path

4 Summary

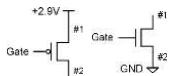
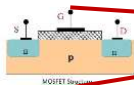
Approach: Bottom Up



Integrated Circuit Design
100 Modules/ IC
0.25M~20G Devices



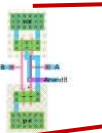
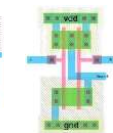
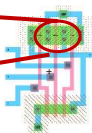
Register Transfer Level (RTL) Design
1K~10K Cells/Module
(100K Devices)



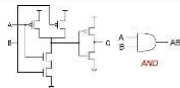
Transistor Physical Layout



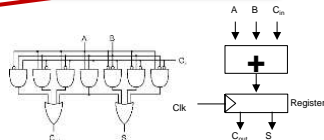
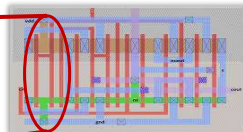
Scheme for Representing Information



Gate Level Design



Circuit Level Design
(Transistor Level Design)
(2~8 Devices/Gate)



Register Transfer Level (RTL) Design
2~16 Gates/Cell
(16~64 Devices)

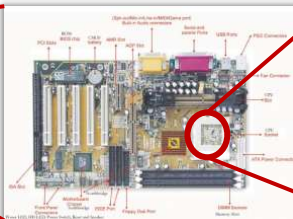
Approach: Bottom Up

Now, You are Here.

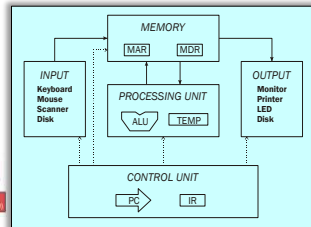
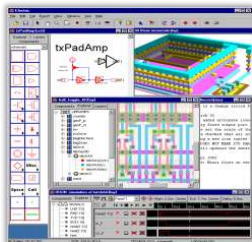
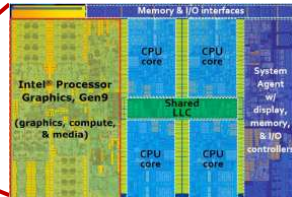
**Personal Computer:
Hardware & Software Design**
1~10PCBs/System



Motherboard Circuit Design
10 ICs/ PCB
1~50G Devices

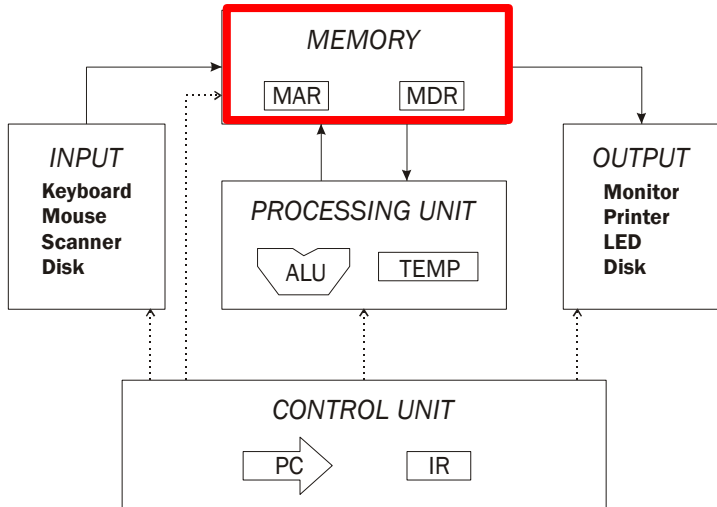


Integrated Circuit Design
100 Modules/ IC
0.25M~20G Devices

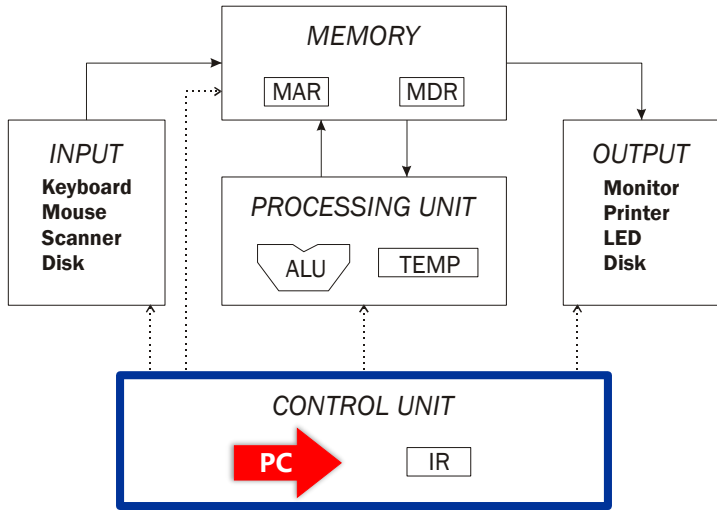


Electronic System Level (ESL) Design

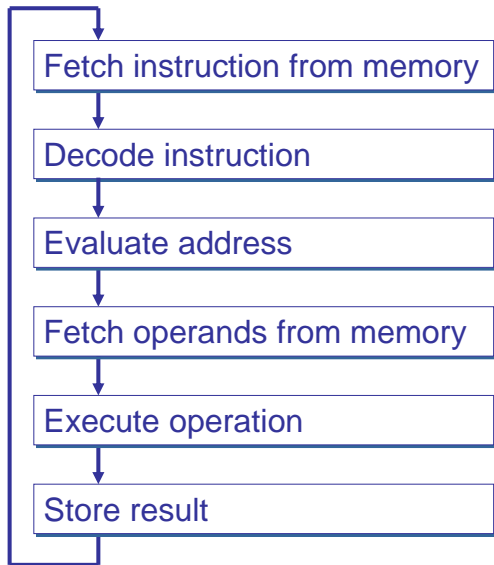
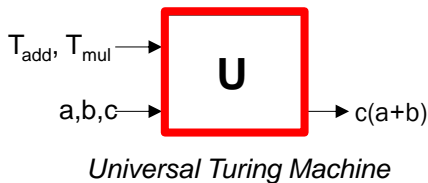
Great Idea #2 Von Neumann Structure (Architecture Model)



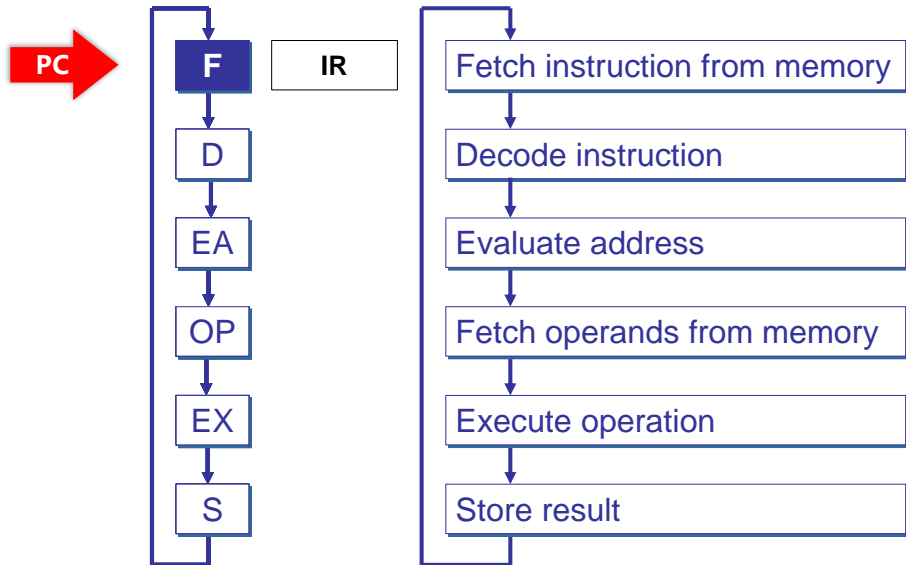
Great Idea #1 Turing Machine (Computational Model)



Great Idea #1 Turing Machine (Computational Model)



Instruction Processing: State Transtion



Instruction Processing: Finite State Automata



568

appendix c The Microarchitecture of the LC-3

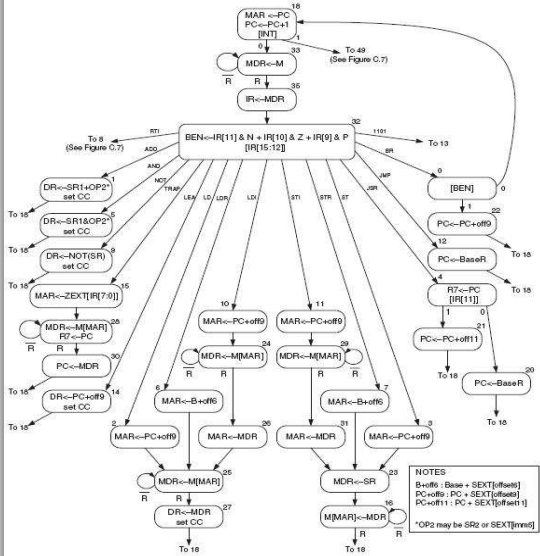
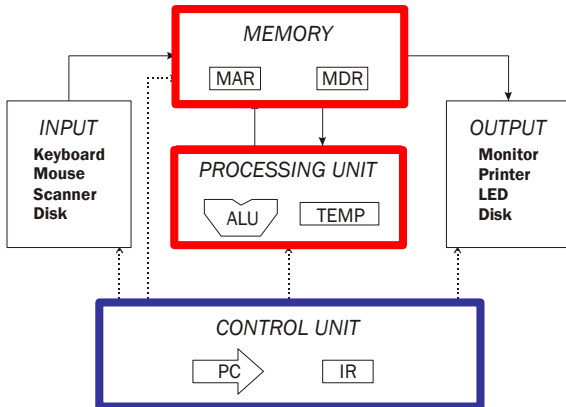


Figure C.2 A state machine for the LC-3

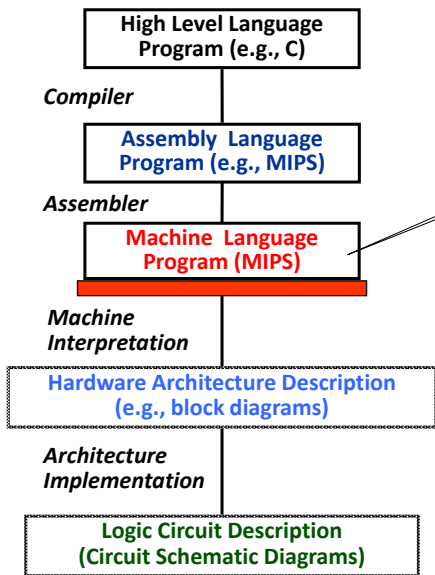


■ We are going to learn how to:

- **compute with values in registers**
- load data from memory to registers
- store data from registers to memory



How do we get the electrons to do the work?



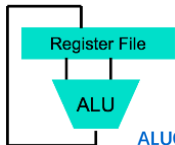
```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

Now, You
are Here.

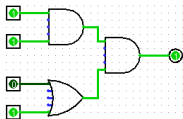
```
lw $t0, 0($2)  
lw $t1, 4($2)  
sw $t1, 0($2)  
sw $t0, 4($2)
```

Anything can be represented
as a *number*,
i.e., data or instructions

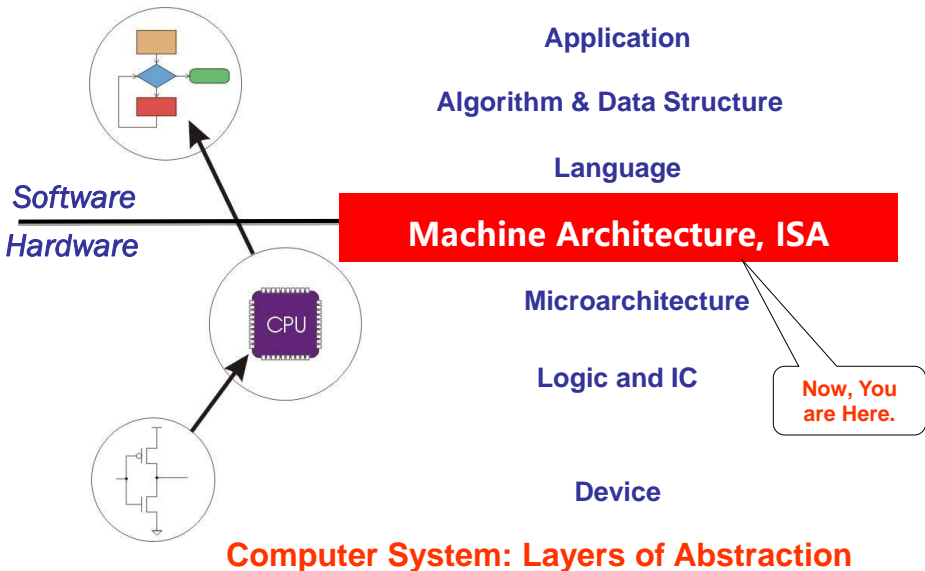
```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```



ALUOP[0:3] <= InstReg[9:11] & MASK



Great Idea #4: Software and Hardware Co-design



Great Idea #3: Abstraction Helps Us Manage Complexity

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

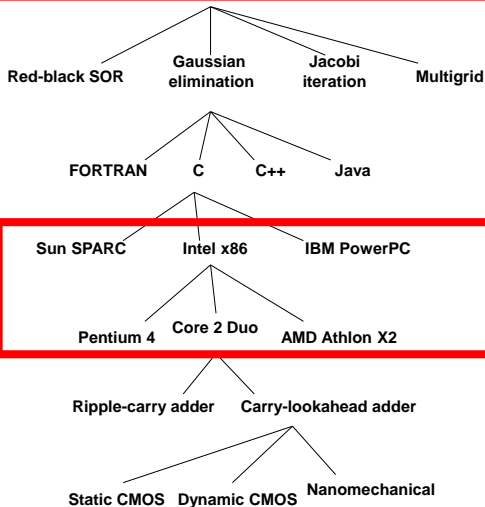
Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

Electronic Devices

Physics

Solve a system of equations





1 Review

2 **LC-3 ISA Overview**

3 LC-3 Operate Instructions and Data Path

4 Summary

Instruction Set Architecture

ISA = All of the *programmer-visible* components and operations of the computer

- **memory organization**

- address space -- how many locations can be addressed?
- addressability -- how many bits per location?

- **register set**

- how many? what size? how are they used?

- **instruction set**

- opcodes
- data types
- addressing modes

ISA provides all information needed for someone that wants to write a program in **machine language** (or translate from a high-level language to machine language).

LC-3 Overview: Memory and Registers

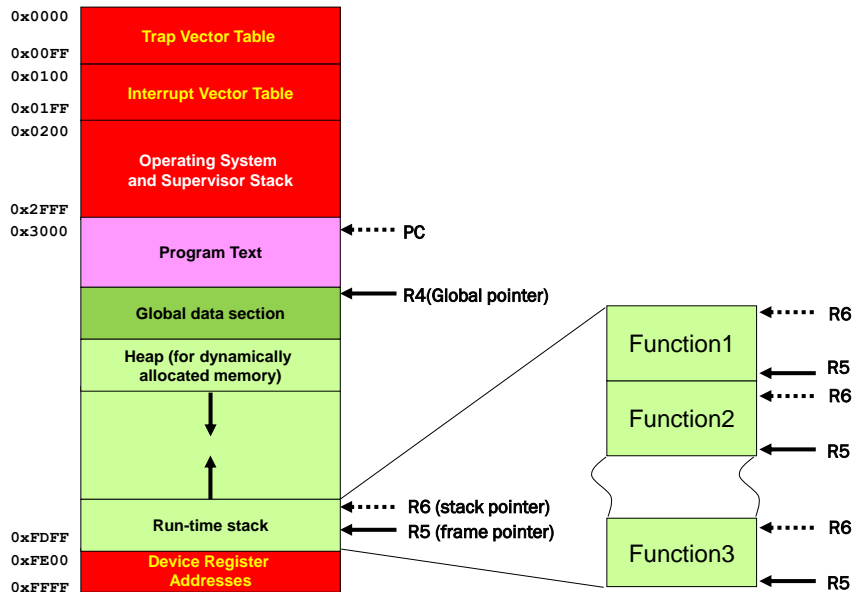
Memory

- address space: 2^{16} locations(16-bit addresses)
- addressability: 16 bits

Registers

- temporary storage, accessed in a single machine cycle
 - accessing memory generally takes longer than a single cycle
- eight general-purpose registers: R0 - R7
 - each 16 bits wide
 - how many bits to uniquely identify a register?
- other registers
 - not directly addressable, but used by (and affected by) instructions
 - PC (program counter), condition codes

LC-3 Overview: Memory Map



LC-3 Overview: Instruction Set

Opcodes

- 15 opcodes
- *Operate* instructions: ADD, AND, NOT
- *Data movement* instructions: LD, LDI, LDR, LEA, ST, STR, STI
- *Control* instructions: BR, JSR/JSRR, JMP(RET), RTI, TRAP
- some opcodes(ADD, AND, NOT; LD, LDI, LDR, LEA) set/clear *condition codes*, based on result:
 - N = negative, Z = zero, P = positive (> 0)

Data Types

- 16-bit 2's complement integer

Addressing Modes

- How is the location of an operand specified?
- non-memory addresses: *immediate, register*
- memory addresses: *PC-relative, indirect, base+offset*

LC-3 ISA Overview

运算指令(Operate Instructions)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|----|----|----|----|----|----|---|-----|---|---|---|------|---|-----|---|---|
| ADD | 0 | 0 | 0 | 1 | DR | | | SR1 | | | 0 | 0 | 0 | SR2 | | |
| ADD | 0 | 0 | 0 | 1 | DR | | | SR1 | | | 1 | Imm5 | | | | |
| AND | 0 | 1 | 0 | 1 | DR | | | SR1 | | | 0 | 0 | 0 | SR2 | | |
| AND | 0 | 1 | 0 | 1 | DR | | | SR1 | | | 1 | Imm5 | | | | |
| NOT | 1 | 0 | 0 | 1 | DR | | | SR1 | | | 1 | 1 | 1 | 1 | 1 | 1 |
| Reserved | 1 | 1 | 0 | 1 | | | | | | | | | | | | |

控制指令(Control Instructions)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|------------|---|-----------|-------------|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | n | z | p | PCOffset9 | | | | | | | | |
| JSR | 0 | 1 | 0 | 0 | 1 | PCOffset11 | | | | | | | | | | |
| JSRR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RET | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| TRAP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | TrapVector8 | | | | | | | |

数据移动指令 (Data Movement Instructions)

取数指令(Load)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|-----------|---|---|---|---|---|---|
| LD | 0 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LDR | 0 | 1 | 1 | 0 | DR | | | BaseR | | PCOffset6 | | | | | | |
| LDI | 1 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LEA | 1 | 1 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |

存数指令(Store)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|---|-----------|---|---|---|---|---|
| ST | 0 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |
| STR | 0 | 1 | 1 | 1 | SR | | | BaseR | | | PCOffset6 | | | | | |
| STI | 1 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |



1 Review

2 LC-3 ISA Overview

3 LC-3 Operate Instructions and Data Path

4 Summary

Operate Instructions

Only three operations: **ADD, AND, NOT**

Source and destination operands are **registers**

- These instructions do not reference memory.
- ADD and AND can use “immediate” mode, where one operand is hard-wired into the instruction.

Will show **dataflow diagram** with each instruction.

- illustrates when and where data moves to accomplish the desired operation

LC-3 ISA Operate Instructions

运算指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|----|----|----|----|----|----|-----|---|---|------|---|-----|---|---|---|---|
| ADD | 0 | 0 | 0 | 1 | DR | | SR1 | | 0 | 0 | 0 | SR2 | | | | |
| ADD | 0 | 0 | 0 | 1 | DR | | SR1 | | 1 | Imm5 | | | | | | |
| AND | 0 | 1 | 0 | 1 | DR | | SR1 | | 0 | 0 | 0 | SR2 | | | | |
| AND | 0 | 1 | 0 | 1 | DR | | SR1 | | 1 | Imm5 | | | | | | |
| NOT | 1 | 0 | 0 | 1 | DR | | SR1 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Reserved | 1 | 1 | 0 | 1 | | | | | | | | | | | | |

控制指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|------------|---|-----------|-------------|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | n | z | p | PCOffset9 | | | | | | | | |
| JSR | 0 | 1 | 0 | 0 | 1 | PCOffset11 | | | | | | | | | | |
| JSRR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RET | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| TRAP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | TrapVector8 | | | | | | | |

移动数据指令

取数指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|-----------|---|---|---|---|---|---|
| LD | 0 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LDR | 0 | 1 | 1 | 0 | DR | | | BaseR | | PCOffset6 | | | | | | |
| LDI | 1 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LEA | 1 | 1 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |

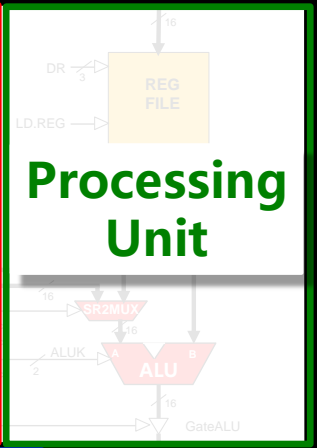
存数指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|---|-----------|---|---|---|---|---|
| ST | 0 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |
| STR | 0 | 1 | 1 | 1 | SR | | | BaseR | | | PCOffset6 | | | | | |
| STI | 1 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |

Operate Instructions Overview

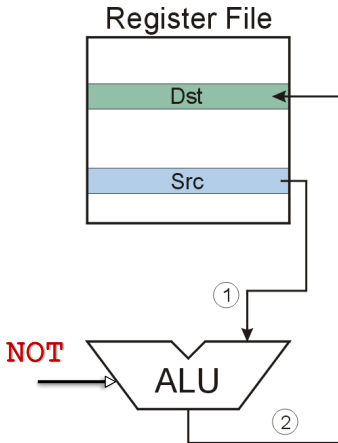
| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|----|----|----|----|----|----|---|-----|---|---|---|------|---|-----|---|---|
| ADD | 0 | 0 | 0 | 1 | DR | | | SR1 | | | 0 | 0 | 0 | SR2 | | |
| ADD | 0 | 0 | 0 | 1 | DR | | | SR1 | | | 1 | Imm5 | | | | |
| AND | 0 | 1 | 0 | 1 | DR | | | SR1 | | | 0 | 0 | 0 | SR2 | | |
| AND | 0 | 1 | 0 | 1 | DR | | | SR1 | | | 1 | Imm5 | | | | |
| NOT | 1 | 0 | 0 | 1 | DR | | | SR1 | | | 1 | 1 | 1 | 1 | 1 | 1 |
| Reserved | 1 | 1 | 0 | 1 | | | | | | | | | | | | |

```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



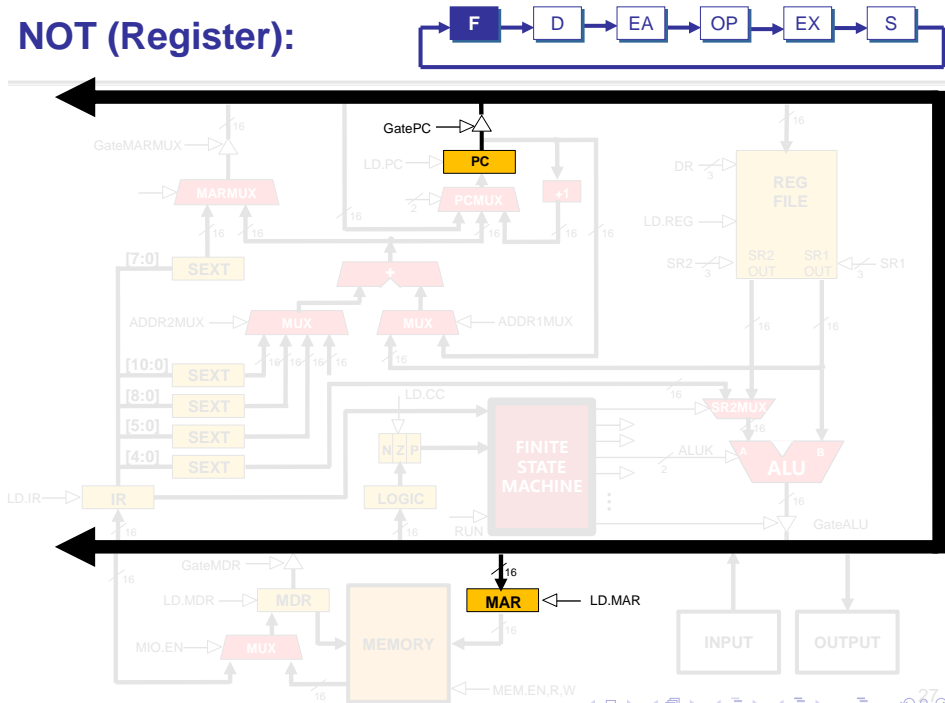
NOT (Register)

| | | | | | | | | | | | | | | | | |
|-----|----|----|----|----|-----|----|---|-----|---|---|---|---|---|---|---|---|
| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| NOT | 1 | 0 | 0 | 1 | Dst | | | Src | | | 1 | 1 | 1 | 1 | 1 | 1 |

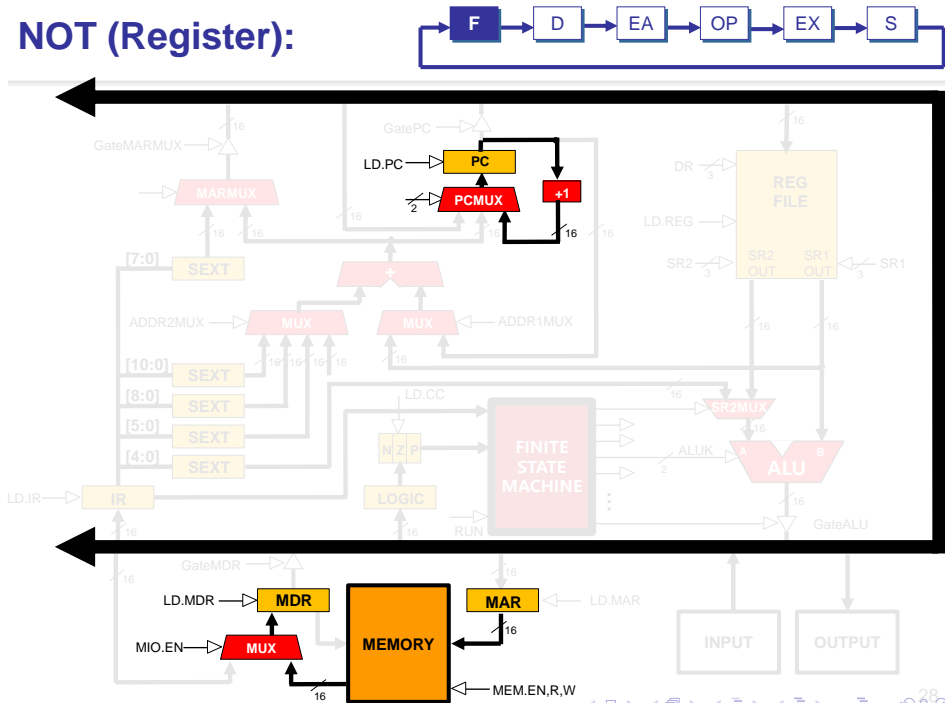


Note: Src and Dst could be the same register.

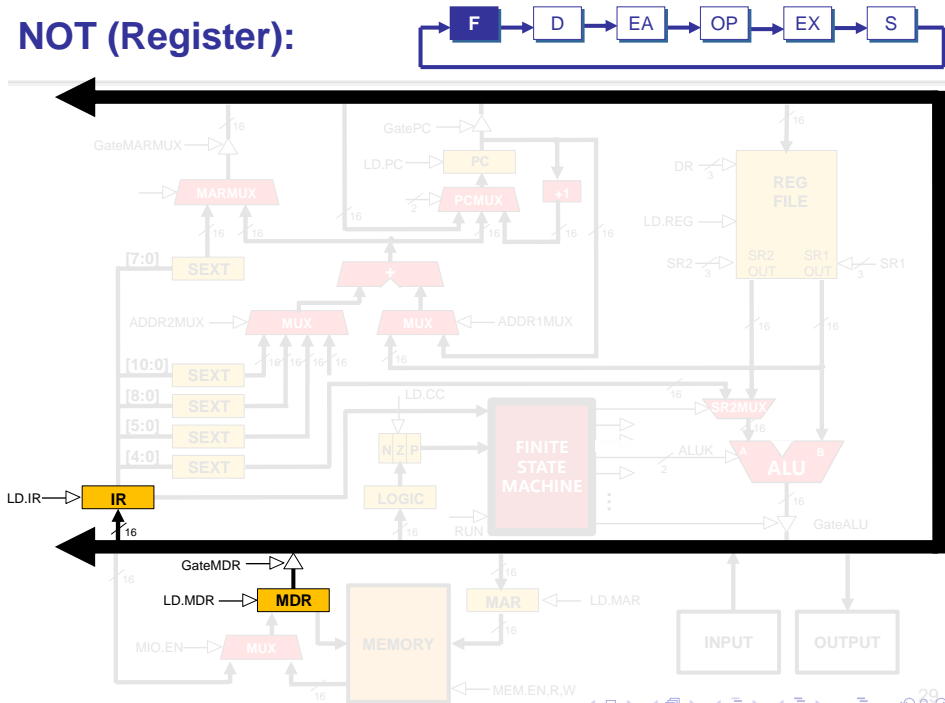
NOT (Register):



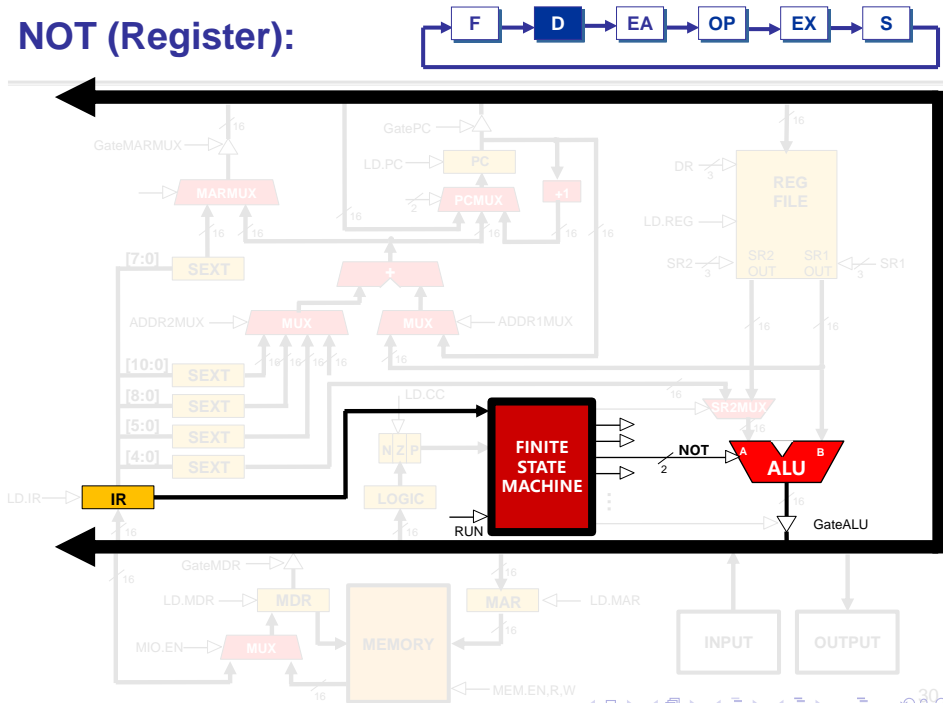
NOT (Register):



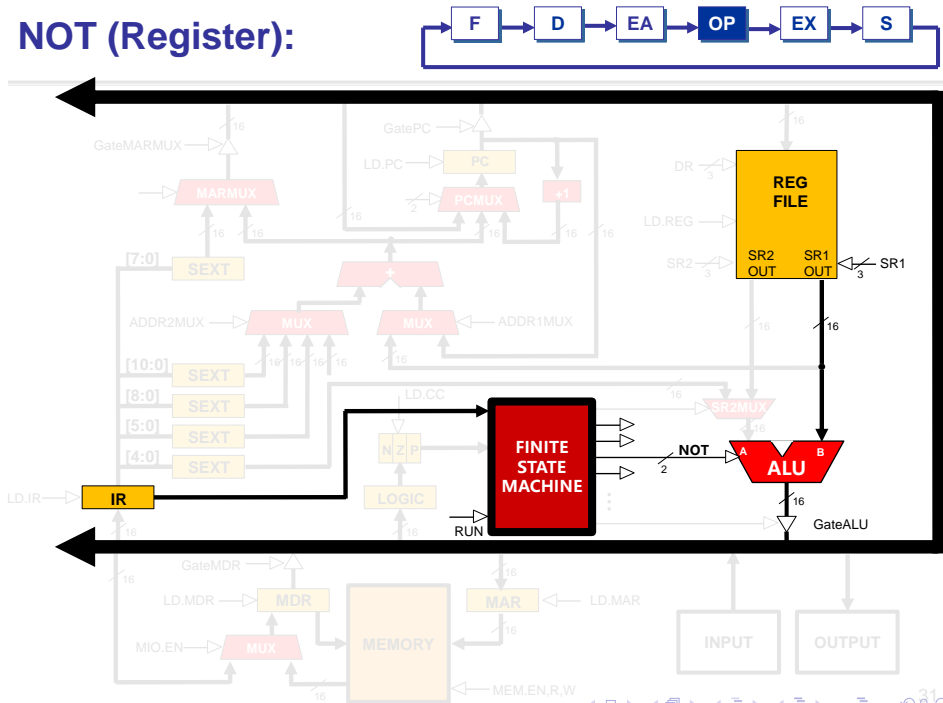
NOT (Register):



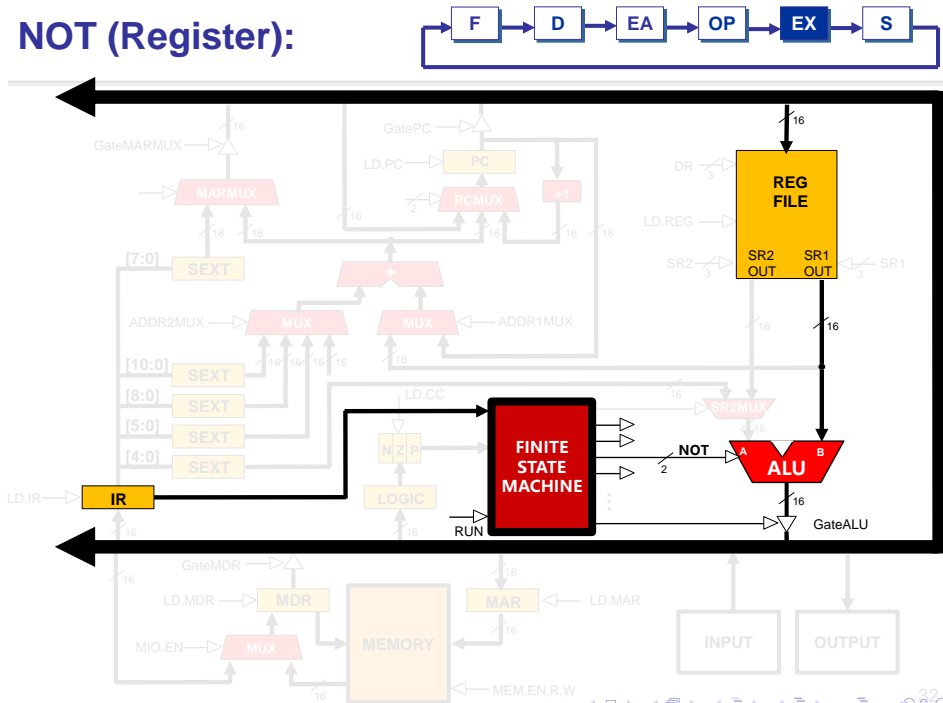
NOT (Register):



NOT (Register):



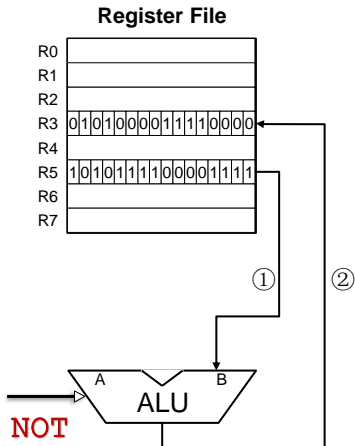
NOT (Register):



```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```

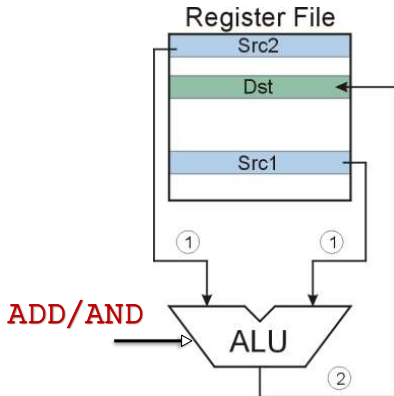


NOT (Register): NOT R3, R5



ADD/AND (Register)

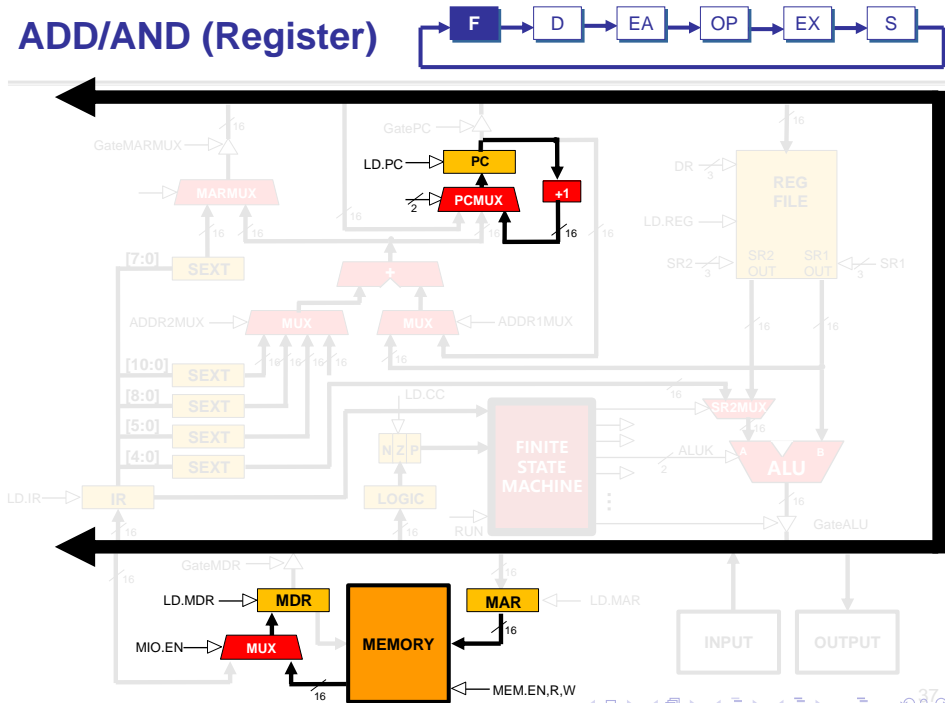
this zero means "register mode"



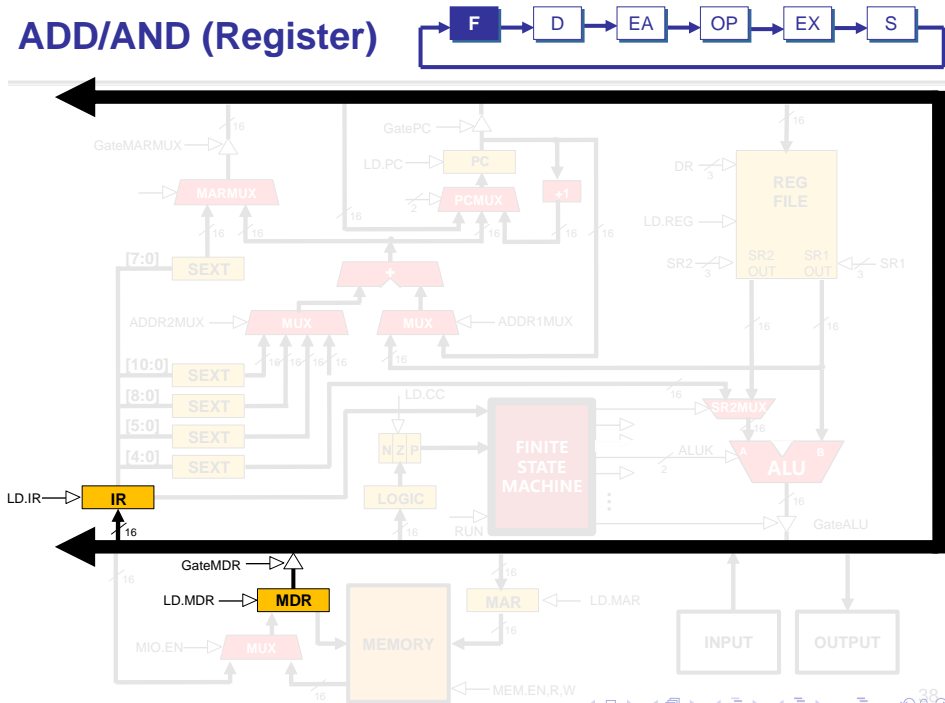
```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



ADD/AND (Register)



ADD/AND (Register)

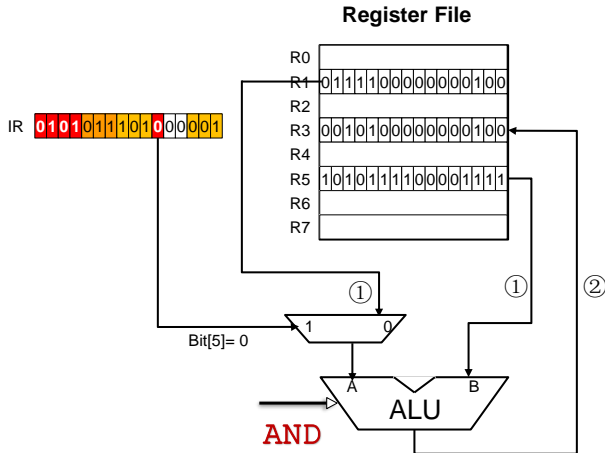



```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



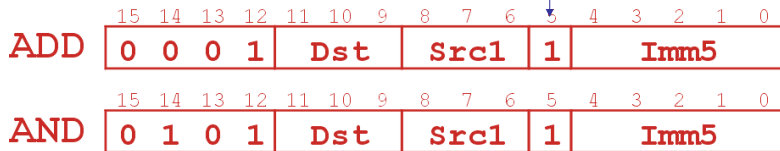
AND (Register): AND R3, R5, R1

| | | | | | | | | | | | | | | | | |
|-----|----|----|----|----|-----|----|---|------|---|---|---|---|---|------|---|---|
| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| AND | 0 | 1 | 0 | 1 | Dst | | | Src1 | | | 0 | 0 | 0 | Src2 | | |

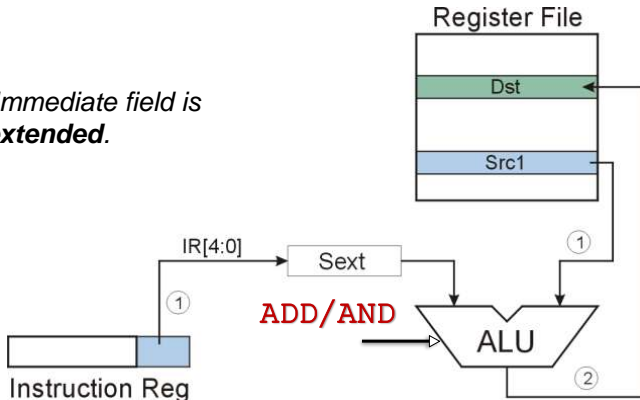


ADD/AND (Immediate)

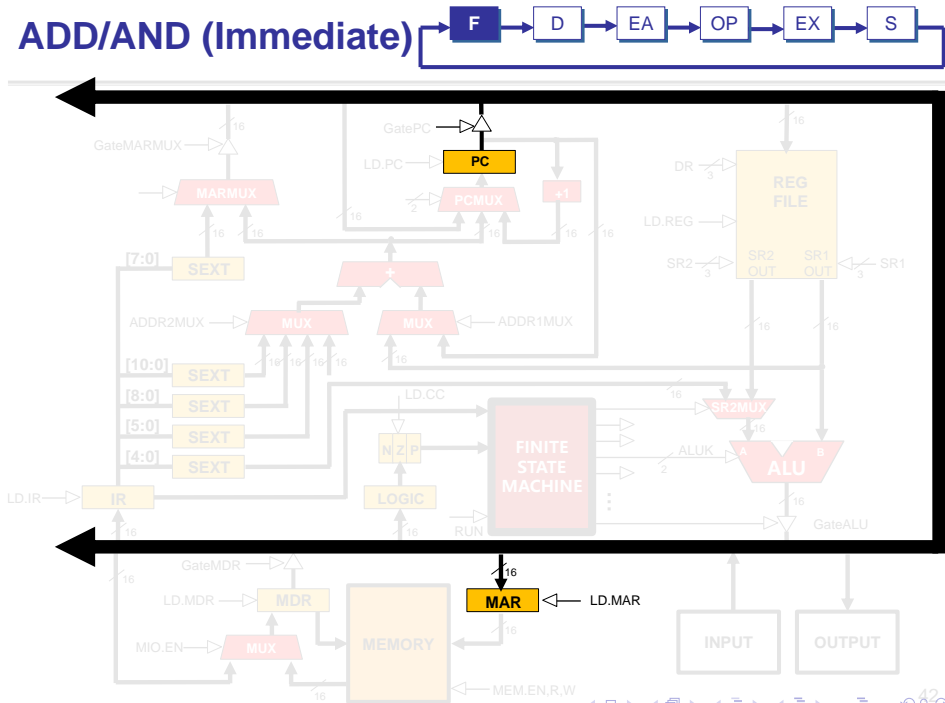
this one means "immediate mode"



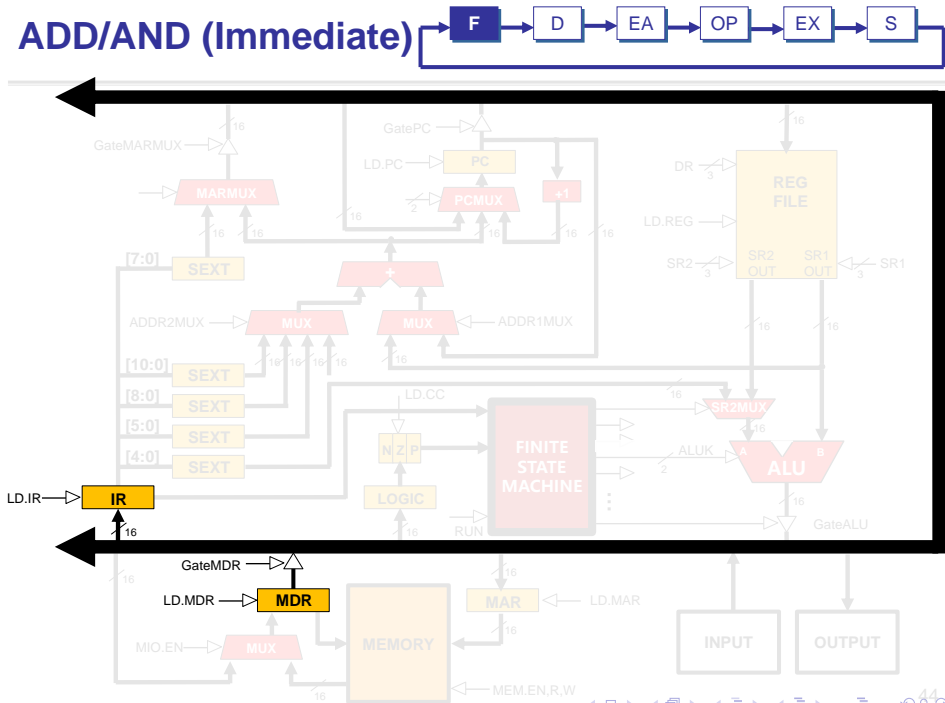
Note: Immediate field is **sign-extended**.



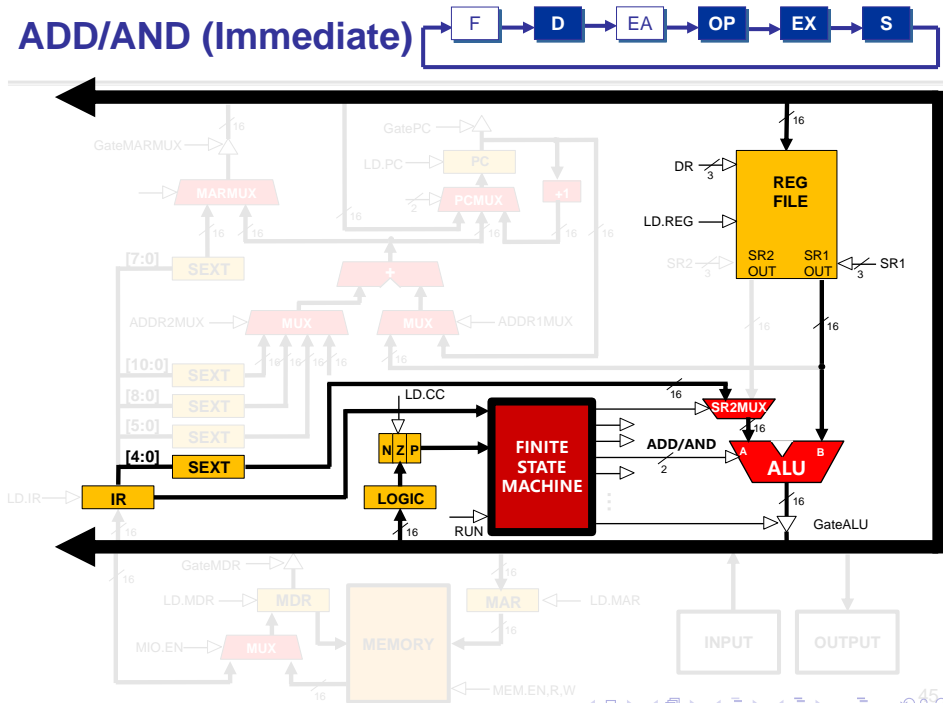
ADD/AND (Immediate)



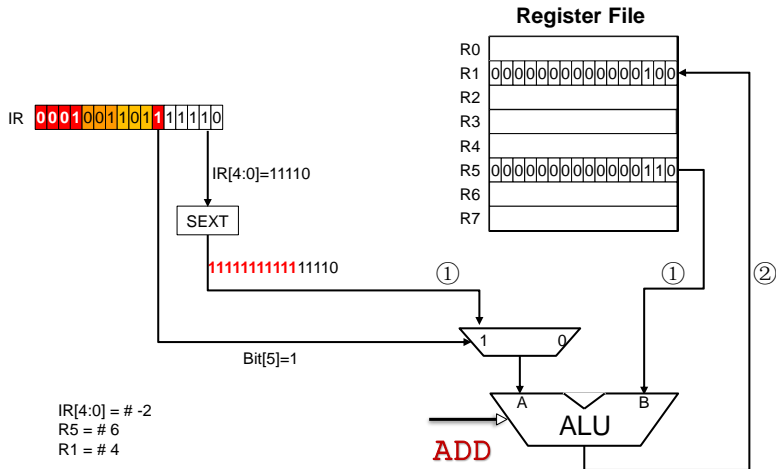
ADD/AND (Immediate)



ADD/AND (Immediate)



ADD (Immediate) ADD R1, R5, #-2



Using Operate Instructions

With only ADD, AND, NOT...

- How do we subtract?
- How do we OR?
- How do we copy from one register to another?
- How do we initialize a register to zero?



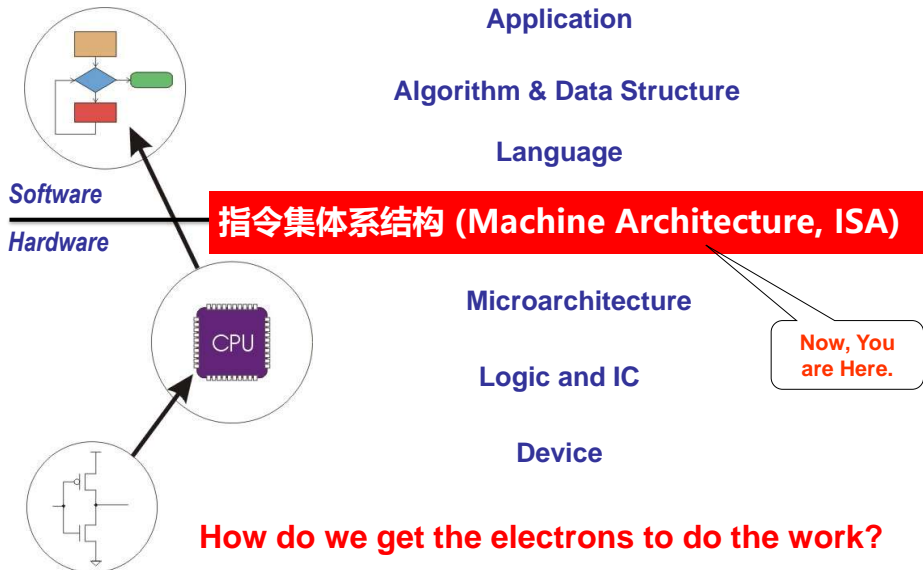
1 Review

2 LC-3 ISA Overview

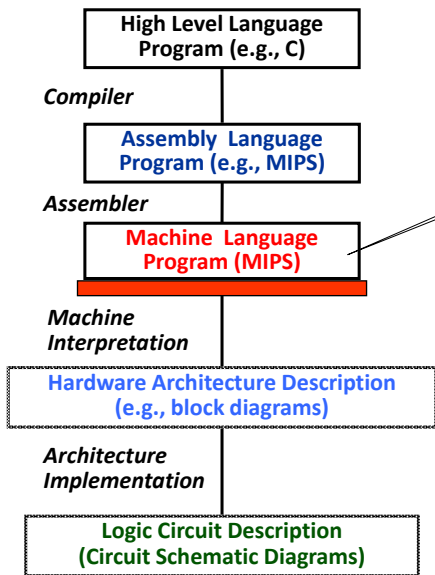
3 LC-3 Operate Instructions and Data Path

4 Summary

Great Idea #4: Software and Hardware Co-design



How do we get the electrons to do the work?



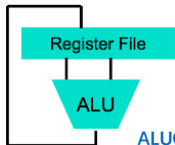
```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

Now, You
are Here.

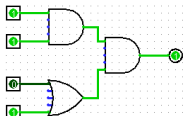
```
lw $t0, 0($2)  
lw $t1, 4($2)  
sw $t1, 0($2)  
sw $t0, 4($2)
```

Anything can be represented
as a *number*,
i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```



ALUOP[0:3] <= InstReg[9:11] & MASK



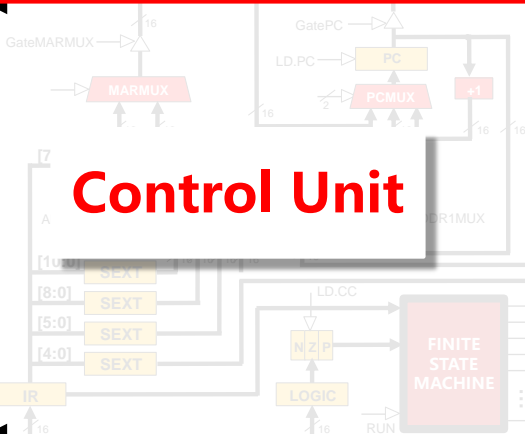
Operate Instructions

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|----|----|----|----|----|----|---|-----|---|---|---|------|---|-----|---|---|
| ADD | 0 | 0 | 0 | 1 | DR | | | SR1 | | | 0 | 0 | 0 | SR2 | | |
| ADD | 0 | 0 | 0 | 1 | DR | | | SR1 | | | 1 | Imm5 | | | | |
| AND | 0 | 1 | 0 | 1 | DR | | | SR1 | | | 0 | 0 | 0 | SR2 | | |
| AND | 0 | 1 | 0 | 1 | DR | | | SR1 | | | 1 | Imm5 | | | | |
| NOT | 1 | 0 | 0 | 1 | DR | | | SR1 | | | 1 | 1 | 1 | 1 | 1 | 1 |
| Reserved | 1 | 1 | 0 | 1 | | | | | | | | | | | | |

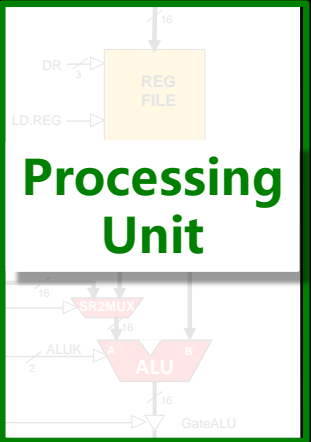
LC-3 Data Path



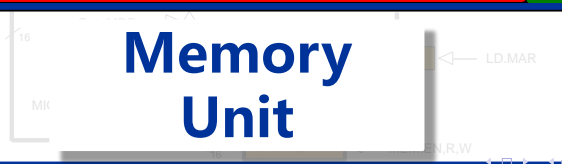
Control Unit



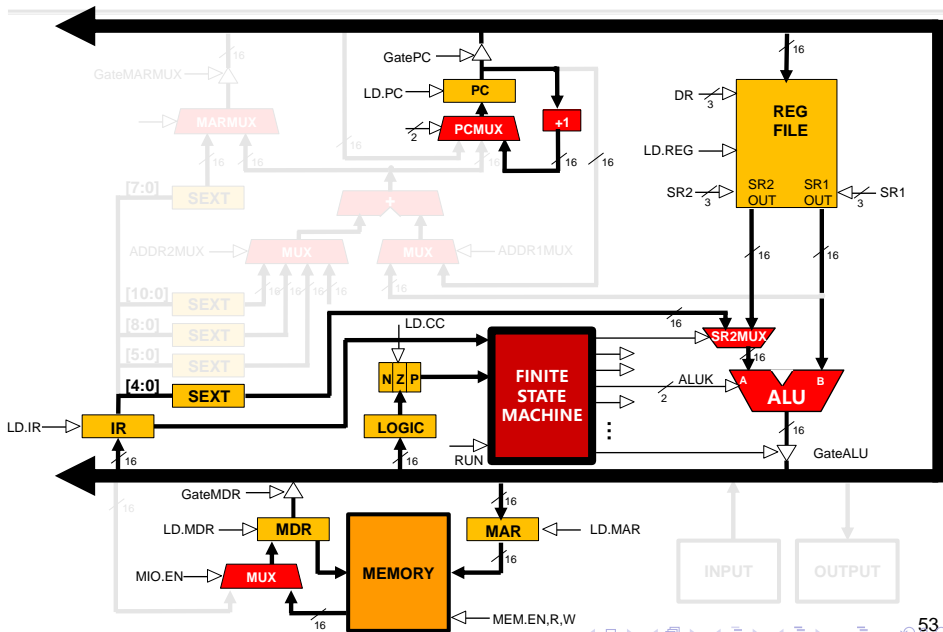
Processing Unit



Memory Unit



LC-3 Data Path After Operate Instruction



Next Lecture: Data Movement Instructions

取数指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|-----------|---|---|---|---|---|---|
| LD | 0 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LDR | 0 | 1 | 1 | 0 | DR | | | BaseR | | PCOffset6 | | | | | | |
| LDI | 1 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LEA | 1 | 1 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |

存数指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|-----------|---|---|---|---|---|---|
| ST | 0 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |
| STR | 0 | 1 | 1 | 1 | SR | | | BaseR | | PCOffset6 | | | | | | |
| STI | 1 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |

■ 5.3 Data Movement Instructions



中国科学技术大学
University of Science and Technology of China

计算机系统概论

Introduction to Computing Systems
(CS1002A.02)

Chapter 5-2

The LC-3 Data Movement Instructions

计算机科学与技术学院
School of Computer Science and Technology



1 Review

2 LC-3 PC-Relative Load/Store

3 LC-3 Indirect, Base+offset Load/Store

4 Summary



1 Review

2 LC-3 PC-Relative Load/Store

3 LC-3 Indirect, Base+offset Load/Store

4 Summary

Great Idea #3 : Abstraction

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

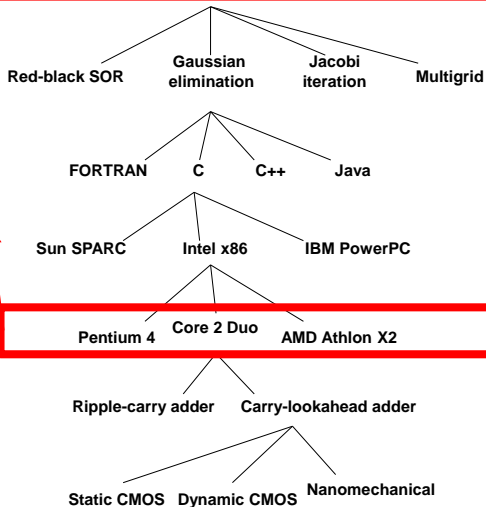
Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

Electronic Devices

Physics

Solve a system of equations



LC-3 ISA Operate Instructions

运算指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|----|----|----|----|----|----|-----|---|---|------|---|-----|---|---|---|---|
| ADD | 0 | 0 | 0 | 1 | DR | | SR1 | | 0 | 0 | 0 | SR2 | | | | |
| ADD | 0 | 0 | 0 | 1 | DR | | SR1 | | 1 | Imm5 | | | | | | |
| AND | 0 | 1 | 0 | 1 | DR | | SR1 | | 0 | 0 | 0 | SR2 | | | | |
| AND | 0 | 1 | 0 | 1 | DR | | SR1 | | 1 | Imm5 | | | | | | |
| NOT | 1 | 0 | 0 | 1 | DR | | SR1 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Reserved | 1 | 1 | 0 | 1 | | | | | | | | | | | | |

控制指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|------------|---|-----------|-------------|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | n | z | p | PCOffset9 | | | | | | | | |
| JSR | 0 | 1 | 0 | 0 | 1 | PCOffset11 | | | | | | | | | | |
| JSRR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RET | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| TRAP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | TrapVector8 | | | | | | | |

移动数据指令

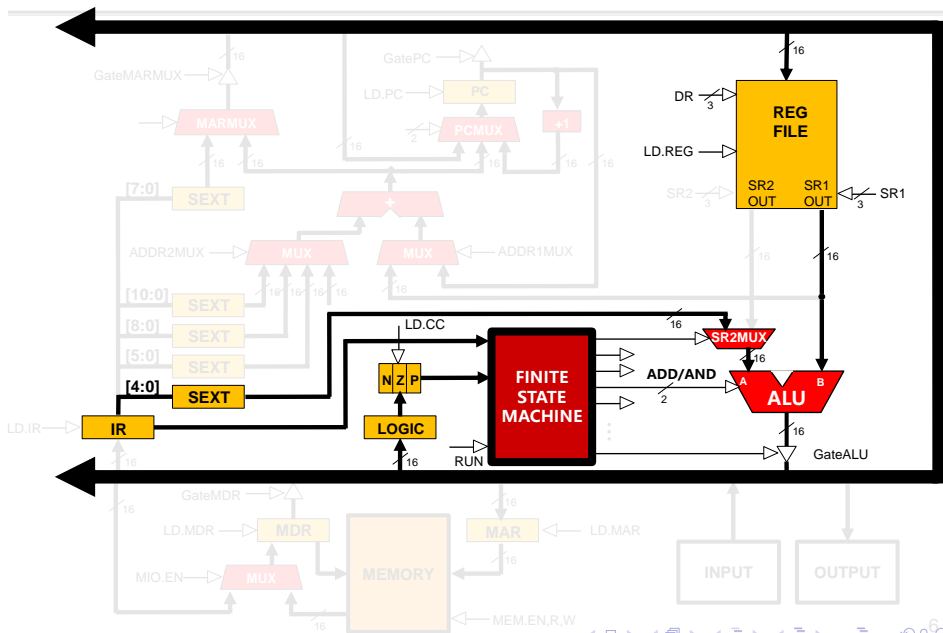
取数指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|-----------|---|---|---|---|---|---|
| LD | 0 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LDR | 0 | 1 | 1 | 0 | DR | | | BaseR | | PCOffset6 | | | | | | |
| LDI | 1 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LEA | 1 | 1 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |

存数指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|---|-----------|---|---|---|---|---|
| ST | 0 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |
| STR | 0 | 1 | 1 | 1 | SR | | | BaseR | | | PCOffset6 | | | | | |
| STI | 1 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |

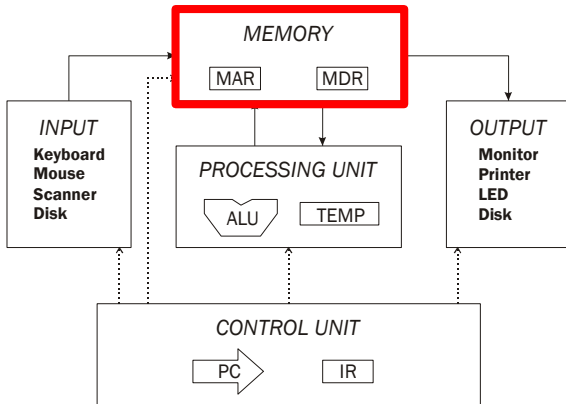
LC-3 Data Path After Operate Instruction





■ We are going to learn how to:

- load data from memory to registers
- store data from registers to memory



Outline

1 Review

2 **LC-3 PC-Relative Load/Store**

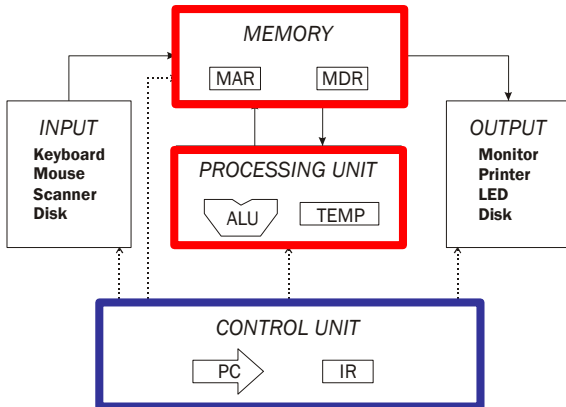
3 LC-3 Indirect, Base+offset Load/Store

4 Summary

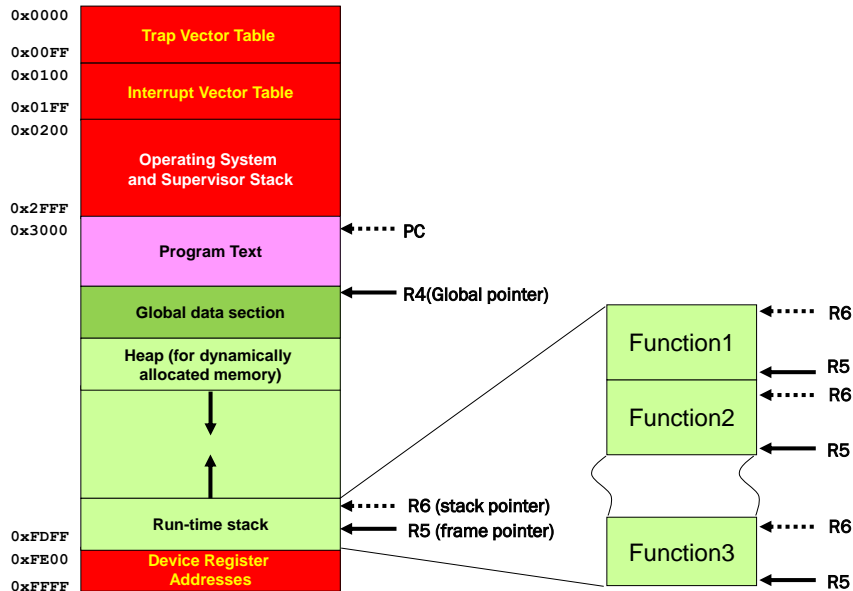


■ We are going to learn how to:

- compute with values in registers
- **load data from memory to registers**
- **store data from registers to memory**



LC-3 Overview: Memory Map



Data Movement Instructions

取数指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|-----------|---|---|---|---|---|---|
| LD | 0 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LDR | 0 | 1 | 1 | 0 | DR | | | BaseR | | PCOffset6 | | | | | | |
| LDI | 1 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LEA | 1 | 1 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |

存数指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|-----------|---|---|---|---|---|---|
| ST | 0 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |
| STR | 0 | 1 | 1 | 1 | SR | | | BaseR | | PCOffset6 | | | | | | |
| STI | 1 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |

Data Movement Instructions

Load -- read data **from memory to register**

- **LD:** PC-relative mode
- **LDR:** base+offset mode
- **LDI:** indirect mode

Store -- write data **from register to memory**

- **ST:** PC-relative mode
- **STR:** base+offset mode
- **STI:** indirect mode

Load effective address -- compute address, save in register

- **LEA:** immediate mode
- *does not access memory*

PC-Relative Addressing Mode

Want to specify address directly in the instruction

- But an address is 16 bits, and so is an instruction!
- After subtracting 4 bits for opcode and 3 bits for register, we have 9 bits available for address.

Solution:

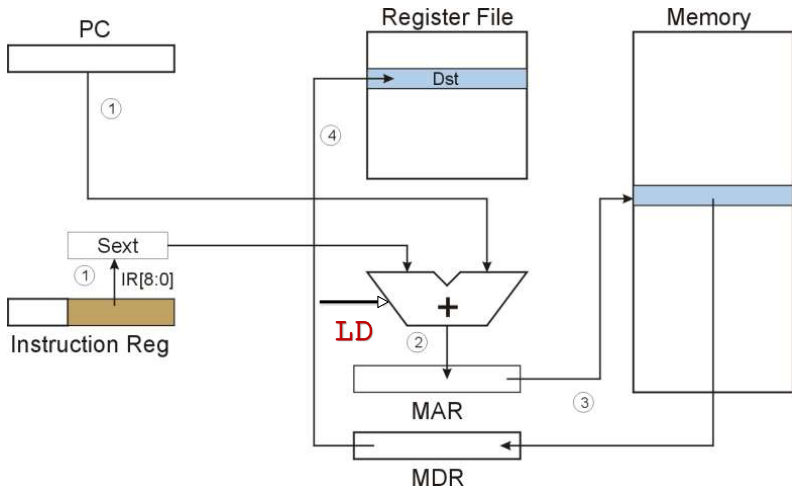
- Use the 9 bits as a signed offset from the current PC.

9 bits: $-256 \leq \text{offset} \leq +255$

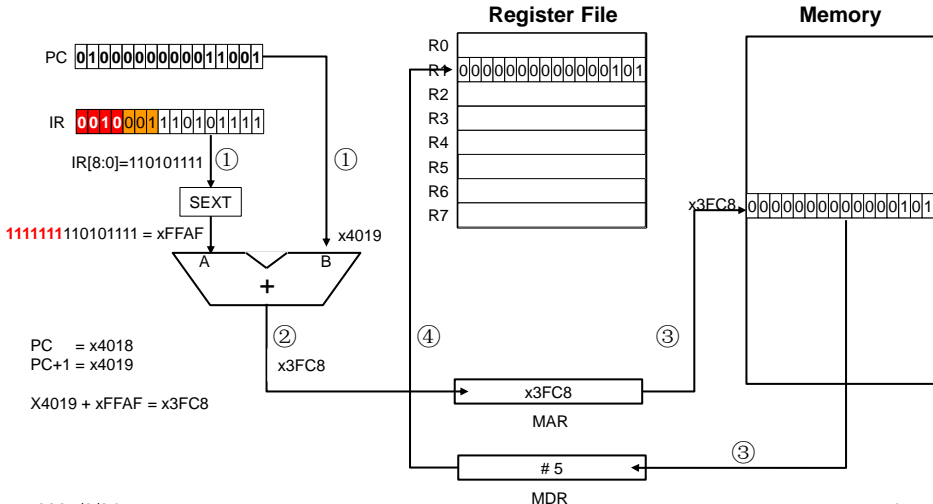
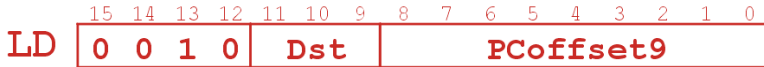
Can form any address X, such that: $\text{PC} - 256 \leq X \leq \text{PC} + 255$

**Remember that PC is incremented as part of the FETCH phase;
This is done before the EVALUATE ADDRESS stage.**

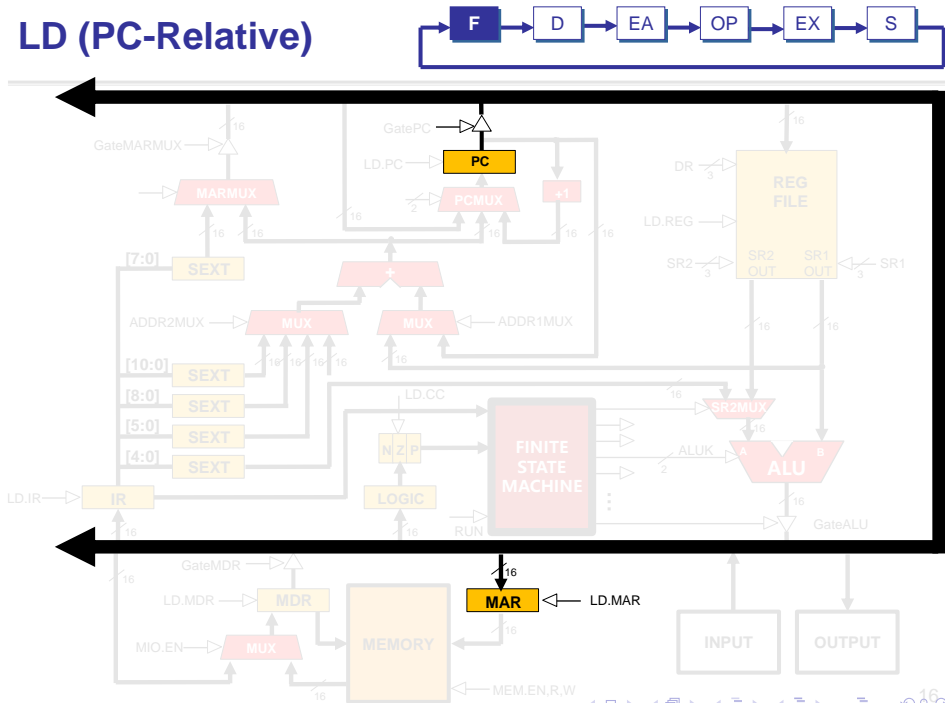
LD (PC-Relative) LD DR, PCoffset9



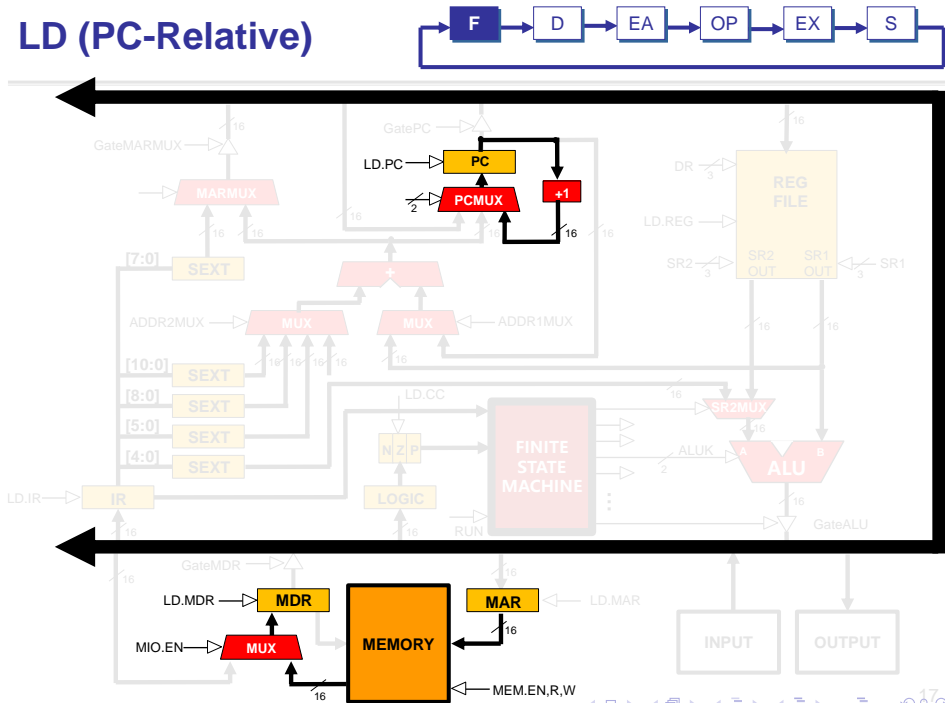
LD (PC-Relative) : LD R1, x1AF



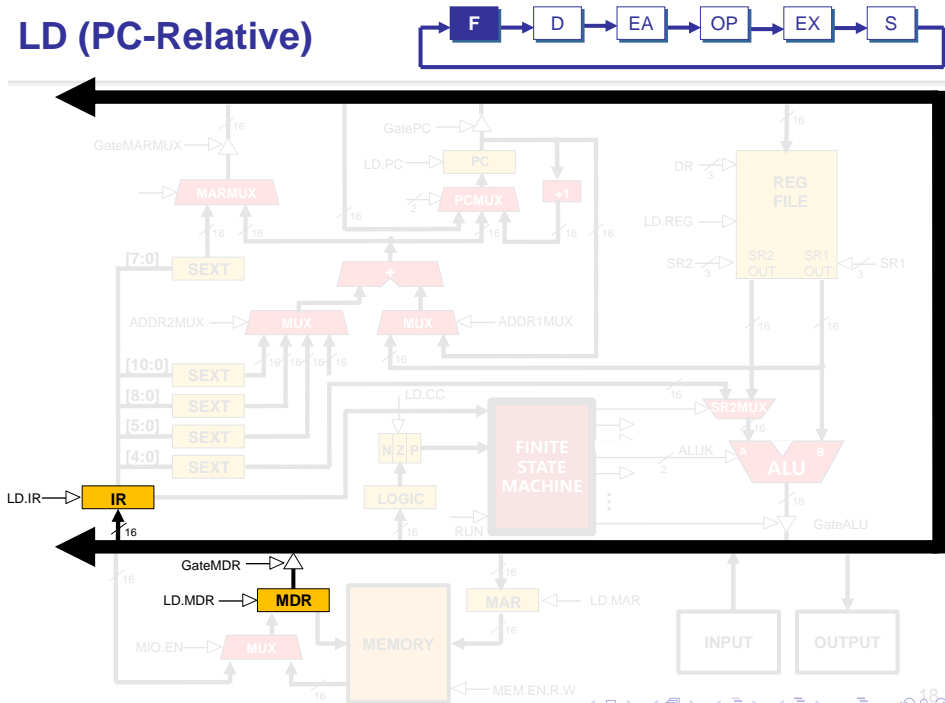
LD (PC-Relative)



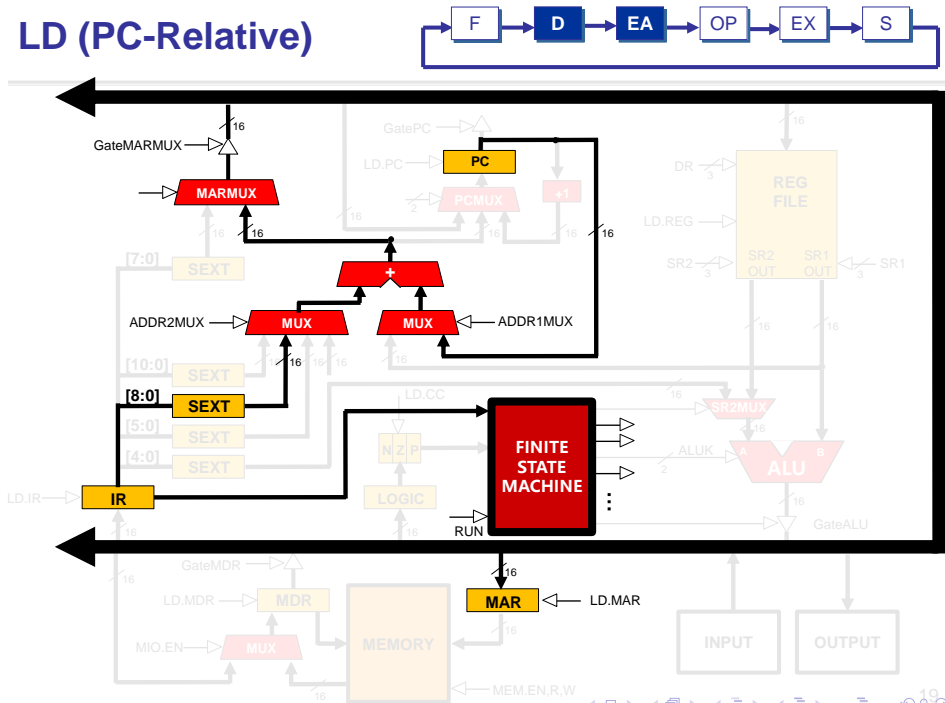
LD (PC-Relative)



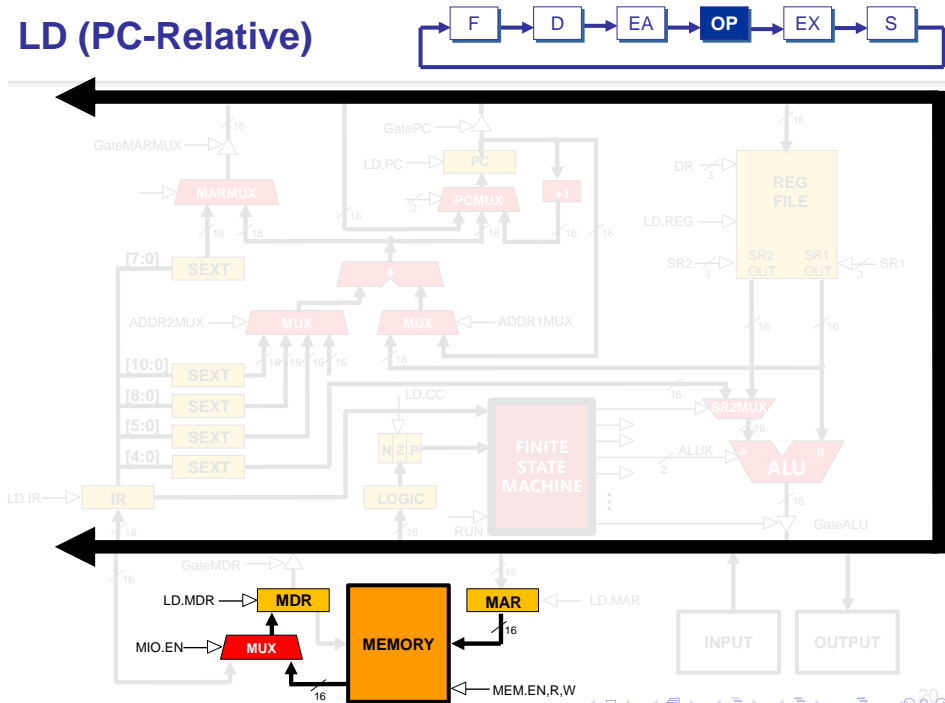
LD (PC-Relative)



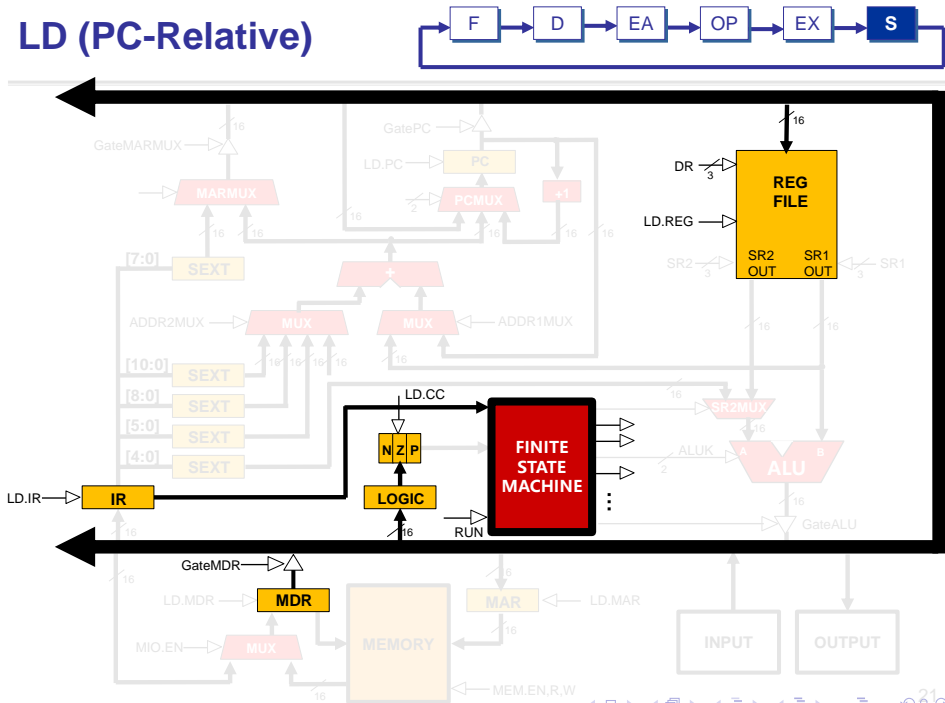
LD (PC-Relative)



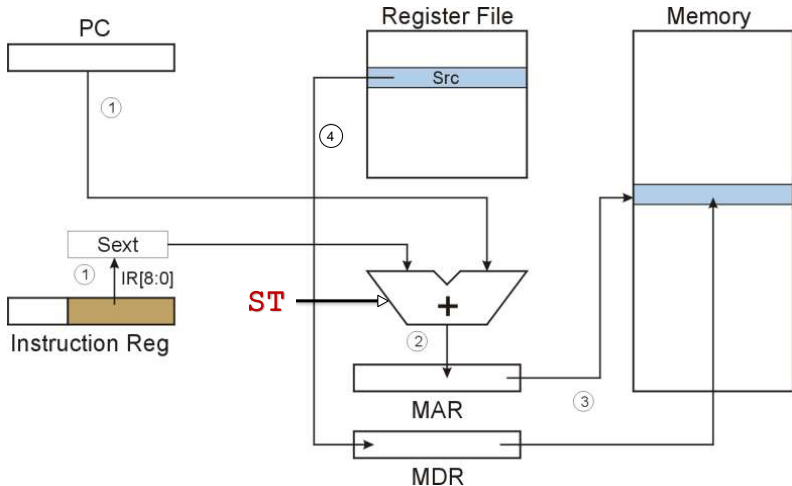
LD (PC-Relative)



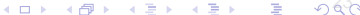
LD (PC-Relative)



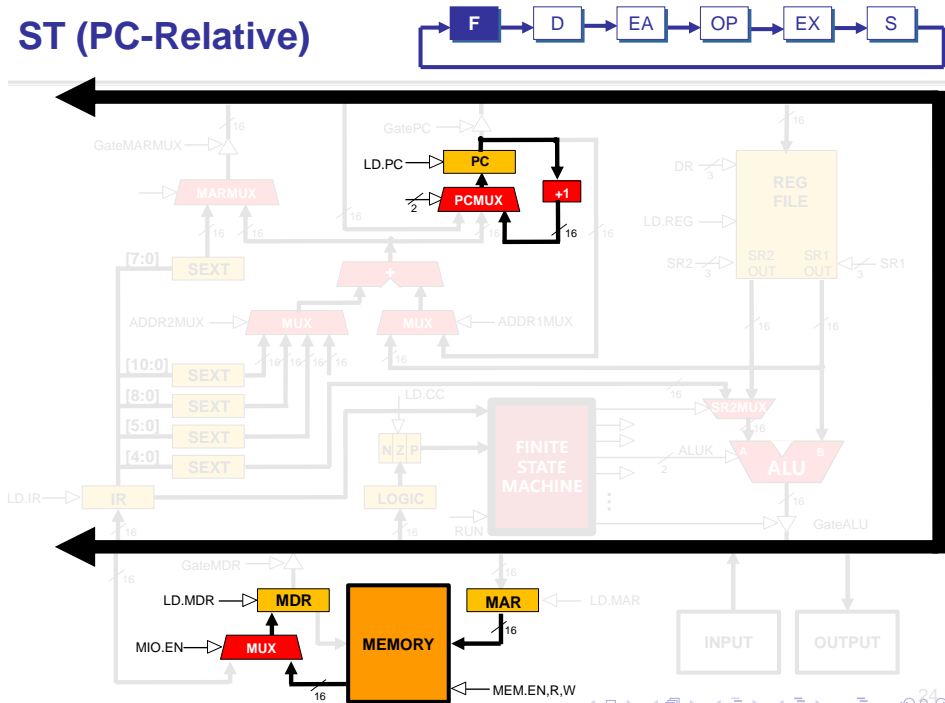
ST (PC-Relative) ST SR, PCoffset9



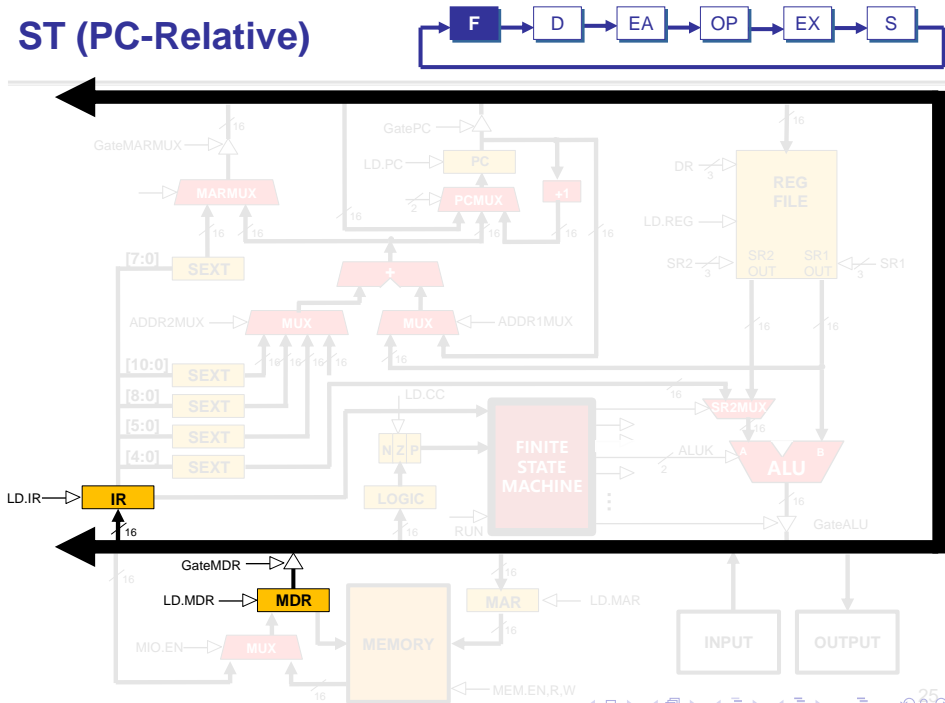
```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



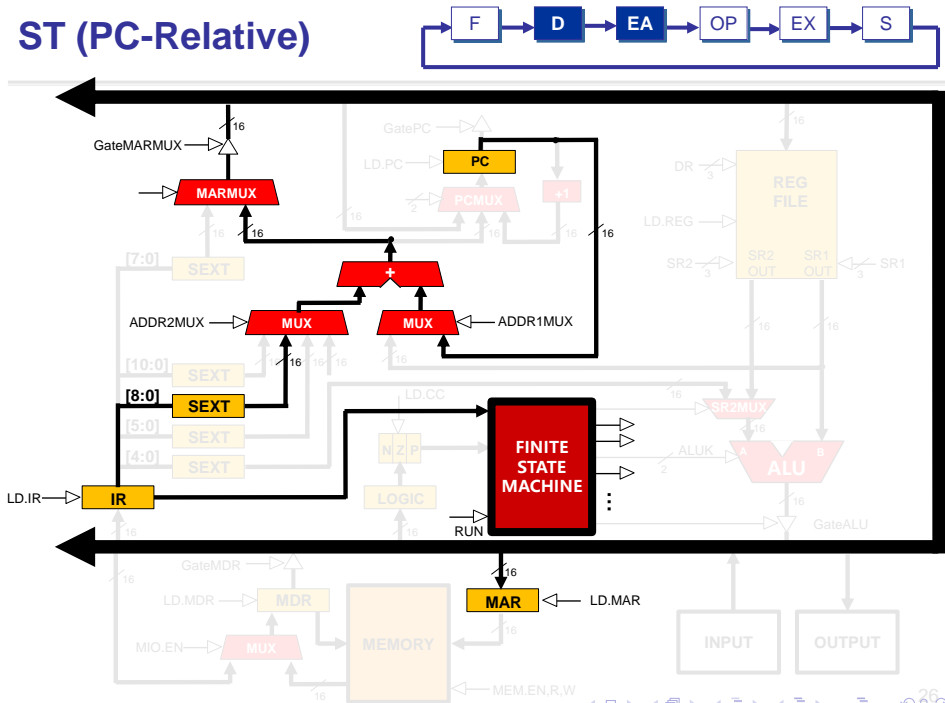
ST (PC-Relative)



ST (PC-Relative)



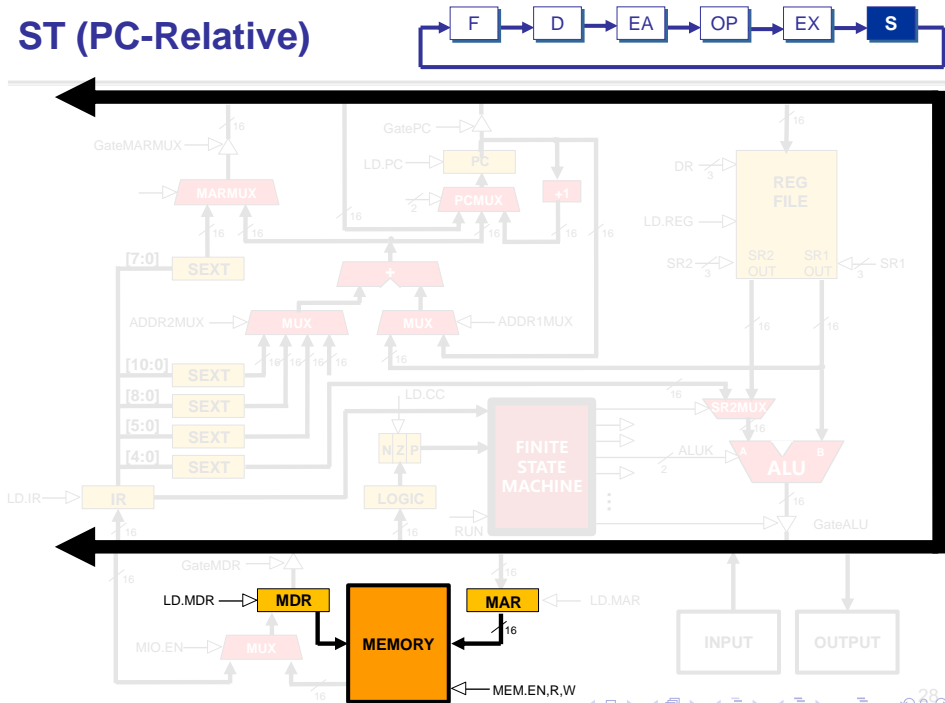
ST (PC-Relative)



```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



ST (PC-Relative)





1 Review

2 LC-3 PC-Relative Load/Store

3 LC-3 Indirect, Base+offset Load/Store

4 Summary

Indirect Addressing Mode

With PC-relative mode, can only address data within 256 words of the instruction.

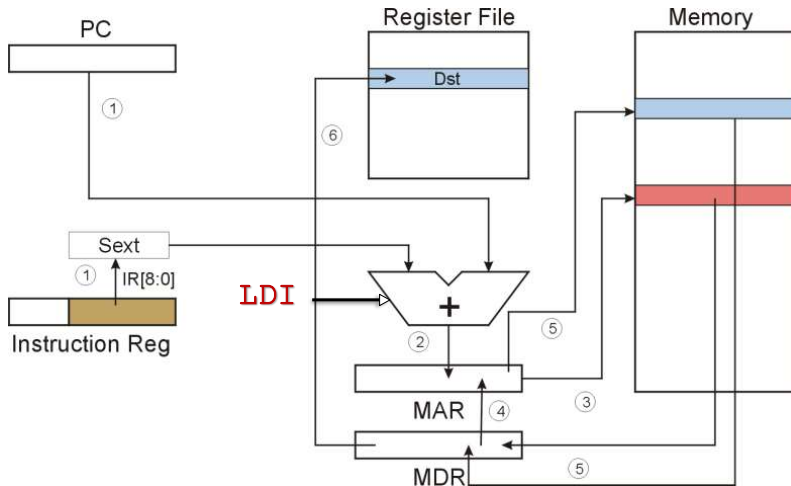
- What about the rest of memory?

Solution #1:

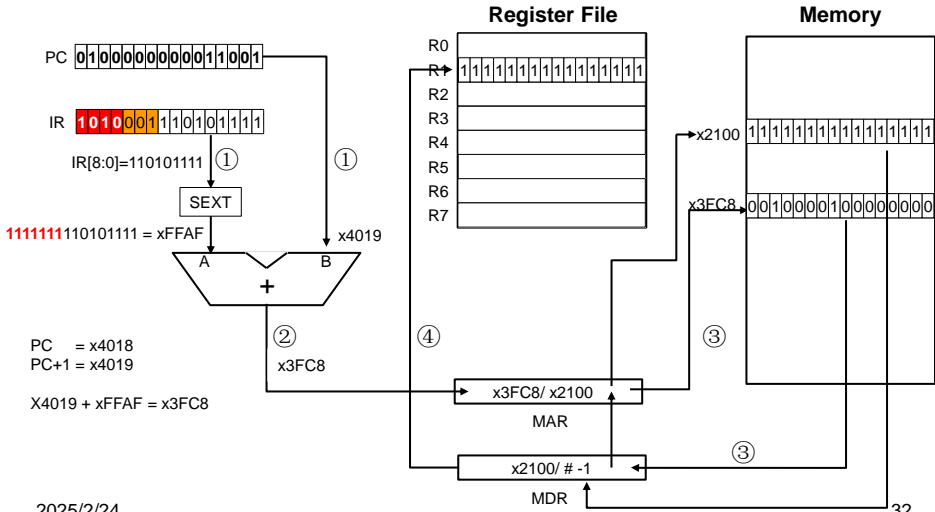
- Read address from memory location, then load/store to that address.

First address is generated from PC and IR (just like PC-relative addressing), then content of that address is used as target for load/store.

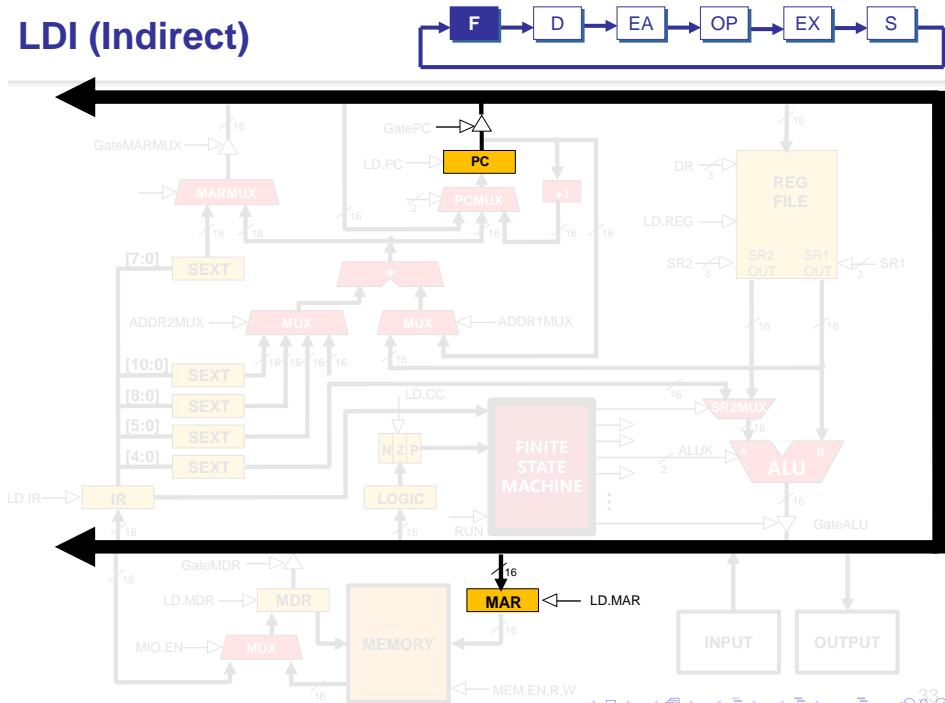
LDI (Indirect) LDI DR, PCoffset9



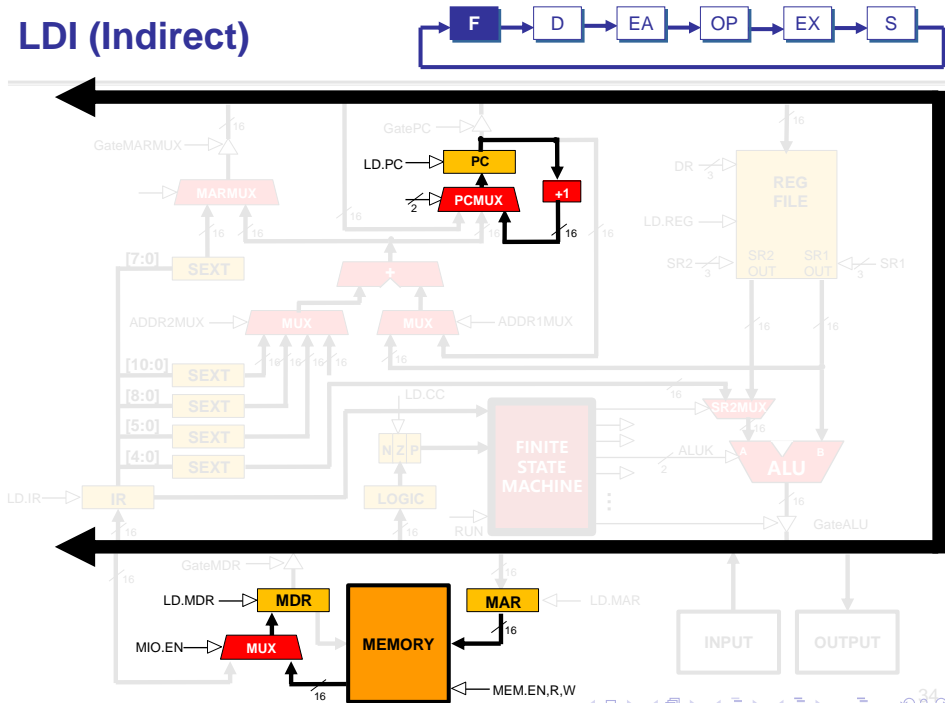
LDI (Indirect) : LDI R1, x1AF



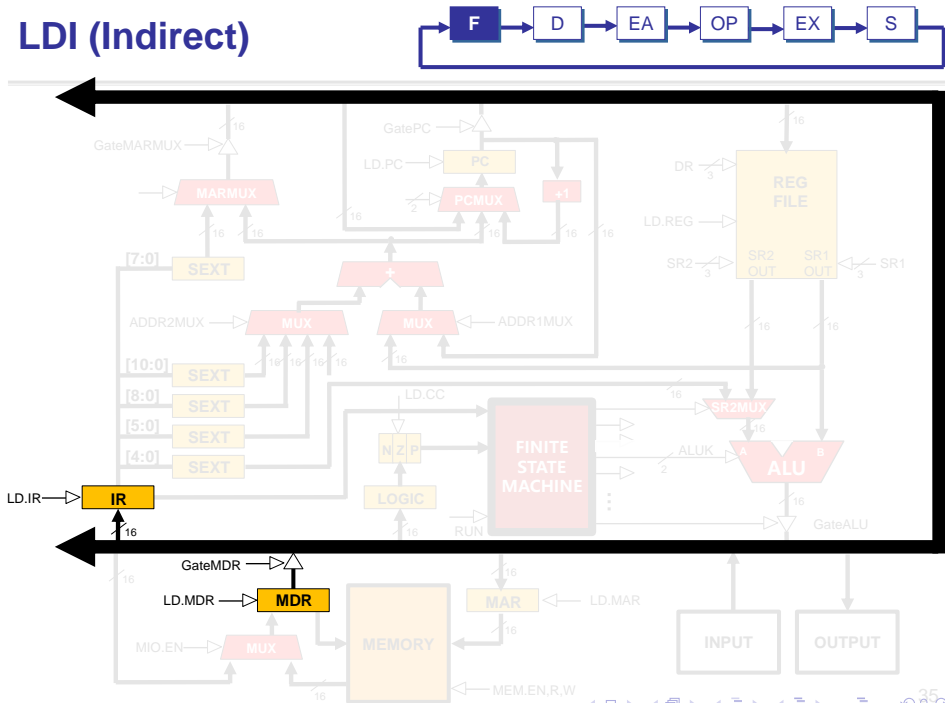
LDI (Indirect)



LDI (Indirect)



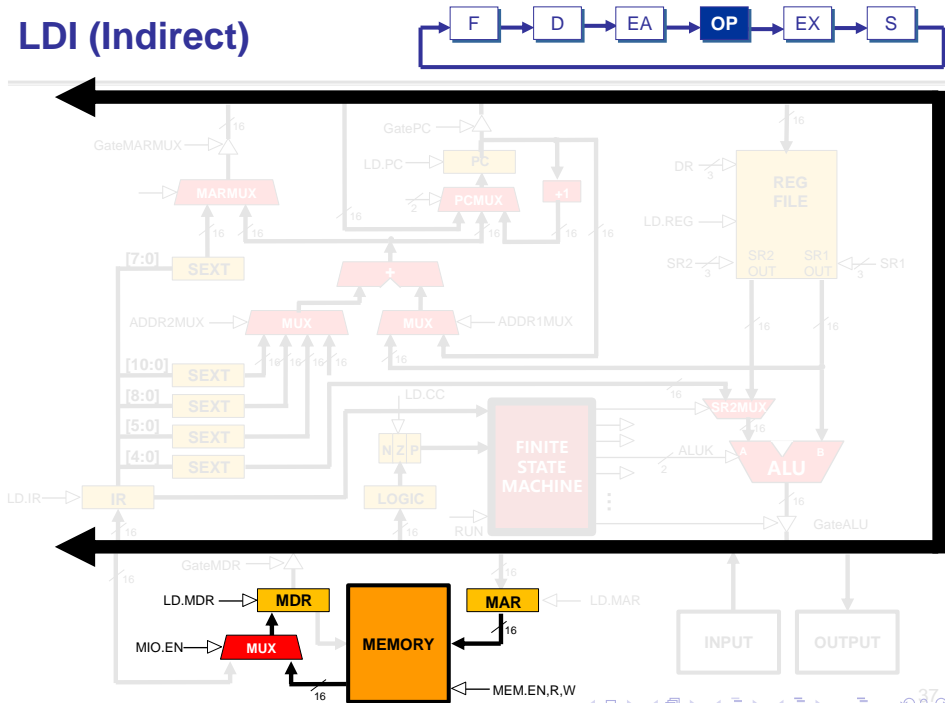
LDI (Indirect)



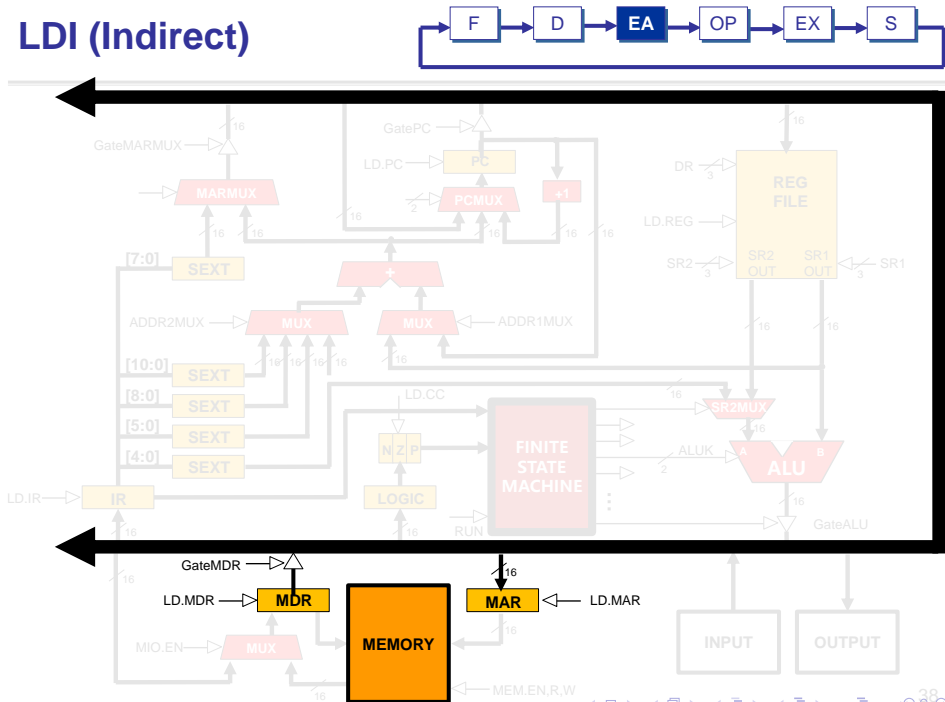
```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



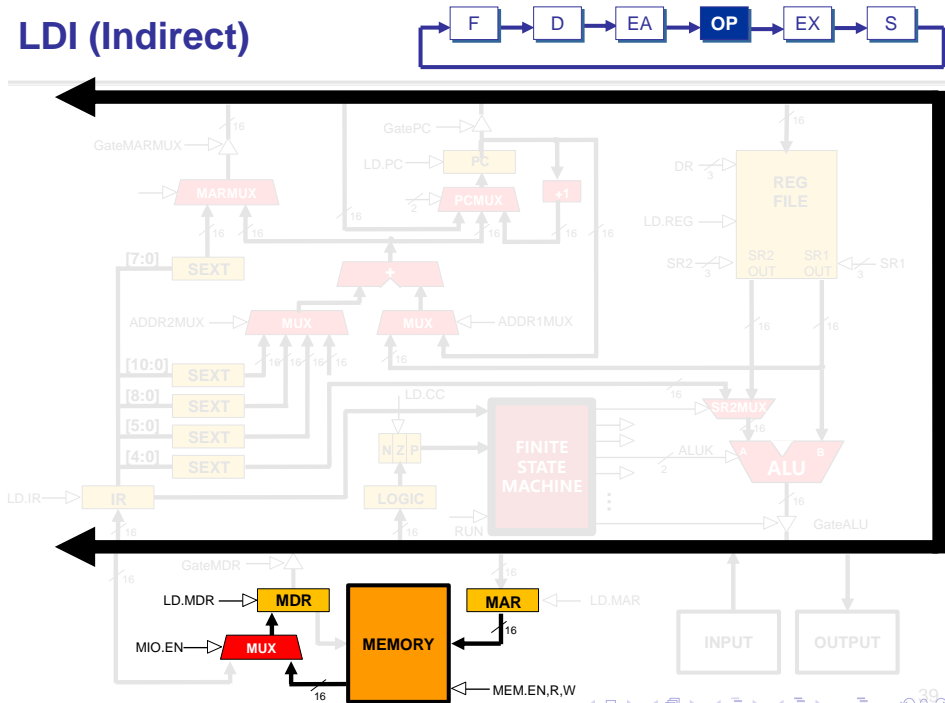
LDI (Indirect)



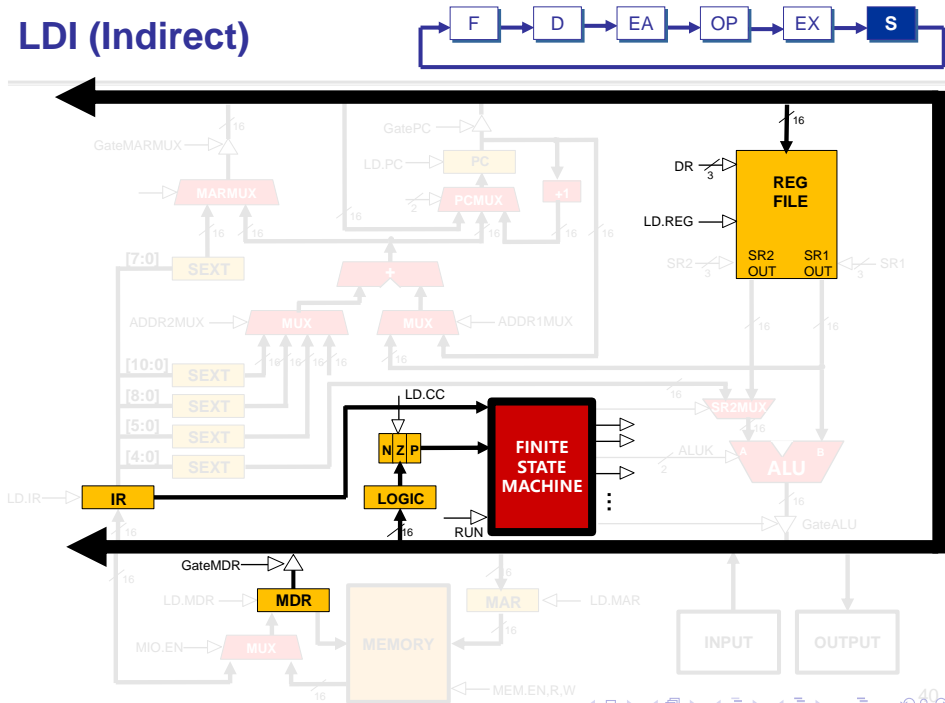
LDI (Indirect)



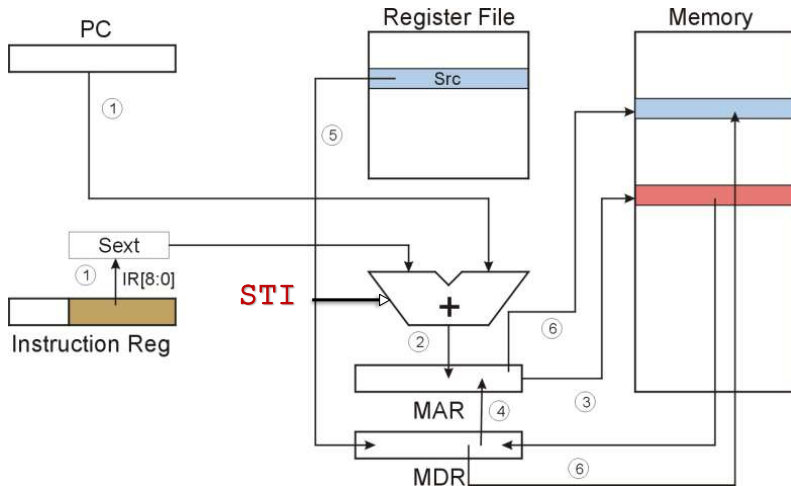
LDI (Indirect)



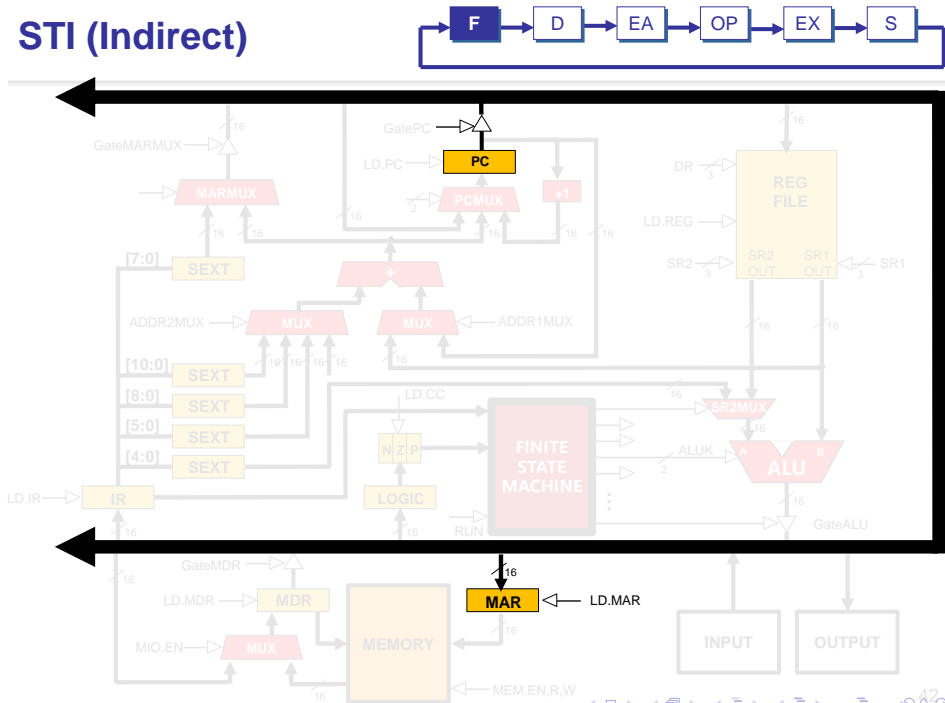
LDI (Indirect)



STI (Indirect) STI SR, PCOffset9



STI (Indirect)



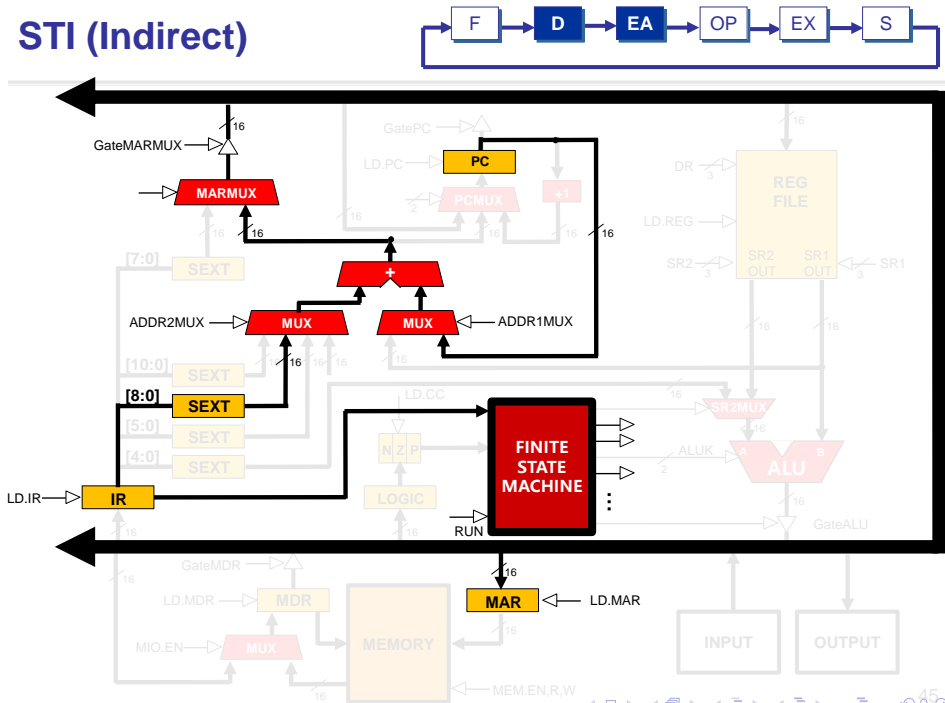
```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



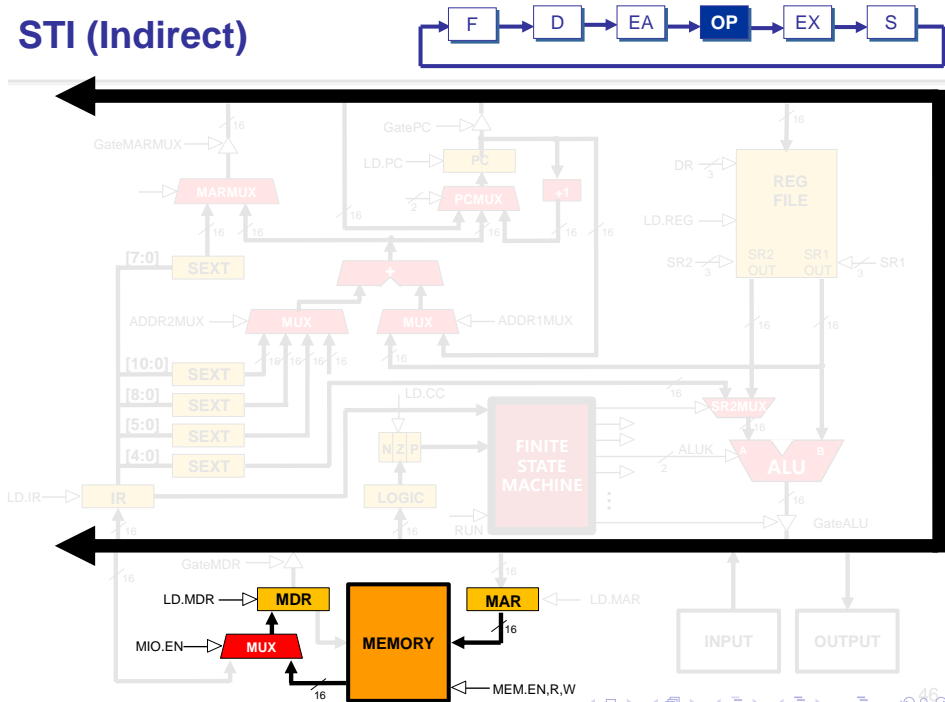
```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



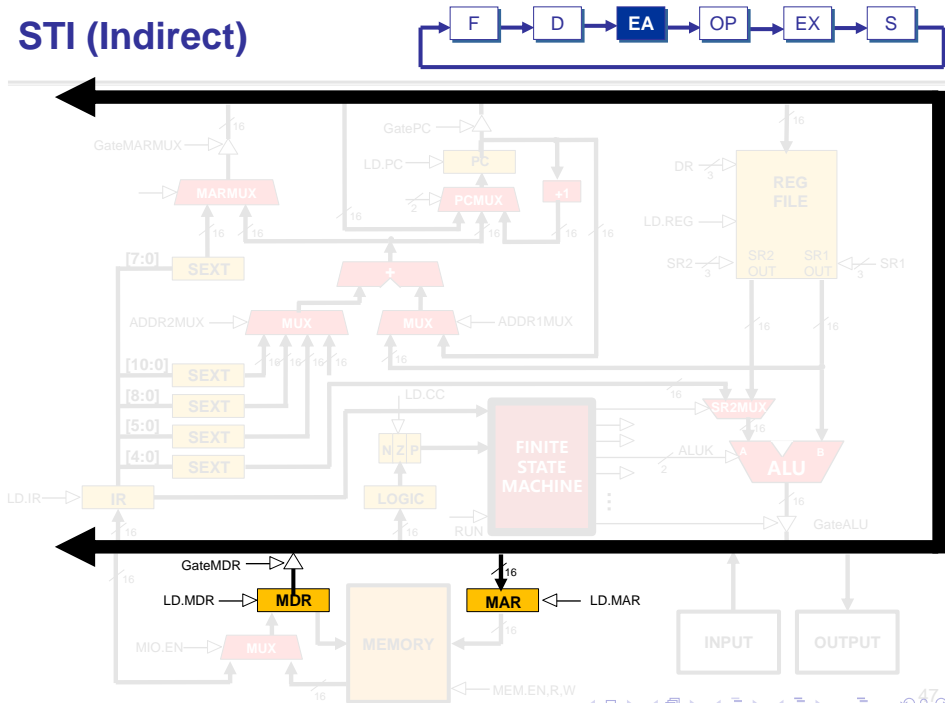
STI (Indirect)



STI (Indirect)



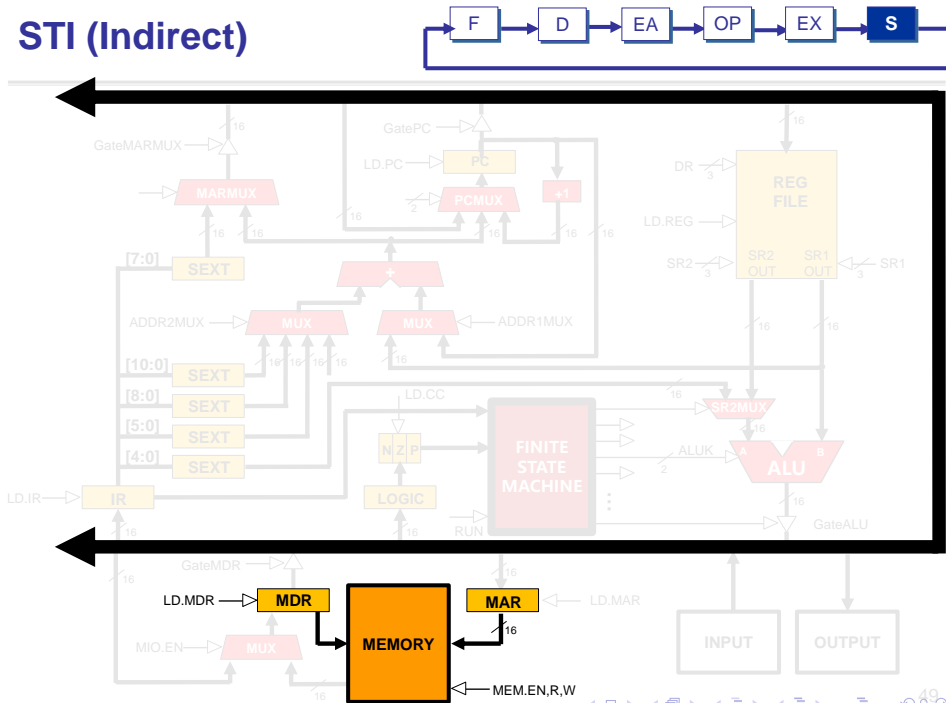
STI (Indirect)




```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP1[OP]; OP1 --> EX[EX]; EX --> OP2[OP]; OP2 --> Out[ ]; Out --> F;
```



STI (Indirect)



Base + Offset Addressing Mode

With PC-relative mode, can only address data within 256 words of the instruction.

- What about the rest of memory?

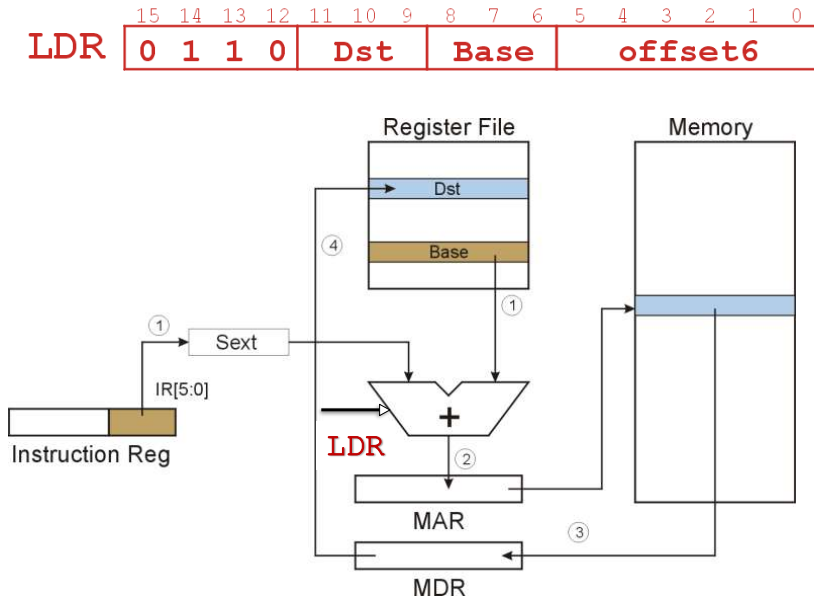
Solution #2:

- Use a register to generate a full 16-bit address.

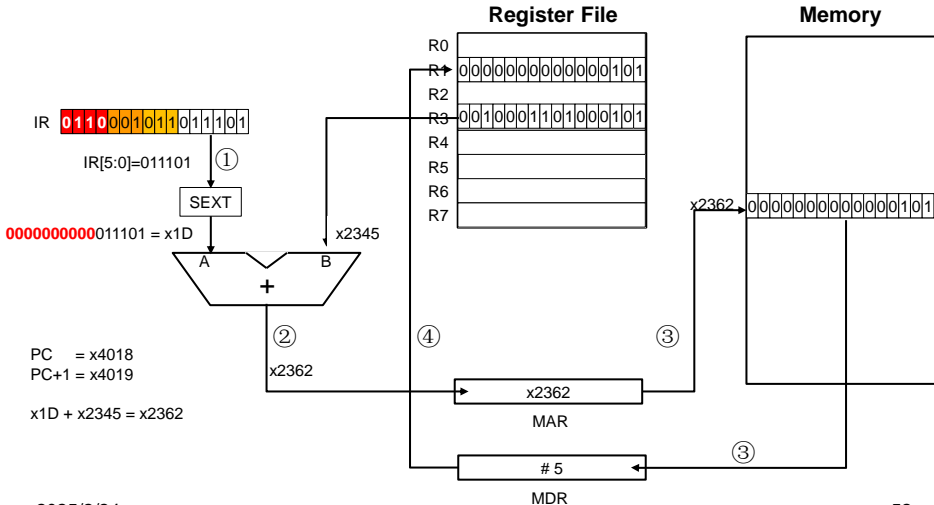
4 bits for opcode, 3 for src/dest register,
3 bits for *base* register -- remaining 6 bits are used
as a *signed offset*.

- Offset is *sign-extended* before adding to base register.

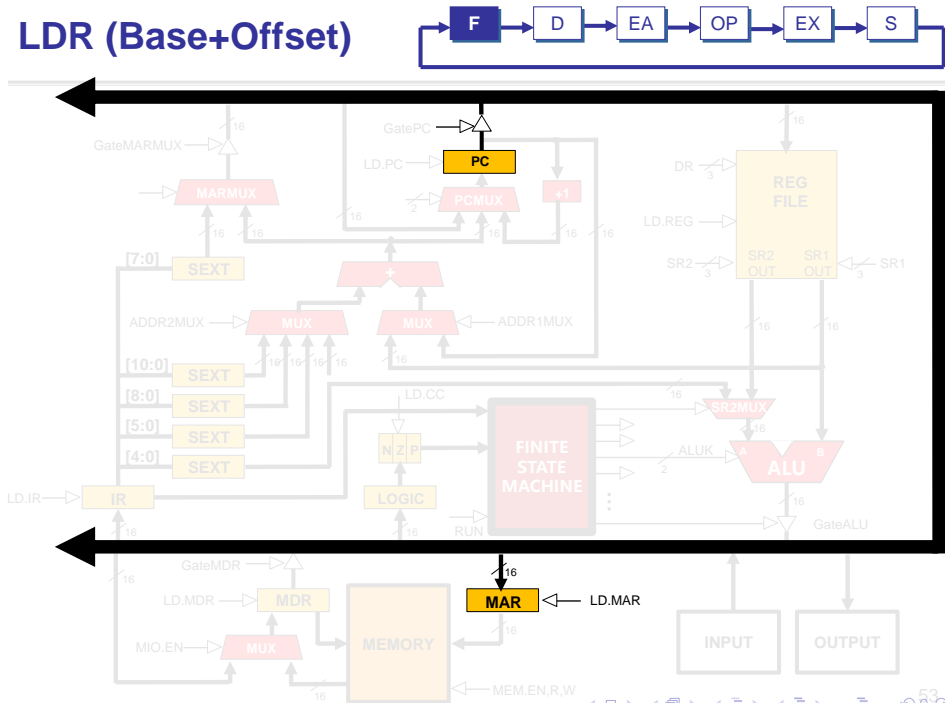
LDR (Base+Offset) LDR DR, BaseR, offset6



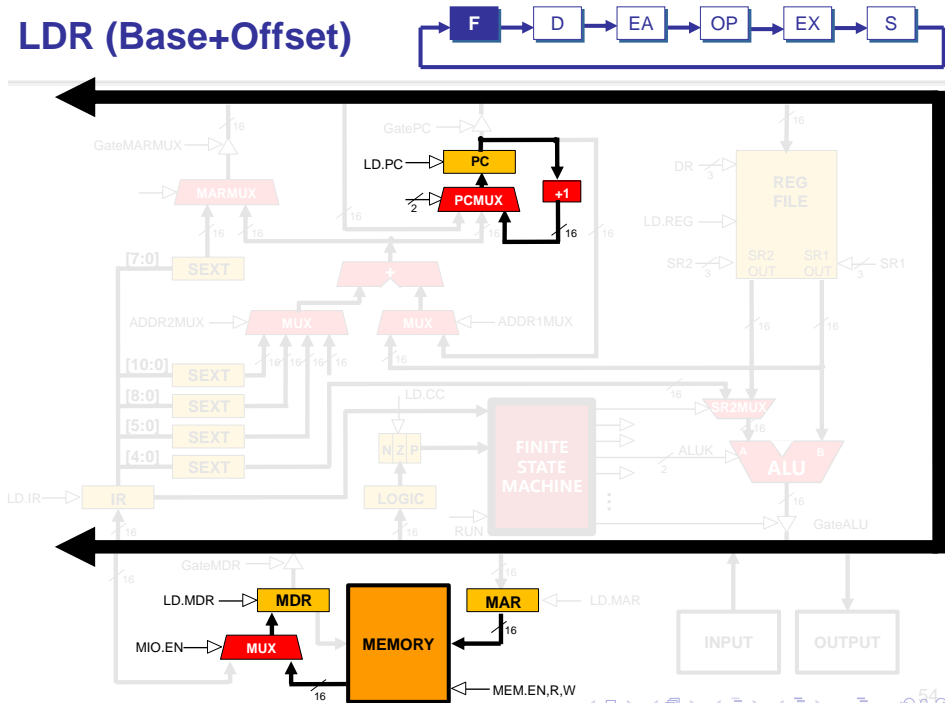
LDR (Base+Offset) : LD R1, R3, x1D



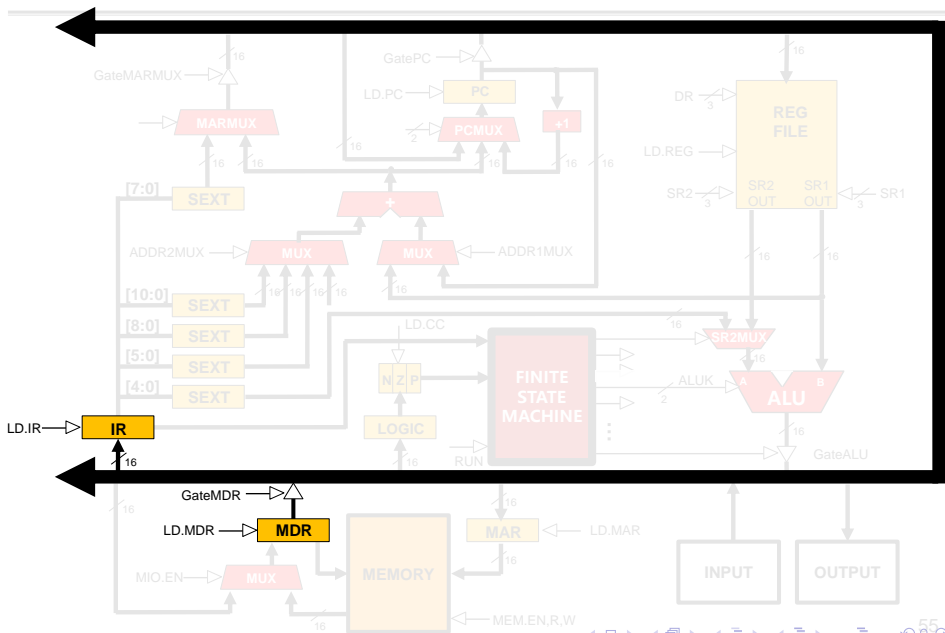
LDR (Base+Offset)



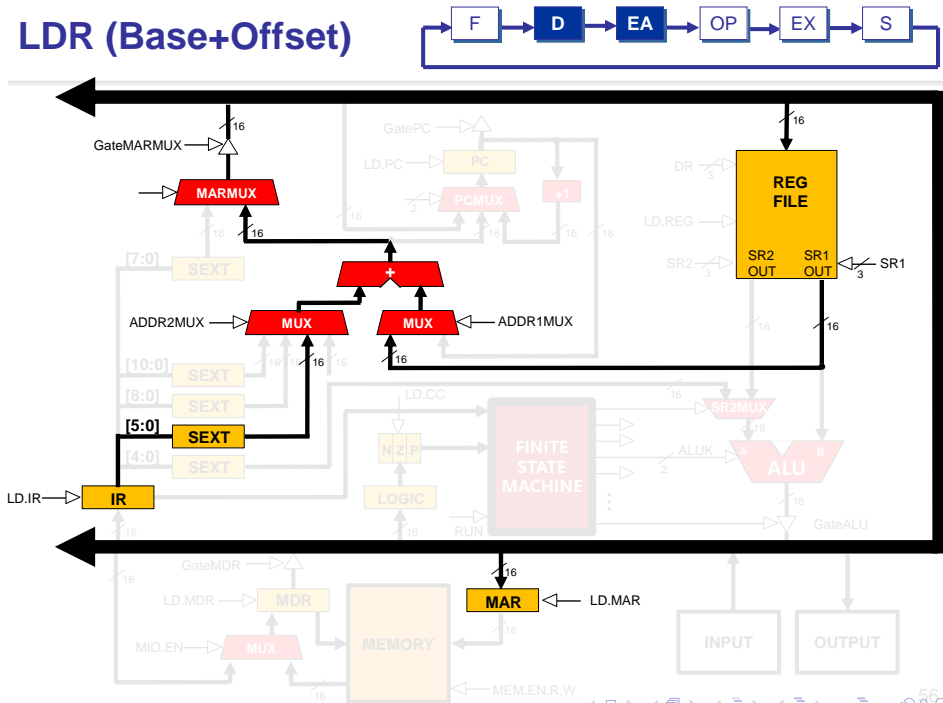
LDR (Base+Offset)



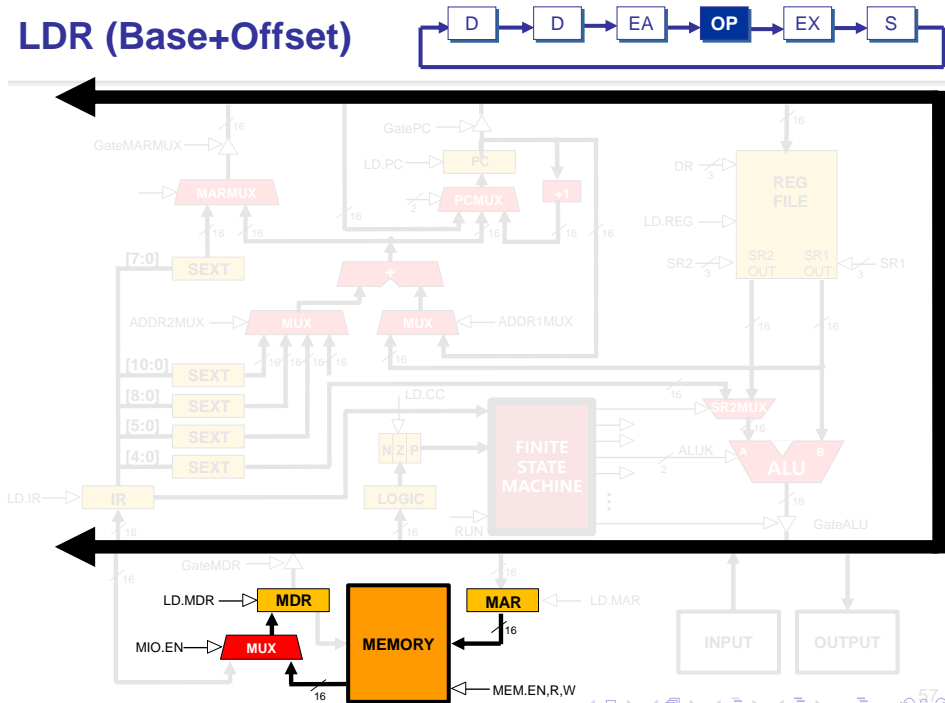
LDR (Base+Offset)



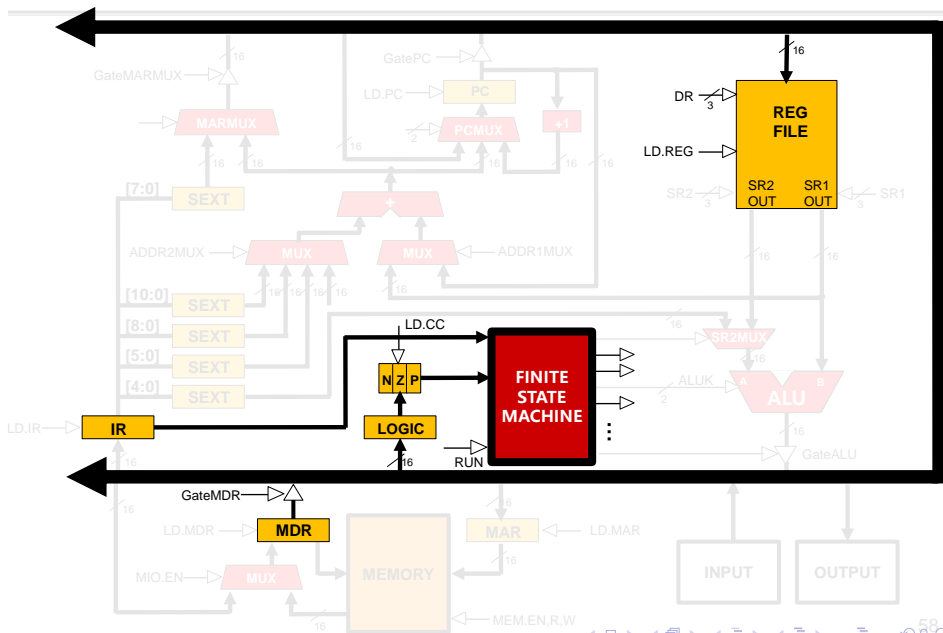
LDR (Base+Offset)



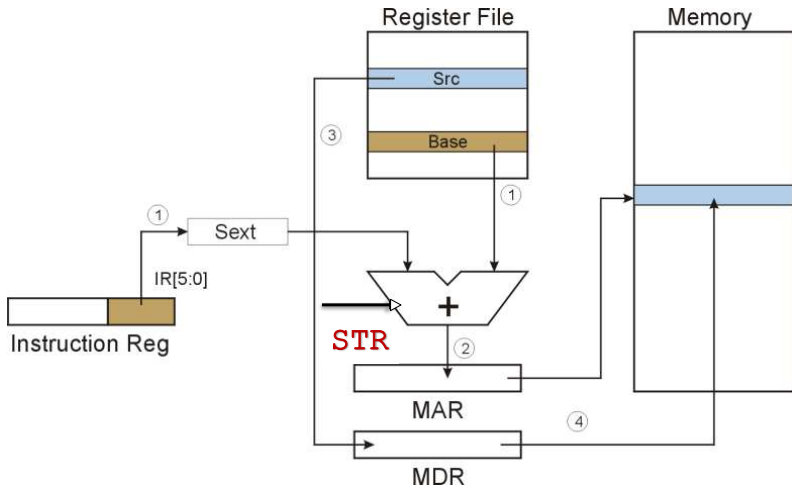
LDR (Base+Offset)



LDR (Base+Offset)



STR (Base+Offset) STR SR, BaseR, offset6



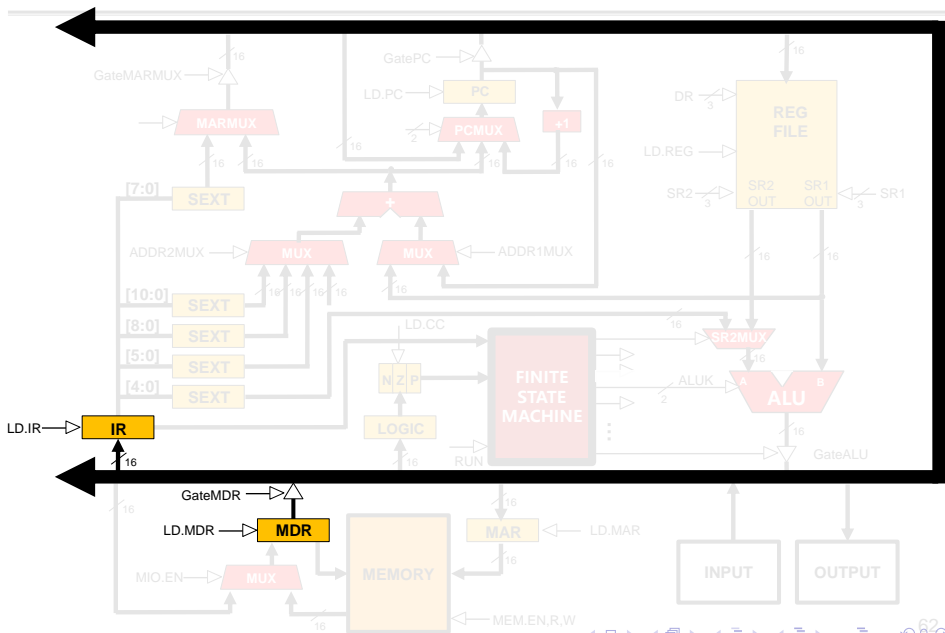
```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



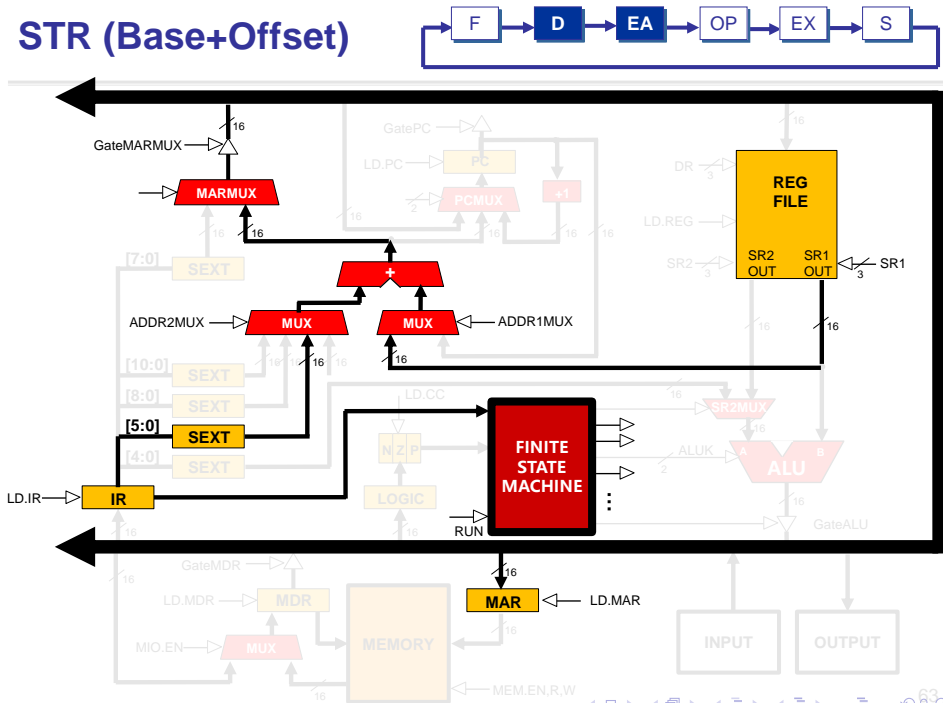
```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



STR (Base+Offset)



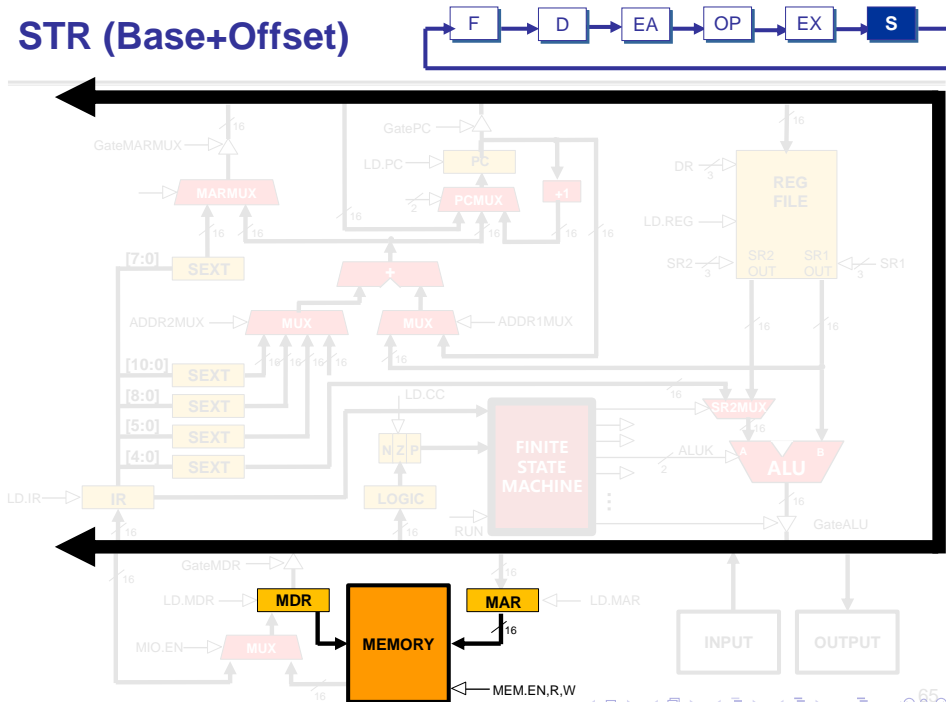
STR (Base+Offset)




```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP1[OP]; OP1 --> EX[EX]; EX --> OP2[OP]; OP2 --> F;
```



STR (Base+Offset)

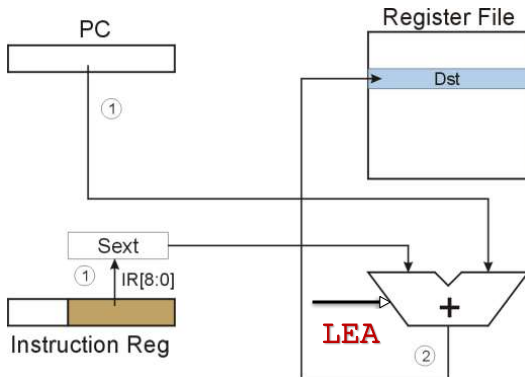


Load Effective Address

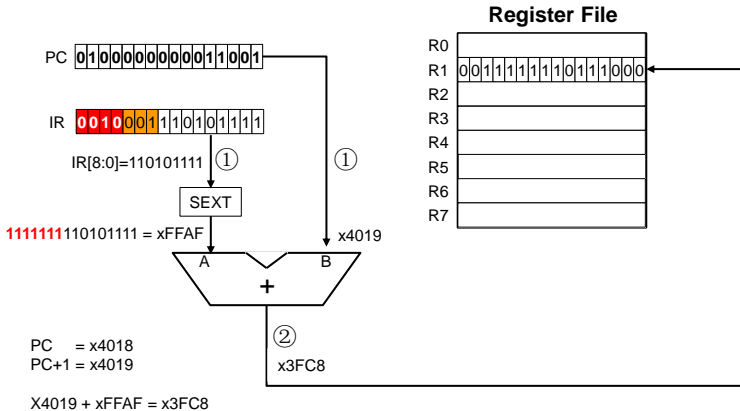
Computes address like PC-relative (PC plus signed offset) and **stores the result into a register**.

Note: The address is stored in the register, not the contents of the memory location.

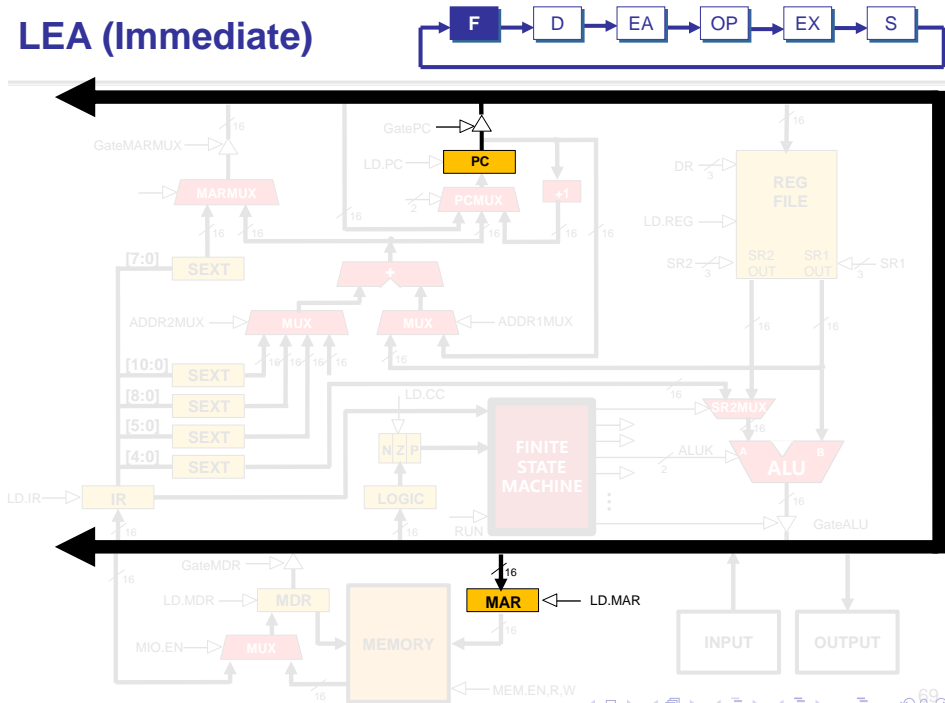
LEA (Immediate) LD DR, PCOffset9



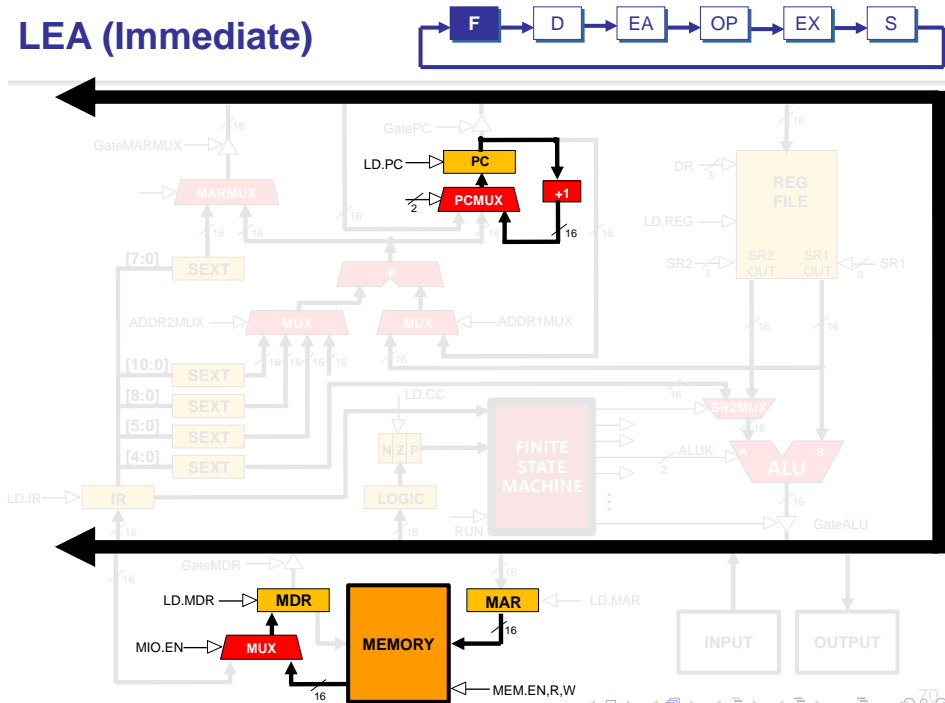
LEA (Immediate): LEA R1, x1AF



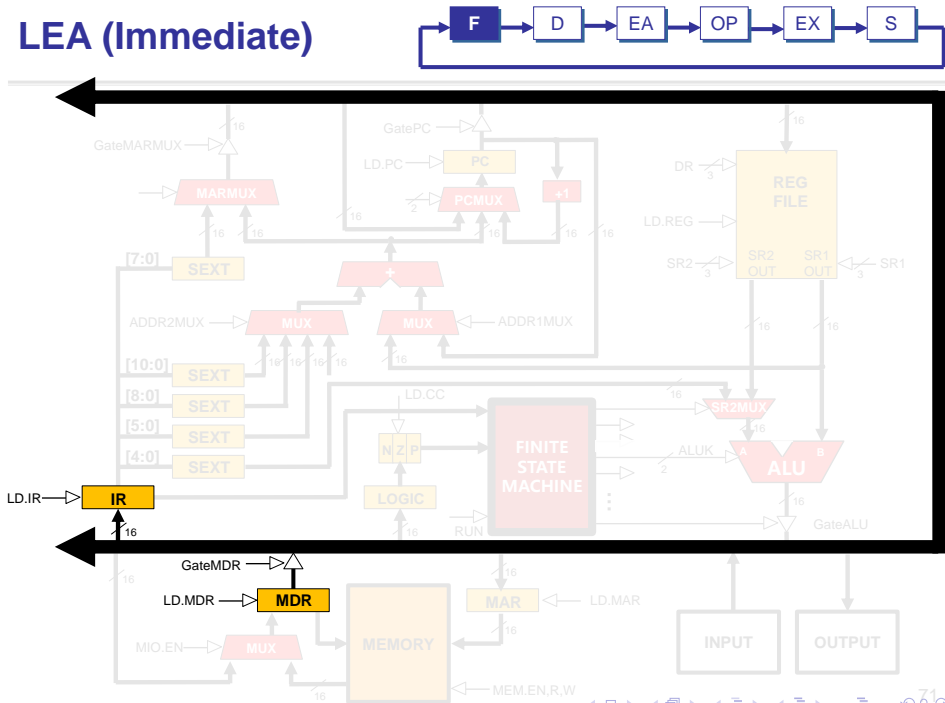
LEA (Immediate)



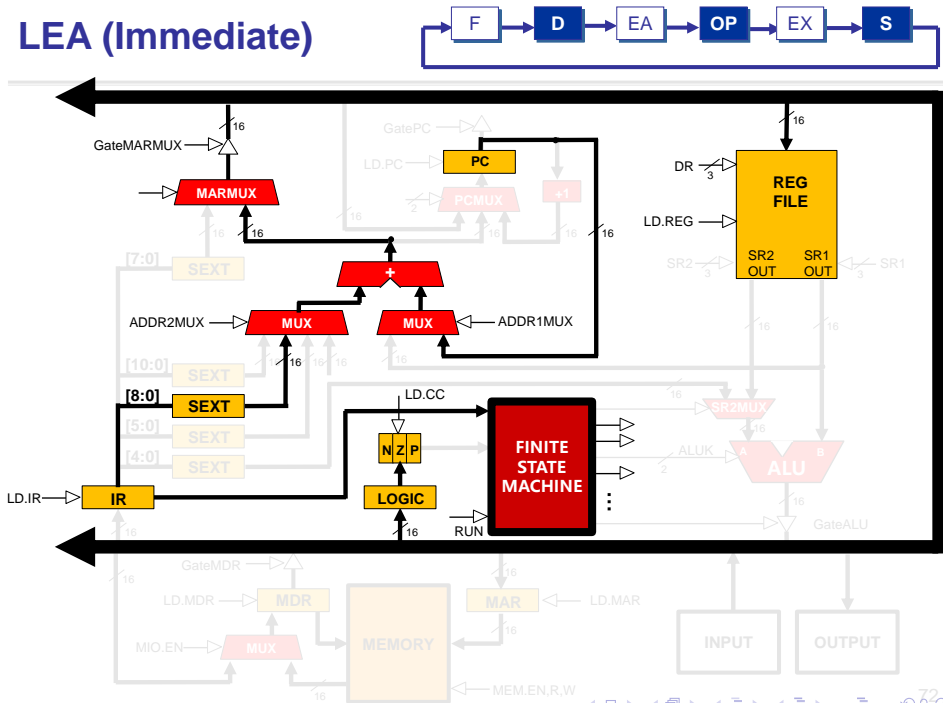
LEA (Immediate)



LEA (Immediate)



LEA (Immediate)



Example

| Address | Instruction | | | | | | | | | | | | | | | | Comments |
|---------|-------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--|
| x30F6 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | $R1 \leftarrow PC - 3 = x30F4$ |
| x30F7 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | $R2 \leftarrow R1 + 14 = x3102$ |
| x30F8 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | $M[PC - 5] \leftarrow R2$; i.e. $M[x30F4] \leftarrow x3102$ |
| x30F9 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | $R2 \leftarrow 0$ |
| x30FA | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | $R2 \leftarrow R2 + 5 = 5$ |
| x30FB | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | $M[R1+14] \leftarrow R2$; i.e. $M[x3102] \leftarrow 5$ |
| x30FC | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | $R3 \leftarrow M[M[PC-9]]$ $= M[M[x30F4]]$ $= M[x3102]$ $= 5$ |

opcode



1 Review

2 LC-3 PC-Relative Load/Store

3 LC-3 Indirect, Base+offset Load/Store

4 **Summary**

Today: Data Movement Instructions

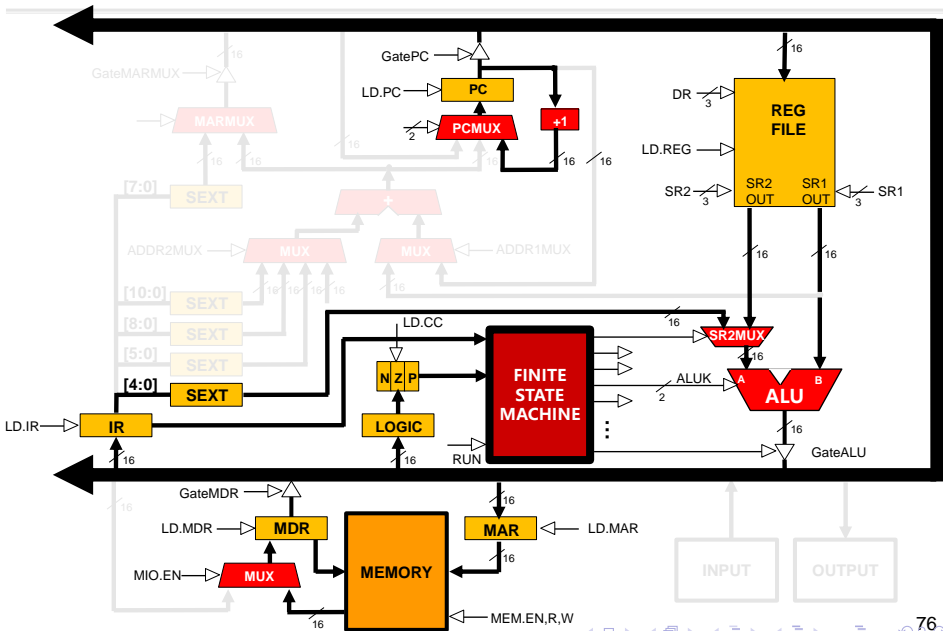
取数指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|-----------|---|---|---|---|---|---|
| LD | 0 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LDR | 0 | 1 | 1 | 0 | DR | | | BaseR | | PCOffset6 | | | | | | |
| LDI | 1 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LEA | 1 | 1 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |

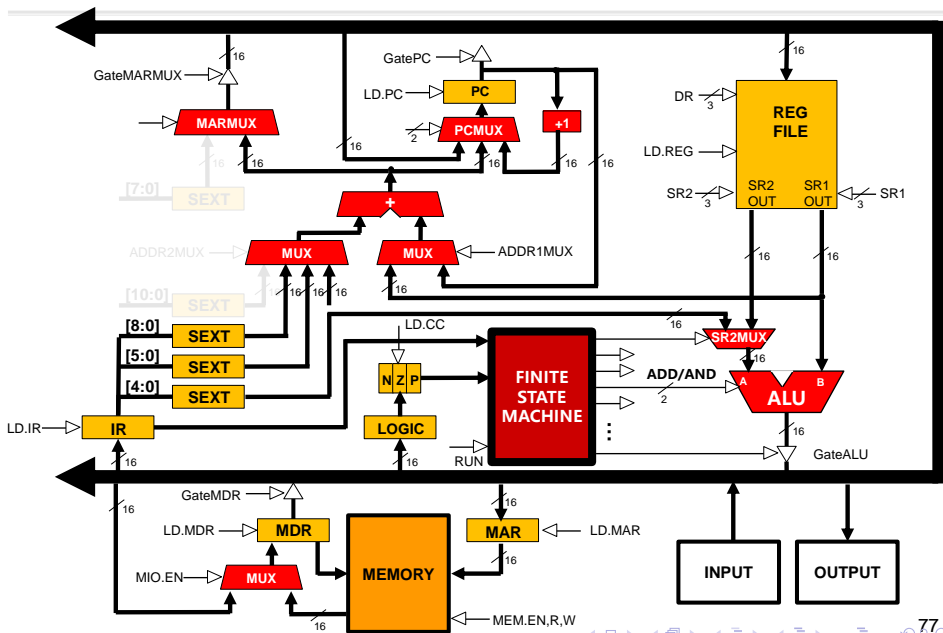
存数指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|-----------|---|---|---|---|---|---|
| ST | 0 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |
| STR | 0 | 1 | 1 | 1 | SR | | | BaseR | | PCOffset6 | | | | | | |
| STI | 1 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |

LC-3 Data Path After Operate Instruction



LC-3 Data Path After Load/Store Instruction



Next Lecture: Control Instructions

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|------------|---|-----------|-------------|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | n | z | p | PCOffset9 | | | | | | | | |
| JSR | 0 | 1 | 0 | 0 | 1 | PCOffset11 | | | | | | | | | | |
| JSRR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | |
| RET | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | | | | | | | | | | | | | | |
| TRAP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | TrapVector8 | | | | | | | |



中国科学技术大学
University of Science and Technology of China

计算机系统概论

Introduction to Computing Systems
(CS1002A.02)

Chapter 5-3

The LC-3 Control Instructions

计算机科学与技术学院
School of Computer Science and Technology



- 1 Review
- 2 LC-3 Control Instructions Overview
- 3 Conditional Branch Instruction and Loop Control Example
- 4 Jump & TRAP Instruction
- 5 Summary



1

Review

2

LC-3 Control Instructions Overview

3

Conditional Branch Instruction and Loop Control Example

4

Jump & TRAP Instruction

5

Summary

Great Idea #3: Abstraction Helps Us Manage Complexity

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

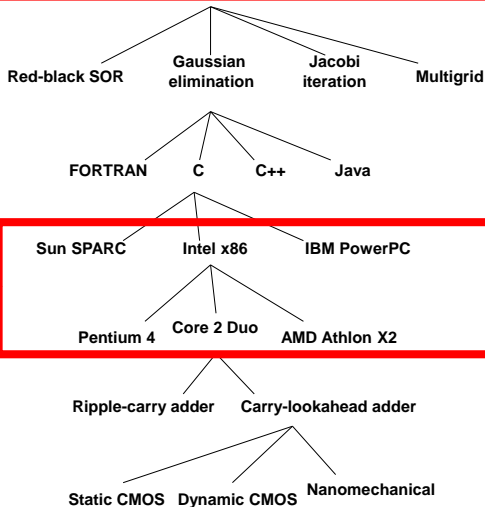
Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

Electronic Devices

Physics

Solve a system of equations



LC-3 ISA Overview

运算指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|----|----|----|----|----|----|---|-----|---|---|---|------|---|-----|---|---|
| ADD | 0 | 0 | 0 | 1 | DR | | | SR1 | | | 0 | 0 | 0 | SR2 | | |
| ADD | 0 | 0 | 0 | 1 | DR | | | SR1 | | | 1 | Imm5 | | | | |
| AND | 0 | 1 | 0 | 1 | DR | | | SR1 | | | 0 | 0 | 0 | SR2 | | |
| AND | 0 | 1 | 0 | 1 | DR | | | SR1 | | | 1 | Imm5 | | | | |
| NOT | 1 | 0 | 0 | 1 | DR | | | SR1 | | | 1 | 1 | 1 | 1 | 1 | 1 |
| Reserved | 1 | 1 | 0 | 1 | | | | | | | | | | | | |

控制指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|------------|---|-----------|-------------|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | n | z | p | PCOffset9 | | | | | | | | |
| JSR | 0 | 1 | 0 | 0 | 1 | PCOffset11 | | | | | | | | | | |
| JSRR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RET | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| TRAP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | TrapVector8 | | | | | | | |

移动数据指令

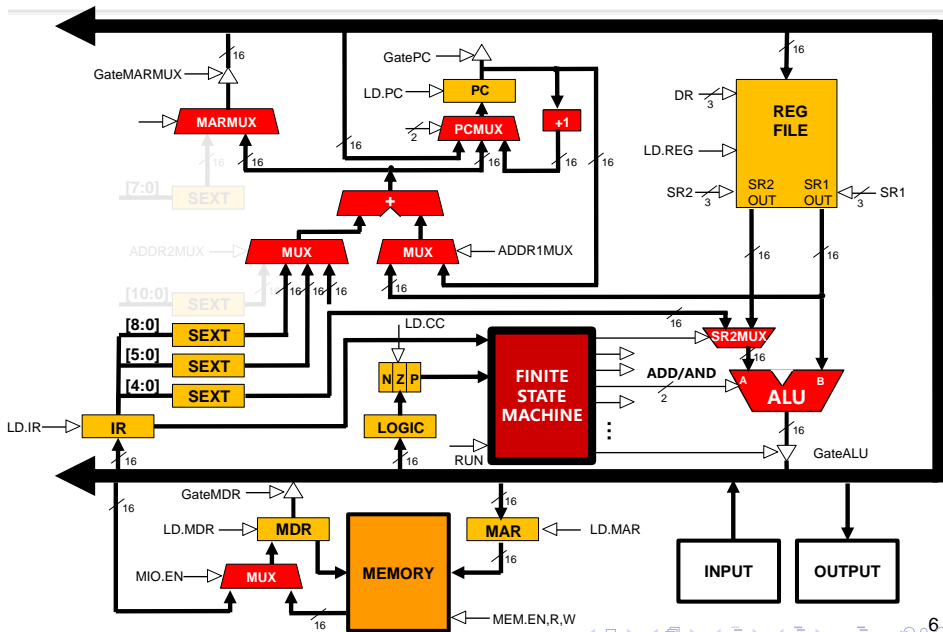
取数指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|-----------|---|---|---|---|---|---|
| LD | 0 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LDR | 0 | 1 | 1 | 0 | DR | | | BaseR | | PCOffset6 | | | | | | |
| LDI | 1 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LEA | 1 | 1 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |

存数指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|---|-----------|---|---|---|---|---|
| ST | 0 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |
| STR | 0 | 1 | 1 | 1 | SR | | | BaseR | | | PCOffset6 | | | | | |
| STI | 1 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |

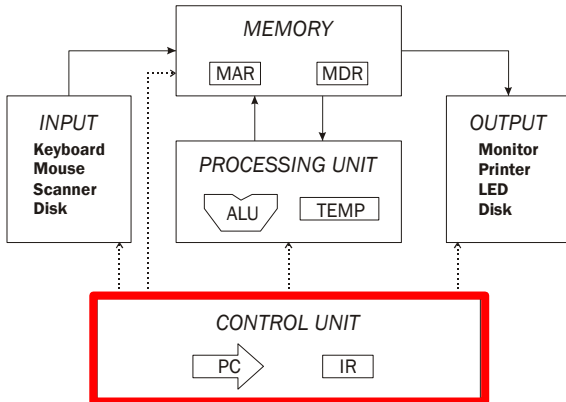
LC-3 Data Path After Load/Store Instruction





■ We are going to learn how to:

- Used to alter the sequence of instructions (by changing the Program Counter)



2

3

5

Control Instructions

Conditional Branch

- branch is *taken* if a specified condition is true
 - signed offset is added to PC to yield new PC
- else, the branch is *not taken*
 - PC is not changed, points to the next sequential instruction

Unconditional Branch (or Jump)

- always changes the PC

TRAP

- changes PC to the address of an OS “service routine”
- routine will return control to the next instruction (after TRAP)

LC-3 ISA Overview

运算指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|----|----|----|----|----|----|-----|---|---|------|---|-----|---|---|---|---|
| ADD | 0 | 0 | 0 | 1 | DR | | SR1 | | 0 | 0 | 0 | SR2 | | | | |
| ADD | 0 | 0 | 0 | 1 | DR | | SR1 | | 1 | Imm5 | | | | | | |
| AND | 0 | 1 | 0 | 1 | DR | | SR1 | | 0 | 0 | 0 | SR2 | | | | |
| AND | 0 | 1 | 0 | 1 | DR | | SR1 | | 1 | Imm5 | | | | | | |
| NOT | 1 | 0 | 0 | 1 | DR | | SR1 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Reserved | 1 | 1 | 0 | 1 | | | | | | | | | | | | |

控制指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|------------|---|-----------|-------------|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | n | z | p | PCOffset9 | | | | | | | | |
| JSR | 0 | 1 | 0 | 0 | 1 | PCOffset11 | | | | | | | | | | |
| JSRR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RET | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| TRAP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | TrapVector8 | | | | | | | |

移动数据指令

取数指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|-----------|---|---|---|---|---|---|
| LD | 0 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LDR | 0 | 1 | 1 | 0 | DR | | | BaseR | | PCOffset6 | | | | | | |
| LDI | 1 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LEA | 1 | 1 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |

存数指令

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|---|-----------|---|---|---|---|---|
| ST | 0 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |
| STR | 0 | 1 | 1 | 1 | SR | | | BaseR | | | PCOffset6 | | | | | |
| STI | 1 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |

Control Instructions

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|------------|---|-----------|-------------|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | n | z | p | PCOffset9 | | | | | | | | |
| JSR | 0 | 1 | 0 | 0 | 1 | PCOffset11 | | | | | | | | | | |
| JSRR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | |
| RET | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| TRAP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | TrapVector8 | | | | | | | |

Condition Codes

LC-3 has three **condition code** registers:

N -- negative

Z -- zero

P -- positive (greater than zero)

Set by any instruction that **writes a value** to a register
(ADD, AND, NOT, LD, LDR, LDI, LEA)

Exactly one will be set at all times

- Based on the last instruction that altered a register



1

Review

2

LC-3 Control Instructions Overview

3

Conditional Branch Instruction and Loop Control Example

4

Jump & TRAP Instruction

5

Summary

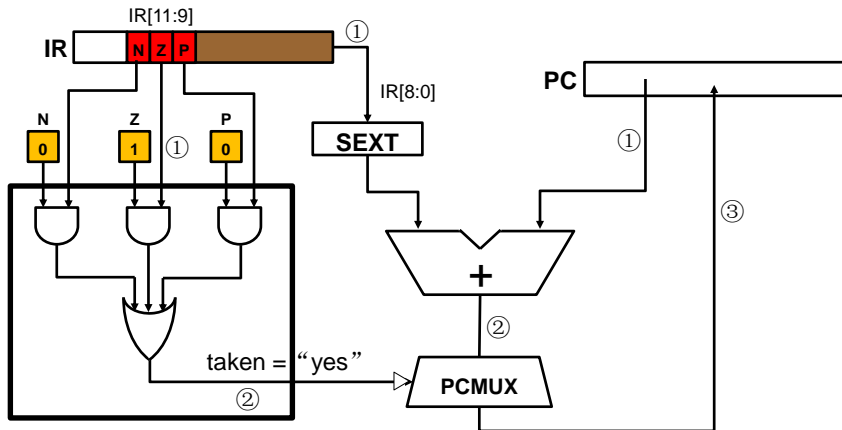
Conditional Branch Instruction

**Branch specifies one or more condition codes.
If the specified bit is set, the branch is taken.**

- PC-relative addressing:
target address is made by adding signed offset (IR[8:0]) to current PC.
- Note: PC has already been incremented by FETCH stage.
- Note: Target must be within 256 words of BR instruction.

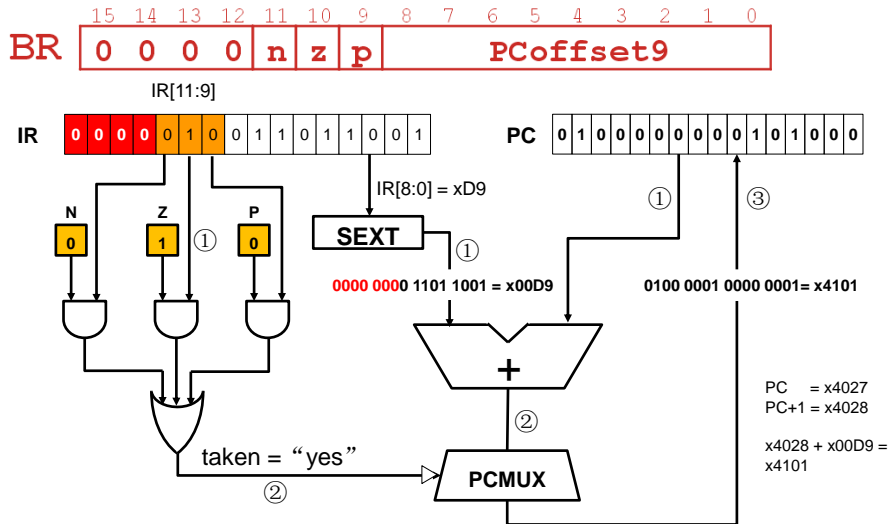
If the branch is not taken, the next sequential instruction is executed.

BR (PC-Relative)



What happens if bits [11:9] are all zero?

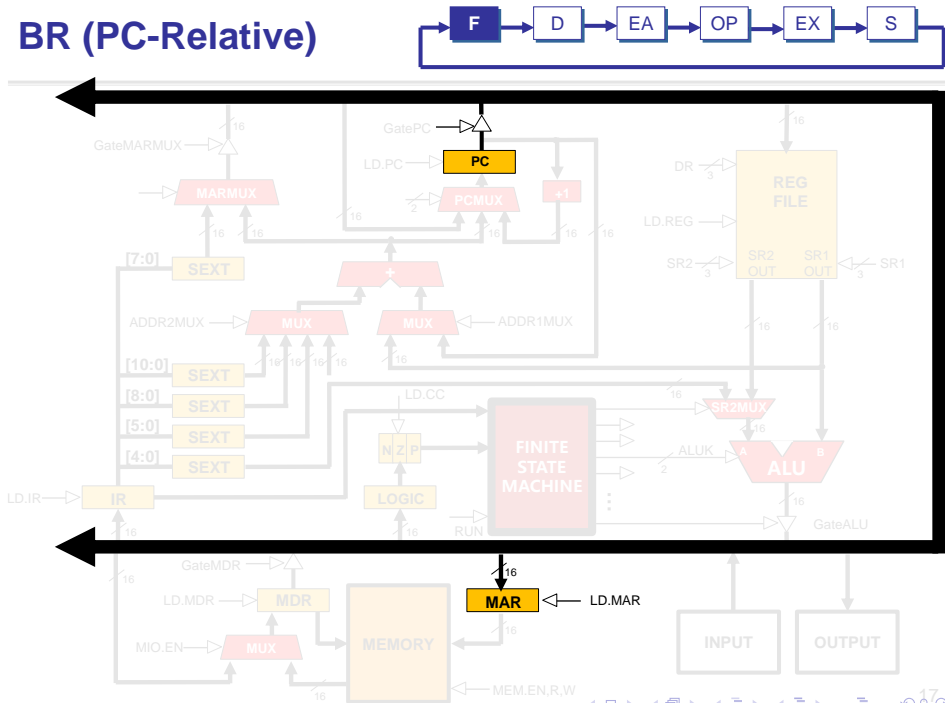
What happens if bits [11:9] are all one?

BR (PC-Relative): BR₇ x4101

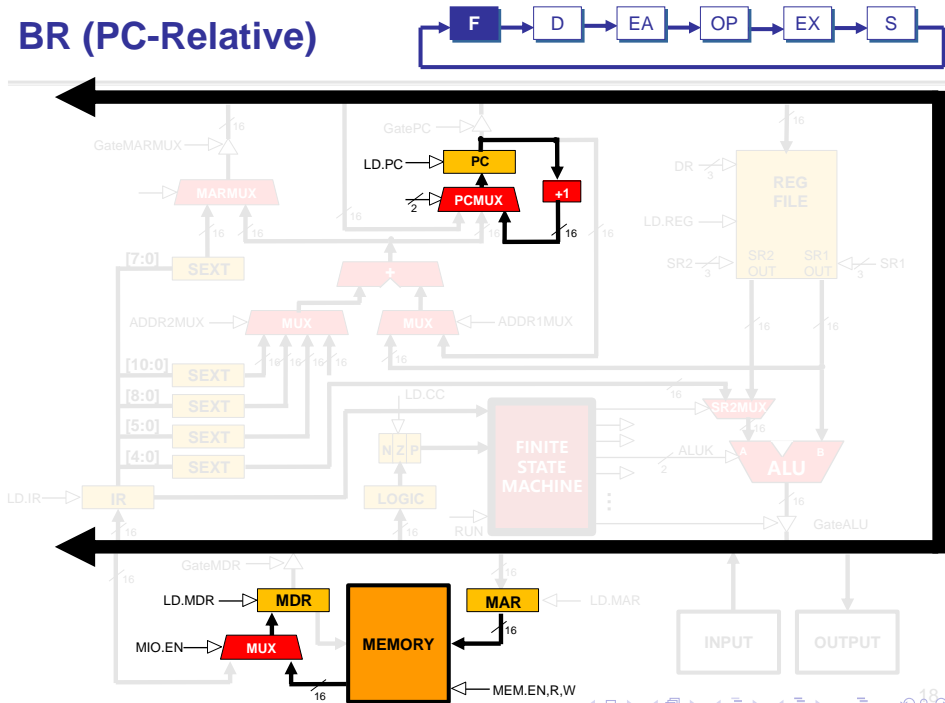
What happens if bits [11:9] are all zero?

What happens if bits [11:9] are all one?

BR (PC-Relative)



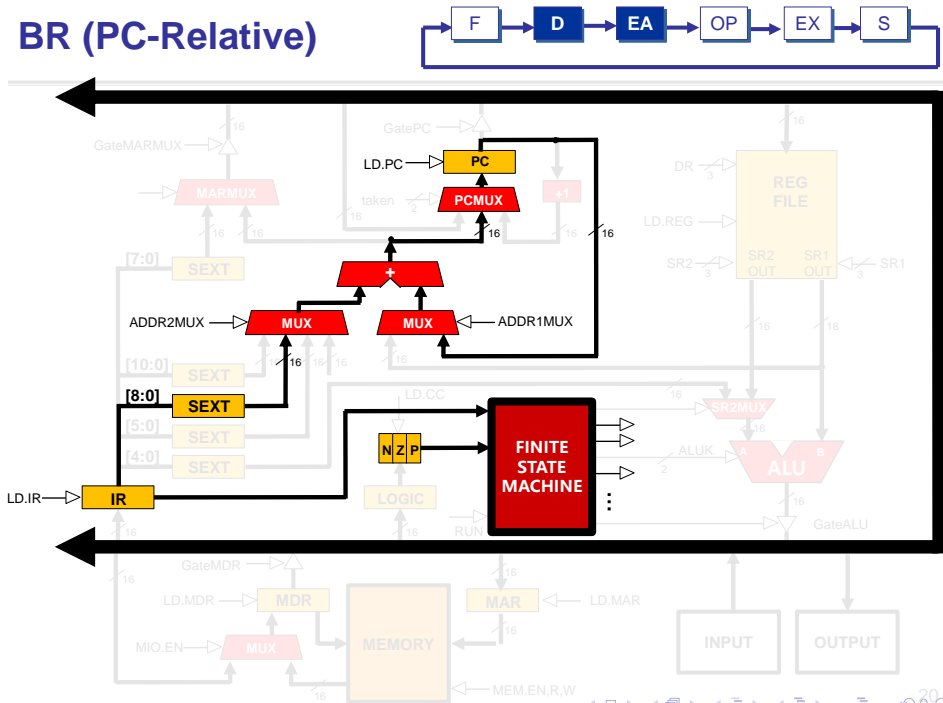
BR (PC-Relative)



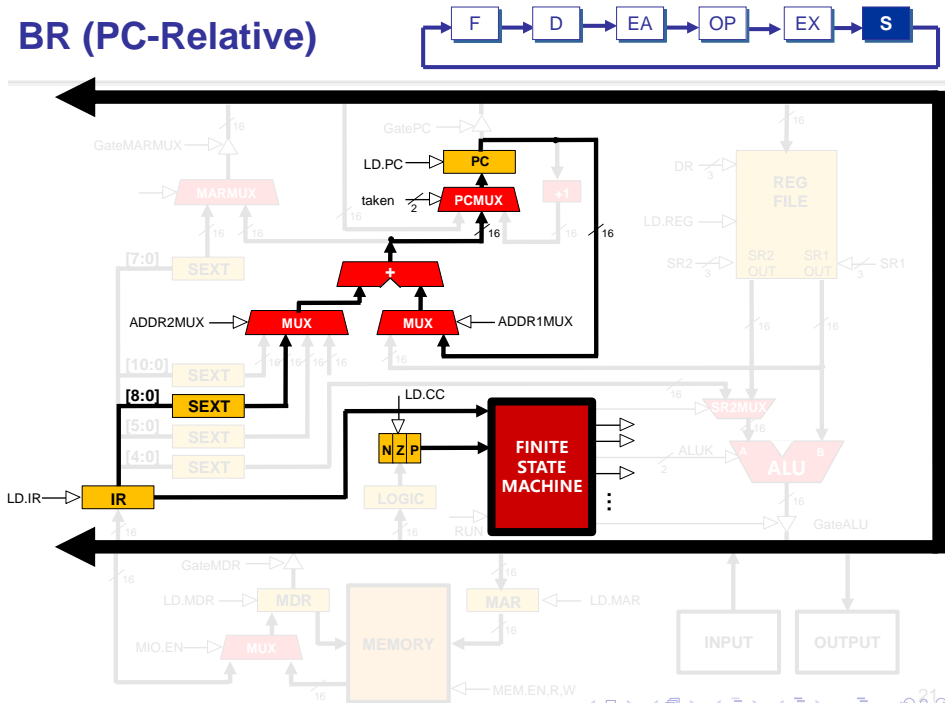
```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



BR (PC-Relative)



BR (PC-Relative)



BR (PC-Relative)

■ Check

- BR_{nzp} x4101 ; if ($n=1$ or $z=1$ or $p=1$) , JMP x4101
- BR_n x4101 ; if ($n=1$)
- BR_z x4101 ; if ($z=1$)
- BR_p x4101 ; if ($p=1$)
- BR_{nz} x4101 ; if ($n=1$ or $z=1$)
- BR_{np} x4101 ; if ($n=1$ or $p=1$)
- BR_{zp} x4101 ; if ($z=1$ or $p=1$)
- BR x4101 ; $PC=PC+1$

■ Set

- If $DR < 0$, set $N=1$ and $Z=0$ and $P=0$
- If $DR = 0$, set $N=0$ and $Z=1$ and $P=0$
- If $DR > 0$, set $N=0$ and $Z=0$ and $P=1$

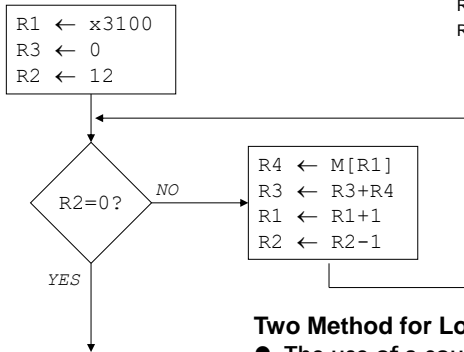
Using Branch Instructions

Compute sum of 12 integers.

Numbers start at location x3100.

Program starts at location x3000.

The use of a counter



Two Method for Loop Control

- The use of a counter
- The use of a sentinel

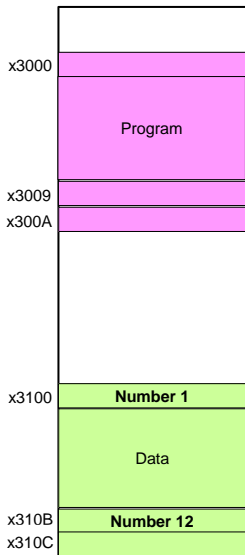
Register File

| | |
|----|-------|
| R0 | x3100 |
| R1 | |
| R2 | 12 |
| R3 | 0 |
| R4 | temp |
| R5 | |
| R6 | |
| R7 | |

PC

| |
|-------|
| x3000 |
|-------|

Memory



Sample Program(The use of a counter)

| Address | Instruction | | | | | | | | | | | | Comments | | |
|---------|-------------|---|---|---|---|---|---|---|---|---|---|---|----------|---|-------------------------------|
| x3000 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | R1 ← x3100 (PC+0xFF); LEA |
| x3001 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | R3 ← 0; AND |
| x3002 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | R2 ← 0; AND |
| x3003 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | R2 ← 12; ADD |
| x3004 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | If Z, goto (PC+5) = x300A; BR |
| x3005 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | Load next value to R4; LDR |
| x3006 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | R3 ← R3 + R4; ADD |
| x3007 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | Increment R1 (pointer); ADD |
| x3008 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | Decrement R2 (counter); ADD |
| x3009 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | Goto (PC-6)=x3004 ;BR |

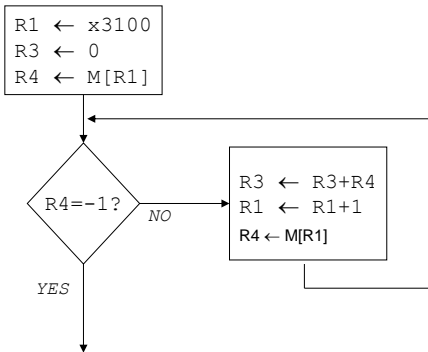
Using Branch Instructions

Compute sum of 12 integers.

Numbers start at location x3100.

Program starts at location x3000.

The use of a sentinel



A special character used to indicate the end of a sequence is often called a **sentinel**.

- Useful when you don't know ahead of time how many times to execute a loop.

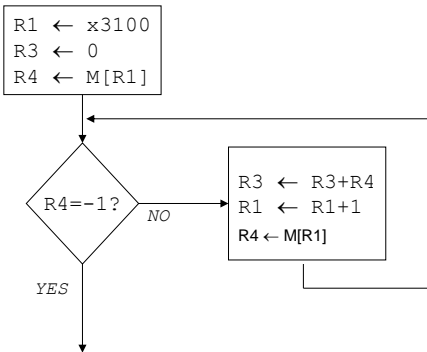
Using Branch Instructions

Compute sum of 12 integers.

Numbers start at location x3100.

Program starts at location x3000.

The use of a sentinel



Register File

| | |
|----|-------|
| R0 | x3100 |
| R1 | |
| R2 | 12 |
| R3 | 0 |
| R4 | temp |
| R5 | |
| R6 | |
| R7 | |

x3000

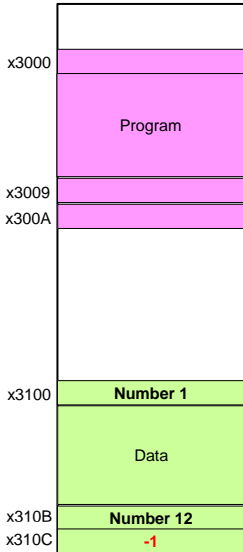
x3009

x300A

PC

| |
|-------|
| x3000 |
|-------|

Memory



Sample Program(The use of a sentinel)

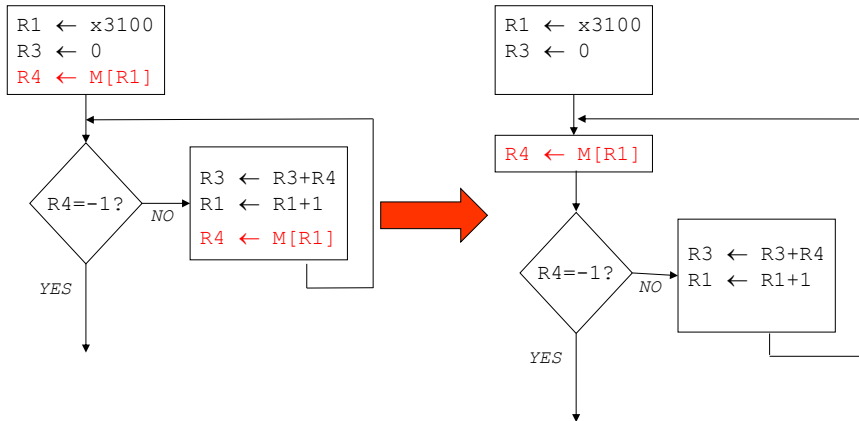
| Address | Instruction | | | | | | | | | | | | Comments | | |
|---------|-------------|---|---|---|---|---|---|---|---|---|---|---|----------|---|---------------------------------------|
| x3000 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | $R1 \leftarrow (PC+0xFF)=x3100$; LEA |
| x3001 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | $R3 \leftarrow 0$; AND |
| x3002 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | $R4 \leftarrow M[R1]$; LDR |
| x3003 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | If N, goto (PC+ 4)=X3008; BR |
| x3004 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | $R3 \leftarrow R3 + R4$; ADD |
| x3005 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | $R1 \leftarrow R1 + 1$; ADD |
| x3006 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | $R4 \leftarrow M[R1]$; LDR |
| x3007 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | Goto (PC-5)= x3003; BR |

Using Branch Instructions(Code Optimization)

Compute sum of 12 integers.

Numbers start at location x3100.

Program starts at location x3000.



Sample Program

| Address | Instruction | | | | | | | | | | | | | | | | Comments |
|---------|-------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|-----------------------------------|
| x3000 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | $R1 \leftarrow (PC+0xFF) = x3100$ |
| x3001 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | $R3 \leftarrow 0$ |
| x3002 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | $R4 \leftarrow M[R1]$ |
| x3003 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | If N, goto $(PC+3)=x3007$ |
| x3004 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | $R3 \leftarrow R3 + R4$ |
| x3005 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | $R1 \leftarrow R1 + 1$ |
| x3006 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | Goto $(PC-5)=x3002$ |



1

Review

2

LC-3 Control Instructions Overview

3

Conditional Branch Instruction and Loop Control Example

4

Jump & TRAP Instruction

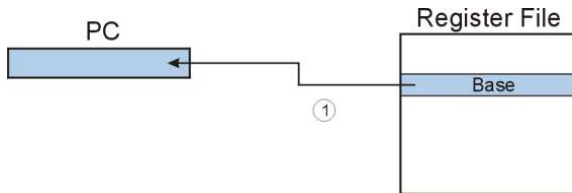
5

Summary

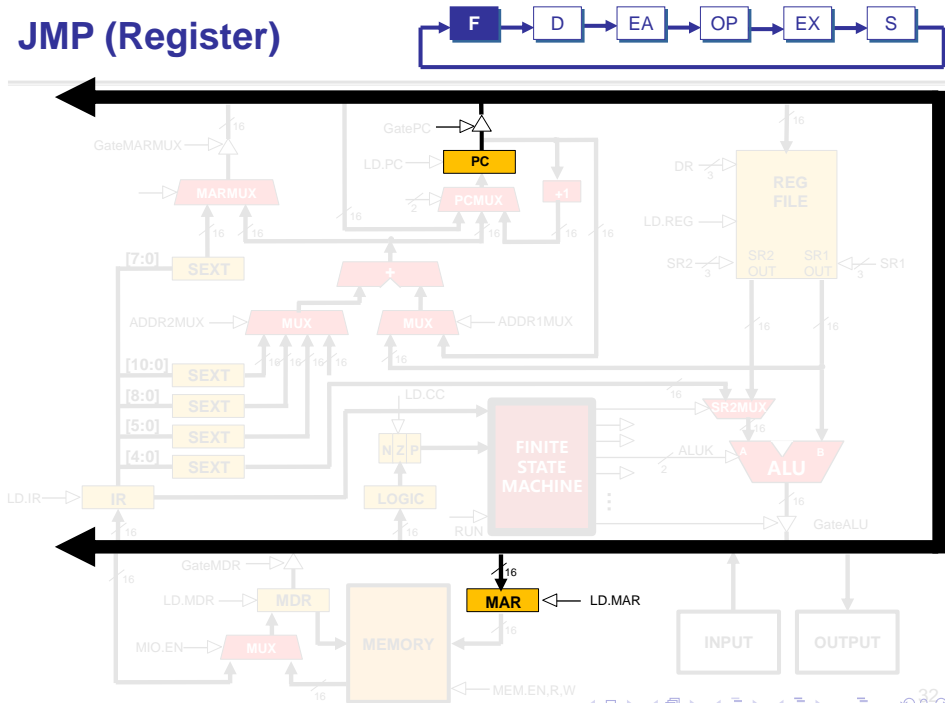
JMP (Register)

Jump is an unconditional branch -- *always* taken.

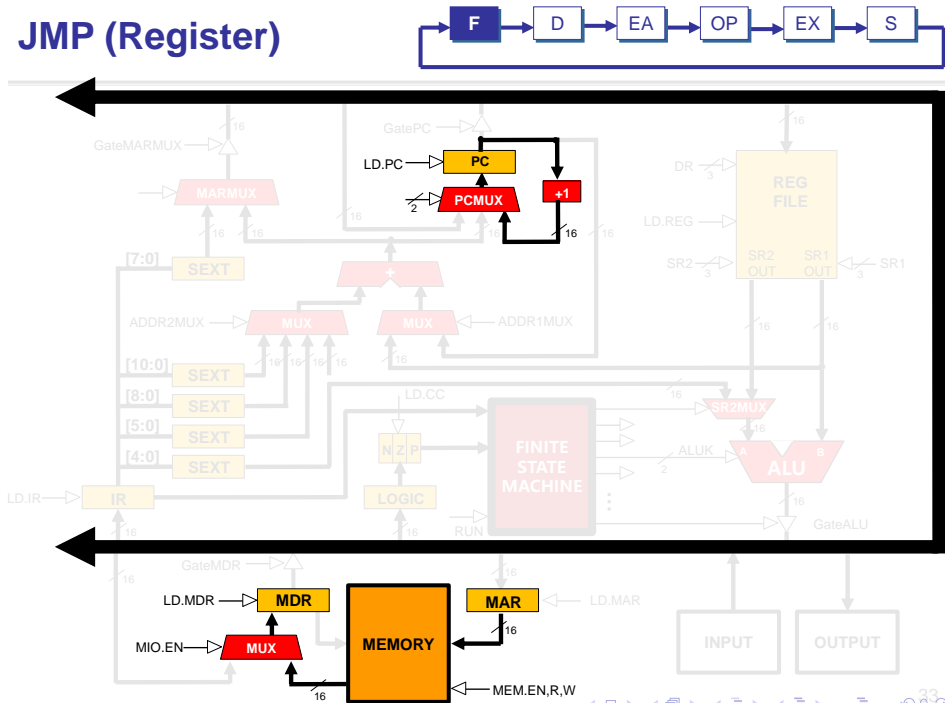
- Target address is the contents of a register.
- Allows any target address.



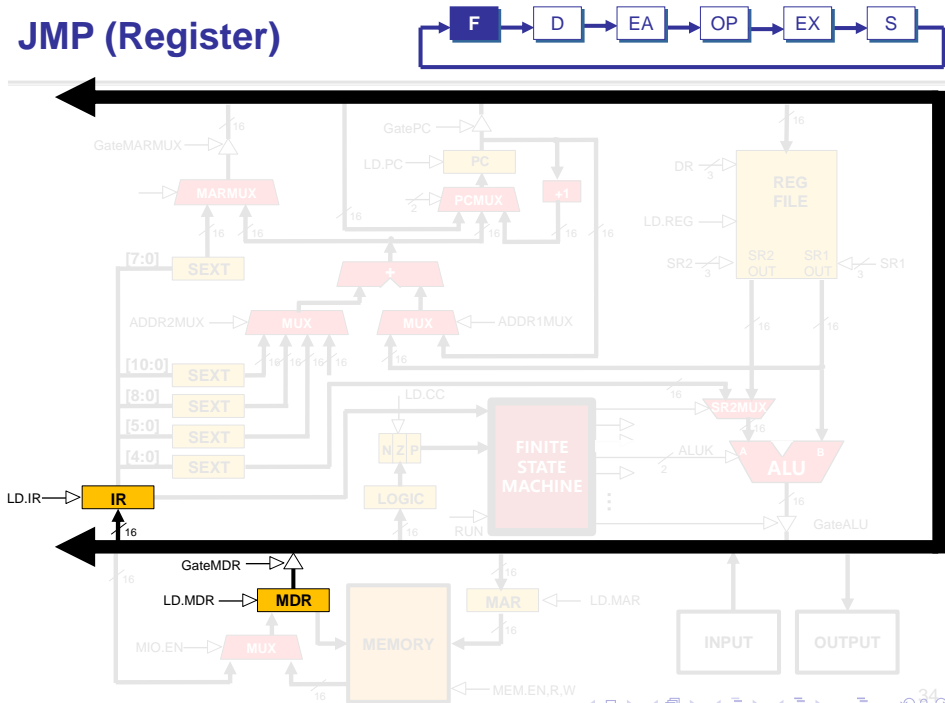
JMP (Register)



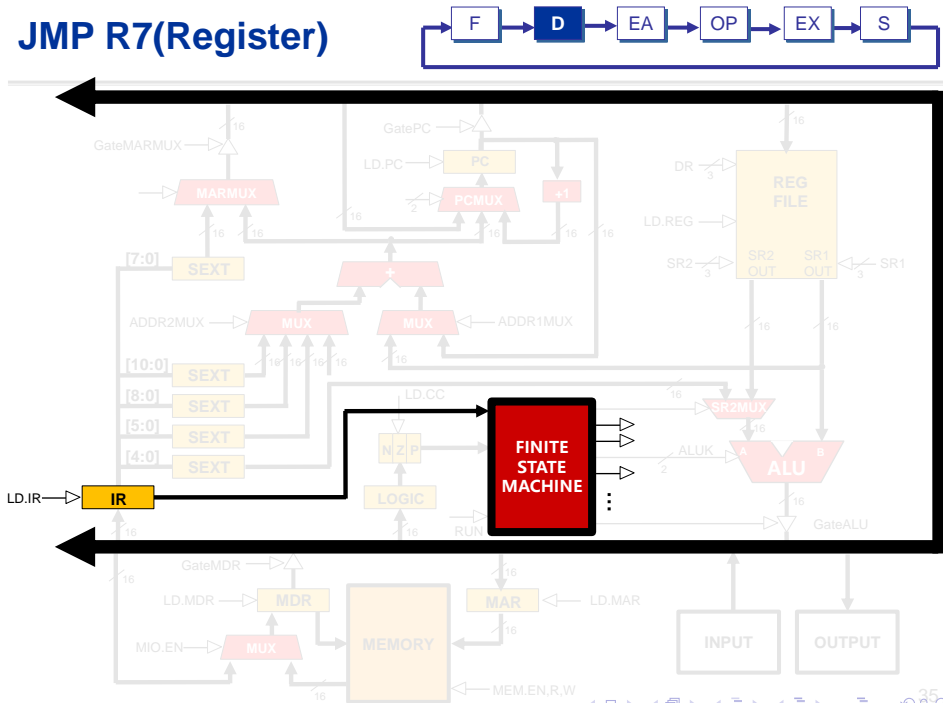
JMP (Register)



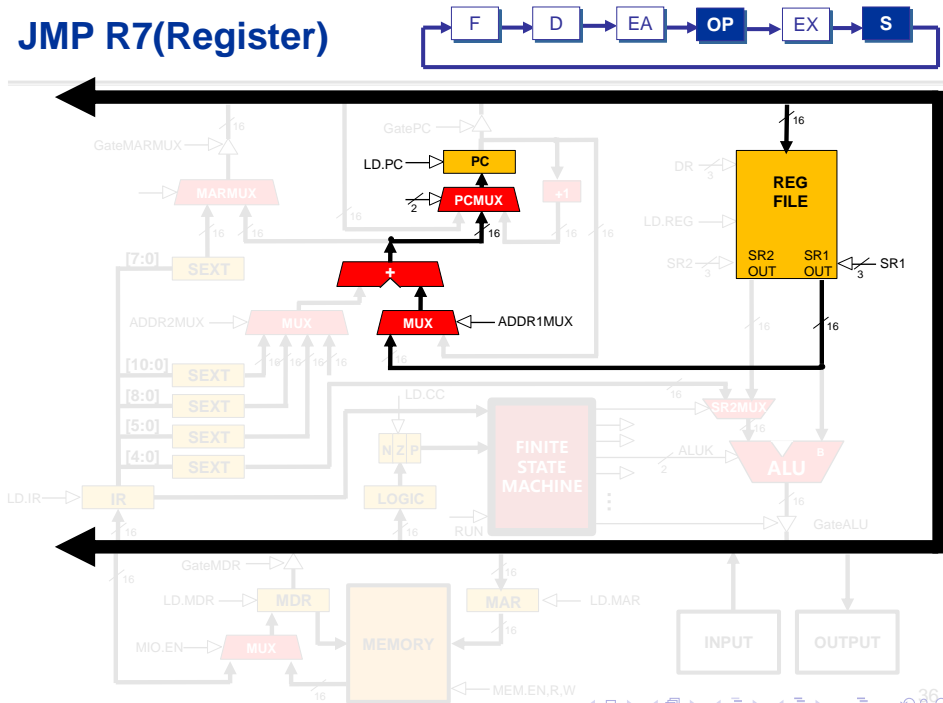
JMP (Register)



JMP R7(Register)



JMP R7(Register)



TRAP



Calls a **service routine**, identified by 8-bit “trap vector.”

| vector | routine |
|--------|-------------------------------------|
| x23 | input a character from the keyboard |
| x21 | output a character to the monitor |
| x25 | halt the program |

Example:

TRAP x23

; Directs the operating system to execute the **IN** system call.

; The starting address of this system call is contained in **memory location x0023**.

TRAP



Calls a **service routine**, identified by 8-bit “trap vector.”

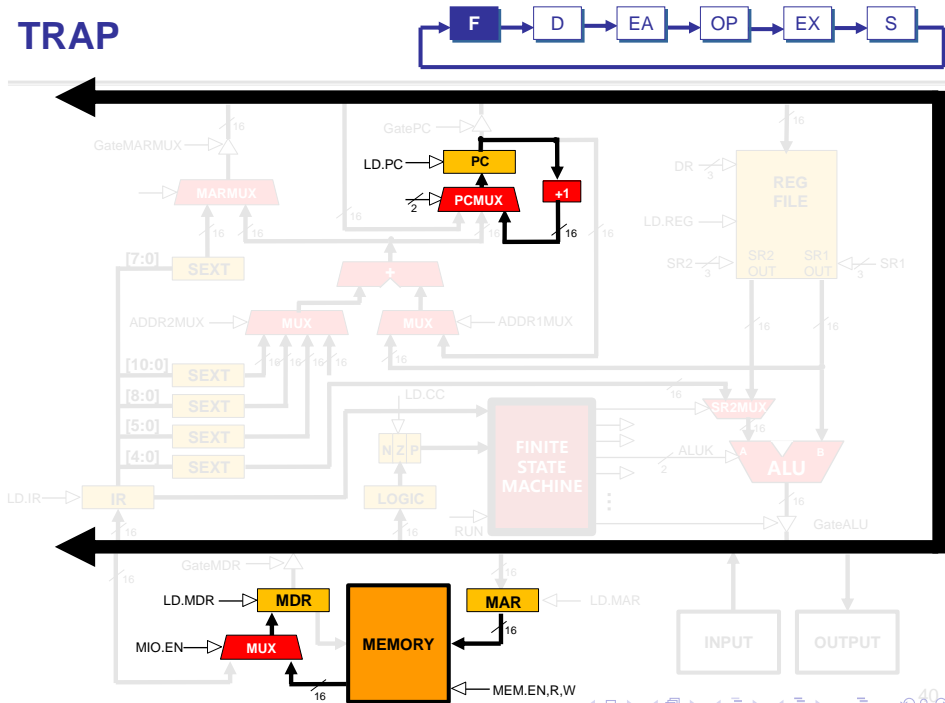
| vector | routine |
|--------|-------------------------------------|
| x23 | input a character from the keyboard |
| x21 | output a character to the monitor |
| x25 | halt the program |

When routine is done,
PC is set to the instruction following TRAP.
(We’ ll talk about how this works later.)

```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



TRAP



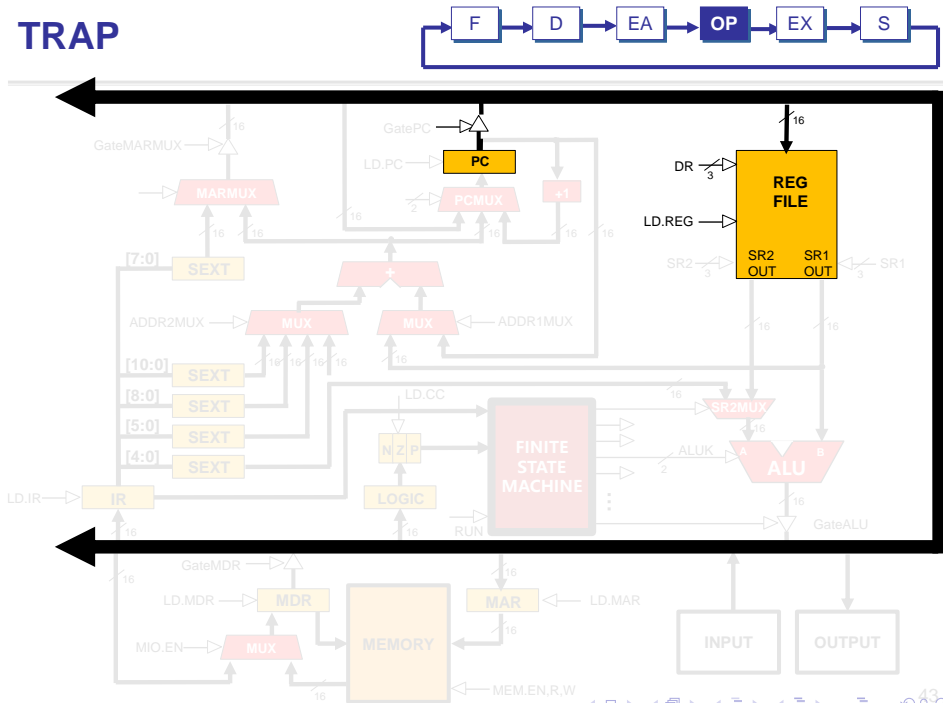
```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



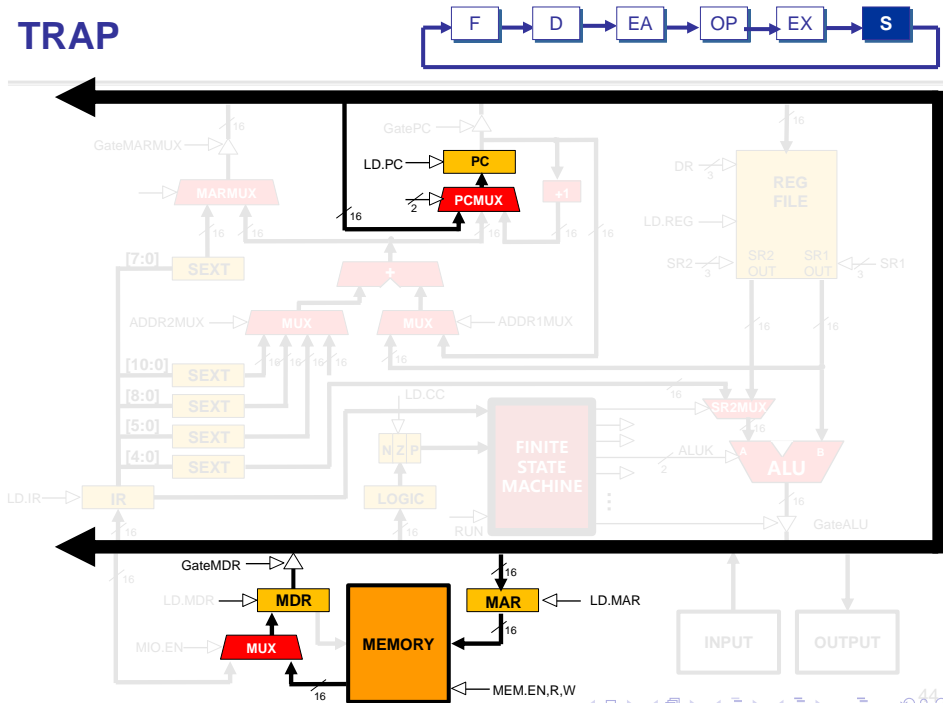

```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



TRAP



TRAP





1

Review

2

LC-3 Control Instructions Overview

3

Conditional Branch Instruction and Loop Control Example

4

Jump & TRAP Instruction

5

Summary

运算指令(Operate Instructions)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|----|----|----|----|----|----|---|-----|---|---|---|------|---|-----|---|---|
| ADD | 0 | 0 | 0 | 1 | DR | | | SR1 | | | 0 | 0 | 0 | SR2 | | |
| ADD | 0 | 0 | 0 | 1 | DR | | | SR1 | | | 1 | Imm5 | | | | |
| AND | 0 | 1 | 0 | 1 | DR | | | SR1 | | | 0 | 0 | 0 | SR2 | | |
| AND | 0 | 1 | 0 | 1 | DR | | | SR1 | | | 1 | Imm5 | | | | |
| NOT | 1 | 0 | 0 | 1 | DR | | | SR1 | | | 1 | 1 | 1 | 1 | 1 | 1 |
| Reserved | 1 | 1 | 0 | 1 | | | | | | | | | | | | |

控制指令(Control Instructions)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|------------|---|-----------|-------------|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | n | z | p | PCoffset9 | | | | | | | | |
| JSR | 0 | 1 | 0 | 0 | 1 | PCoffset11 | | | | | | | | | | |
| JSRR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RET | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| TRAP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | TrapVector8 | | | | | | | |

数据移动指令 (Data Movement Instructions)

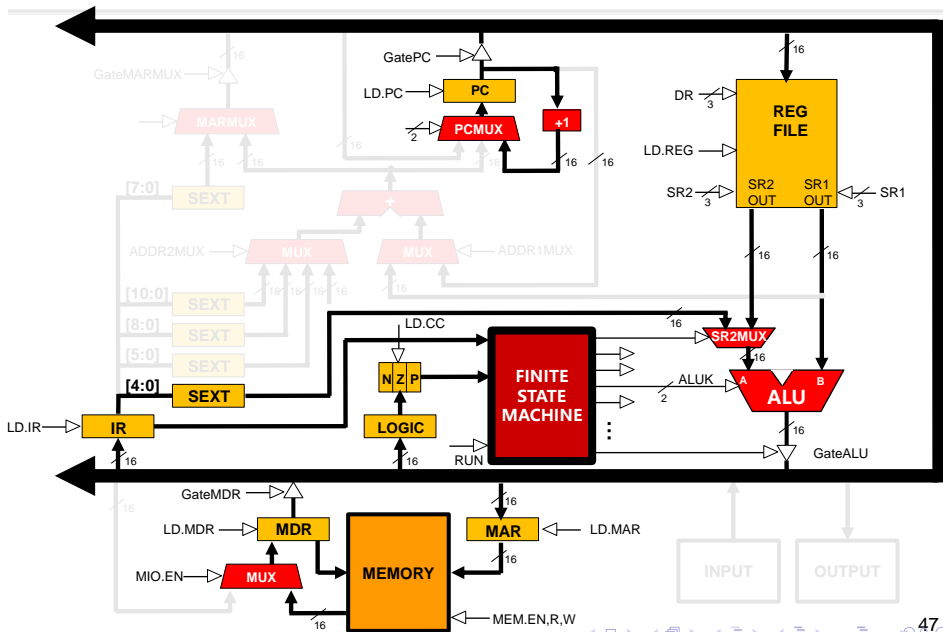
取数指令(Load)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|-----------|---|---|---|---|---|---|
| LD | 0 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LDR | 0 | 1 | 1 | 0 | DR | | | BaseR | | PCOffset6 | | | | | | |
| LDI | 1 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LEA | 1 | 1 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |

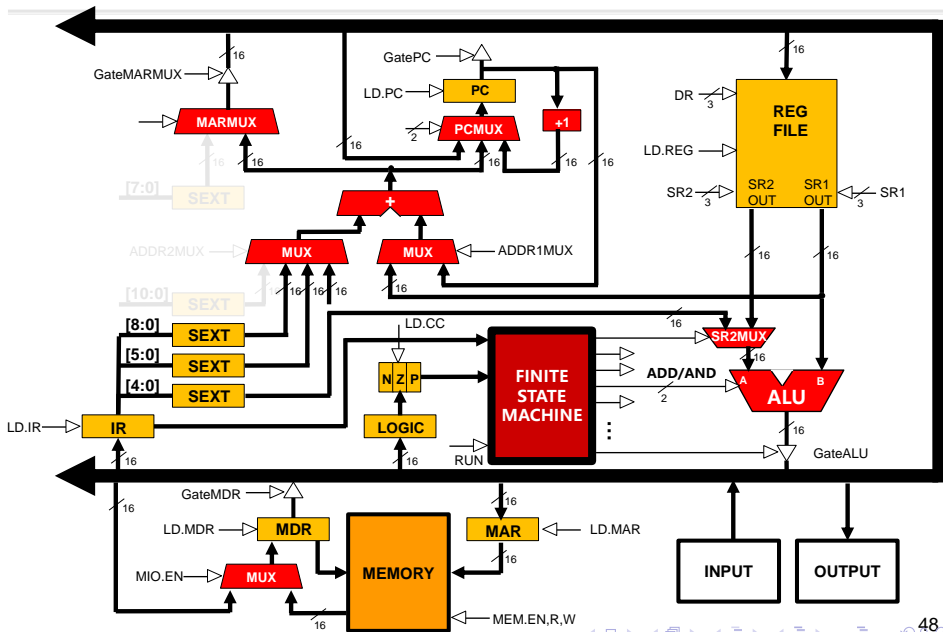
存数指令(Store)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|---|-----------|---|---|---|---|---|
| ST | 0 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |
| STR | 0 | 1 | 1 | 1 | SR | | | BaseR | | | PCOffset6 | | | | | |
| STI | 1 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |

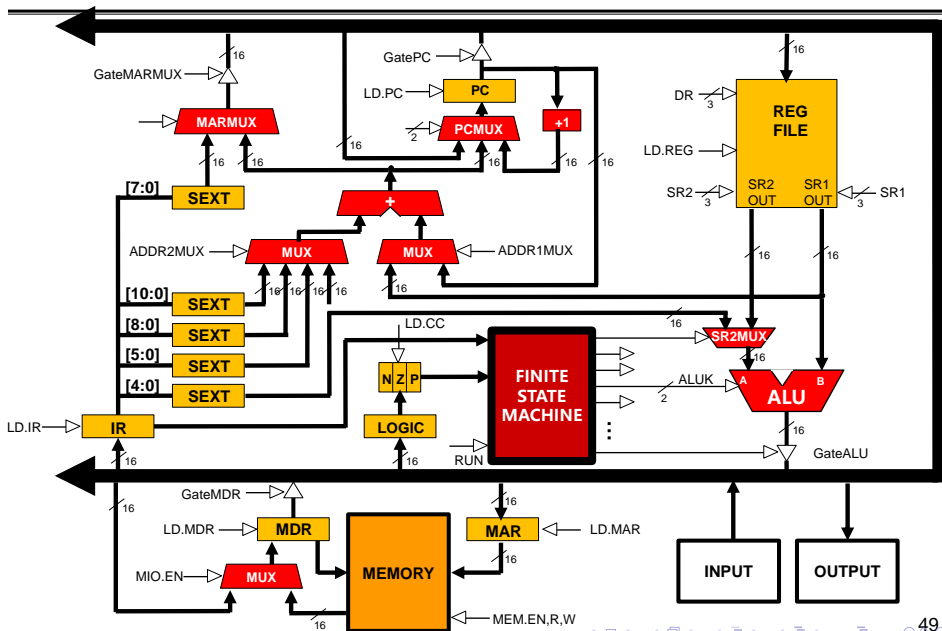
LC-3 Data Path After Operate Instruction



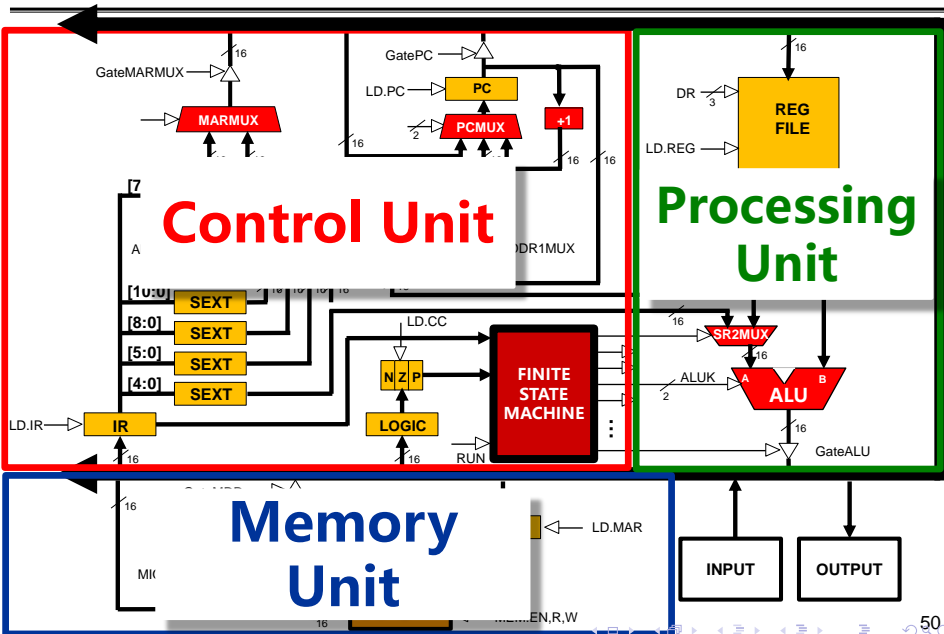
LC-3 Data Path After Load/Store Instruction



LC-3 Data Path After Control Instruction



LC-3 Data Path





中国科学技术大学
University of Science and Technology of China

计算机系统概论

Introduction to Computing Systems
(CS1002A.02)

Chapter 5-4 Tying It All Together

计算机科学与技术学院
School of Computer Science and Technology



1 Review

2 An Example: Counting Occurrences of a Character

3 ISA & Data Path Revisited

4 Summary

Control Instructions

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|------------|---|-----------|-------------|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | n | z | p | PCOffset9 | | | | | | | | |
| JSR | 0 | 1 | 0 | 0 | 1 | PCOffset11 | | | | | | | | | | |
| JSRR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RET | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| TRAP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | TrapVector8 | | | | | | | |

Condition Codes

LC-3 has three **condition code** registers:

N -- negative

Z -- zero

P -- positive (greater than zero)

**Set by any instruction that writes a value to a register
(ADD, AND, NOT, LD, LDR, LDI, LEA)**

Exactly one will be set at all times

- Based on the last instruction that altered a register

1

2

3

4

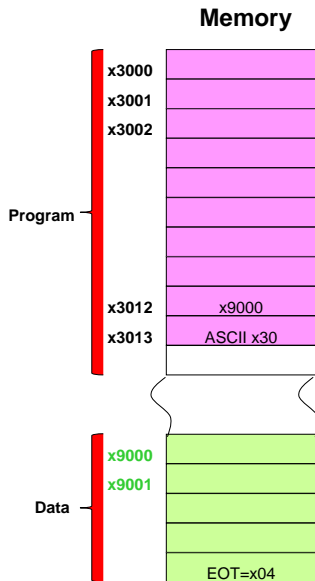
Counting the occurrences of a character in a file

- Program begins at location x3000
- Read character from keyboard
- Load each character from a “file”
 - File is a sequence of memory locations
 - Starting address of file is stored in the memory location immediately after the program
- If file character equals input character, increment counter
- End of file is indicated by a special ASCII value: **EOT (x04)**
- At the end, print the number of characters and halt (assume there will be less than 10 occurrences of the character)

Counting the occurrences of a character in a file

A special character used to indicate the end of a sequence is often called a **sentinel**.

- Useful when you don't know ahead of time how many times to execute a loop.



Register and Memory

Register

R0: hold the character that is being counted (typed from keyboard)

R1: hold, in turn, each character that we get from the file being examined

R2: keep track of the number of occurrences

R3: at first,
 $M[x3012] = x9000$

R4: temp, checking $R4 = R1 - \text{ASCII}(\text{EOT})$

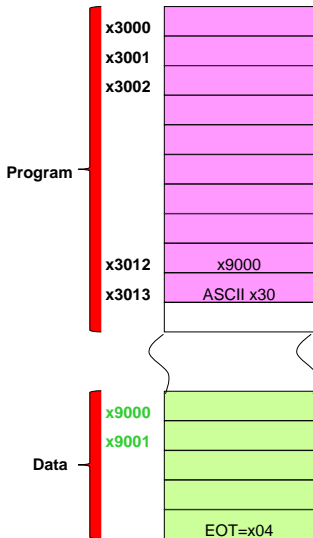
Register File

| | |
|----|-------|
| R0 | x3100 |
| R1 | |
| R2 | count |
| R3 | x9000 |
| R4 | temp |
| R5 | |
| R6 | |
| R7 | |

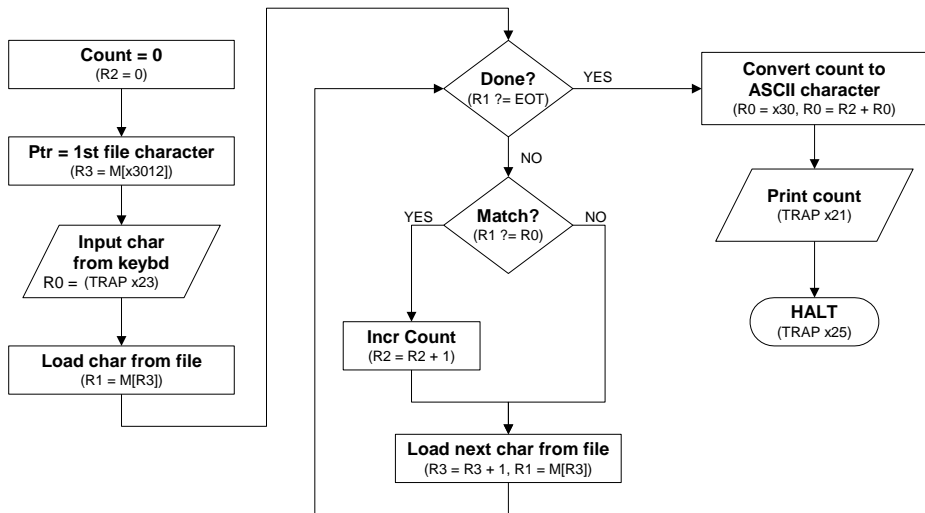
PC

| |
|-------|
| x3000 |
|-------|

Memory



Flow Chart



Counting the occurrences of a character in a file

```
.ORIG x3000
    AND R2, R2, #0
    LD R3, PTR
    TRAP x23
    LDR R1, R3, #0
TEST  ADD R4, R1, #-4
      BRz OUTPUT
      NOT R1, R1
      ADD R1, R1, #1
      ADD R1, R1, R0
      BRnp GETCHAR
      ADD R2, R2, #1
GETCHAR ADD R3, R3, #1
      LDR R1, R3, #0
      BRnzp TEST
OUTPUT LD R0, ASCII
      ADD R0, R0, R2
      TRAP x21
;
      HALT
PTR    .FILL x9000
ASCII  .FILL x30
.END
;
.ORIG X9000
.FILL x0031
.FILL x0032
.FILL x0031
.FILL x0033
.FILL x0043
.FILL x04
.END
```

```
.ORIG x3000
AND R2, R2, #0
LD R3, PTR
TRAP x23
LDR R1, R3, #0
ADD R4, R1, #-4
TEST  BRz OUTPUT
      NOT R1, R1
      ADD R1, R1, #1
      ADD R1, R1, R0
      BRnp GETCHAR
      ADD R2, R2, #1
GETCHAR ADD R3, R3, #1
      LDR R1, R3, #0
      BRnzp TEST
OUTPUT LD R0, ASCII
      ADD R0, R0, R2
      TRAP x21
;
      HALT
PTR    .FILL x9000
ASCII  .FILL x30
.END
;
.ORIG X9000
.FILL x0031
.FILL x0032
.FILL x0031
.FILL x0033
.FILL x0043
.FILL x04
.END
```

LC3Tools v1.6.6
Application Edit View

Registers

| Register | Value | Comment |
|----------|-------|---------|
| R0 | x0000 | 0 |
| R1 | x7FFF | 32767 |
| R2 | x0002 | 2 |
| R3 | x301A | 12314 |
| R4 | x0000 | 0 |
| R5 | x0000 | 0 |
| R6 | x2FFE | 12286 |
| R7 | x0000 | 0 |
| PSR | x0002 | 2 CC: Z |
| PC | x0263 | 611 |
| ISR | x0000 | 0 |

Memory

| Address | Value | Comment |
|---------|-------|-----------------------------|
| x3000 | x74A0 | 11664 AND R2, R2, #0 |
| x3001 | x2410 | 9744 LD R3, PTR |
| x3002 | xF023 | 61475 TRAP x23 |
| x3003 | x62C0 | 25280 LDR R1, R3, #0 |
| x3004 | x187C | 6268 TEST ADD R4, R1, #-4 |
| x3005 | x0408 | 1032 RRs OUTPUT |
| x3006 | x927F | 37503 NOT R1, R1 |
| x3007 | x1261 | 4705 ADD R1, R1, #1 |
| x3008 | x1240 | 4672 ADD R1, R1, R0 |
| x3009 | x1A01 | 2561 BRnp GETCHAR |
| x300A | x14A1 | 5281 ADD R2, R2, #1 |
| x300B | x16E1 | 5657 GETCHAR ADD R3, R3, #1 |
| x300C | x62C0 | 25280 LDR R1, R3, #0 |
| x300D | x07FE | 4086 BRnp TEST |
| x300E | x2004 | 8196 OUTPUT LD R0, ASCII |
| x300F | x1002 | 4098 ADD R0, R0, R2 |
| x3010 | xF021 | 61473 TRAP x21 |
| x3011 | xF025 | 61477 HALT |
| x3012 | x3015 | 12309 PTR .FILL x3015 |
| x3013 | x0030 | 48 ASCII .FILL x30 |
| x3014 | x0000 | 0 |
| x3015 | x0031 | 49 .FILL x0031 |
| x3016 | x0032 | 50 .FILL x0032 |
| x3017 | x0031 | 49 .FILL x0031 |
| x3018 | x0033 | 51 .FILL x0033 |
| x3019 | x0043 | 67 .FILL x0043 |
| x301A | x0004 | 4 .FILL x04 |
| x301B | x0000 | 0 |
| x301C | x0000 | 0 |
| x301D | x0000 | 0 |
| x301E | x0000 | 0 |
| x301F | x0000 | 0 |

Console (click to focus)

```

Input a character> 1
3
--- Halting the LC-3 ---

```

Jump To Location

PC ← ← → →

File
EOT

Program (1 of 2)

| Address | Instruction | | | | | | | | | | | | | | | | Comments |
|---------|-------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x3000 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | $R2 \leftarrow 0$ (counter) AND R2,R2,#0 |
| x3001 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | $R3 \leftarrow M[x3012]$ (ptr) LD R3, x3012 (LD R3, PTR) |
| x3002 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | Input to R0 (TRAP x23) TRAP x23 (GETC) |
| x3003 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | $R1 \leftarrow M[R3]$ LDR R1, R3, #0 |
| x3004 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | $R4 \leftarrow R1 - 4$ (EOT) ADD R4,R1,#-4 |
| x3005 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | If Z, goto x300E BRz x300E (BRz OUTPUT) |
| x3006 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | $R1 \leftarrow \text{NOT } R1$ NOT R1,R1 |
| x3007 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | $R1 \leftarrow R1 + 1$ ADD R1,R1,#1 |
| x3008 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | $R1 \leftarrow R1 + R0$ ADD R1,R1,R0 |
| x3009 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | If N or P, goto x300B BRnp x300B (BRnp GETCHAR) |

Program (2 of 2)

| Address | Instruction | | | | | | | | | | | | Comments | | | | |
|---------|-------------|---|---|---|---|---|---|---|---|---|---|---|----------|---|---|---|--|
| x300A | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | $R2 \leftarrow R2 + 1$ ADD R2,R2,#1 |
| x300B | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | $R3 \leftarrow R3 + 1$ ADD R3,R3,#1 |
| x300C | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | $R1 \leftarrow M[R3]$ LDR R1,R3,#0 |
| x300D | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Goto x3004 BRnzp x3004 (BRnzp TEST) |
| x300E | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | $R0 \leftarrow M[x3013]$ LD R0,x3013 (LD R0, ASCII) |
| x300F | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | $R0 \leftarrow R0 + R2$ ADD R0,R0,R2 |
| x3010 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | Print R0 TRAP x21 (OUT) |
| x3011 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | HALT TRAP x25 (HALT) |
| X3012 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Starting Address of File (X9000) |
| x3013 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | ASCII x30 ('0') |



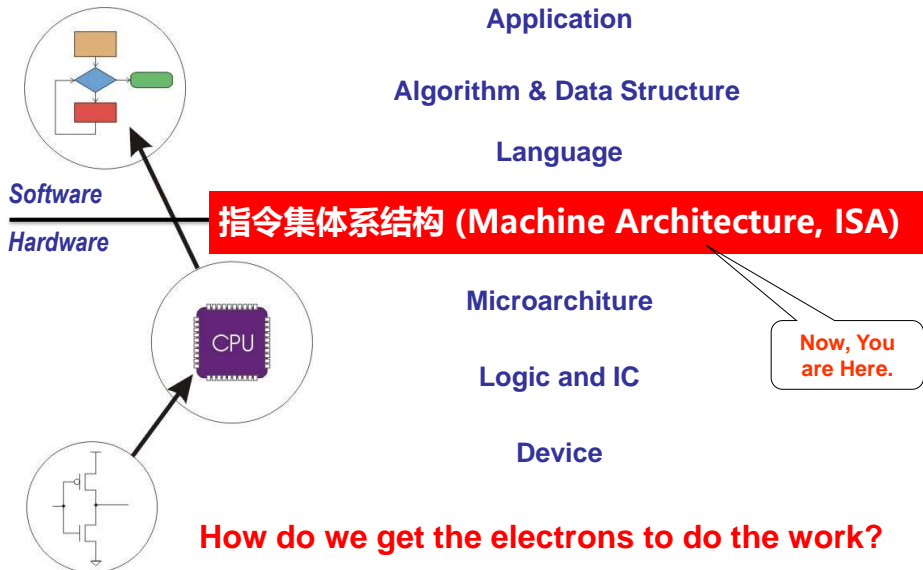
1 Review

2 An Example: Counting Occurrences of a Character

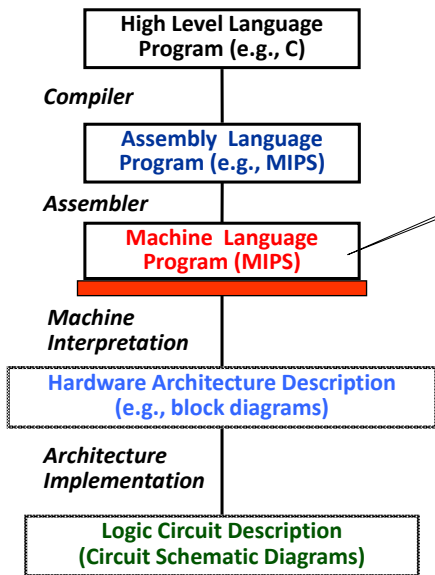
3 ISA & Data Path Revisited

4 Summary

Great Idea #4: Software and Hardware Co-design



How do we get the electrons to do the work?



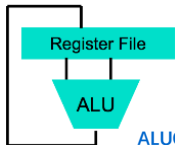
```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

Now, You
are Here.

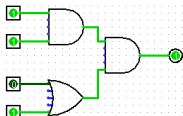
```
lw $t0, 0($2)  
lw $t1, 4($2)  
sw $t1, 0($2)  
sw $t0, 4($2)
```

Anything can be represented
as a *number*,
i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```



ALUOP[0:3] <= InstReg[9:11] & MASK



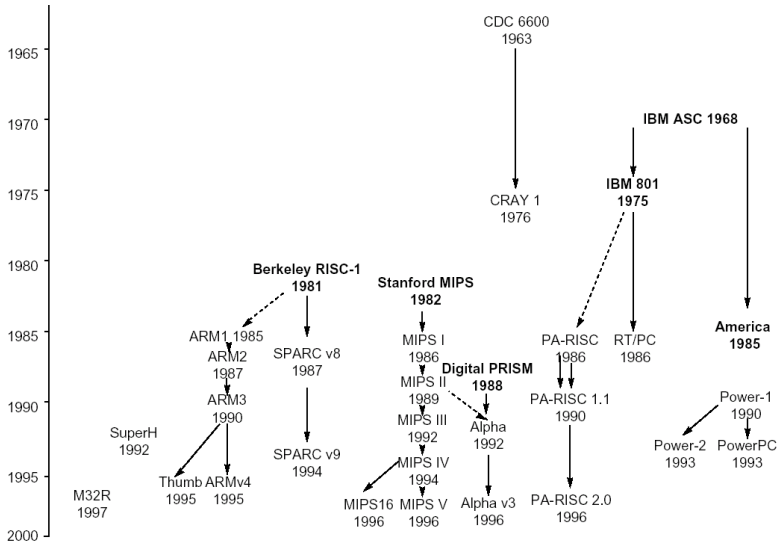
Instruction Set Architecture (ISA)

- Computer's native operations called **instructions**.
- Job of a CPU (Central Processing Unit, aka Core):
execute instructions
 - Instructions: CPU's primitives operations
 - Instructions performed one after another in sequence
 - Each instruction does a small amount of work (a tiny part of a larger program).
 - Each instruction has an operation applied to operands, and might be used change the sequence of instruction.
- Instruction set architecture (ISA) specifies the set of commands (instructions) a computer can execute
- Hardware registers provide a few very fast variables for instructions to operate on

Instruction Set Architecture (ISA)

- The instruction set defines all the valid instructions.
- CPUs belong to “families,” each implementing its own set of instructions
- CPU’ s particular set of instructions implements an Instruction Set Architecture (ISA)
- Examples:
 - ARM,
 - Intel x86
 - MIPS
 - RISC-V
 - IBM/Motorola PowerPC (old Mac)
 - Intel IA64,
 - ...

Instruction set architecture evolution



运算指令(Operate Instructions)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|----|----|----|----|----|----|-----|---|---|------|---|-----|---|---|---|---|
| ADD | 0 | 0 | 0 | 1 | DR | | SR1 | | 0 | 0 | 0 | SR2 | | | | |
| ADD | 0 | 0 | 0 | 1 | DR | | SR1 | | 1 | Imm5 | | | | | | |
| AND | 0 | 1 | 0 | 1 | DR | | SR1 | | 0 | 0 | 0 | SR2 | | | | |
| AND | 0 | 1 | 0 | 1 | DR | | SR1 | | 1 | Imm5 | | | | | | |
| NOT | 1 | 0 | 0 | 1 | DR | | SR1 | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Reserved | 1 | 1 | 0 | 1 | | | | | | | | | | | | |

控制指令(Control Instructions)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|------------|---|-----------|-------------|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | n | z | p | PCoffset9 | | | | | | | | |
| JSR | 0 | 1 | 0 | 0 | 1 | PCoffset11 | | | | | | | | | | |
| JSRR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RET | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| TRAP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | TrapVector8 | | | | | | | |

数据移动指令 (Data Movement Instructions)

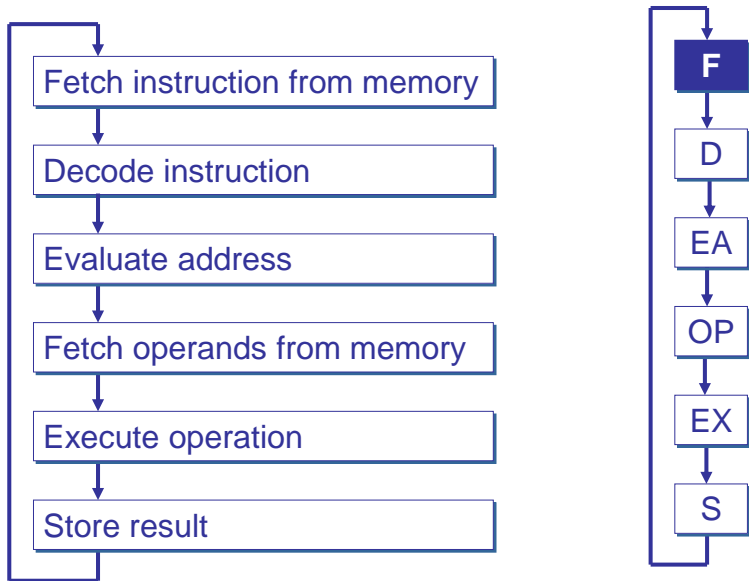
取数指令(Load)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|-----------|---|---|---|---|---|---|
| LD | 0 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LDR | 0 | 1 | 1 | 0 | DR | | | BaseR | | PCOffset6 | | | | | | |
| LDI | 1 | 0 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |
| LEA | 1 | 1 | 1 | 0 | DR | | | PCOffset9 | | | | | | | | |

存数指令(Store)

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|---|-----------|---|---|-----------|---|---|---|---|---|
| ST | 0 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |
| STR | 0 | 1 | 1 | 1 | SR | | | BaseR | | | PCOffset6 | | | | | |
| STI | 1 | 0 | 1 | 1 | SR | | | PCOffset9 | | | | | | | | |

Instruction Processing (state transtion)



Instruction Set Architecture (ISA) vs. Finite State Automata

A.3 The Instruction Set

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------------------|------|----|----|----|--------------|----|-----------|---|------------|---|---|---|---------|---|------|---|
| ADD ⁺ | 0001 | | | | DR | | | | SR1 | | | | 0 00 | | SR2 | |
| ADD ⁺ | 0001 | | | | DR | | | | SR1 | | | | 1 | | imm5 | |
| AND ⁺ | 0101 | | | | DR | | | | SR1 | | | | 0 00 | | SR2 | |
| AND ⁺ | 0101 | | | | DR | | | | SR1 | | | | 1 | | imm5 | |
| BR | 0000 | | | | n z p | | PCoffset0 | | | | | | | | | |
| JMP | 1100 | | | | 000 | | | | BaseR | | | | 000000 | | | |
| JSR | 0100 | | | | 1 | | | | PCoffset11 | | | | | | | |
| JSRR | 0100 | | | | 0 00 | | | | BaseR | | | | 000000 | | | |
| LD ⁺ | 0010 | | | | DR | | | | PCoffset0 | | | | | | | |
| LDI ⁺ | 1010 | | | | DR | | | | PCoffset0 | | | | | | | |
| LDR ⁺ | 0110 | | | | DR | | | | BaseR | | | | offset6 | | | |
| LEA ⁺ | 1110 | | | | DR | | | | PCoffset0 | | | | | | | |
| NOT ⁺ | 1001 | | | | DR | | | | SR | | | | 111111 | | | |
| RET | 1100 | | | | 000 | | | | 111 | | | | 000000 | | | |
| RTI | 1000 | | | | 000000000000 | | | | | | | | | | | |
| ST | 0011 | | | | SR | | | | PCoffset0 | | | | | | | |
| STI | 1011 | | | | SR | | | | PCoffset0 | | | | | | | |
| STR | 0111 | | | | SR | | | | BaseR | | | | offset6 | | | |
| TRAP | 1111 | | | | 0000 | | | | trapvect8 | | | | | | | |
| reserved | 1101 | | | | | | | | | | | | | | | |

Figure A.2 Format of the entire LC-3 instruction set. Note: + indicates instructions that modify condition codes

568

appendix c The Microarchitecture of the LC-3

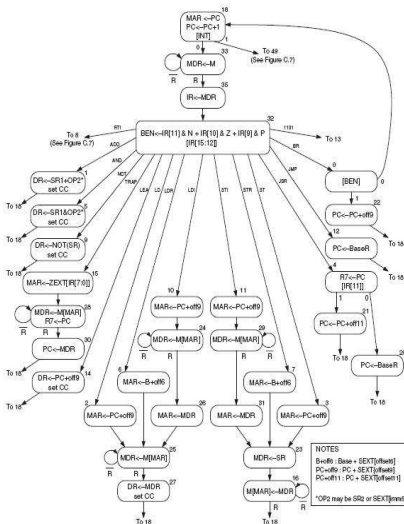
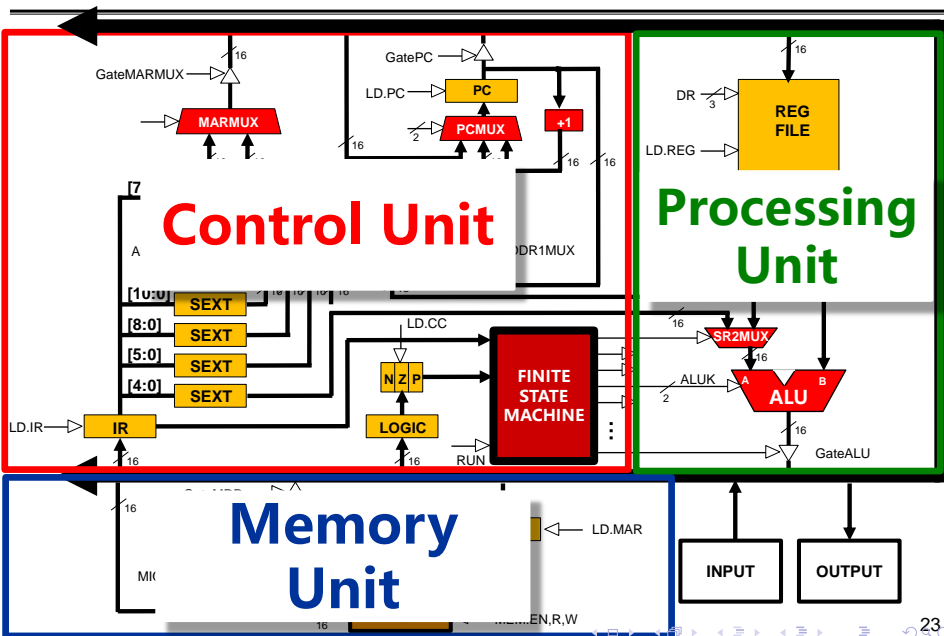
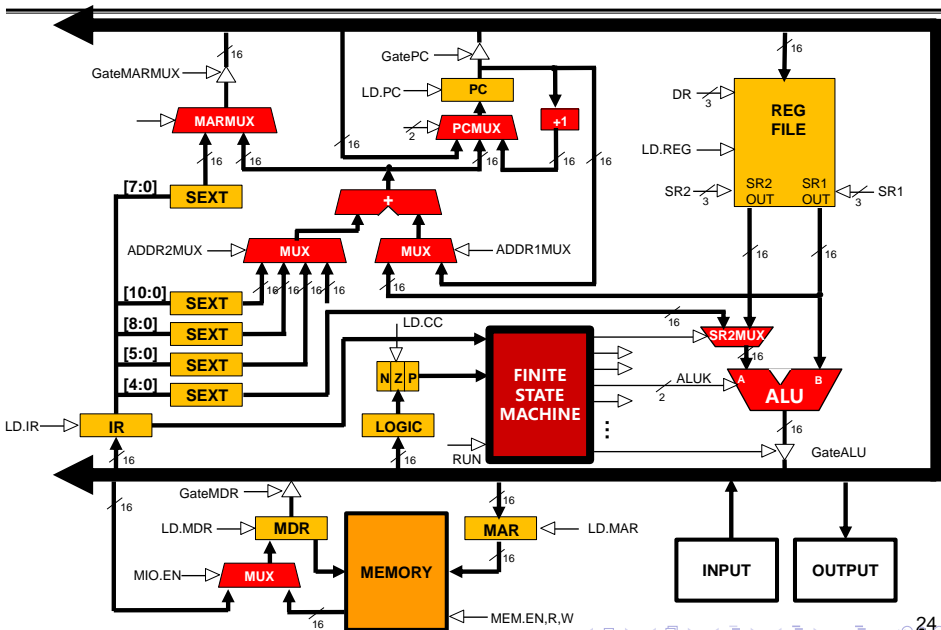


Figure C.2 A state machine for the LC-3

LC-3 Data Path



LC-3 Data Path



Data Path Components

Global bus

- special set of wires that carry a 16-bit signal to many components
- inputs to the bus are “tri-state devices,” that only place a signal on the bus when they are enabled
- only one (16-bit) signal should be enabled at any time
 - control unit decides which signal “drives” the bus
- any number of components can read the bus
 - register only captures bus data if it is write-enabled by the control unit

Memory

- Control and data registers for memory and I/O devices
- memory: MAR, MDR (also control signal for read/write)

Data Path Components

ALU

- **Accepts inputs from register file and from sign-extended bits from IR (immediate field).**
- **Output goes to bus.**
 - used by condition code logic, register file, memory

Register File

- **Two read addresses (SR1, SR2), one write address (DR)**
- **Input from bus**
 - result of ALU operation or memory read
- **Two 16-bit outputs**
 - used by ALU, PC, memory address
 - data for store instructions passes through ALU

Data Path Components

PC and PCMUX

- **Three inputs to PC, controlled by PCMUX**
 1. PC+1 - FETCH stage
 2. Address adder - BR, JMP
 3. bus - TRAP (discussed later)

MAR and MARMUX

- **Two inputs to MAR, controlled by MARMUX**
 1. Address adder - LD/ST, LDR/STR
 2. Zero-extended IR[7:0] -- TRAP (discussed later)

Data Path Components

Condition Code Logic

- Looks at value on bus and generates N, Z, P signals
- Registers set only when control unit enables them (LD.CC)
 - only certain instructions set the codes (ADD, AND, NOT, LD, LDI, LDR, LEA)

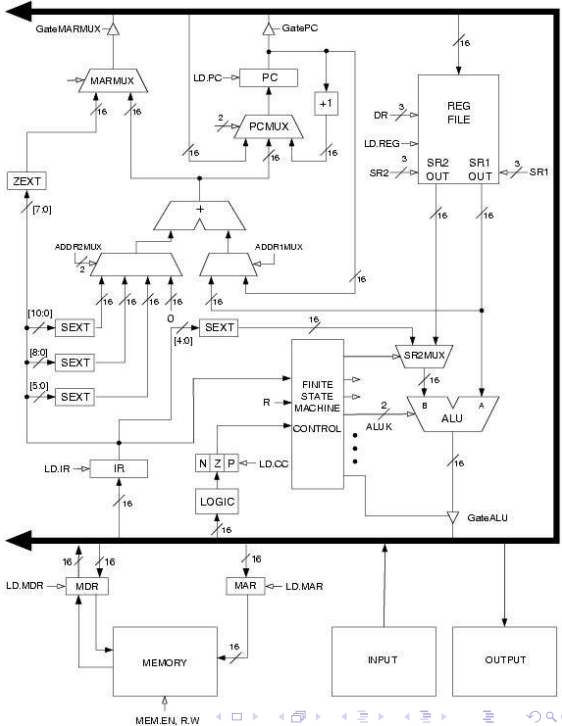
Control Unit – Finite State Machine

- On each machine cycle, changes control signals for next phase of instruction processing
 - who drives the bus? (GatePC, GateALU, ...)
 - which registers are write enabled? (LD.IR, LD.REG, ...)
 - which operation should ALU perform? (ALUK)
 - ...
- Logic includes decoder for opcode, etc.

LC-3 Data Path

Filled arrow
= info to be processed.

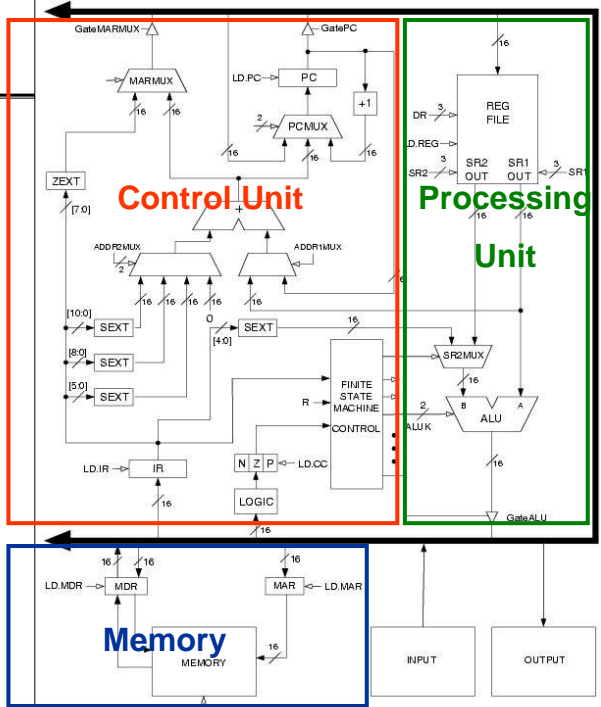
Unfilled arrow
= control signal.



LC-3 Data Path

Filled arrow
= info to be processed.

Unfilled arrow
= control signal.





1 Review

2 An Example: Counting Occurrences of a Character

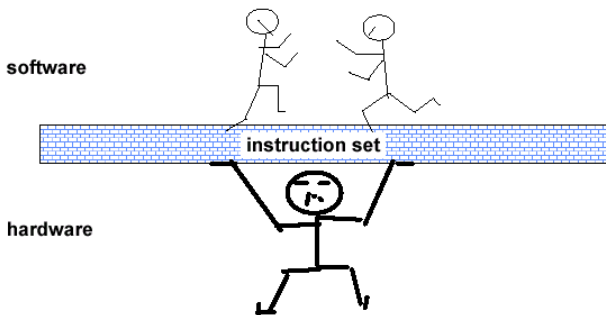
3 ISA & Data Path Revisited

4 Summary

Definition of computer architecture: classical definition

... the attributes of a [computing] system as seen by the programmer, *i.e.* **the conceptual structure and functional behavior**, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.

– Amdahl, Blaaw, and Brooks, 1964



Instruction Set Architecture: What does each cycle do?

Stage1:从存储系统中获得指令

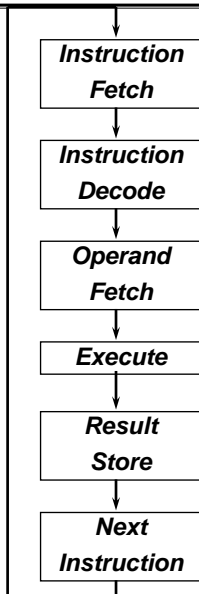
Stage2:确定做何动作

Stage3:获得操作数

Stage4:产生运算结果或状态

Stage5:向存储系统中存放运算结果

Stage6:确定下一条要执行的指令





中国科学技术大学
University of Science and Technology of China

计算机系统概论

Introduction to Computing Systems
(CS1002A.02)

Chapter 7

Assembly Language Program

计算机科学与技术学院
School of Computer Science and Technology



1 Review

2 Assembly Language Overview

3 Assembly Process

4 Summary



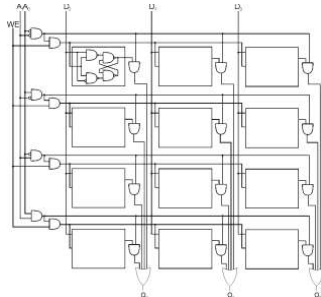
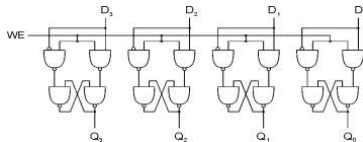
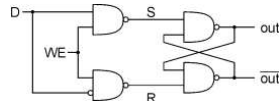
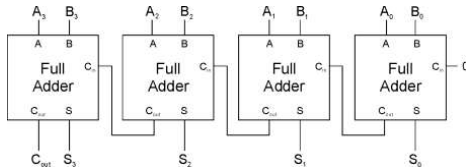
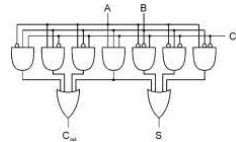
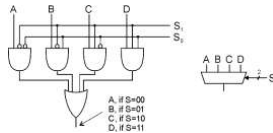
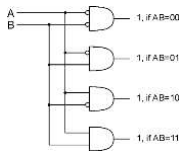
1 Review

2 Assembly Language Overview

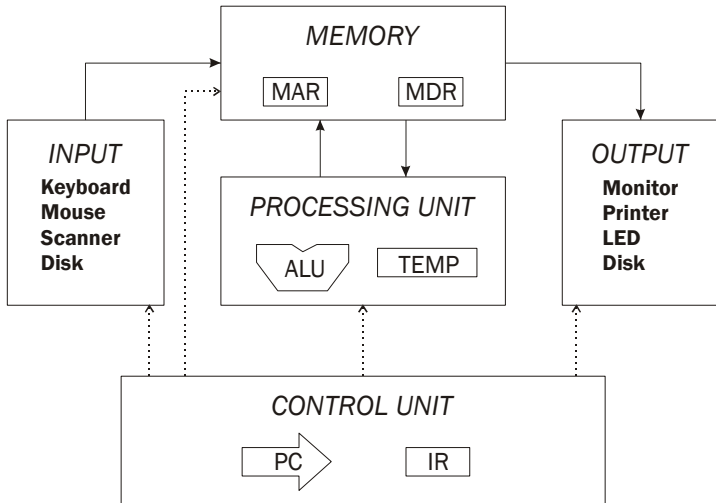
3 Assembly Process

4 Summary

Review: The Transistor & Basic Logical Structure



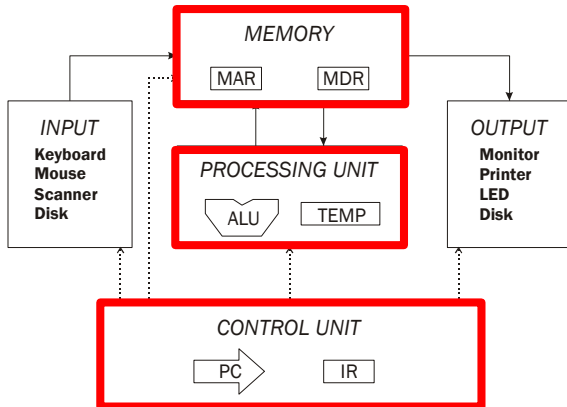
Review: Von Neumann Model



Review: Von Neumann Model

■ So far, we' ve learned how to:

- compute with values in registers
- load data from memory to registers
- store data from registers to memory



Review: The ISA

A.3 The Instruction Set

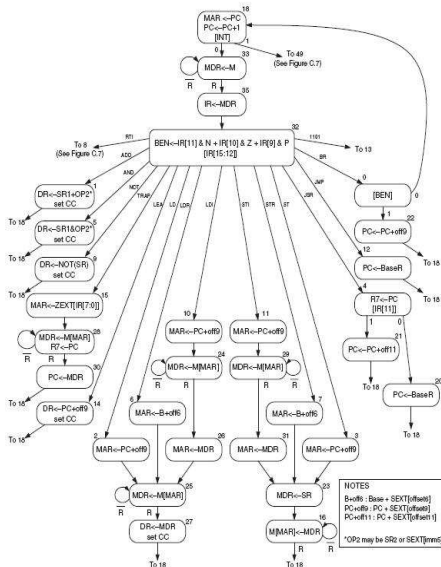
| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------------------|------|----|----|----|------|----|---|-------|---|---|----|---|------|---|---------|--------------|
| ADD ⁺ | 0001 | | | | DR | | | SR1 | | 0 | 00 | | | | SR2 | |
| ADD ⁺ | 0001 | | | | DR | | | SR1 | | 1 | | | imm5 | | | |
| AND ⁺ | 0101 | | | | DR | | | SR1 | | 0 | 00 | | | | SR2 | |
| AND ⁺ | 0101 | | | | DR | | | SR1 | | 1 | | | imm5 | | | |
| BR | 0000 | | | n | z | p | | | | | | | | | | PCoffset9 |
| JMP | 1100 | | | | 000 | | | BaseR | | | | | | | 000000 | |
| JSR | 0100 | | 1 | | | | | | | | | | | | | PCoffset11 |
| JSRR | 0100 | | 0 | | 00 | | | BaseR | | | | | | | 000000 | |
| LD ⁺ | 0010 | | | | DR | | | | | | | | | | | PCoffset9 |
| LDI ⁺ | 1010 | | | | DR | | | | | | | | | | | PCoffset9 |
| LDR ⁺ | 0110 | | | | DR | | | BaseR | | | | | | | offset8 | |
| LEA ⁺ | 1110 | | | | DR | | | | | | | | | | | PCoffset9 |
| NOT ⁺ | 1001 | | | | DR | | | SR | | | | | | | 111111 | |
| RET | 1100 | | | | 000 | | | 111 | | | | | | | 000000 | |
| RTI | 1000 | | | | | | | | | | | | | | | 000000000000 |
| ST | 0011 | | | | SR | | | | | | | | | | | PCoffset9 |
| STI | 1011 | | | | SR | | | | | | | | | | | PCoffset9 |
| STR | 0111 | | | | SR | | | BaseR | | | | | | | offset8 | |
| TRAP | 1111 | | | | 0000 | | | | | | | | | | | trapvect8 |
| reserved | 1101 | | | | | | | | | | | | | | | |

Figure A.2 Format of the entire LC-3 instruction set. Note: + indicates instructions that modify condition codes

Review: The State Machine(Turing Machine equivalent)

568

appendix c The Microarchitecture of the LC-3



Review: The Data Path(von Neumann Model)

appendix c The Microarchitecture of the LC-3

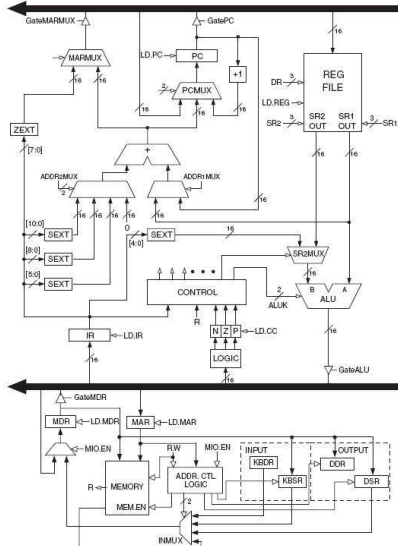


Figure C.3 The LC-3 data path



1 Review

2 Assembly Language Overview

3 Assembly Process

4 Summary

A LC-3 Program

| | | | | | | | | | | | | | | | | |
|-------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X4101 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| X4102 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| X4103 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| X4104 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| X4105 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| X4106 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| X4107 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| X4108 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| X4109 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| X410A | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| X410B | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| X410C | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| X410D | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| X410E | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| X410F | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| X4110 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| X4101 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| X4102 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| X4103 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| X4104 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

| | | | | | | | | | | | | | | | | | | |
|-------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| X8001 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | |
| X8002 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | |
| X8003 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | |
| X8004 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | |
| X8005 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | |
| X8006 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | |
| X8007 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | |
| X8008 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | |
| X8009 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | |
| X800A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| X800B | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| X800C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| X800D | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | |
| X800E | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | |
| X800F | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | |
| X8010 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | |
| X8011 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| X8012 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | |
| X8013 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | |
| X8014 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| X8015 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| X8016 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| X8017 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | |
| X8018 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Human-Readable Machine Language

■ Computers like ones and zeros...

0001110010000110

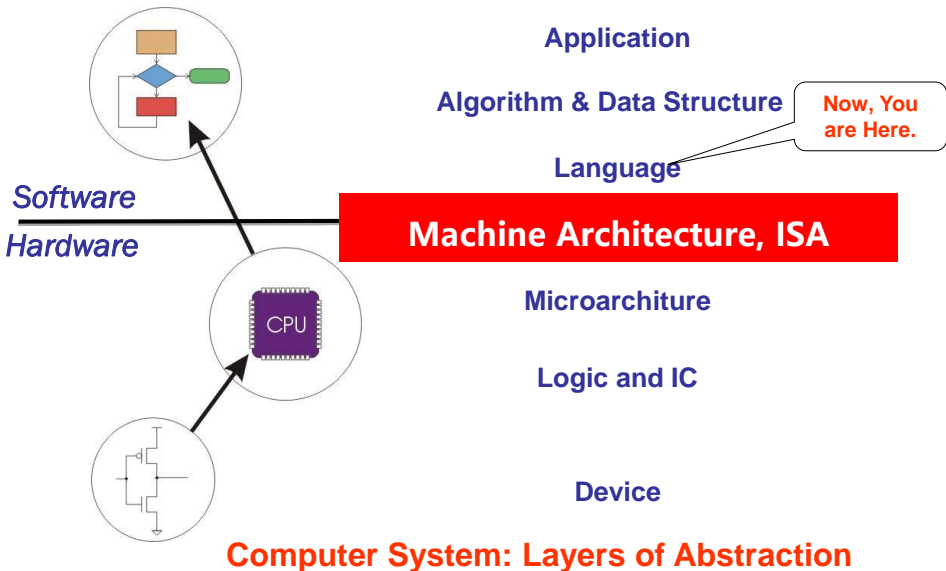
■ Humans like symbols...

ADD R6,R2,R6 ; *increment index reg.*
or
 $C = a + b;$

■ **Assembler** is a program that turns symbols into machine instructions.

- ISA-specific: close correspondence between symbols and instruction set
 - mnemonics for opcodes
 - labels for memory locations
- additional operations for allocating storage and initializing data

Great Idea #4: Software and Hardware Co-design



Great Idea #3: Abstraction Helps Us Manage Complexity

Application

Algorithm and Data Structure

Programming Language/Compiler

Operating System/Virtual Machines

Instruction Set Architecture (ISA)

Microarchitecture

Gates/Register-Transfer Level (RTL)

Analog/Digital Circuits

Electronic Devices

Physics

Solve a system of equations

Red-black SOR Gaussian elimination Jacobi iteration Multigrid

FORTRAN C C++ Java

Sun SPARC Intel x86 IBM PowerPC

Pentium 4 Core 2 Duo AMD Athlon X2

Ripple-carry adder Carry-lookahead adder

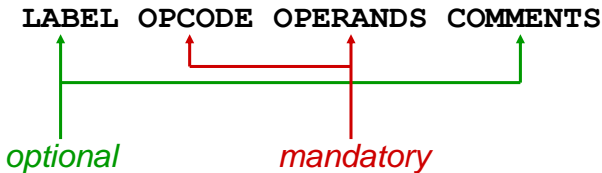
Static CMOS Dynamic CMOS Nanomechanical

An Assembly Language Program

```
;
; Program to multiply a number by the constant 6
;
        .ORIG    x3050
        LD       R1, SIX
        LD       R2, NUMBER
        AND      R3, R3, #0      ; Clear R3.  It will
                                ; contain the product.
; The inner loop
;
AGAIN  ADD      R3, R3, R2
        ADD      R1, R1, #-1     ; R1 keeps track of
        BRp     AGAIN          ; the iteration.
;
        HALT
;
NUMBER .BLKW    1
SIX    .FILL    x0006
;
        .END
```

LC-3 Assembly Language Syntax

- Each line of a program is one of the following:
 - an instruction
 - an assembler directive (or pseudo-op)
 - a comment
- Whitespace (between symbols) and case are ignored.
- Comments (beginning with “;”) are also ignored.
- An instruction has the following format:



Opcodes and Operands

■ Opcodes

- reserved symbols that correspond to LC-3 instructions
- listed in Appendix A
 - ex: ADD, AND, LD, LDR, ...

■ Operands

- registers -- specified by Rn, where n is the register number
- numbers -- indicated by # (decimal) or x (hex)
- label -- symbolic name of memory location
- separated by comma
- number, order, and type correspond to instruction format

```
-ex:  ADD    R1,R1,R3
      ADD    R1,R1,#3
      LD     R6,NUMBER
4     BRz    LOOP
```


Assembler Directives

■ Pseudo-operations

- do not refer to operations executed by program
- used by assembler
- look like instruction, but "opcode" starts with dot

| <i>Opcode</i> | <i>Operand</i> | <i>Meaning</i> |
|-----------------|---------------------------|---|
| .ORIG | address | starting address of program |
| .END | | end of program |
| .BLKW | n | allocate n words of storage |
| .FILL | n | allocate one word, initialize with value n |
| .STRINGZ | n-character string | allocate n+1 locations, initialize w/characters and null terminator |

Example

```
.ORIG    X3010  
HELLO    .STRINGZ    " Hello, World! "
```

```
x3010:  x0048  
x3011:  x0065  
x3012:  x006C  
x3013:  x006C  
x3014:  x006F  
x3015:  x002C  
x3016:  x0020  
x3017:  x0057  
x3018:  x006F  
x3019:  x0072  
x301A:  x006C  
x301B:  x0064  
x301C:  x0021  
x301D:  x0000
```

Trap Codes

- LC-3 assembler provides “pseudo-instructions” for each trap code, so you don’t have to remember them.

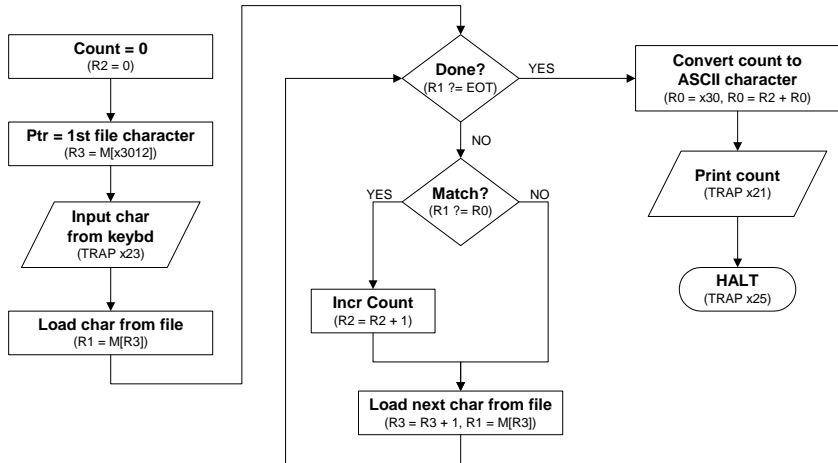
| <i>Code</i> | <i>Equivalent</i> | <i>Description</i> |
|-------------|-------------------|---|
| HALT | TRAP x25 | Halt execution and print message to console. |
| IN | TRAP x23 | Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0]. |
| OUT | TRAP x21 | Write one character (in R0[7:0]) to console. |
| GETC | TRAP x20 | Read one character from keyboard. Character stored in R0[7:0]. |
| PUTS | TRAP x22 | Write null-terminated string to console. Address of string is in R0. |

Style Guidelines

- **Use the following style guidelines to improve the readability and understandability of your programs:**
 1. Provide a program header, with author's name, date, etc., and purpose of program.
 2. Start labels, opcode, operands, and comments in same column for each line. (Unless entire line is a comment.)
 3. Use comments to explain what each register does.
 4. Give explanatory comment for most instructions.
 5. Use meaningful symbolic names.
 - Mixed upper and lower case for readability.
 - ASCIItoBinary, InputRoutine, SaveR1
 6. Provide comments between program sections.
 7. Each line must fit on the page -- no wraparound or truncations.
 - Long statements split in aesthetically pleasing manner.

Sample Program

- Count the occurrences of a character in a file.
Remember this?



Program (1 of 2)

| Address | Instruction | | | | | | | | | | | | | | | | Comments |
|---------|-------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x3000 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | $R2 \leftarrow 0$ (counter) AND R2,R2,#0 |
| x3001 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | $R3 \leftarrow M[x3012]$ (ptr) LD R3, x3012 (LD R3, PTR) |
| x3002 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | Input to R0 (TRAP x23) TRAP x23 (GETC) |
| x3003 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | $R1 \leftarrow M[R3]$ LDR R1, R3, #0 |
| x3004 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | $R4 \leftarrow R1 - 4$ (EOT) ADD R4,R1,#-4 |
| x3005 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | If Z, goto x300E BRz x300E (BRz OUTPUT) |
| x3006 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | $R1 \leftarrow \text{NOT } R1$ NOT R1,R1 |
| x3007 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | $R1 \leftarrow R1 + 1$ ADD R1,R1,#1 |
| x3008 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | $R1 \leftarrow R1 + R0$ ADD R1,R1,R0 |
| x3009 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | If N or P, goto x300B BRnp x300B (BRnp GETCHAR) |

Program (2 of 2)

| Address | Instruction | | | | | | | | | | | | Comments | | | | |
|---------|-------------|---|---|---|---|---|---|---|---|---|---|---|----------|---|---|---|---|
| x300A | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | $R2 \leftarrow R2 + 1$ <i>ADD R2,R2,#1</i> |
| x300B | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | $R3 \leftarrow R3 + 1$ <i>ADD R3,R3,#1</i> |
| x300C | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | $R1 \leftarrow M[R3]$ <i>LDR R1,R3,#0</i> |
| x300D | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | <i>Goto x3004</i> <i>BRnzp x3004 (BRnzp TEST)</i> |
| x300E | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | $R0 \leftarrow M[x3013]$ <i>LD R0,x3013 (LD R0, ASCII)</i> |
| x300F | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | $R0 \leftarrow R0 + R2$ <i>ADD R0,R0,R2</i> |
| x3010 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | <i>Print R0</i> <i>TRAP x21 (OUT)</i> |
| x3011 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | <i>HALT</i> <i>TRAP x25 (HALT)</i> |
| X3012 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Starting Address of File (X9000) |
| x3013 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | <i>ASCII x30 ('0')</i> |

Char Count in Assembly Language (1 of 3)

```
01 ;
02 ; Program to count occurrences of a character in a file.
03 ; Character to be input from the keyboard.
04 ; Result to be displayed on the monitor.
05 ; Program only works if no more than 9 occurrences are
06 ; found.
07 ;
08 ; Initialization
09 ;
0A      .ORIG    x3000
0B      AND      R2, R2, #0          ; R2 is counter, initially 0
0C      LD        R3, PTR            ; R3 is pointer to characters
0D      GETC                      ; TRAP x23
0E                      ; R0 gets character input
0F      LDR       R1, R3, #0         ; R1 gets first character
10 ;
11 ; Test character for end of file
12 ;
13 TEST ADD      R4, R1, #-4          ; Test for EOT
14                      ; (ASCII x04)
15      BRz       OUTPUT              ; If done, prepare the output
```

Char Count in Assembly Language (2 of 3)

```
16 ;  
17 ; Test character for match.  If a match, increment count.  
18 ;  
19     NOT    R1, R1  
1A     ADD    R1, R1, R0 ; If match, R1 = xFFFF  
1B     NOT    R1, R1     ; If match, R1 = x0000  
1C     BRnp   GETCHAR    ; If no match, do not increment  
1D     ADD    R2, R2, #1  
1E ;  
1F ; Get next character from file.  
20 ;  
21 GETCHAR ADD    R3, R3, #1 ; Point to next character.  
22         LDR    R1, R3, #0 ; R1 gets next char to test  
23         BRnzp  TEST  
24 ;  
25 ; Output the count.  
26 ;  
27 OUTPUT LD      R0, ASCII ; Load the ASCII template  
28         ADD    R0, R0, R2 ; Covert binary count to ASCII  
29         OUT  
2A         ; ASCII code in R0 is displayed.  
2B         HALT    ; TRAP x25,Halt machine
```

Char Count in Assembly Language (3 of 3)

```
2C ;  
2D ; Storage for pointer and ASCII template  
2E ;  
2F ASCII .FILL    x0030  
30 PTR   .FILL    x9000  
31      .END
```



1 Review

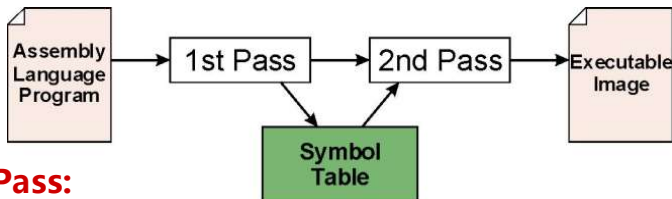
2 Assembly Language Overview

3 Assembly Process

4 Summary

Assembly Process

- Convert assembly language file (.asm) into an executable file (.obj) for the LC-3 simulator.



■ First Pass:

- scan program file
- find all labels and calculate the corresponding addresses; this is called the symbol table

■ Second Pass:

- convert instructions to machine language, using information from symbol table

First Pass: Constructing the Symbol Table

1. Find the `.ORIG` statement,
which tells us the address of the first instruction.
 - Initialize location counter (LC), which keeps track of the current instruction.
2. For each non-empty line in the program:
 - a) If line contains a label, add label and LC to symbol table.
 - b) Increment LC.
 - NOTE: If statement is `.BLKW` or `.STRINGZ`, increment LC by the number of words allocated.
3. Stop when `.END` statement is reached. **Right?**

- NOTE: A line that contains only a comment is considered an empty line.

Practice

- Construct the symbol table for the program in Figure 7.2

| Symbol | Address |
|--------|---------|
| | |
| | |
| | |
| | |
| | |

Practice

- Construct the symbol table for the program in Figure 7.2

| Symbol | Address |
|---------|---------|
| TEST | X3004 |
| GETCHAR | X300B |
| OUTPUT | X300E |
| ASCII | X3012 |
| PTR | X3013 |

Second Pass: Generating Machine Language

- For each executable assembly language statement, generate the corresponding machine language instruction.
 - If operand is a label, look up the address from the symbol table.
- Potential problems:
 - Improper number or type of arguments
 - ex: NOT R1,#7
 ADD R1,R2
 ADD R3,R3,NUMBER
 - Immediate argument too large
 - ex: ADD R1,R2,#1023
 - Address (associated with label) not on the same page
 - can't use direct addressing mode

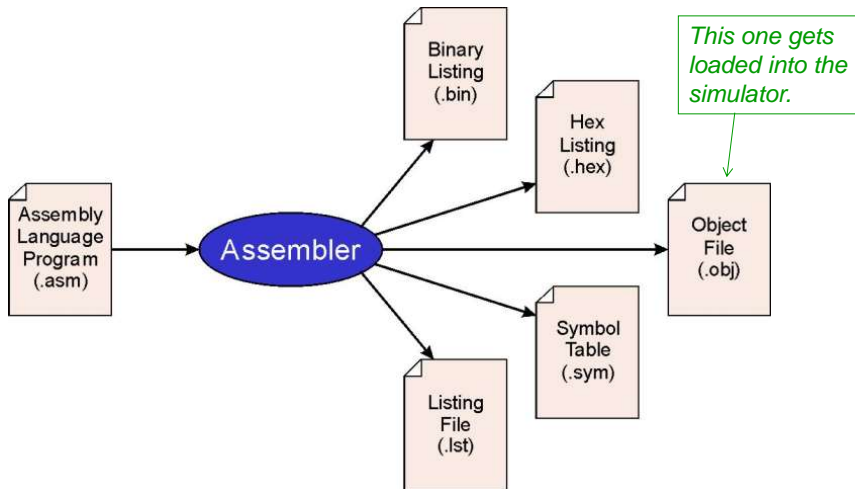
Practice

- Using the symbol table constructed earlier, translate these statements into LC-3 machine language.
- *(Assume all addresses are on the current page.)*

| Statement | Machine Language |
|-------------------|------------------|
| LD R3 , PTR | |
| ADD R4 , R1 , #-4 | |
| LDR R1 , R3 , #0 | |
| BRnp GETCHAR | |

LC-3 Assembler

- Using “assemble” (Unix) or LC3 Edit (Windows), generates several different output files.



Object File Format

■ LC-3 object file contains

- Starting address (location where program must be loaded), followed by...
- Machine instructions

■ Example

- Beginning of “count character” object file looks like this:

| | |
|------------------|------------------|
| 0011000000000000 | ← .ORIG x3000 |
| 0101010010100000 | ← AND R2, R2, #0 |
| 0010011000010100 | ← LD R3, PTR |
| 1111000000100011 | ← TRAP x23 |
| · | |
| · | |
| · | |

Multiple Object Files

- An object file is not necessarily a complete program.
 - system-provided library routines
 - code blocks written by multiple developers
- For LC-3, can load multiple object files into memory, then start executing at a desired address.
 - system routines, such as keyboard input, are loaded automatically
 - loaded into “system memory,” below x1000
 - by convention, user code should be loaded between x3000 and xCFFF
 - each object file includes a starting address
 - be careful not to load overlapping object files



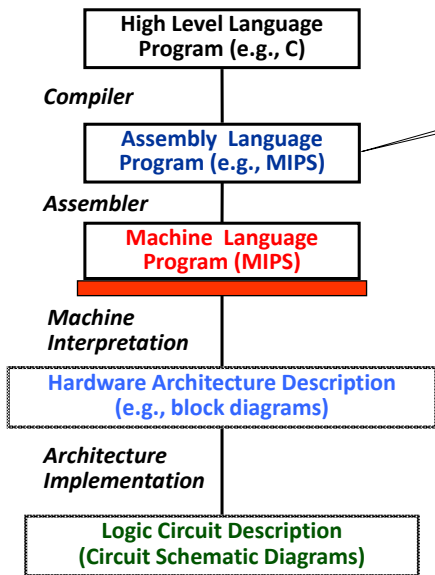
1 Review

2 Assembly Language Overview

3 Assembly Process

4 Summary

Summary: Assembly Language



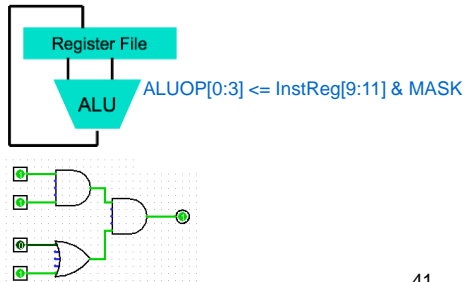
```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

Now, You
are Here.

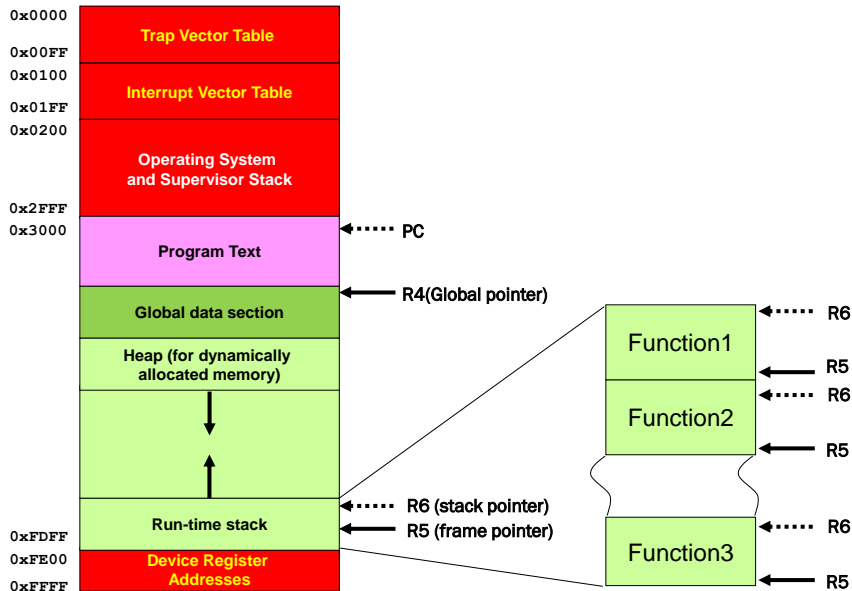
```
lw $t0, 0($2)  
lw $t1, 4($2)  
sw $t1, 0($2)  
sw $t0, 4($2)
```

Anything can be represented
as a *number*,
i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```



Memory map of the LC-3





中国科学技术大学
University of Science and Technology of China

计算系统概论A

Introduction to Computing Systems
(CS1002A.02)

Chapter 8-1 Subroutines

计算机科学与技术学院
School of Computer Science and Technology



1 Review

2 Subroutines

3 Control Instructions for Subroutines

4 Summary

Abstract Data Types: Data Structures

- Up to now, we have processed a single value
 - an integer
 - an ASCII character
- The information in the real world is far more complex than simple, single numbers. We call these complex items of information **abstract data types**, or more colloquially **data structures**, E.g.
 - a company's organization chart
 - a list of items arranged in alphabetical order
- In this chapter, we will study three abstract data types:
 - stacks
 - queues
 - and character strings

Abstract Data Types: Data Structures

- We will write programs to solve problems that require expressing information according to its structure.
- Before we get to **stacks, queues, and character strings**, however, we introduce a new concept that will prove very useful in manipulating data structures: **subroutines**, or what is also called **functions**.

Subroutines

■ A subroutine is a program fragment that. . .

- Resides in **user** space (i.e, not in OS)
- Performs a **well-defined task**
- Is invoked (called) **multiple times** by a user program
- **Returns control** to the calling program when finished

■ Virtues

- **Reuse code** without re-typing it (and debugging it!)
- Divide task into parts (or among multiple programmers)
- Use vendor-supplied **library** of useful routines that one software engineer writes a program that requires such fragments and another software engineer writes the fragments.
 - math library
 - square root, sine, and arctangent, etc.

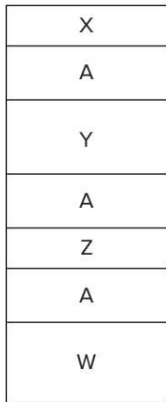
A simple illustration of a part of a program

```
01 START    ST R1,SaveR1                ; Save registers needed
02          ST R2,SaveR2                ; by this routine
03          ST R3,SaveR3
04 ;
05          LD R2,Newline
06 L1        LDI R3,DSR
07          BRzp L1                      ; Loop until monitor is ready
08          STI R2,DDR                    ; Move cursor to new clean line
09 ;
0A          LEA R1,Prompt                ; Starting address of prompt string
0B Loop     LDR R0,R1,#0                 ; Write the input prompt
0C          BRz Input                    ; End of prompt string
0D L2        LDI R3,DSR
0E          BRzp L2                      ; Loop until monitor is ready
0F          STI R0,DDR                    ; Write next prompt character
10          ADD R1,R1,#1                 ; Increment prompt pointer
11          BRnzp Loop                   ; Get next prompt character
12 ;
13 Input     LDI R3,KBSR
14          BRzp Input                    ; Poll until a character is typed
15          LDI R0,KBDR                    ; Load input character into R0
16 L3        LDI R3,DSR
17          BRzp L3                      ; Loop until monitor is ready
18          STI R0,DDR                    ; Echo input character
19 ;
1A L4        LDI R3,DSR
1B          BRzp L4                      ; Loop until monitor is ready
1C          STI R2,DDR                    ; Move cursor to new clean line
1D          LD R1,SaveR1                 ; Restore registers
1E          LD R2,SaveR2                 ; to original values
1F          LD R3,SaveR3
20          JMP R7 ; Do the program's next task
```

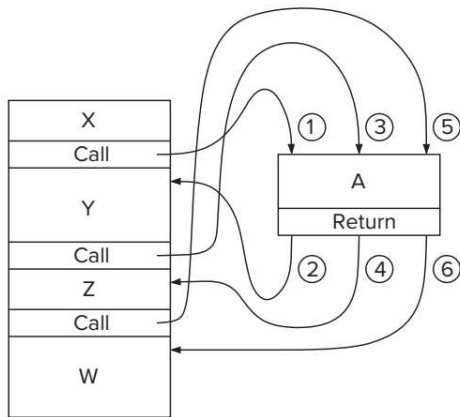
A simple illustration of a part of a program

```
21 ;  
22 SaveR1 .BLKW 1 ; Memory for registers saved  
23 SaveR2 .BLKW 1  
24 SaveR3 .BLKW 1  
25 DSR .FILL xFE04  
26 DDR .FILL xFE06  
27 KBSR .FILL xFE00  
28 KBDR .FILL xFE02  
29 Newline .FILL x000A ; ASCII code for newline  
2A Prompt .STRINGZ ``Input a character>''
```

The Call/Return Mechanism



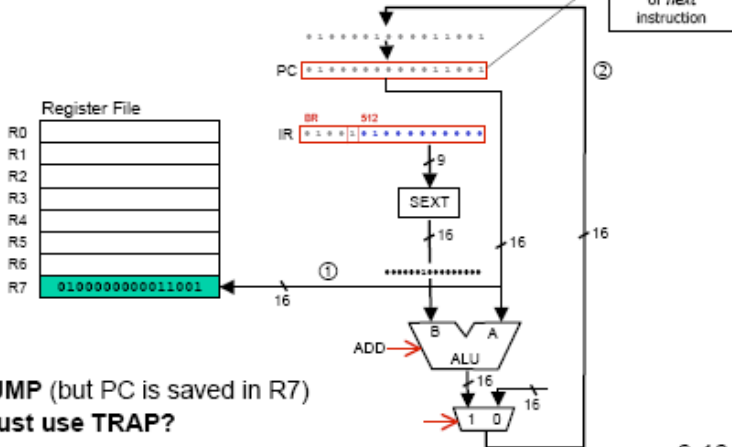
(a) Without subroutines



(b) With subroutines

Control Instructions for Subroutines

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|------------|---|-----------|-------------|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | n | z | p | PCOffset9 | | | | | | | | |
| JSR | 0 | 1 | 0 | 0 | 1 | PCOffset11 | | | | | | | | | | |
| JSRR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RET | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | | | | | | | | | | | | | | |
| TRAP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | TrapVector8 | | | | | | | |



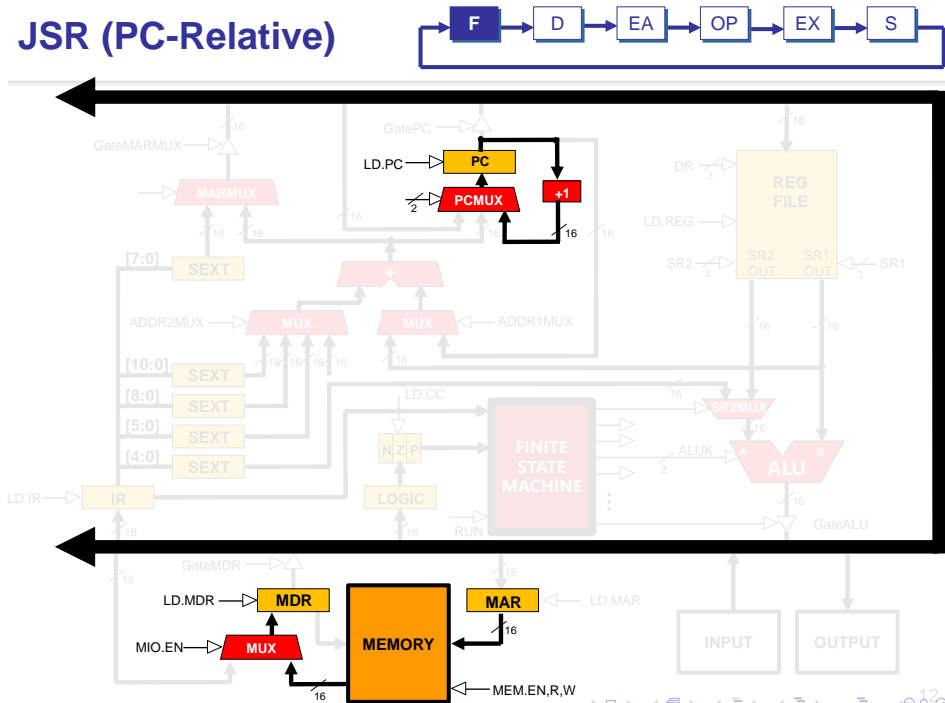
Just like JMP (but PC is saved in R7)
Why not just use TRAP?

CSF 240

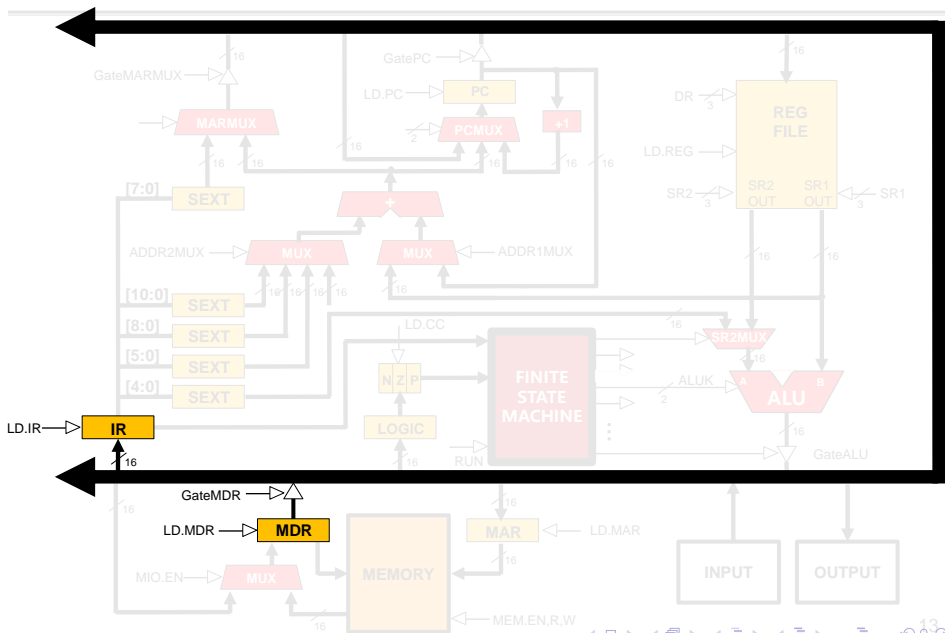
```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



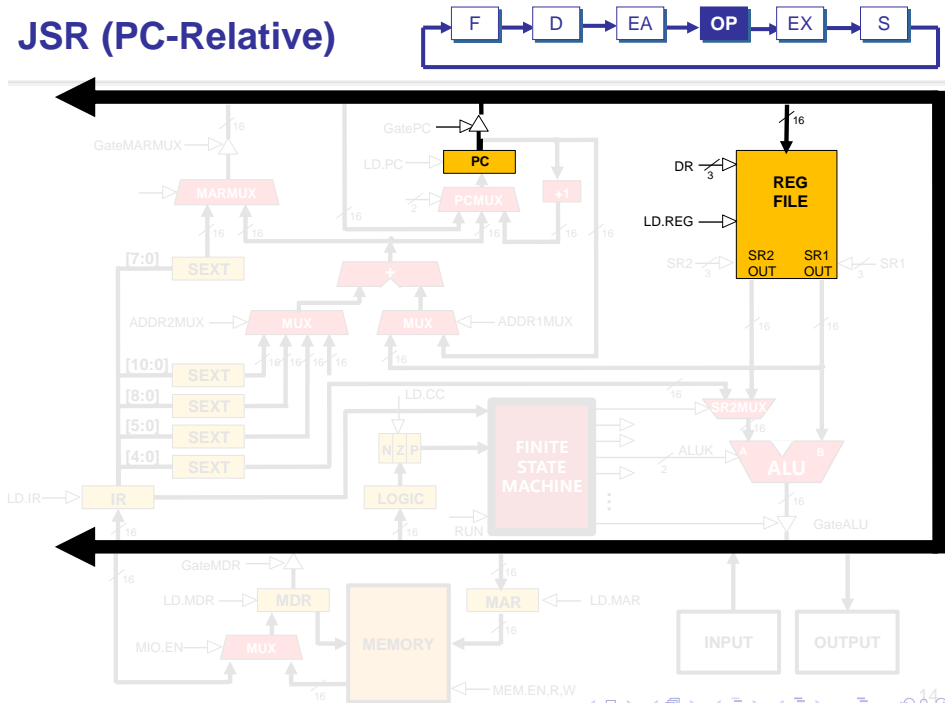
JSR (PC-Relative)



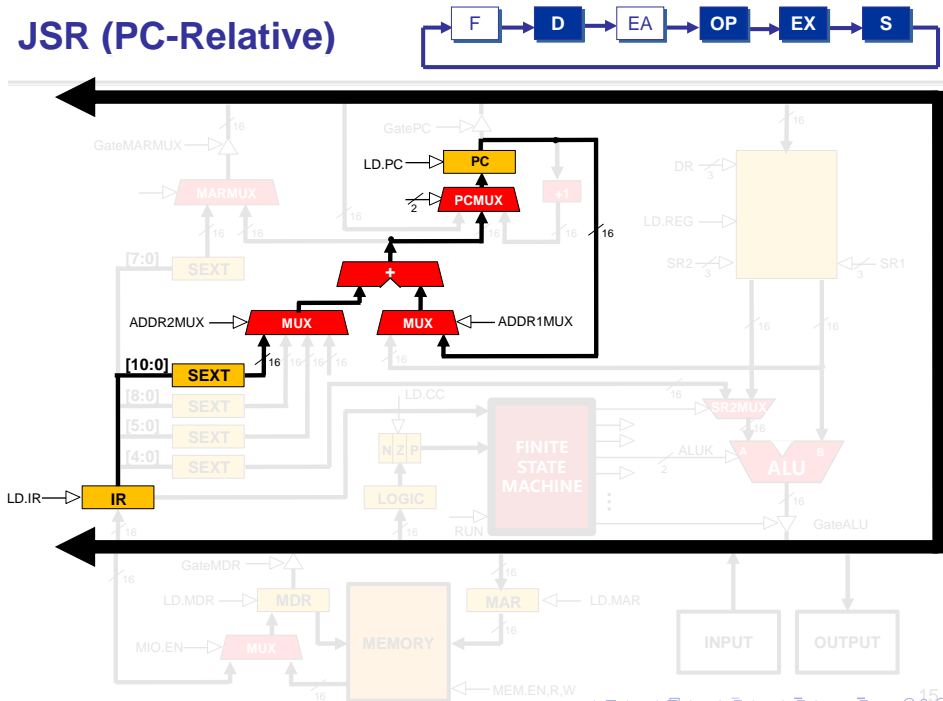
JSR (PC-Relative)



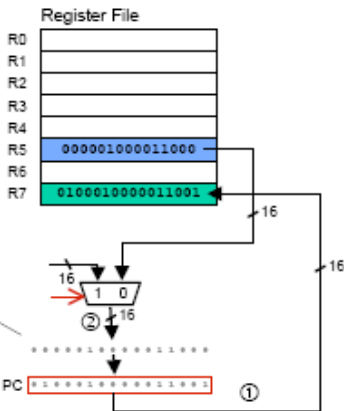
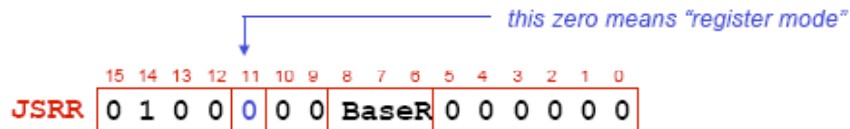
JSR (PC-Relative)



JSR (PC-Relative)

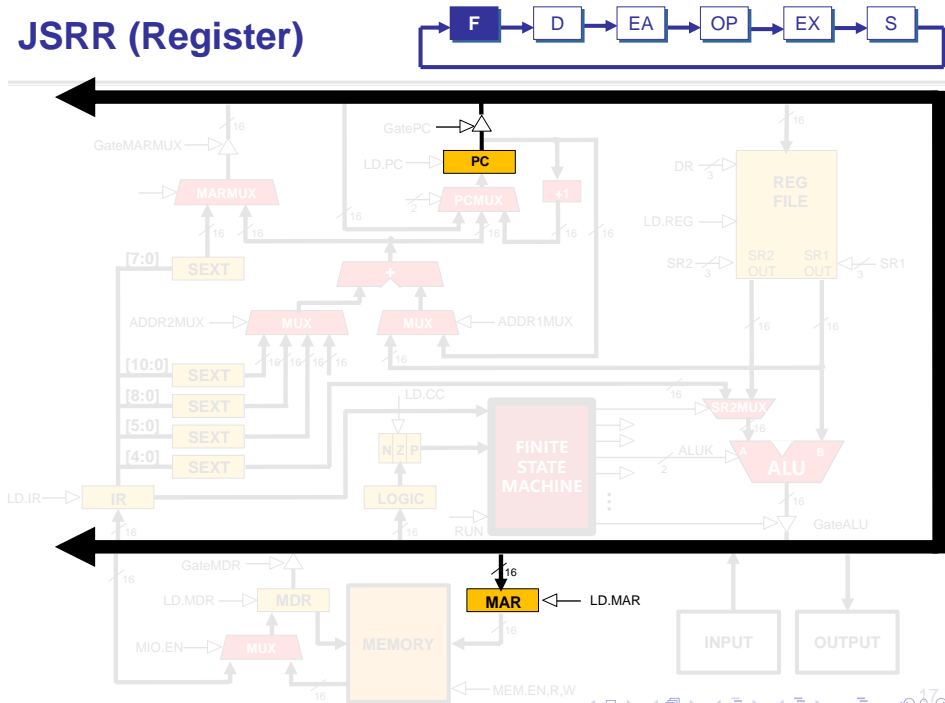


JSRR



Virtues of JSRR?

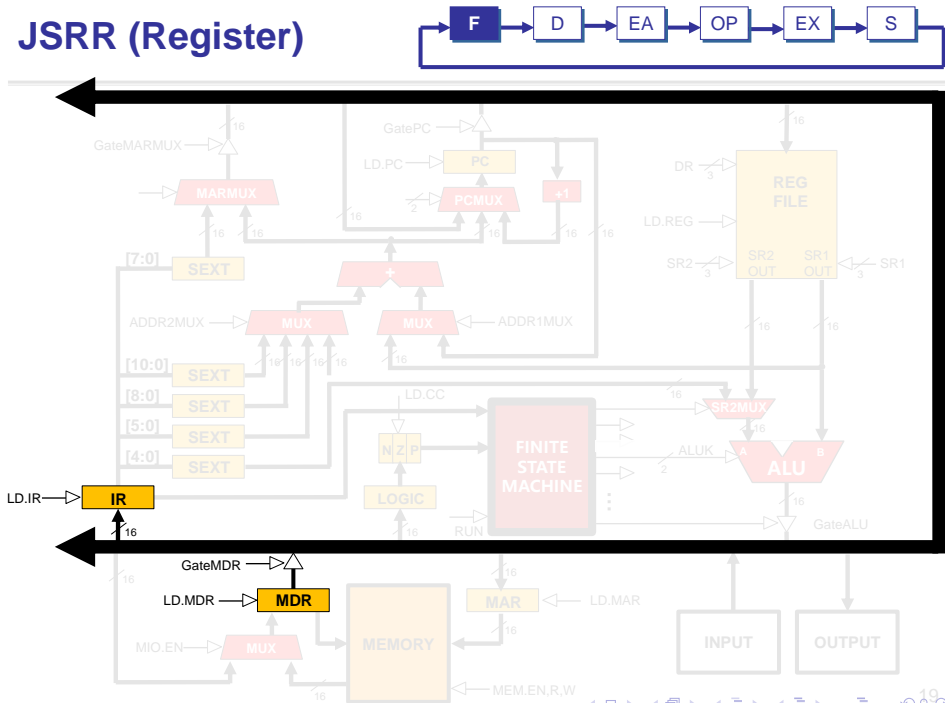
JSRR (Register)



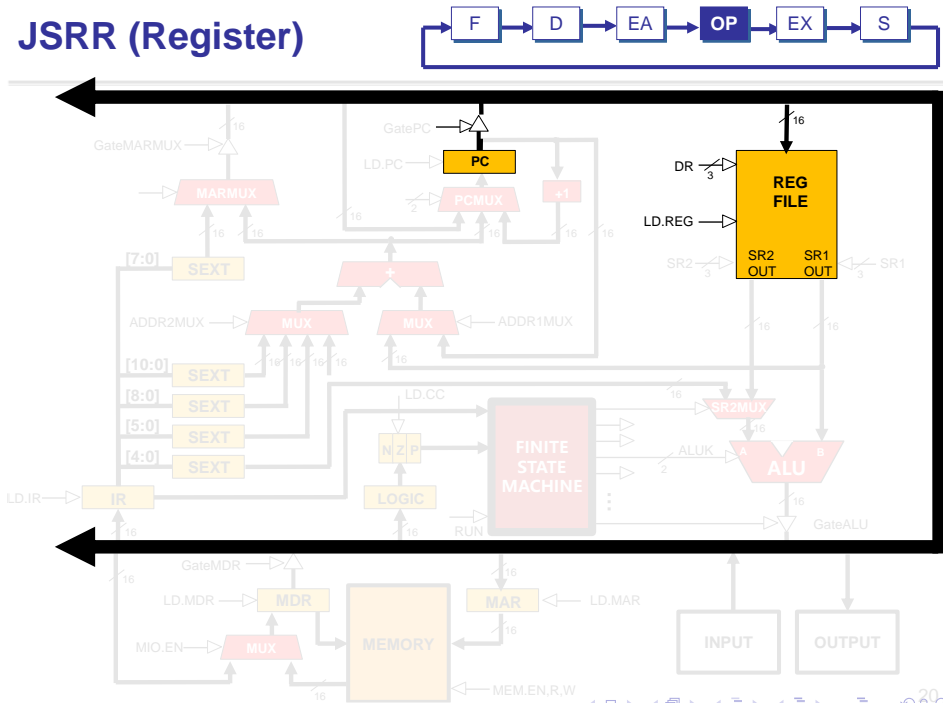
```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



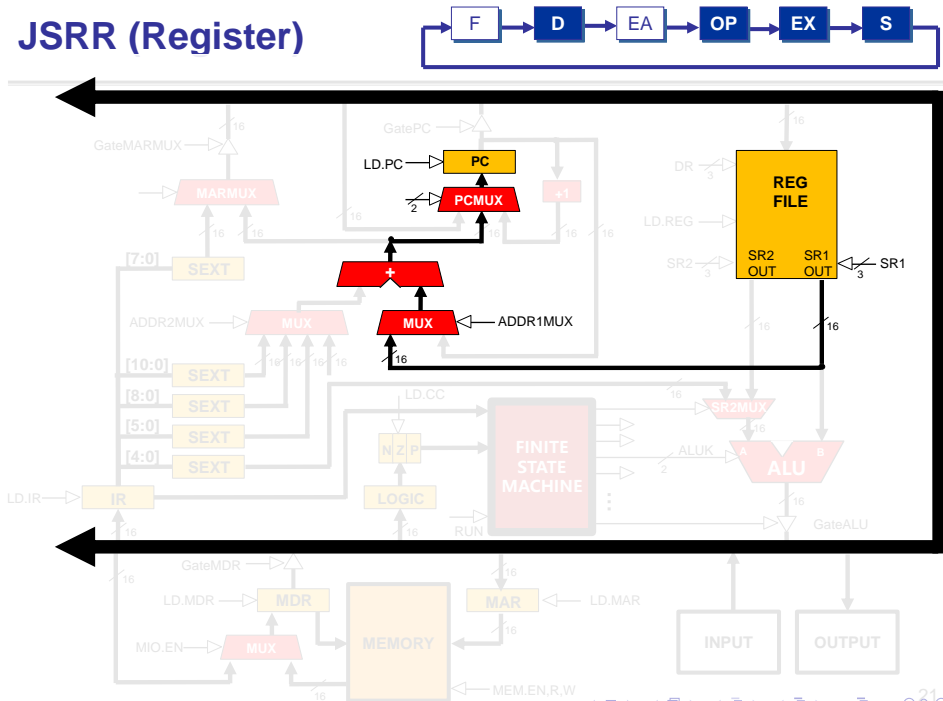
JSRR (Register)



JSRR (Register)



JSRR (Register)



RET instruction

■ RET – return instruction

- How to return

- Place address in R7 in PC, Return the execution to the last calling point.

- $PC \leftarrow (R7)$

RET
(JMP R7)

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Example: Negate the value in R0

```
TwosComp    NOT    R0,R0        ;flip bits
            ADD    R0,R0,#1     ;add one
            RET                     ;return to caller
```

To call from a program

```
;need to compute R4 = R1-R3
            ADD    R0,R3,#0     ;copy R3 to R0
            JSR    TwosComp     ;negate
            ADD    R4,R1,R0     ;add to R1
            ...
```

Using Subroutines

■ Programmer must know

- **Address:** or at least a label that will be bound to its address
- **Function:** what it does
 - NOTE: The programmer does not need to know *how* the subroutine works, but what changes are visible in the machine's state after the routine has run
- **Arguments:** what they are and where they are placed
- **Return values:** what they are and where they are placed

Passing Information To Subroutines

■ Argument(s)

- Value **passed in** to a subroutine is called an argument
- This is a value needed by the subroutine to do its job
- Examples
 - TwosComp: R0 is number to be negated
 - OUT: R0 is character to be printed
 - PUTS: R0 is *address* of string to be printed

■ How?

- In registers (simple, fast, but limited number)
- In memory (many, but awkward, expensive)
- Both

Getting Values From Subroutines

■ Return Values

- A value **passed out** of a subroutine is called a return value
- This is the value that you called the subroutine to compute
- Examples
 - TwosComp: negated value is returned in R0
 - GETC: character read from the keyboard is returned in R0

■ How?

- Registers, memory, or both
- Single return value in register most common

Saving and Restore Registers

■ Like service routines, must save and restore registers

- Who saves what is part of the calling convention

■ Generally use “callee-save” strategy, except for return values

- Same as trap service routines
- Save anything that subroutine alters internally that shouldn't be visible when the subroutine returns
- Restore incoming arguments to original values (unless overwritten by return value)

■ Remember

- You MUST save R7 if you call any other subroutine or trap
- Otherwise, you won't be able to return!

Subroutine Template

```
01 SUB_NAME
02     ;Register Saving
03     ST R0, SUB_R0
04     ST R1, SUB_R1
05     ...
06     ST R6, SUB_R6
07     ST R7, SUB_R7;Return address
08
09     ;***Code***
10
11     ;Register Restoring
12     LD R0, SUB_R0
13     LD R1, SUB_R1
14     ...
15     LD R6, SUB_R6
16     LD R7, SUB_R7      ;Return address
17     RET
```



中国科学技术大学
University of Science and Technology of China

计算机系统概论A

Introduction to Computing Systems
(CS1002A.02)

Chapter 8-2 Memory Model for Program Execution & the Stack

计算机科学与技术学院
School of Computer Science and Technology



1 Review

2 Memory Model for Program Execution

3 The Stack

4 Implementing Functions in C Using a Stack



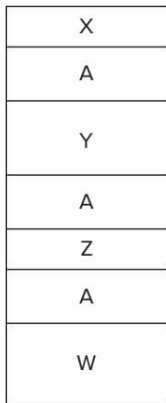
1 Review

2 Memory Model for Program Execution

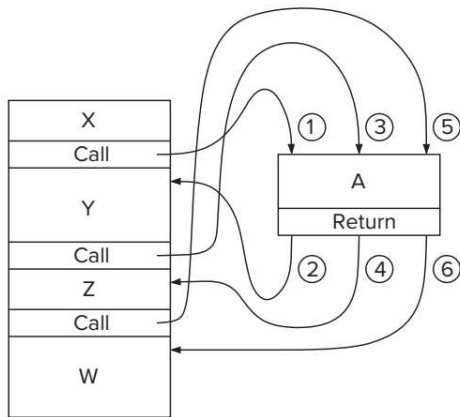
3 The Stack

4 Implementing Functions in C Using a Stack

Review: The Call/Return Mechanism



(a) Without subroutines

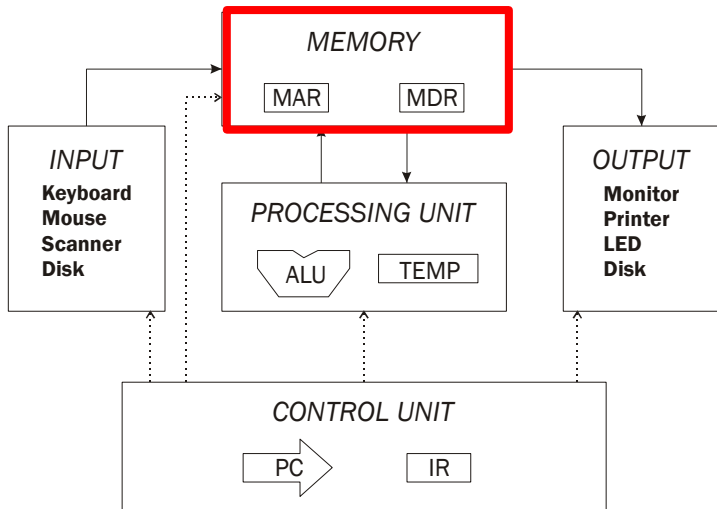


(b) With subroutines

Review: Control Instructions for Subroutines

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|------------|---|-----------|-------------|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | n | z | p | PCoffset9 | | | | | | | | |
| JSR | 0 | 1 | 0 | 0 | 1 | PCoffset11 | | | | | | | | | | |
| JSRR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | |
| RET | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| TRAP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | TrapVector8 | | | | | | | |

Review: Memory in Von Neumann Model



Review: Using Memory

■ Memory

- Just a big "array"
- "Indexed" by address
- Accessed with loads and stores

■ LD/LDR/LDI

- Read a word out of memory
- Use different addressing mode

■ ST/STR/STI

- Place a word in memory
- Use different addressing mode

| Memory | |
|---------|-------|
| Address | Value |
| x0000 | x00A0 |
| x0001 | x5007 |
| x0002 | x0201 |
| x0003 | x0203 |
| x0004 | x3002 |
| ... | |
| xFFFC | x5007 |
| xFFFD | x0201 |
| xFFFE | x0203 |
| xFFFF | x3002 |

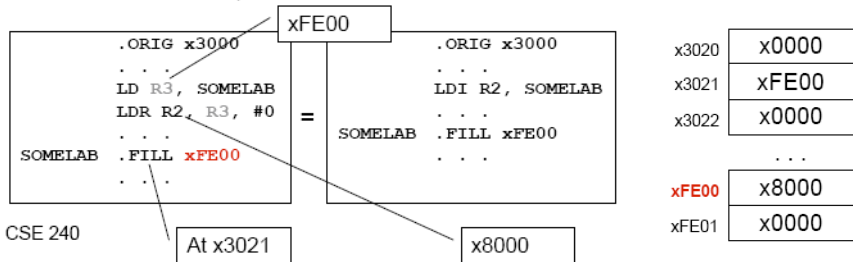
Review: Using Memory

■ Problem

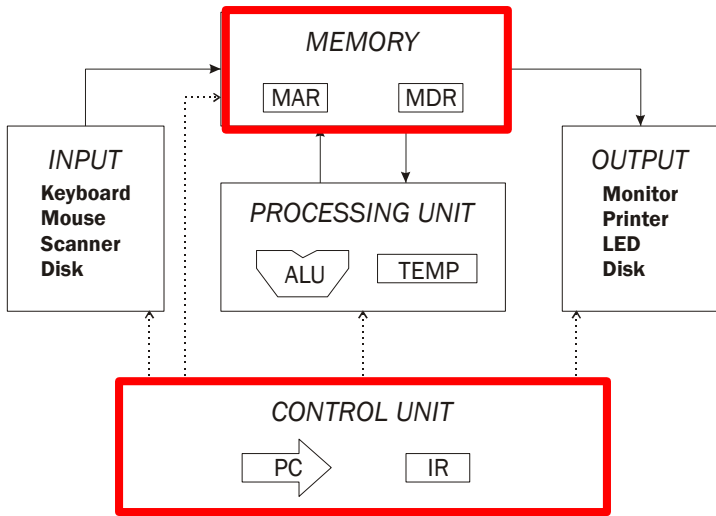
- What if the memory you want to access is far away?
- LD/ST won't work (PC-relative)
- LDR/STR won't work alone (need to get address in register)

■ Solution:LDI/STI

- Place *address* of far away value nearby
- Load address, then load/store from that



Today: Memory Model for Function Calls





1 Review

2 **Memory Model for Program Execution**

3 The Stack

4 Implementing Functions in C Using a Stack

Problem

■ How do we allocate memory during the execution of a program written in C?

- Programs need memory for **code and data** such as instructions, global and local variables, etc.
- Modern programming practices encourage many **(reusable) functions**, callable from anywhere.
- Some memory **can be statically allocated**, since the size and type is known at compile time.
- Some memory **must be allocated dynamically**, size and type is unknown at compile time.

Motivation

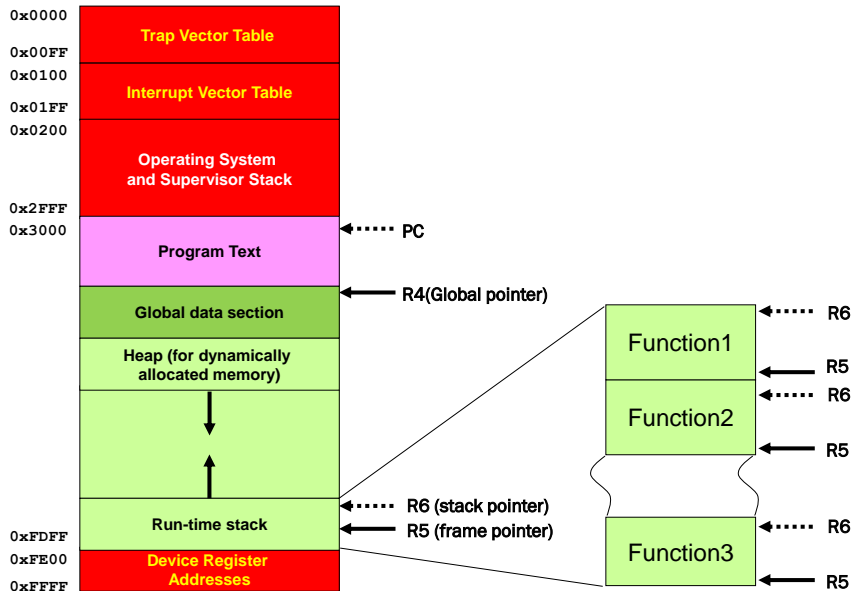
■ Why is memory allocation important? Why not just use **a memory manager**?

- Allocation affects the performance and memory usage of every C, C++, Java program.
- Current systems do not have **enough registers** to store everything that is required.
- Memory management is too **slow and cumbersome** to solve the problem.
- Static allocation of memory resources is too **inflexible and inefficient**, as we will see.

■ What do we care about?

- Fast program execution
- Efficient memory usage
- Avoid memory fragmentation
- Maintain data locality
- Allow recursive calls
- Support parallel execution
- Minimize resource allocation
- Memory should never be allocated for functions that are not executed.

Memory Model in the LC-3





1 Review

2 Memory Model for Program Execution

3 The Stack

4 Implementing Functions in C Using a Stack

Stack: An Abstract Data Type

- An important abstraction that you will encounter in many applications.
- The fundamental model for execution of C, Java, Fortran, and many other languages.
- We will describe two uses of the stack:
 - Evaluating arithmetic expressions
 - Store intermediate results on stack instead of in registers
 - Function calls
 - Store parameters, return values, return address, dynamic link
 - Interrupt-Driven I/O
 - Store processor state for currently executing program

Stack Data Structure

■ A LIFO (last-in first-out) storage structure

- The **first** thing you put in is the **last** thing you take out
- The **last** thing you put in is the **first** thing you take out
- This means of access is what defines a stack, not the specific implementation.

■ Two main operations

- **PUSH**: add an item to the stack
- **POP**: remove an item from the stack

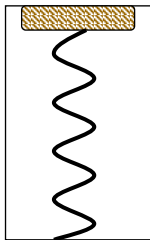
■ Error conditions:

- Underflow (try to pop from empty stack)
- Overflow (try to push onto full stack)

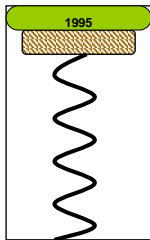
■ A register (eg. R6) holds address of top of stack (TOS)

A Physical Stack

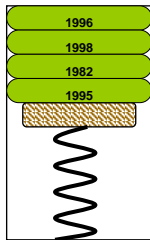
■ Coin holder



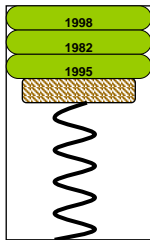
Initial State



After
One Push



After Three
More Pushes

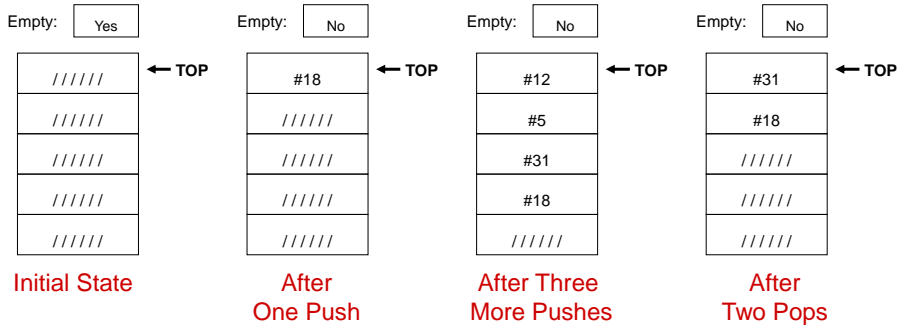


After
One Pop

Last quarter in is the first quarter out (LIFO)

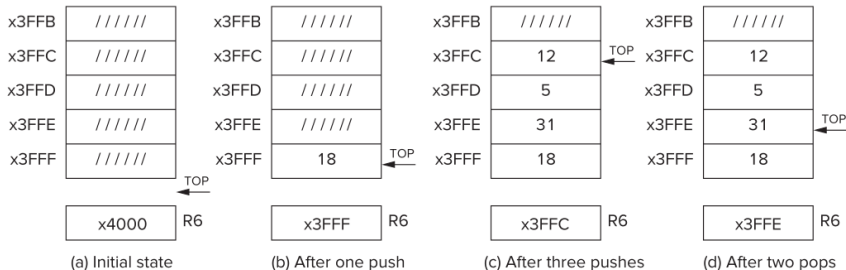
A Hardware Stack Implementation

■ Data items move between registers



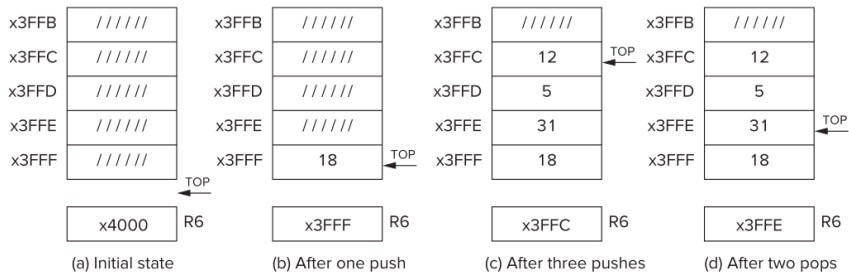
A Software Stack Implementation

- Data items don't move in memory, just our idea about where TOP of the stack is



By convention, R6 holds the Top of Stack (TOS) Pointer (SP)

Basic Push and Pop Code



PUSH

```
ADD R6, R6, #-1      ; increment stack ptr
STR R0, R6, #0        ; store data(R0) to TOS
```

POP

```
LDR R0, R6, #0        ; load data(R0) from TOS
ADD R6, R6, #1        ; decrement stack ptr
```

■ **Note:** Stacks can grow in either direction (toward higher address or toward lower addresses)

Pop with Underflow Detection

■ If we try to pop too many items off the stack, an **underflow** condition occurs.

- Check for underflow by checking TOS before removing data.
- Return status code in R5 (0 for success, 1 for underflow)

```
POP    LD    R1, EMPTY
        ADD  R2, R6, R1    ; Compare stack pointer
        BRz UNDER        ; with x4000
;
        LDR  R0, R6, #0    ; The actual 'pop'
        ADD  R6, R6, #1    ; Adjust stack pointer
;
        AND  R5, R5, #0    ; Success: return R5 = 0
        RET
UNDER  AND  R5, R5, #0    ; Underflow: return R5 = 1
        ADD  R5, R5, #1
        RET
EMPTY  .FILL xC000        ; EMPTY = -x4000
```

Push with Overflow Detection

- If we try to push too many items onto the stack, an **overflow** condition occurs.

- Check for underflow by checking TOS before adding data.
- Return status code in R5 (0 for success, 1 for overflow)

```
PUSH  LD  R1, MAX
      ADD R2, R6, R1      ; Compare stack pointer
      BRz OVER           ; with MAX
      ADD R6, R6, #-1     ; Adjust stack pointer
      STR R0, R6, #0      ; The actual 'push'
      AND R5, R5, #0      ; Success: return R5 = 0
      RET
OVER  AND R5, R5, #0
      ADD R5, R5, #1      ; Overflow: return R5 = 1
      RET
MAX   .FILL xC005        ; MAX = -x3FFB
```


PUSH & POP in LC-3 - 1

```
01 ;
02 ; Subroutines for carrying out the PUSH and POP functions. This
03 ; program works with a stack consisting of memory locations x3FFF
04 ; through x3FFB. R6 is the stack pointer .
05 ;
06 POP  AND R5,R5,#0      ; R5 <-- success
07      ST R1,Save1      ; Save registers that
08      ST R2,Save2      ; are needed by POP
090D ; LD R1,EMPTY        ; EMPTY contains -x4000
0B      ADD R2,R6,R1      ; Compare stack pointer to x4000
0C      BRz fail_exit    ; Branch if stack is empty

0E      LDR R0,R6,#0      ; The actual "pop"
0F      ADD R6,R6,#1      ; Adjust stack pointer
10      BRnzp success_exit
11 ;
12 PUSH AND R5,R5,#0
13      ST R1,Save1      ; Save registers that
14      ST R2,Save2      ; are needed by PUSH
15      LD R1,FULL        ; FULL contains -x3FFB
16      ADD R2,R6,R1      ; Compare stack pointer to x3FFB
17      BRz fail_exit    ; Branch if stack is full
18 ;
```

PUSH & POP in LC-3 - 2

```
19          ADD R6,R6,#-1 ; Adjust stack pointer
1A          STR R0,R6,#0  ; The actual "push"
1B success_exit
            LD R2,Save2 ; Restore original
1C          LD R1,Save1 ; register values
1D          RET
1E ;
1F fail_exit LD R2,Save2 ; Restore original
20          LD R1,Save1 ; register values
21          ADD R5,R5,#1 ; R5 <-- failure
22          RET
23 ;
24 EMPTY .FILL xC000 ; EMPTY contains -x4000
25 FULL .FILL xC005 ; FULL contains -x3FFB
26 Save1 .FILL x0000
27 Save2 .FILL x0000
```

Arithmetic Using a Stack (p387, chapter 10.2)

■ Instead of registers, some ISA's use a stack for source and destination operations. The computer always pops and pushes operands from the stack, and hence no addresses need to be specified explicitly. Therefore, stack machines are sometimes referred to as **zero-address machines**.

- Example: ADD instruction pops two numbers from the stack, adds them, and pushes the result to the stack.

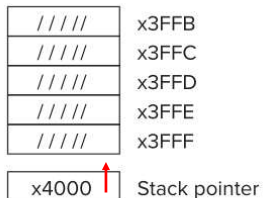
ADD vs. `ADD R0,R1,R2`

■ Evaluating $(A+B) \cdot (C+D)$ using a stack:

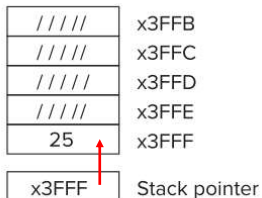
```
push A
push B
ADD
push C
push D
ADD
MULTIPLY
pop result
```



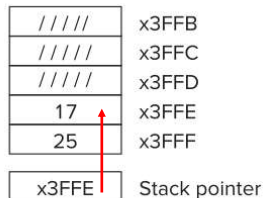
$(25+17) \times (3+2)$



(a) Before

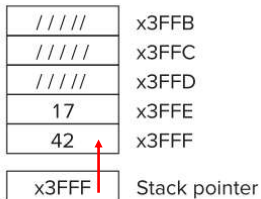


(b) After first push

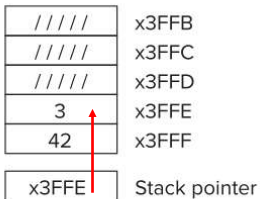


(c) After second push

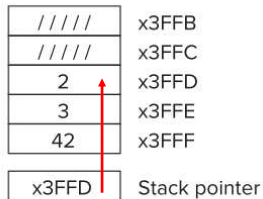
$(25+17) \times (3+2)$



(d) After first add

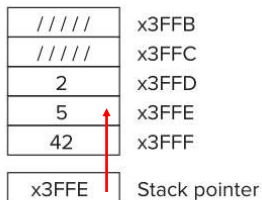


(e) After third push

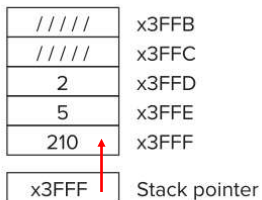


(f) After fourth push

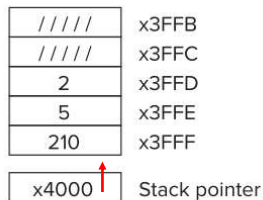
$(25+17) \times (3+2)$



(g) After second add



(h) After multiply



(i) After pop



1 Review

2 Memory Model for Program Execution

3 The Stack

4 Implementing Functions in C Using a Stack

Function in C

- **Smaller, simpler, subcomponent of program**
- **Provides abstraction**
 - hide low-level details
 - give high-level structure to programmer, easier to understand overall program flow
 - enables separable, independent development
- **C functions**
 - zero or multiple arguments passed in
 - single result returned (optional)
 - return value is always a particular type
- **In other languages, called **procedures, subroutines**, ...**

Example of High-Level Structure

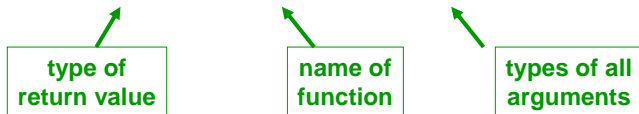
```
main()  
{  
    SetupBoard();    /* place pieces on board */  
    DetermineSides(); /* choose black/white */  
  
    /* Play game */  
    do {  
        WhitesTurn();  
        BlacksTurn();  
    } while (NoOutcomeYet());  
}
```

Structure of program is evident, even without knowing implementation.

Functions in C

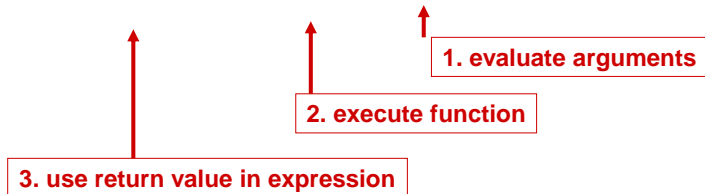
■ Declaration (also called prototype)

```
int Factorial(int n);
```



■ Function call -- used in expression

```
a = x + Factorial(f + g);
```



Function Definition

■ State type, name, types of arguments

- must match function declaration
- give name to each argument (doesn't have to match declaration)

```
int Factorial(int n)
{
    int i;
    int result = 1;
    for (i = 1; i <= n; i++)
        result *= i;
    return result;
}
```

gives control back to calling function and returns value

Why Declaration?

- Since function definition also includes return and argument types, why is declaration needed?
- Use might be seen before definition.
 - Compiler needs to know return and arg types and number of arguments.
- Definition might be in a different file, written by a different programmer.
 - include a "header" file with function declarations only
 - compile separately, link together to make executable

Example

```
double ValueInDollars(double amount, double rate);
```

 **declaration**

```
main()
```

```
{
```

```
...
```

```
dollars = ValueInDollars(francs,  
                        DOLLARS_PER_FRANC);
```

```
printf("%f francs equals %f dollars.\n",  
       francs, dollars);
```

```
...
```

```
}
```

function call (invocation)

definition

```
double ValueInDollars(double amount, double rate)
```

```
{
```

```
    return amount * rate;
```

```
}
```

Storage Requirements

- **Code** must be stored in memory so that we can execute the function.
- **Parameters** must be sent from the caller to the callee so that the function receives them.
- **Local variables** for the function must be stored somewhere, is one copy enough?
- **Return address** must be stored so that control can be returned to the caller.
- **Return values** must be sent from the callee to the caller, that's how results are returned.

Function Call in C

■ Consider the following code:

```
// main program
Int a = 10;
Int b = 20;
Int c = foo(a,b);
Int foo(int x,int y)
{
    Int z;
    z= x+y;
    return z;
}
```

■ What needs to be stored?

- Code, parameters, local/global variables, return address/values

Possible Solution: Mixed Code and Data

■ Function implementation:

```
foo          BR   foo_begin      ;skip over data
foo_rv       .BLKW 1             ;return value
foo_ra       .BLKW 1             ;return address
foo_paramx   .BLKW 1             ;'x' parameter
foo_paramy   .BLKW 1             ;'y' parameter
foo_localz   .BLKW 1             ;'z' local
foo_begin    ST   R7, foo_ra      ;save return
...
             LD   R7, foo_ra      ;restore return
             RET
```

■ Can construct data section by appending foo_

Corresponding to the option 1 in text book p.497

Possible Solution: Mixed Code and Data

■ Calling sequence

```
ST R1, foo_paramx      ; R1 has 'x'  
ST R2, foo_paramy      ; R2 has 'y'  
JSR foo                ; Function call  
LD R3, foo_rv          ; R3 = return value
```

- Code generation is relatively simple.
- Few instructions are spent on moving data.

Possible Solution: Mixed Code and Data

■ Advantages:

- Code and data are close together
- Conceptually easy to understand
- Minimizes register usage for variables
- Data persists through life of program

■ Disadvantages:

- **Cannot handle recursion** or parallel execution
- Code is vulnerable to self-modification
- Consumes resource for inactive functions

Possible Solution: Separate Code and Data

■ Memory allocation

```
foo_rv      .BLKW 1      ; foo return value
foo_ra      .BLKW 1      ; foo return address
foo_paramx  .BLKW 1      ; foo 'x' parameter
foo_paramy  .BLKW 1      ; foo 'y' parameter
foo_localz  .BLKW 1      ; foo 'z' local
bar_rv      .BLKW 1      ; bar return value
bar_ra      .BLKW 1      ; bar return address
bar_paramw  .BLKW 1      ; bar 'w' parameter
```

■ Code for foo() and bar() are somewhere else

■ Function code call is similar to mixed solution

Possible Solution: Separate Code and Data

■ Advantages:

- Code can be marked 'read only'
- Conceptually easy to understand
- Early Fortran used this scheme
- Data persists through life of program

■ Disadvantages:

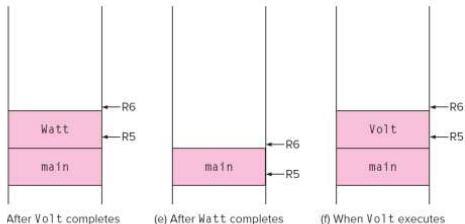
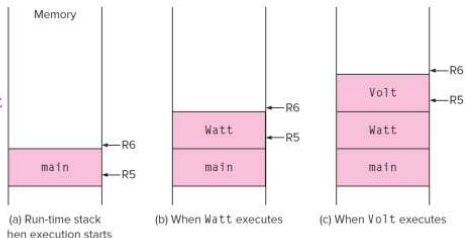
- **Cannot handle recursion** or parallel execution
- Consumes resource for inactive functions

Run-time Stack: Stack frame

- Each function has a **memory template** where it stores its local variables, some bookkeeping information, and its parameter variables. This template is called its **stack frame** or **activation record**.
- Whenever a function is called, its stack frame will be allocated somewhere **in memory**.
- Because the calling pattern of functions naturally follows a stack-like pattern, this allocation and deallocation will follow the pushes and pops of a stack.

Run-time Stack: stack-like nature of function calls

```
1 int main(void)
2 {
3     int a;
4     int b;
5
6     :
7     b = Watt(a);           // main calls Watt first
8     b = Volt(a, b);        // then calls Volt
9 }
10
11 int Watt(int a)
12 {
13     int w;
14
15     :
16     w = Volt(w, 10);       // Watt calls Volt
17
18     return w;
19 }
20
21 int Volt(int q; int r)
22 {
23     int k;
24     int m;
25
26     :
27     return k;
28 }
```



14.5 Several snapshots of the run-time stack while the program outlined in Figure 14.4 executes.

Run-time Stack: frame pointer & stack pointer

- We need some easy way to access the data in each function's stack frame and also to manage the pushing and popping of stack frames.
- For this, we will use R5 and R6. R5 points to some **internal location** within the **stack frame** at the top of the stack—it may point to the base of the local variables for the currently executing function. We call it the **frame pointer (FP)**.
- R6 always points to the very top of the stack. We call it the **stack pointer (SP)**.

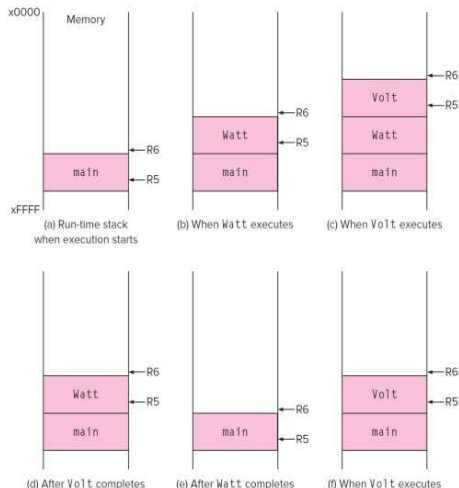


Figure 14.5 Several snapshots of the run-time stack while the program outlined in Figure 14.4 executes.

Run-time Stack: Stack frame

■ Consider what has to happen in a function call:

- Caller must pass parameters to the callee.
- Caller must transfer control to the callee.
- Caller need to allocate space for the return value.
- Caller need to save the return address.

- Callee requires space for local variables.
- Callee must return control to the caller.

- Callee need to save the frame pointer of the caller

■ So, parameters, return value, return address, frame pointer, and local variables are stored on the stack.

Run-time Stack: Stack frame

stack frame points to the **base of the local variables** for the currently executing function.

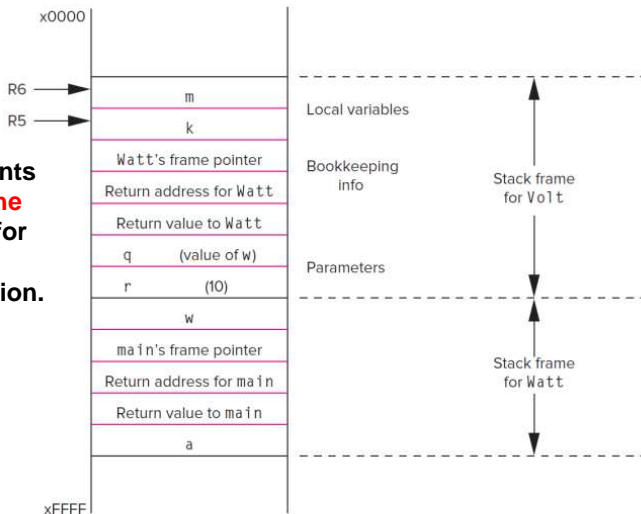
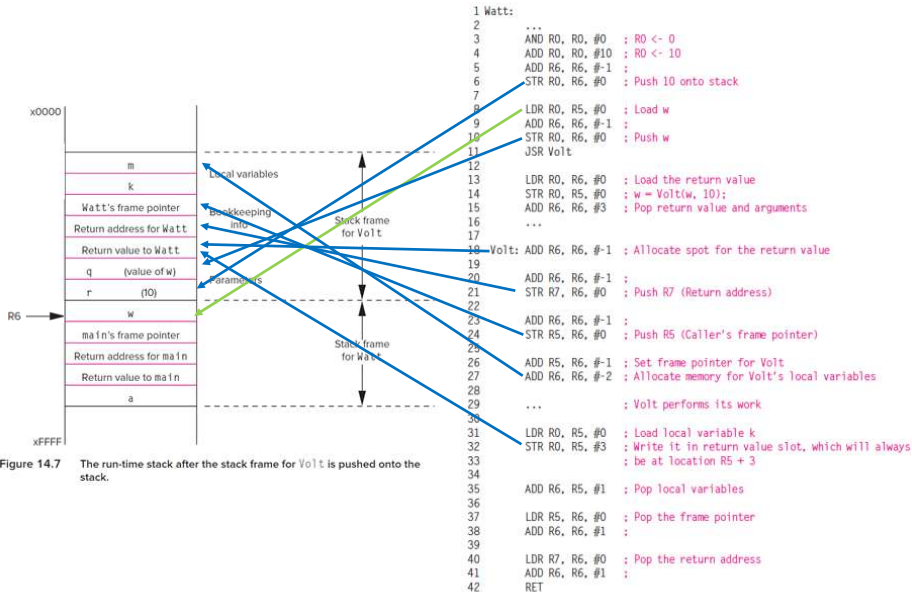


Figure 14.7 The run-time stack after the stack frame for `Volt` is pushed onto the stack.

Run-time Stack: an example



Run-time Stack: Stack frame

```
#include <stdio.h>

int add(char* s, int a, int b, int c, int d, int e, int f) {
    return a+b+c+d+e+f;
}

int main() {
    int x = 12;
    int y = 23;
    char * s = "hehe, hahahaha!!!!";
    int out = add( s, x, y, 1, 1, 1, 1);
    printf("%d\n", out);
    return 0;
}
```

```
00010400 <add>:
10400: e52db004 push {fp} ; (str fp, [sp, #-4])!
10404: e28db000 add fp, sp, #0
10408: e24dd014 sub sp, sp, #20
1040c: e50b0008 str r0, [fp, #-8]
10410: e50b100c str r1, [fp, #-12]
10414: e50b2010 str r2, [fp, #-16]
10418: e50b3014 str r3, [fp, #-20] ; 0xffffffffec
1041c: e51b200c ldr r2, [fp, #-12]
10420: e51b3010 ldr r3, [fp, #-16]
10424: e0822003 add r2, r2, r3
10428: e51b3014 ldr r3, [fp, #-20] ; 0xffffffffec
1042c: e0822003 add r2, r2, r3
10430: e59b3004 ldr r3, [fp, #4]
10434: e0822003 add r2, r2, r3
10438: e59b3008 ldr r3, [fp, #8]
1043c: e0822003 add r2, r2, r3
10440: e59b300c ldr r3, [fp, #12]
10444: e0823003 add r3, r2, r3
10448: e1a00003 mov r0, r3
1044c: e28db000 add sp, fp, #0
10450: e49db004 pop {fp} ; (ldr fp, [sp], #4)
10454: e12fff1e bx lr
```

```
00010458 <main>:
10458: e92d4800 push {fp, lr}
1045c: e28db004 add fp, sp, #4
10460: e24dd020 sub sp, sp, #32
10464: e3a0300c mov r3, #12
10468: e50b3014 str r3, [fp, #-20] ; 0xffffffffec
1046c: e3a03017 mov r3, #23
10470: e50b3010 str r3, [fp, #-16]
10474: e59f304c ldr r3, [pc, #76] ; 104c8 <main+0x70>
10478: e50b300c str r3, [fp, #-12]
1047c: e3a03001 mov r3, #1
10480: e58d3008 str r3, [sp, #8]
10484: e3a03001 mov r3, #1
10488: e58d3004 str r3, [sp, #4]
1048c: e3a03001 mov r3, #1
10490: e58d3000 str r3, [sp]
10494: e3a03001 mov r3, #1
10498: e51b2010 ldr r2, [fp, #-16]
1049c: e51b1014 ldr r1, [fp, #-20] ; 0xffffffffec
104a0: e51b000c ldr r0, [fp, #-12]
104a4: ebffffd5 bl 10400 <add>
104a8: e50b0008 str r0, [fp, #-8]
104ac: e51b1008 ldr r1, [fp, #-8]
104b0: e59f0014 ldr r0, [pc, #20] ; 104cc <main+0x74>
104b4: ebffff89 bl 102e0 <printf@plt>
104b8: e3a03000 mov r3, #0
104bc: e1a00003 mov r0, r3
104c0: e24bd004 sub sp, fp, #4
104c4: e8bd8800 pop {fp, pc}
104c8: 00010540 .word 0x00010540
104cc: 00010554 .word 0x00010554
```

Contents of section .rodata:

```
1053c 01000200 68656865 2c206861 68616861 ....hehe, hahaha
1054c 68612121 21210000 25640a00 hahahaha!!!!..%d..
```

Run-time Stack: Stack frame

```
#include <stdio.h>

int myadd(int c, int d) {
    return c+d;
}

int add(int a, int b) {
    return myadd(a,b);
}

int main() {

    int x = 12;
    int y = 23;
    int out = add(x, y);
    printf("%d\n", out);
    return 0;
}
```

```
00010400 <myadd>:
10400: e52db004      push    {fp}                ; (str fp, [sp, #-4]!)
10404: e28db000      add     fp, sp, #0
10408: e24dd00c      sub     sp, sp, #12
1040c: e50b0008      str     r0, [fp, #-8]
10410: e50b100c      str     r1, [fp, #-12]
10414: e51b2008      ldr     r2, [fp, #-8]
10418: e51b300c      ldr     r3, [fp, #-12]
1041c: e0823003      add     r3, r2, r3
10420: e1a00003      mov     r0, r3
10424: e28bd000      add     sp, fp, #0
10428: e49db004      pop     {fp}                ; (ldr fp, [sp], #4)
1042c: e12fff1e      bx      lr

00010430 <add>:
10430: e92d4800      push    {fp, lr}
10434: e28db004      add     fp, sp, #4
10438: e24dd008      sub     sp, sp, #8
1043c: e50b0008      str     r0, [fp, #-8]
10440: e50b100c      str     r1, [fp, #-12]
10444: e51b100c      ldr     r1, [fp, #-12]
10448: e51b0008      ldr     r0, [fp, #-8]
1044c: ebffffeb      bl      10400 <myadd>
10450: e1a03000      mov     r3, r0
10454: e1a00003      mov     r0, r3
10458: e24bd004      sub     sp, fp, #4
1045c: e8bd8800      pop     {fp, pc}

00010460 <main>:
10460: e92d4800      push    {fp, lr}
10464: e28db004      add     fp, sp, #4
10468: e24dd010      sub     sp, sp, #16
1046c: e3a0300c      mov     r3, #12
10470: e50b3010      str     r3, [fp, #-16]
10474: e3a03017      mov     r3, #23
10478: e50b300c      str     r3, [fp, #-12]
1047c: e51b100c      ldr     r1, [fp, #-12]
10480: e51b0010      ldr     r0, [fp, #-16]
10484: ebffffe9      bl      10430 <add>
10488: e50b0008      str     r0, [fp, #-8]
1048c: e51b1008      ldr     r1, [fp, #-8]
10490: e59f0010      ldr     r0, [pc, #16]        ; 104a8 <main+0x48>
10494: ebffff91      bl      102e0 <printf@plt>
10498: e3a03000      mov     r3, #0
1049c: e1a00003      mov     r0, r3
104a0: e24bd004      sub     sp, fp, #4
104a4: e8bd8800      pop     {fp, pc}
104a8: 0001051c      <main+0x48>
```



中国科学技术大学
University of Science and Technology of China

计算机系统概论

Introduction to Computing Systems
(CS1002A.02)

Chapter 8-3 Recursion, Queue and Character Strings

计算机科学与技术学院

School of Computer Science and Technology



1 Recursion

2 The Queue

3 Character Strings

1. Recursion

- Recursion is a mechanism for expressing a function *in terms of itself*.
- When used appropriately, the expressive power of recursion is going to save us a lot of headaches
- Otherwise, it results in longer execution time and wasted energy

Factorial: compute $n! = n \cdot (n-1)!$, A BAD EXAMPLE

```
FACT    ST R1, Save1      ; Callee save R1
        ADD R1,R0,#-1      ; Test if R0=1
        BRz DONE          ; If R0=1, R0 also contains (1)!, so we are done
        ADD R1,R0,#0       ; Save n in R1, to be used after we compute (n-1)!
        ADD R0,R1, #-1     ; Set R0 to n-1, and then call FACT
B       JSR FACT           ; On RET, R0 will contain (n-1)!
        MUL R0,R0,R1       ; Multiply n times (n-1)!, yielding n! in R0
DONE    LD R1, Save1      ; Callee restore R1
        RET
Save1   .BLKW 1
```

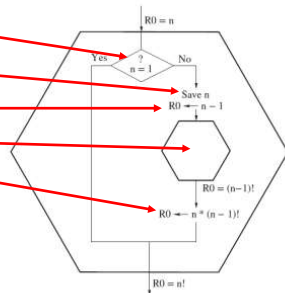


Figure 8.12 Flowchart for a recursive FACTORIAL subroutine.

Can it work properly?

Expected execution flow

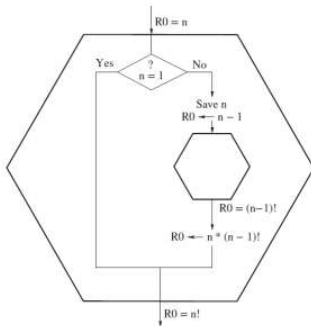
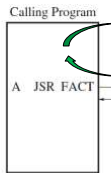


Figure 8.12 Flowchart for a recursive FACTORIAL subroutine.

Figure 8.13 Execution flow for recursive FACTORIAL subroutines.

Factorial: compute $n! = n \cdot (n-1)!$

```
FACT  ST R1, Save1    ; Callee save R1
      ADD R1,R0,#-1    ; Test if R0=1
      BRz DONE        ; If R0=1, R0 also contains (1)!, so we are done
      ADD R1,R0,#0     ; Save n in R1, to be used after we compute (n-1)!
      ADD R0,R1, #-1   ; Set R0 to n-1, and then call FACT
      B     JSR FACT    ; On RET, R0 will contain (n-1)!
      MUL R0,R0,R1     ; Multiply n times (n-1)!, yielding n! in R0
DONE  LD R1, Save1    ; Callee restore R1
      RET
Save1 .BLKW 1
```

Problem 1

In Calling Program:

A JSR FACT: A+1 \rightarrow R7

In #1:

B JSR FACT: B+1 \rightarrow R7

So, **R7 = A+1** is wiped out by **B+1**, and the execution **can not return to A+1**.

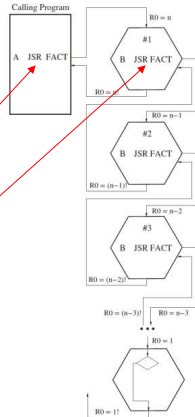


Figure 8.13 Execution flow for recursive FACTORIAL subroutines.

Factorial: compute $n! = n \cdot (n-1)!$

```
FACT  ST R1, Save1    ; Callee saves R1
      ADD R1,R0,#-1    ; Test if R0=1
      BRz DONE        ; If R0=1, R0 also contains (1)!, so we are done
      ADD R1,R0,#0     ; Save n in R1, to be used after we compute (n-1)!
      ADD R0,R1, #-1   ; Set R0 to n-1, and then call FACT
B     JSR FACT         ; On RET, R0 will contain (n-1)!
      MUL R0,R0,R1     ; Multiply n times (n-1)!, yielding n! in R0
DONE  LD R1, Save1    ; Callee restores R1
      RET
Save1 .BLKW 1
```

Problem 2

In #1: ADD R1,R0,#0 stores the value n into R1

In #2: ADD R1,R0,#0 stores the value n-1 into R1

So, #2 **wipes out** the value n that had been put in R1 by the code in #1. when the instruction flow gets back to #1, where the value n is needed by the instruction MUL R0,R0,R1, it is no longer there.

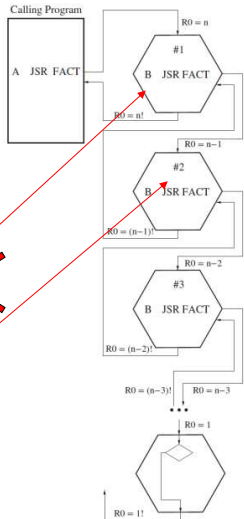


Figure 8.13 Execution flow for recursive FACTORIAL subroutines.

Factorial: compute $n! = n \cdot (n-1)!$

```
FACT  ST R1, Save1    ; Callee saves R1
      ADD R1,R0,#-1    ; Test if R0=1
      BRz DONE         ; If R0=1, R0 also contains (1)!, so we are done
      ADD R1,R0,#0     ; Save n in R1, to be used after we compute (n-1)!
      ADD R0,R1,#-1    ; Set R0 to n-1, and then call FACT
      JSR FACT          ; On RET, R0 will contain (n-1)!
      MUL R0,R0,R1      ; Multiply n times (n-1)!, yielding n! in R0
DONE  LD R1, Save1     ; Callee restores R1
      RET
Save1 .BLKW 1
```

Problem 3

In #1:

ST R1, Save1 ; #1 saves R1 (n) to save1

In #2:

ST R1, Save1 ; #2 saves R1 (n-1) to save1

So, after the JSR FACT instruction is executed, the first instruction of the recursively called subroutine FACT (#2) will save that value to **Save1**, wiping out the value that the main program (#1) had stored in R1 when it called FACT.

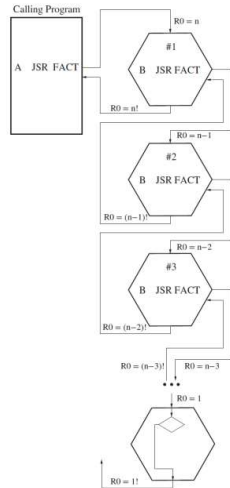


Figure 8.13 Execution flow for recursive FACTORIAL subroutines.

A new solution: using stack

FACT

ADD R6,R6,#-1
STR R1,R6,#0 ; Push Caller's R1 on the stack, so we can use R1.

ADD R1,R0,#-1 ; If n=1, we are done since 1! = 1
BRz NO_RECURSE

ADD R6,R6,#-1
STR R7,R6,#0 ; Push return linkage onto stack

ADD R6,R6,#-1
STR R0,R6,#0 ; Push n on the stack

ADD R0,R0,#-1 ; Form n-1, argument of JSR
JSR FACT

LDR R1,R6,#0 ; Pop n from the stack
ADD R6,R6,#1

MUL R0,R0,R1 ; form n*(n-1)!

LDR R7,R6,#0 ; Pop return linkage into R7

ADD R6,R6,#1

NO_RECURSE LDR R1,R6,#0 ; Pop caller's R1 back into R1

ADD R6,R6,#1

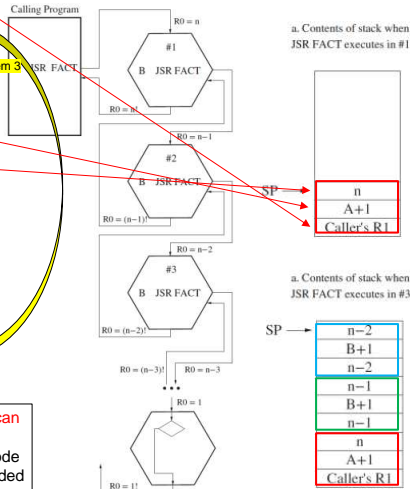
RET

P1,Return address: R7 = A+1 is wiped out by B+1, and the execution can not return to A+1.

P2, registers: #2 wipes out the value n that had been put in R1 by the code in #1. when the instruction flow gets back to #1, where the value n is needed by the instruction MUL R0,R0,R1, it is no longer there.

P3,static memory address: The first instruction of the recursively called subroutine FACT (#2) will save that value to **Save1**, wiping out the value that the main program (#1) had stored in R1 when it called FACT.

Figure 8.13 Execution flow for recursive FACTORIAL subroutines.



A new solution: using stack

FACT ADD R6,R6,#-1
STR R1,R6,#0 ; Push Caller's R1 on the stack, so we can use R1.

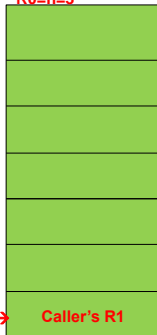
ADD R1,R0,#-1 ; If n=1, we are done since 1! = 1
BRz NO_RECURSE

ADD R6,R6,#-1
STR R7,R6,#0 ; Push return linkage onto stack
ADD R6,R6,#-1
STR R0,R6,#0 ; Push n on the stack

B ADD R0,R0,#-1 ; Form n-1, argument of JSR
JSR **FACT**
LDR R1,R6,#0 ; Pop n from the stack
ADD R6,R6,#1
MUL R0,R0,R1 ; form $n*(n-1)!$

LDR R7,R6,#0 ; Pop return linkage into R7
ADD R6,R6,#1
NO_RECURSE LDR R1,R6,#0 ; Pop caller's R1 back into R1
ADD R6,R6,#1
RET

**Example:
R0=n=3**



R0=3, R1=2

A new solution: using stack

FACT `ADD R6,R6,#-1`
 `STR R1,R6,#0` ; Push Caller' s R1 on the stack, so we can use R1.

`ADD R1,R0,#-1` ; If $n=1$, we are done since $1! = 1$
`BRz NO_RECURSE`

`ADD R6,R6,#-1`
`STR R7,R6,#0` ; Push return linkage onto stack
`ADD R6,R6,#-1`
`STR R0,R6,#0` ; Push n on the stack

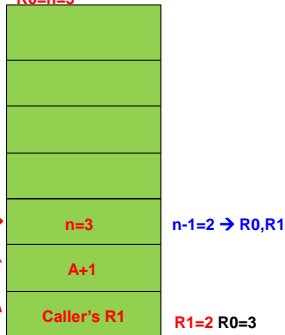
`ADD R0,R0,#-1` ; Form $n-1$, argument of JSR

B `JSR FACT`
 `LDR R1,R6,#0` ; Pop n from the stack
 `ADD R6,R6,#1`
 `MUL R0,R0,R1` ; form $n*(n-1)!$

`LDR R7,R6,#0` ; Pop return linkage into R7

`ADD R6,R6,#1`
NO_RECURSE `LDR R1,R6,#0` ; Pop caller' s R1 back into R1
 `ADD R6,R6,#1`
 `RET`

Example:
 $R0=n=3$



A new solution: using stack

FACT ADD R6,R6,#-1
 STR R1,R6,#0

; Push Caller' s R1 on the stack, so we can use R1.

ADD R1,R0,#-1
BRz NO_RECURSE

; If n=1, we are done since $1! = 1$

ADD R6,R6,#-1
STR R7,R6,#0
ADD R6,R6,#-1
STR R0,R6,#0

; Push return linkage onto stack

; Push n on the stack

ADD R0,R0,#-1
JSR FACT

; Form n-1, argument of JSR

LDR R1,R6,#0
ADD R6,R6,#1
MUL R0,R0,R1

; Pop n from the stack

; form $n*(n-1)!$

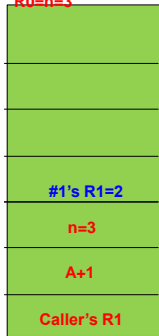
LDR R7,R6,#0
ADD R6,R6,#1

; Pop return linkage into R7

NO_RECURSE LDR R1,R6,#0
 ADD R6,R6,#1
 RET

; Pop caller' s R1 back into R1

Example:
R0=n=3



A new solution: using stack

FACT `ADD R6,R6,#-1`
 `STR R1,R6,#0` ; Push Caller' s R1 on the stack, so we can use R1.

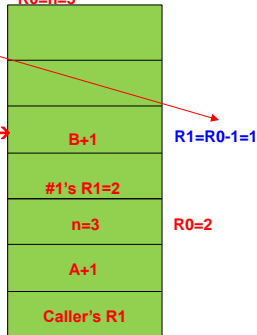
`ADD R1,R0,#-1` ; If $n=1$, we are done since $1! = 1$
`BRz NO_RECURSE`

`ADD R6,R6,#-1`
`STR R7,R6,#0` ; Push return linkage onto stack
`ADD R6,R6,#-1`
`STR R0,R6,#0` ; Push n on the stack

B `ADD R0,R0,#-1` ; Form $n-1$, argument of JSR
 `JSR FACT`
 `LDR R1,R6,#0` ; Pop n from the stack
 `ADD R6,R6,#1`
 `MUL R0,R0,R1` ; form $n*(n-1)!$

`LDR R7,R6,#0` ; Pop return linkage into R7
 `ADD R6,R6,#1`
NO_RECURSE `LDR R1,R6,#0` ; Pop caller' s R1 back into R1
 `ADD R6,R6,#1`
 `RET`

Example:
 $R0=n=3$



A new solution: using stack

FACT `ADD R6,R6,#-1`
 `STR R1,R6,#0` ; Push Caller' s R1 on the stack, so we can use R1.

`ADD R1,R0,#-1` ; If $n=1$, we are done since $1! = 1$
`BRz NO_RECURSE`

`ADD R6,R6,#-1`
`STR R7,R6,#0` ; Push return linkage onto stack
`ADD R6,R6,#-1`
`STR R0,R6,#0` ; Push n on the stack

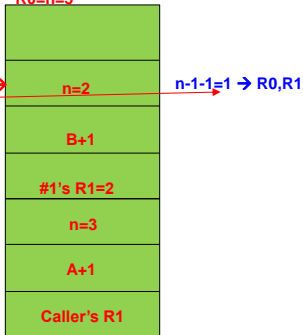
`ADD R0,R0,#-1` ; Form $n-1$, argument of JSR

B `JSR FACT`
 `LDR R1,R6,#0` ; Pop n from the stack
 `ADD R6,R6,#1`
 `MUL R0,R0,R1` ; form $n*(n-1)!$

`LDR R7,R6,#0` ; Pop return linkage into R7

`ADD R6,R6,#1`
NO_RECURSE `LDR R1,R6,#0` ; Pop caller' s R1 back into R1
 `ADD R6,R6,#1`
 `RET`

Example:
 $R0=n=3$



A new solution: using stack

FACT **ADD R6,R6,#-1**
 STR R1,R6,#0 ; Push Caller' s R1 on the stack, so we can use R1.

ADD R1,R0,#-1 ; If n=1, we are done since 1! = 1
BRz NO_RECURSE

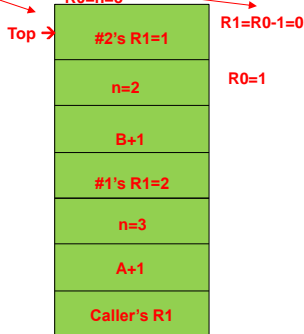
ADD R6,R6,#-1
STR R7,R6,#0 ; Push return linkage onto stack
ADD R6,R6,#-1
STR R0,R6,#0 ; Push n on the stack

ADD R0,R0,#-1 ; Form n-1, argument of JSR
B **JSR FACT**
LDR R1,R6,#0 ; Pop n from the stack
ADD R6,R6,#1
MUL R0,R0,R1 ; form n*(n-1)!

LDR R7,R6,#0 ; Pop return linkage into R7

NO_RECURSE ADD R6,R6,#1 ; Pop caller' s R1 back into R1
 ADD R6,R6,#1
 RET

Example:
R0=n=3



A new solution: using stack

FACT `ADD R6,R6,#-1`
 `STR R1,R6,#0` ; Push Caller' s R1 on the stack, so we can use R1.

`ADD R1,R0,#-1` ; If n=1, we are done since $1! = 1$
`BRz NO_RECURSE`

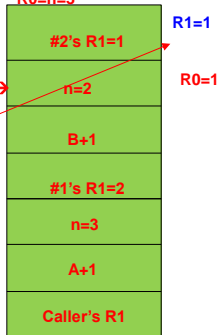
`ADD R6,R6,#-1`
`STR R7,R6,#0` ; Push return linkage onto stack
`ADD R6,R6,#-1`
`STR R0,R6,#0` ; Push n on the stack

B `ADD R0,R0,#-1` ; Form n-1, argument of JSR
 `JSR FACT`
 `LDR R1,R6,#0` ; Pop n from the stack
 `ADD R6,R6,#1`
 `MUL R0,R0,R1` ; form $n*(n-1)!$

`LDR R7,R6,#0` ; Pop return linkage into R7

`ADD R6,R6,#1`
NO_RECURSE `LDR R1,R6,#0` ; Pop caller' s R1 back into R1
 `ADD R6,R6,#1`
 `RET`

Example:
 $R0=n=3$



A new solution: using stack

FACT `ADD R6,R6,#-1`
 `STR R1,R6,#0` ; Push Caller' s R1 on the stack, so we can use R1.

`ADD R1,R0,#-1` ; If n=1, we are done since $1! = 1$
`BRz NO_RECURSE`

`ADD R6,R6,#-1`
`STR R7,R6,#0` ; Push return linkage onto stack
`ADD R6,R6,#-1`
`STR R0,R6,#0` ; Push n on the stack

B `ADD R0,R0,#-1` ; Form n-1, argument of JSR
 `JSR FACT`

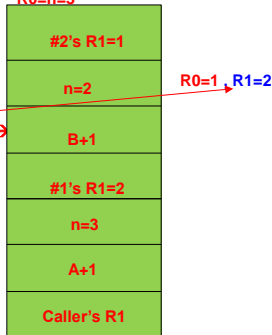
`LDR R1,R6,#0` ; Pop n from the stack

`ADD R6,R6,#1`
`MUL R0,R0,R1` ; form $n*(n-1)!$

`LDR R7,R6,#0` ; Pop return linkage into R7

`ADD R6,R6,#1`
NO_RECURSE `LDR R1,R6,#0` ; Pop caller' s R1 back into R1
`ADD R6,R6,#1`
`RET`

Example:
R0=n=3



A new solution: using stack

FACT `ADD R6,R6,#-1`
 `STR R1,R6,#0` ; Push Caller' s R1 on the stack, so we can use R1.

`ADD R1,R0,#-1` ; If n=1, we are done since $1! = 1$
`BRz NO_RECURSE`

`ADD R6,R6,#-1`
`STR R7,R6,#0` ; Push return linkage onto stack
`ADD R6,R6,#-1`
`STR R0,R6,#0` ; Push n on the stack

B `ADD R0,R0,#-1` ; Form n-1, argument of JSR
 `JSR FACT`

`LDR R1,R6,#0` ; Pop n from the stack

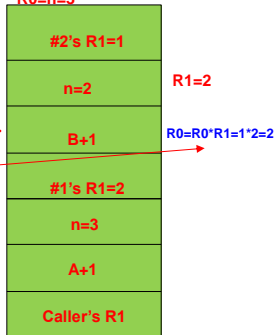
`ADD R6,R6,#1`
`MUL R0,R0,R1` ; form $n*(n-1)!$

`LDR R7,R6,#0` ; Pop return linkage into R7

`ADD R6,R6,#1`
NO_RECURSE `LDR R1,R6,#0` ; Pop caller' s R1 back into R1

`ADD R6,R6,#1`
`RET`

Example:
R0=n=3



A new solution: using stack

FACT `ADD R6,R6,#-1`
 `STR R1,R6,#0` ; Push Caller' s R1 on the stack, so we can use R1.

`ADD R1,R0,#-1` ; If $n=1$, we are done since $1! = 1$
`BRz NO_RECURSE`

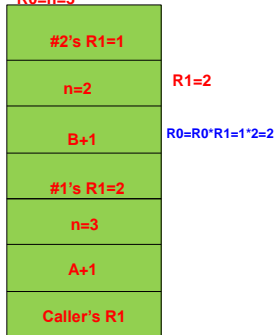
`ADD R6,R6,#-1`
`STR R7,R6,#0` ; Push return linkage onto stack
`ADD R6,R6,#-1`
`STR R0,R6,#0` ; Push n on the stack

B `ADD R0,R0,#-1` ; Form $n-1$, argument of JSR
 `JSR FACT`
 `LDR R1,R6,#0` ; Pop n from the stack
 `ADD R6,R6,#1`
 `MUL R0,R0,R1` ; form $n*(n-1)!$

`LDR R7,R6,#0` ; Pop return linkage into R7

`ADD R6,R6,#1`
NO_RECURSE `LDR R1,R6,#0` ; Pop caller' s R1 back into R1
 `ADD R6,R6,#1`
 `RET`

Example:
 $R0=n=3$



A new solution: using stack

FACT `ADD R6,R6,#-1`
 `STR R1,R6,#0` ; Push Caller' s R1 on the stack, so we can use R1.

`ADD R1,R0,#-1` ; If n=1, we are done since $1! = 1$
`BRz NO_RECURSE`

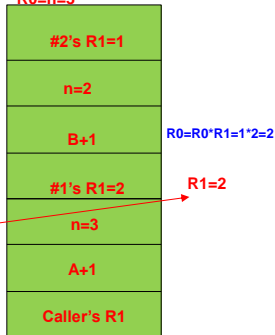
`ADD R6,R6,#-1`
`STR R7,R6,#0` ; Push return linkage onto stack
`ADD R6,R6,#-1`
`STR R0,R6,#0` ; Push n on the stack

B `ADD R0,R0,#-1` ; Form n-1, argument of JSR
`JSR FACT`
`LDR R1,R6,#0` ; Pop n from the stack
`ADD R6,R6,#1`
`MUL R0,R0,R1` ; form $n*(n-1)!$

`LDR R7,R6,#0` ; Pop return linkage into R7

`ADD R6,R6,#1`
NO_RECURSE `LDR R1,R6,#0` ; Pop caller' s R1 back into R1
`ADD R6,R6,#1`
`RET`

Example:
R0=n=3



A new solution: using stack

FACT `ADD R6,R6,#-1`
 `STR R1,R6,#0` ; Push Caller' s R1 on the stack, so we can use R1.

`ADD R1,R0,#-1` ; If n=1, we are done since $1! = 1$
`BRz NO_RECURSE`

`ADD R6,R6,#-1`
`STR R7,R6,#0` ; Push return linkage onto stack
`ADD R6,R6,#-1`
`STR R0,R6,#0` ; Push n on the stack

B `ADD R0,R0,#-1` ; Form n-1, argument of JSR
`JSR FACT`

`LDR R1,R6,#0` ; Pop n from the stack

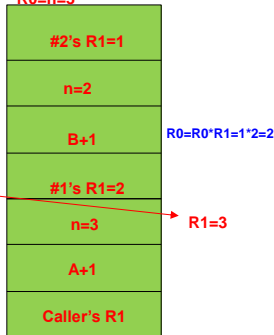
`ADD R6,R6,#1`
`MUL R0,R0,R1` ; form $n*(n-1)!$

`LDR R7,R6,#0` ; Pop return linkage into R7

`ADD R6,R6,#1`
NO_RECURSE `LDR R1,R6,#0` ; Pop caller' s R1 back into R1

`ADD R6,R6,#1`
`RET`

Example:
R0=n=3



A new solution: using stack

FACT `ADD R6,R6,#-1`
 `STR R1,R6,#0` ; Push Caller' s R1 on the stack, so we can use R1.

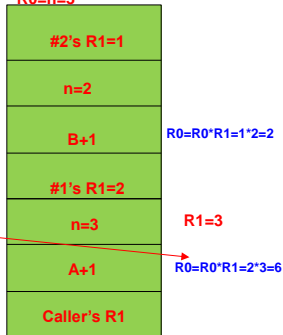
`ADD R1,R0,#-1` ; If $n=1$, we are done since $1! = 1$
`BRz NO_RECURSE`

`ADD R6,R6,#-1`
`STR R7,R6,#0` ; Push return linkage onto stack
`ADD R6,R6,#-1`
`STR R0,R6,#0` ; Push n on the stack

B `ADD R0,R0,#-1` ; Form $n-1$, argument of JSR
 `JSR FACT`
 `LDR R1,R6,#0` ; Pop n from the stack
 `ADD R6,R6,#1`
 `MUL R0,R0,R1` ; form $n*(n-1)!$

`LDR R7,R6,#0` ; Pop return linkage into R7
`ADD R6,R6,#1`
NO_RECURSE `LDR R1,R6,#0` ; Pop caller' s R1 back into R1
 `ADD R6,R6,#1`
 `RET`

Example:
 $R0=n=3$



A new solution: using stack

FACT `ADD R6,R6,#-1`
 `STR R1,R6,#0` ; Push Caller's R1 on the stack, so we can use R1.

`ADD R1,R0,#-1` ; If $n=1$, we are done since $1! = 1$
`BRz NO_RECURSE`

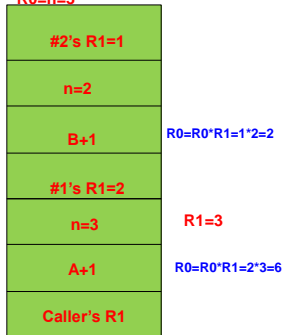
`ADD R6,R6,#-1`
`STR R7,R6,#0` ; Push return linkage onto stack
`ADD R6,R6,#-1`
`STR R0,R6,#0` ; Push n on the stack

B `ADD R0,R0,#-1` ; Form $n-1$, argument of JSR
 `JSR FACT`
 `LDR R1,R6,#0` ; Pop n from the stack
 `ADD R6,R6,#1`
 `MUL R0,R0,R1` ; form $n*(n-1)!$

`LDR R7,R6,#0` ; Pop return linkage into R7

`ADD R6,R6,#1`
NO_RECURSE `LDR R1,R6,#0` ; Pop caller's R1 back into R1
 `ADD R6,R6,#1`
 `RET`

Example:
 $R0=n=3$



A new solution: using stack

FACT `ADD R6,R6,#-1`
 `STR R1,R6,#0` ; Push Caller' s R1 on the stack, so we can use R1.

`ADD R1,R0,#-1` ; If $n=1$, we are done since $1! = 1$
`BRz NO_RECURSE`

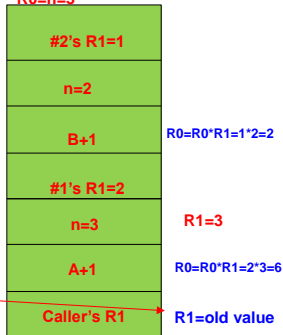
`ADD R6,R6,#-1`
`STR R7,R6,#0` ; Push return linkage onto stack
`ADD R6,R6,#-1`
`STR R0,R6,#0` ; Push n on the stack

B `ADD R0,R0,#-1` ; Form $n-1$, argument of JSR
`JSR FACT`
`LDR R1,R6,#0` ; Pop n from the stack
`ADD R6,R6,#1`
`MUL R0,R0,R1` ; form $n*(n-1)!$

`LDR R7,R6,#0` ; Pop return linkage into R7
`ADD R6,R6,#1`

`NO_RECURSE` `LDR R1,R6,#0` ; Pop caller' s R1 back into R1
`ADD R6,R6,#1`
`RET`

Example:
 $R0=n=3$



A new solution: using stack

FACT `ADD R6,R6,#-1`
 `STR R1,R6,#0` ; Push Caller' s R1 on the stack, so we can use R1.

`ADD R1,R0,#-1` ; If n=1, we are done since $1! = 1$
`BRz NO_RECURSE`

`ADD R6,R6,#-1`
`STR R7,R6,#0` ; Push return linkage onto stack
`ADD R6,R6,#-1`
`STR R0,R6,#0` ; Push n on the stack

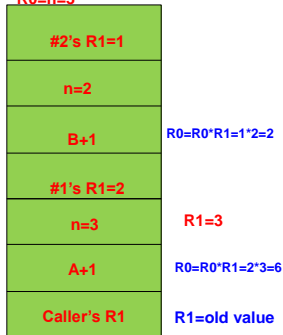
B `ADD R0,R0,#-1` ; Form n-1, argument of JSR
 `JSR FACT`
 `LDR R1,R6,#0` ; Pop n from the stack
 `ADD R6,R6,#1`
 `MUL R0,R0,R1` ; form $n*(n-1)!$

`LDR R7,R6,#0` ; Pop return linkage into R7

`ADD R6,R6,#1`
NO_RECURSE `LDR R1,R6,#0` ; Pop caller' s R1 back into R1
 `ADD R6,R6,#1`

RET

Example:
R0=n=3



Top →

Return to calling program

Implementing FACT iteratively (without recursion)

```
FACT    ST R1,SAVE_R1
        ADD R1,R0,#0           ; R1=R0=n
        ADD R0,R0,#-1         ; R0=n-1
        BRz DONE
AGAIN   MUL R1,R1,R0           ; (n*(n-1))*(n-2)...
        ADD R0,R0,#-1         ; R0 gets next integer for MUL
        BRnp AGAIN
DONE    ADD R0,R1,#0           ; Move n! to R0
        LD R1,SAVE_R1
        RET
SAVE_R1 .BLKW 1
```

The Maze: a Good Example

Given a **maze** and a **starting position** within the maze, write a program that **determines whether or not there is a way out of the maze** from your starting position.

A Maze A maze can be any size, n by m . For example, Figure 8.20 illustrates a 6×6 maze.

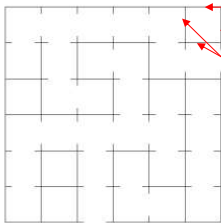


Figure 8.20 Example of a maze.

■ Specification of each cell in the maze

- Bit[4]=1 if there is a door to the **outside world**;
Bit[4]=0 if no door.
- Bit[3]=1 if there is a door to the cell to the **north**;
Bit[3]=0 if no door.
- Bit[2]=1 if there is a door to the cell to the **east**;
Bit[2]=0 if no door.
- Bit[1]=1 if there is a door to the cell to the **south**;
Bit[1]=0 if no door.
- Bit[0]=1 if there is a door to the cell to the **west**;
Bit[0]=0 if no door.

Specification of the maze

```
00 .ORIG x5000
01 MAZE .FILL x0006 ; first row: indices 0 to 5
02     .FILL x0007
03     .FILL x0005
04     .FILL x0005
05     .FILL x0003
06     .FILL x0000
07 ; second row: indices 6 to 11
08     .FILL x0008
09     .FILL x000A
0A     .FILL x0004
0B     .FILL x0003
0C     .FILL x000C
0D     .FILL x0015
0E ; third row: indices 12 to 17
0F     .FILL x0000
10     .FILL x000C
11     .FILL x0001
12     .FILL x000A
13     .FILL x0002
14     .FILL x0002
.....
2A .END
```

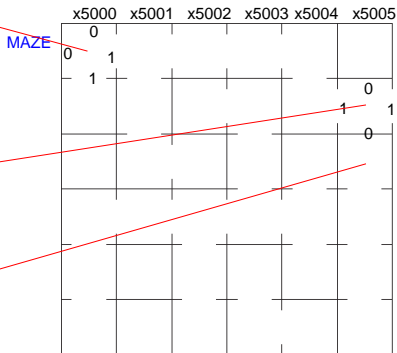
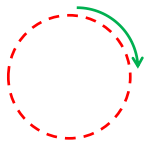
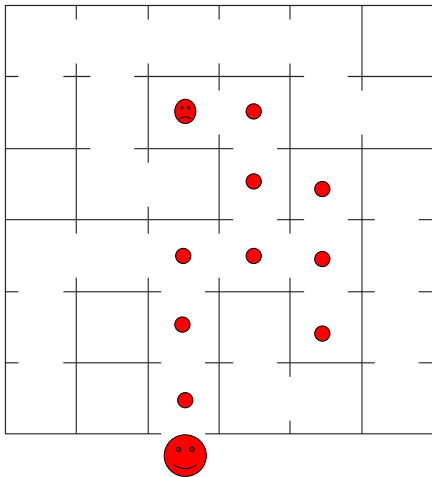


Figure 8.22 Specification of the maze of Figure 8.20.

The Maze: a searching algorithm



Description of the Algorithm

- a. From our cell, we ask if we can exit. If yes, we are done. We exit with $R1=1$.
- b. If not, we put a **breadcrumb** in our cell. Our **breadcrumb** is bit [15] of the word corresponding to our current cell. We set it to 1.
- c. We ask two questions: Is there a door to the north, and have we never visited the cell to the north before? If the answer to both is yes, we set the address to the cell to the north, and JSR FIND_EXIT.
We set the address to the cell to the north by simply subtracting 6 from the address of the current cell. Why 6? Because the cells are stored in row major order, and the number of columns in the maze is 6.
- d. If the answer to either question is no, or if going north resulted in failure, we ask: Is there a door to the east, and have we never visited that cell before? If the answer to both is yes, we set the address to the address of the cell to the east (by adding 1 to the address) and JSR FIND_EXIT.
- e. If going east does not get us out, we repeat the question for south, and if that does not work, then for west.
- f. If we end up with no door to the west to a cell we have not visited, or if there is a door and we haven't visited, but it results in failure, we are done. We cannot exit the maze from our starting position. We set $R1=0$ and return.

Recursive Subroutine to exit the Maze

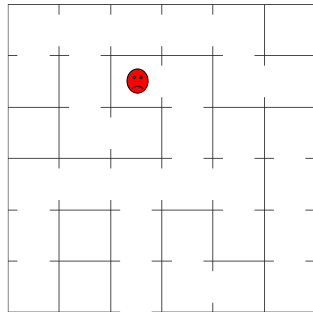
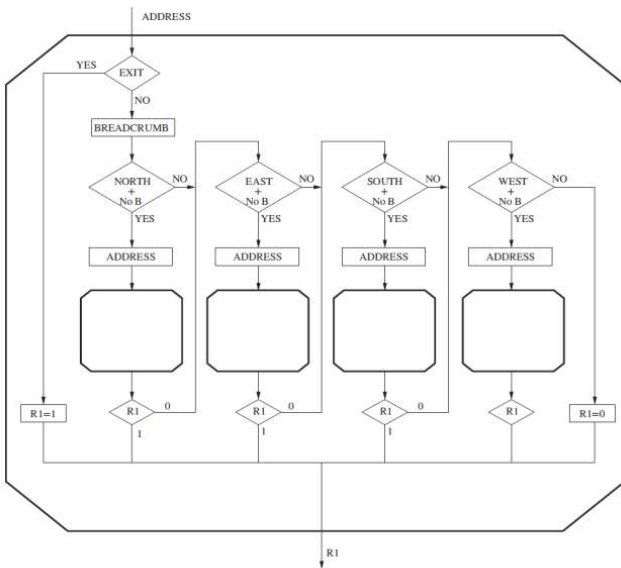


Figure 8.23 Pictorial representation of the recursive subroutine to exit the maze.

2025/2/24

Lecture 1

; Recursive subroutine that determines if there is
 ; a path from current cell to the outside world.
 ; input: R0, current cell address
 ; output: R1, YES (1) or NO (0)

```

.ORIG x4000
01 FIND_EXIT ; save modified registers into the stack.
02 ADD R6, R6, #-1
03 STR R2, R6, #0 ; R2 holds the cell data of the caller
04 ADD R6, R6, #-1
05 STR R3, R6, #0 ; R3 holds the cell address of the caller
06 ADD R6, R6, #-1
07 STR R7, R6, #0 ; R7 holds the PC of the caller
08
09 ; Move cell address to R3, since we need to use R0
0A ; as the input to recursive subroutine calls.
0B ADD R3, R0, #0
0C
0D ; If the exit is in this cell, return YES
0E LDR R2, R0, #0 ; R2 now holds the current cell data
0F LD R7, EXIT_MASK ; EXIT_MASK .FILL x0010
10 AND R7, R2, R7
11 BRnp DONE_YES
12
13 ; Put breadcrumb in the current cell.
14 LD R7, BREADCRUMB ; BREADCRUMB .FILL x8000
15 ADD R2, R2, R7
16 STR R2, R0, #0
17

```

```

18 ; check the north cell for a path to exit
19 CHECK_NORTH LD R7, NORTH_MASK
1A AND R7, R2, R7
1B BRz CHECK_EAST ; If north is blocked, check east
1C LDR R7, R3, #-6
1D BRn CHECK_EAST ; If a breadcrumb in the north
; cell, check east. bit[15]=1, negative
1E ADD R0, R3, #-6
1F JSR FIND_EXIT ; Recursively check the north cell
20 ADD R1, R1, #0
21 BRp DONE_YES ; If a path from north cell found,
22 ; return YES
23 ; check the north cell for a path to exit
24 CHECK_EAST LD R7, EAST_MASK
25 AND R7, R2, R7
26 BRz CHECK_SOUTH ; If the way to east is
; blocked, check south
27 LDR R7, R3, #1
28 BRn CHECK_SOUTH ; If a breadcrumb in the
; east cell, check south
29 ADD R0, R3, #1
2A JSR FIND_EXIT ; Recursively check the east cell
2B ADD R1, R1, #0
2C BRp DONE_YES ; If a path from east cell found,
; return YES,

```

```

2E ; check the south cell for a path to exit
2F CHECK_SOUTH LD R7, SOUTH_MASK
30      AND R7, R2, R7
31      BRz CHECK_WEST ; If the way to south is blocked,
                        ;check west
32      LDR R7, R3, #6
33      BRn CHECK_WEST ; If a breadcrumb in the south
                        ;cell, check west
34      ADD R0, R3, #6
35      JSR FIND_EXIT ; Recursively check the south cell
36      ADD R1, R1, #0
37      BRp DONE_YES ; If a path from south cell found,
                        ;return YES
38
39 ; check the west cell for a path to exit
3A CHECK_WEST LD R7, WEST_MASK
3B      AND R7, R2, R7
3C      BRz DONE_NO ; If the way to west is blocked,
                        ; return NO
3D      LDR R7, R3, #-1
3E      BRn DONE_NO ; If a breadcrumb in the west cell,
                        ; return NO
3F      ADD R0, R3, #-1
40      JSR FIND_EXIT ; Recursively check the west cell
41      ADD R1, R1, #0
42      BRp DONE_YES ; If a path from west cell found,
                        ; return YES
43
44 DONE_NO AND R1, R1, #0
45      BR RESTORE

```

```

46
47 DONE_YES AND R1, R1, #0
48      ADD R1, R1, #1
49
4A RESTORE ADD R0, R3, #0 ; restore R0 from R3
;
4B ; restore the rest of the modified registers
;from the stack.
4C      LDR R7, R6, #0
4D      ADD R6, R6, #1
4E      LDR R3, R6, #0
4F      ADD R6, R6, #1
50      LDR R2, R6, #0
51      ADD R6, R6, #1
52      RET
53
54 BREADCRUMB .FILL x8000
55 EXIT_MASK .FILL x0010
56 NORTH_MASK .FILL x0008
57 EAST_MASK .FILL x0004
58 SOUTH_MASK .FILL x0002
59 WEST_MASK .FILL x0001
5A      .END

```

2.The Queue

- **Definition of Queue**
- **Basic Operations**
- **Wrap-Around**
- **Definition of full and empty queue**
- **Underflow/Overflow**

2.1 The Definition of Queue

■ Queue

- A data structure with the property of “First in First out (FIFO)”;
- **Front** pointer for removing elements from the front of the queue; **Rear** pointer for inserting into the rear of the queue.
- **FRONT** points to the location just **in front of the first element** in the queue; **REAR** points to the location of **the last element** in the queue.

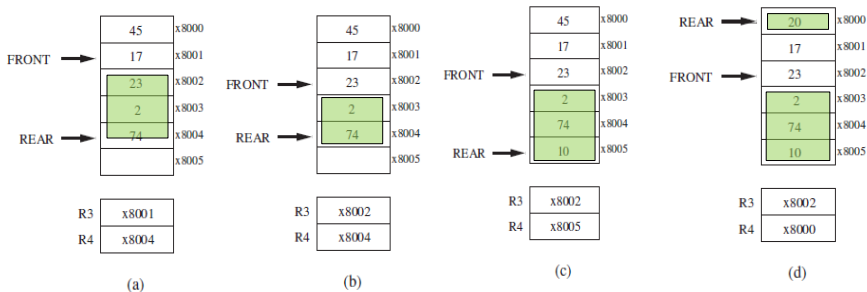


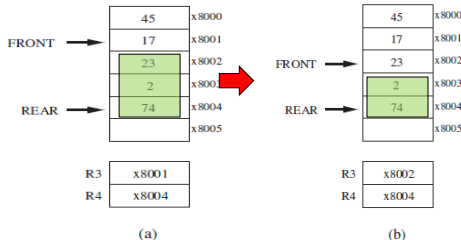
Figure 8.25 A queue allocated to memory locations x8000 to x8005.

2.2 Basic Operations

■ Remove from **Front**

- FRONT points to the location just in front of the first element in the queue; R3 stores the FRONT pointer;
- First incrementing FRONT; then loading the value.

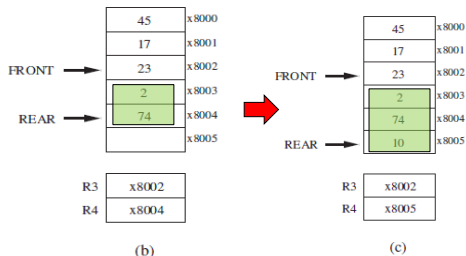
ADD R3,R3,#1
LDR R0,R3,#0



■ Insert at **Rear**

- First incrementing REAR; then storing the value. R4 stores the REAR pointer;

ADD R4,R4,#1
STR R0,R4,#0



2.3 Wrap-Around

■ Remove

;R3 stores FRONT pointer

LD R2, LAST

ADD R2,R3,R2

BRnp SKIP_1

LD R3,FIRST

BR SKIP_2

SKIP_1 ADD R3,R3,#1

SKIP_2 LDR R0,R3,#0

; R0 gets the front of the queue

RET

LAST .FILL x7FFB

; LAST contains the negative of 8005

FIRST .FILL x8000

■ Insert

;R4 stores REAR pointer

LD R2, LAST

ADD R2,R4,R2

BRnp SKIP_1

LD R4,FIRST

BR SKIP_2

SKIP_1 ADD R4,R4,#1

SKIP_2 STR R0,R4,#0

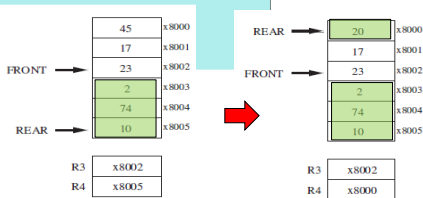
; R0 gets the front of the queue

RET

LAST .FILL 7FFB

; LAST contains the negative of 8005

FIRST .FILL x8000



2.4 Full and Empty Queue

- The queue are allowed to store only **n-1 (why?)** elements for a queue with n locations.
- **Full:** $\text{FRONT} = \text{REAR} + 1$ **OR** $\text{FRONT} + n - 1 = \text{REAR}$
- **Empty:** $\text{FRONT} = \text{REAR}$

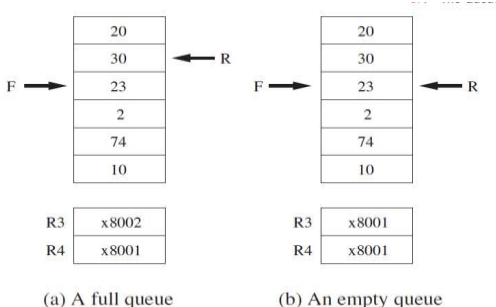


Figure 8.26 A full queue and an empty queue.

2.5 Tests for Underflow, Overflow

R5=1: underflow
R5=0: normal

R3: FRONT
R4: REAR

■ Test for underflow **FRONT =? REAR**

AND R5,R5,#0 ; Initialize R5 to 0

NOT R2,R3

ADD R2,R2,#1 ; R2 contains negative of R3

ADD R2,R2,R4

BRz **UNDERFLOW** ; R3 = R4

.....

; code to remove the front of the ;queue and return
success.

RET

UNDERFLOW ADD R5,R5,#1

RET

■ Test for overflow

● **How?**

2.6 The Complete Story

```
00 ;Input: R0 for item to be inserted, R3 is FRONT, R4 is REAR
01 ;Output: R0 for item to be removed
02 ;
03 INSERT ST R1,SaveR1 ; Save register we need
04     AND R5,R5,#0 ; Set R5 to success code
05     ; Initialization complete
06     LD R1,NEG_LAST
07     ADD R1,R1,R4 ; R1 = REAR MINUS x8005
08     BRnp SKIP1 ; SKIP WRAP AROUND
09     LD R4,FIRST ; WRAP AROUND, R4=x8000
0A     BR SKIP2
0B SKIP1 ADD R4,R4,#1 ; NO WRAP AROUND, R4=R4+1
0C SKIP2 NOT R1,R4
0D     ADD R1,R1,#1 ; R1= NEG REAR
0E     ADD R1,R1,R3 ; R1= FRONT-REAR
0F     BRz FULL
10     STR R0,R4,#0 ; DO THE INSERT
11     BR DONE
12 FULL LD R1,NEG_FIRST ; to decrement R4
13     ADD R1,R1,R4 ; R1 = REAR MINUS x8000
14     BRnp SKIP3
15     LD R4,LAST ; UNDO WRAP AROUND, REAR=x8005
16     BR SKIP4
17 SKIP3 ADD R4,R4,#-1 ; NO WRAP AROUND, R4=R4-1
18 SKIP4 ADD R5,R5,#1 ; R5=FAILURE
19     BR DONE
1A ;
1B REMOVE ST R1,SaveR1 ; Save register we need
1C     AND R5,R5,#0 ; Set R5 to success code
1D ; Initialization complete
1E     NOT R1,R4
1F     ADD R1,R1,#1 ; R1= NEG REAR
20     ADD R1,R1,R3 ; R1= FRONT-REAR
21     BRz EMPTY
22     LD R1, NEG_LAST
23     ADD R1,R1,R3 ; R1= FRONT MINUS x8005
24     BRnp SKIP5
25     LD R3, FIRST ; R3=x8000
26     BR SKIP6
27 SKIP5 ADD R3,R3,#1 ; R3=R3+1
28 SKIP6 LDR R0,R3,#0 ; DO THE REMOVE
29     BR DONE
2A EMPTY ADD R5,R5,#1 ; R5=FAILURE
2B DONE LD R1,SaveR1 ; Restore register
2C     RET
2D FIRST .FILL x8000
2E NEG_FIRST .FILL x8000
2F LAST .FILL x8005
30 NEG_LAST .FILL x7FFB
31 SaveR1 .BLKW 1
```

3. Character Strings

- one-dimensional array of ASCII codes often followed by x0000 (null character).
- Each location stores an ASCII codes of a character.

| | |
|-------|-------|
| x5000 | x0042 |
| | x0069 |
| | x006C |
| | x006C |
| | x0020 |
| | x004C |
| | x0069 |
| | x006E |
| | x0076 |
| | x0069 |
| | x006C |
| | x006C |
| | x0000 |

Figure 8.28 Character string representing the name "Bill Linvill."

Example 1: Personnel Record

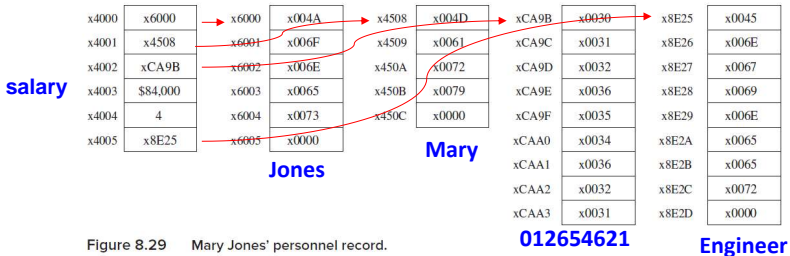


Figure 8.29 Mary Jones' personnel record.

■ 6 words of sequential memory starting at x4000:

1. The first word contains the starting address of a character string containing the person's **last** name. The pointer in location x4000 is the address x6000. The six-word character string, starting at x6000, contains the ASCII code for "**Jones**," terminated with the null character.
2. The second word, at x4001, contains a pointer to the character string of the person's **first** name, in this case "**Mary**," starting at location x4508.
3. The third word, at x4002, contains a pointer (xCA9B) to her **nine-digit social security number**, the unique identifier for all persons working in the United States.
4. The fourth word, at x4003, contains her **salary** (in thousands of dollars).
5. The fifth word contains **how long** she has worked for the company.
6. The sixth word is a pointer (x8E25) to the character string identifying her job title, "**Engineer**."

Subroutine to compare two character strings

■ Given social security number in example 1, how to look for the salary?

```
STRCMP      ST R0,SaveR0
            ST R1,SaveR1
            ST R2,SaveR2
            ST R3,SaveR3
;R0 points to 1st string, R1 points to 2nd string
            AND R5,R5,#0 ; R5 <-- Match;
NEXTCHAR    LDR R2,R0,#0
; R2 contains character from 1st string
            LDR R3,R1,#0
; R3 contains character from 2nd string
            BRnp COMPARE
; String is not done, continue comparing
            ADD R2,R2,#0
            BRz DONE
; If both strings done, match found
COMPARE     NOT R2,R2
            ADD R2,R2,#1
; R2 contains negative of character
            ADD R2,R2,R3
; Compare the 2 characters
```

```
            BRnp FAIL ; Not equal, no match
            ADD R0,R0,#1
            ADD R1,R1,#1

            BRnzp NEXTCHAR
; Move on to next pair of characters

FAIL        ADD R5,R5,#1 ; R5 <-- No match
;
DONE        LD R0,SaveR0
            LD R1,SaveR1
            LD R2,SaveR2
            LD R3,SaveR3
            RET
SaveR0 .BLKW 1
SaveR1 .BLKW 1
SaveR2 .BLKW 1
SaveR3 .BLKW 1
```


Example 2: Character String Containing an “Integer”

- Represent a long integer of any length by character strings, ensuring all characters are within 0-9.

; Input: R0 contains the starting address of the character string

; Output: R5=0, success; R5=1, failure.

;

TEST_INTEGER

ST R1,SaveR1

; Save registers needed by subroutine

ST R2,SaveR2

ST R3,SaveR3

ST R4,SaveR4

;

AND R5,R5,#0 ; Initialize success code to R5=0, success

LD R2,ASCII_0 ; R2=xFFD0, the negative of ASCII code x30

LD R3,ASCII_9 ; R3=xFFC7, the negative of ASCII code x39

;

NEXT_CHAR LDR R1,R0,#0 ; Load next character

BRz SUCCESS ; if current character is null

ADD R4,R1,R2 ; R1 - 0x30h

BRn **BAD** ; R1 is less than x30, not a decimal digit

ADD R4,R1,R3 ; R1 - 0x39h

BRp **BAD** ; R1 is greater than x39, not a decimal digit

ADD R0,R0,#1 ; Character good! Prepare for next character

BR **NEXT_CHAR**

BAD ADD R5,R5,#1 ; R5 contains failure code

SUCCESS LD R4,SaveR4 ; Restore registers

LD R3,SaveR3

LD R2,SaveR2

LD R1,SaveR1

RET

ASCII_0 .FILL xFFD0

ASCII_9 .FILL xFFC7

SaveR1 .BLKW 1

SaveR2 .BLKW 1

SaveR3 .BLKW 1

SaveR4 .BLKW 1

| |
|-------|
| x0037 |
| x0039 |
| x0032 |
| x0034 |
| x0035 |
| x0000 |

Figure 8.31 A character string representing the integer 79,245, with one ASCII code per decimal digit.



中国科学技术大学
University of Science and Technology of China

计算系统概论A

Introduction to Computing Systems
(CS1002A.02)

Chapter 9-1 Memory Mapped I/O

计算机科学与技术学院
School of Computer Science and Technology



1 Review

2 The Memory Address Space

3 Input/Output



1 Review

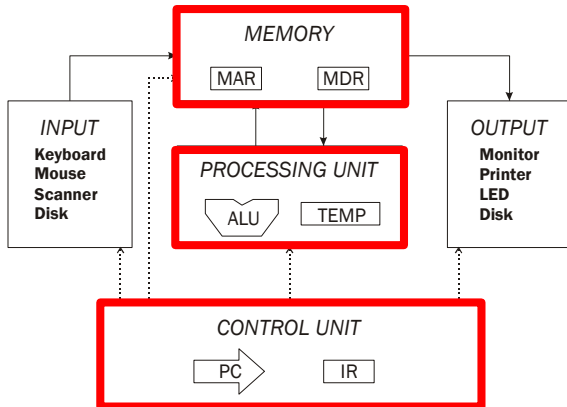
2 The Memory Address Space

3 Input/Output

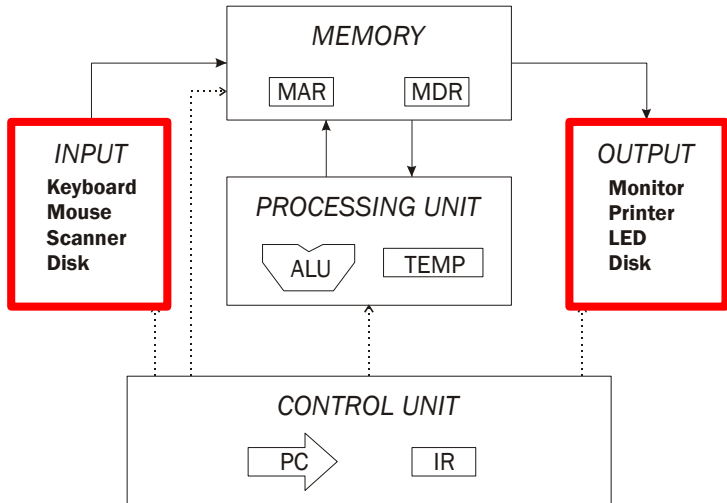
Review

■ So far, we' ve learned how to:

- compute with values in registers
- load data from memory to registers
- store data from registers to memory



Today: I/O in Von Neumann Model





1 Review

2 The Memory Address Space

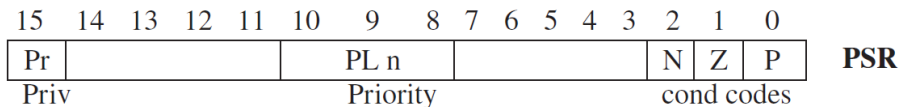
3 Input/Output

Privilege and Priority

- Two very different concepts associated with computer processing are **privilege** and **priority**.
 - **Privilege** is all about the right to do something, such as execute a particular instruction or access a particular memory location. Not all computer programs have the right to execute all instructions.
 - We say a program is executing in **Supervisor mode** to indicate privileged, or **User mode** to indicate unprivileged.
 - **Priority** is all about the urgency of a program to execute.
 - allows programs of greater urgency to interrupt programs of lesser urgency.
- **privilege** and **priority** are two **orthogonal** notions
 - They have nothing to do with each other.

The Processor Status Register (PSR)

- Each program executing on the computer has associated with it two very important registers.
 - The Program Counter (**PC**)
 - and the Processor Status Register (**PSR**) which contains the **privilege** and **priority** assigned to that program.



- PSR[15]=0 means **supervisor privilege**, and PSR[15]=1 means **unprivileged**.
- Bits [10:8] specify the **priority level** (PL) of the program. The highest priority level is 7 (PL7), the lowest is PL0.
- The PSR also contains the current values of the **condition codes**

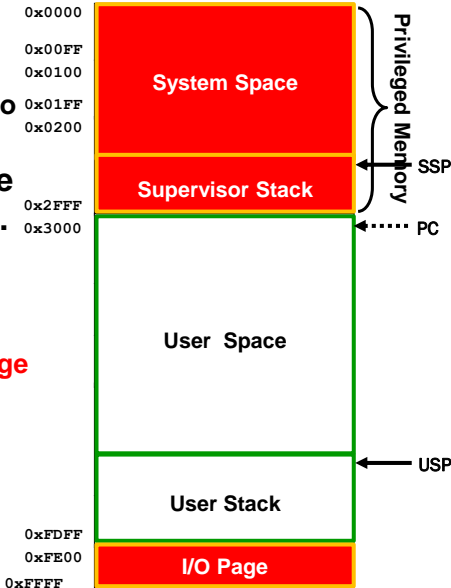
Organization of Memory

■ LC-3 has a 16-bit address space

- memory locations from x0000 to xFFFF.

■ Locations x0000 to x2FFF are **privileged** memory locations.

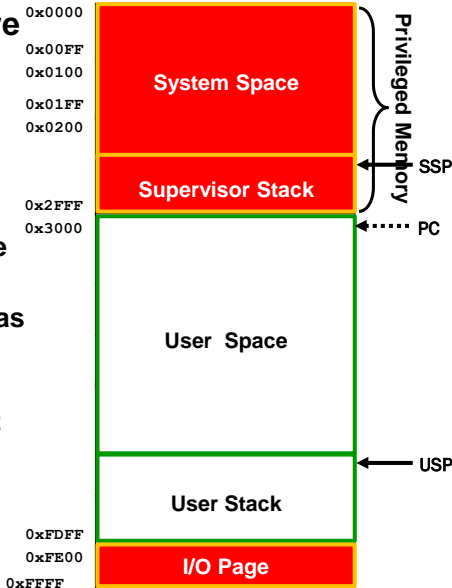
- They contain the various data structures and code of the operating system.
- They require **supervisor privilege** to access.
- They are referred to as system space.
- The **supervisor stack** is controlled by the operating system.



Organization of Memory

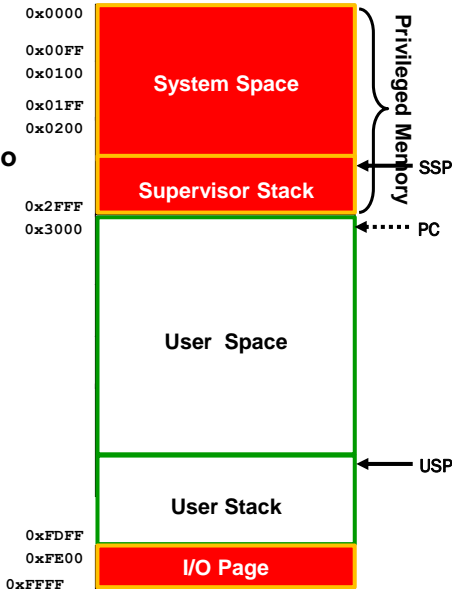
■ Locations x3000 to xFDFF are **unprivileged** memory locations.

- Supervisor privilege is **not required** to access these memory locations.
- All user programs and data use this region of memory.
- The region is often referred to as user space.
- The user stack is controlled by the user program and does not require privilege to access.



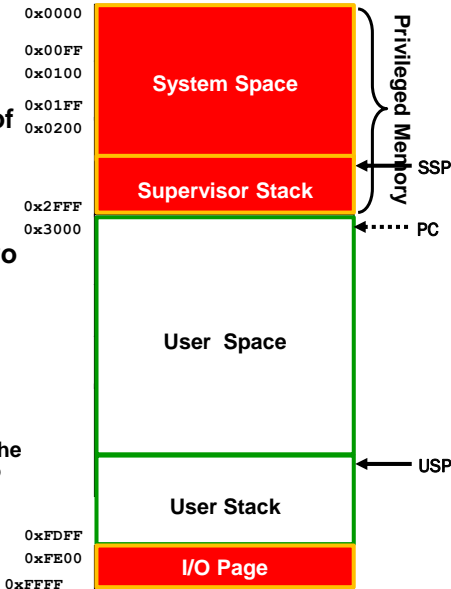
Organization of Memory

- Addresses xFE00 to xFFFF do not correspond to **memory locations**
 - The last address of a memory location is xFDFF.
- The set of addresses from xFE00 to xFFFF is usually referred to as the **I/O page** since most of the addresses are used for **identifying registers** that take part in input or output functions.
- The set of addresses are part of **privileged memory** address space and accessible only to programs that have **supervisor privilege**.



Organization of Memory

- For the two stacks, each has a stack pointer, **Supervisor Stack Pointer (SSP)** and **User Stack Pointer (USP)**, to indicate the top of the stack.
- Since a program can only execute in Supervisor mode or User mode at any one time, only one of the two stacks is active at any one time.
- **Two registers, Saved_SSP and Saved_USP, are provided to save the SP not in use.**
 - When privilege changes, for example, from Supervisor mode to User mode, the SP is stored in Saved_SSP, and the SP is loaded from Saved_USP.





1 Review

2 The Memory Address Space

3 Input/Output

Input / Output (I/O)

- **Computer systems are useless unless they can process information from outside of the computer and output results outside of the computer**
 - But where does data in memory **come from**?
 - And how does data **get out** of the system?
- **I/O is effective communication with the outside of the computer**
 - **I/O device** itself communicates with outside world, e.g., keyboard takes input from user
 - Computer needs to communicate with **I/O device**, e.g., computer takes input from keyboard
- **Communication through shared memory locations**
 - Processor and I/O can read/write those memory locations
 - Sometimes, data in memory locations can be set/cleared automatically (by hardware) depending on a read/write

I/O: Connecting to the Outside World

■ I/O Examples

- **Keyboard/mouse input, video output** on a standard computer
- **Network input/output** that enables web surfing
- Information **from an engine of a car** that a computer uses to determine how to tune the engine (output from computer tunes the engine)
- Requests for **airline reservations** and replies that service those requests

I/O: Connecting to the Outside World

■ Types of I/O devices characterized by:

- **behavior:** input, output
 - **input:** keyboard, motion detector, network interface
 - **output:** display screen(monitor), printer, network interface
- **data rate:** how fast can data be transferred?
 - keyboard: 100 bytes/sec
 - disk: 60-120 MB/s
 - network: 1 Mb/s - 100 Gb/s
- **accessing mode:**
 - **character device:** no buffering is performed. E.g., keyboard
 - **block device:** accessed through a cache, be random access. E.g., disk

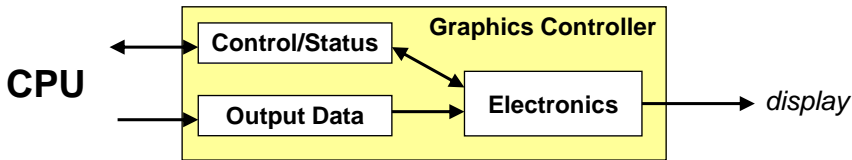
I/O Controller

■ Control/Status Registers

- CPU tells device what to do -- write to control register
- CPU checks whether task is done -- read status register

■ Data Registers

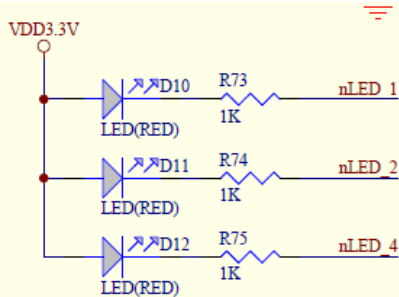
- CPU transfers data to/from device



■ Device electronics

- performs actual operation
 - pixels to screen, bits to/from disk, characters from keyboard

LED, An ARM Example



- 开发板上面的led灯通常接到处理器的GPIO(general purpose input output)
- 通过原理图确定需要控制的IO端口为S3C2440的GPF4、GPF5、GPF6
- 要想点亮LED，需要控制相关的GPIO口输出高电平(1)或低电平(0)

| | | | |
|--------|--------|-----|------------|
| nLED 1 | nLED 1 | M17 | EINT4/GPF4 |
| nLED 2 | nLED 2 | L14 | EINT5/GPF5 |
| nLED 4 | nLED 4 | L15 | EINT6/GPF6 |

LED, An ARM Example

■ 通过S3C2440的DataSheet 可以得知, GPIO主要有三个寄存器需要设置

- GPFCON是**控制寄存器**, 主要是控制GPIO的功能, 主要有输入、输出和中断三个功能。每个GPIO口有寄存器的两位来控制, 00: 输入, **01: 输出**, 10: 中断, 10: 保留。
- GPFDAT是**数据寄存器**, 主要是控制GPIO输出高电平还是低电平, 0: 低电平, 1: 高电平。
- GPFUP 是设置内部上拉电阻的寄存器, 0: 不设置上拉电阻, 1: 设置上拉电阻。

| Register | Address | R/W | Description | Reset Value |
|----------|------------|-----|-------------------------------------|-------------|
| GPFCON | 0x56000050 | R/W | Configures the pins of port F | 0x0 |
| GPFDAT | 0x56000054 | R/W | The data register for port F | Undef. |
| GPFUP | 0x56000058 | R/W | Pull-up disable register for port F | 0x000 |
| Reserved | 0x5600005c | — | — | — |

IO内存地址

LED, An ARM Example

```
.text
.global _start
_start:
    LDR    R0,=0x56000050 /*R0← &GPFCON */
    MOV    R1,#0x00000400 /* R1← 0x00000400 */
    STR    R1,[R0] /* 0x00000400→GPFCON, GPFCON的
                    [11:10]设置为了01, 即是GPF5位输出功能 */
    LDR    R0,=0x56000054 /*R0← &GPFDAT */
    MOV    R1,#0x00000000 /* R1← 0x00000000 */
    STR    R1,[R0] /* 0x00000000→GPFDAT, GPFDAT输出低电平*/
MAIN_LOOP:
    B      MAIN_LOOP
```

Question: ARM 是怎样访问GPIO相关寄存器的?

Some Basic Characteristics of I/O

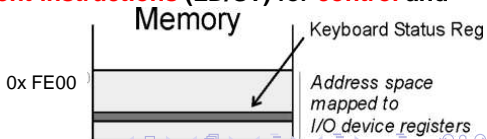
- All I/O activity is controlled by instructions in the computer's ISA. Does the ISA need special instructions for dealing with I/O?
 - Memory-mapped vs. special instructions
- Does the I/O device execute at the same speed as the computer, and if not, what manages the difference in speeds?
 - Asynchronous vs. synchronous
- Is the transfer of information between the computer and the I/O device initiated by a program executing in the computer, or is it initiated by the I/O device?
 - CPU (polling) vs. device (interrupts)

Memory-Mapped I/O vs. Special I/O Instructions

- An instruction that interacts with an input or output device register must **identify the particular input or output device register** with which it is interacting.
- **Special I/O Instructions**
 - These instructions typically allow data to be sent to an I/O device or read from an I/O device.
 - **Coding**: designate opcode(s) for I/O, register and operation encoded in instruction



- **Memory-mapped I/O**
 - assign a memory address to each device register
 - The advantage to this method is that **every instruction** which can **access memory** can be used to **manipulate an I/O device**.
 - In LC3, we can use **data movement instructions (LD/ST)** for **control** and **data transfer**



■ Memory-mapped I/O (Table A.1)

| <i>Location</i> | <i>I/O Register</i> | <i>Function</i> |
|-----------------|--|--|
| xFE00 | Keyboard Status Reg (KBSR) | Bit [15] is one when keyboard has received a new character. |
| xFE02 | Keyboard Data Reg (KBDR) | Bits [7:0] contain the last character typed on keyboard. |
| xFE04 | Display Status Register (DSR) | Bit [15] is one when device ready to display another char on screen. |
| xFE06 | Display Data Register (DDR) | Character written to bits [7:0] will be displayed on screen. |

Asynchronous vs. Synchronous

■ I/O events generally happen much slower than CPU cycles.

- If : CPU 300MHz, 10clocks/character, 6characters/word
- Then: typing speed $(300 \times 10^6) / (10 \times 6) = 5 \times 10^6$ words/s

■ Synchronous

- data supplied at a fixed, predictable rate, and CPU reads/writes every X cycles. (**Limitation?**)

■ Asynchronous

- I/O devices usually operate at speeds **very different** from that of a microprocessor.
- To control processing in an asynchronous world requires some **protocol** or **handshaking** mechanism.
 - In the case of the keyboard, use a **one-bit status register**, called a **flag**, to indicate if someone has or has not typed a character.
- These flags are the simplest form of **synchronization(?)**.

Interrupt-Driven vs. Polling

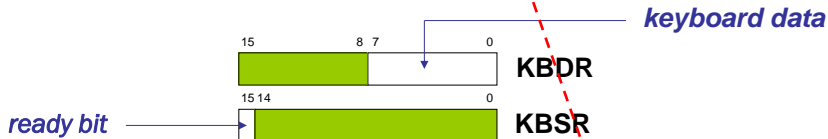
- Who determines when the next data transfer occurs?
 - CPU vs. I/O device
- **Polling** is explicitly looking/examining
 - CPU keeps checking status register until new data arrives OR device ready for next data
 - “Are you there yet? Are you there yet? Are you there yet?”
- **Interrupts** is a nudge, knock on the door, loud noise, which forces you to pay attention
 - Device sends a special signal to CPU when new data arrives OR device ready for next data
 - CPU can be performing other tasks instead of polling device.
 - “Wake me when you get there.”

Example: Input from Keyboard

| Location | I/O Register | Function |
|--------------|-------------------------------------|--|
| xFE00 | Keyboard Status Reg (KBSR) | Bit [15] is one when keyboard has received a new character. |
| xFE02 | Keyboard Data Reg (KBDR) | Bits [7:0] contain the last character typed on keyboard. |

■ When a character is typed:

- its ASCII code is placed in bits [7:0] of KBDR (bits [15:8] are always zero)
- the “ready bit” (KBSR[15]) is set to one (**who sets?**)
- keyboard is disabled -- any typed characters will be ignored



■ When KBDR is read:

- KBSR[15] is set to zero (**who sets?**), meaning no keyboard key is pending
- keyboard is enabled

Electronic circuits associated with the keyboard

Memory-mapped Operations

■ How do we read ready bit?

```
LDI R0, KBSR
```

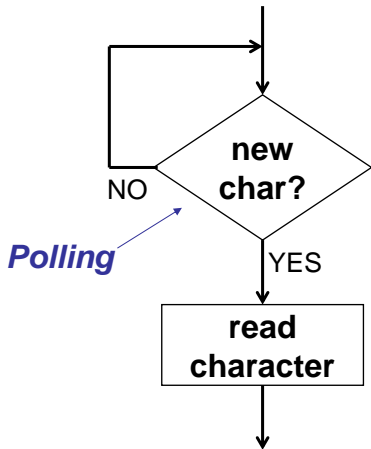
■ How do we test whether the bit is one?

Negative, so BRn, or BRzp

■ How do we read keyboard data?

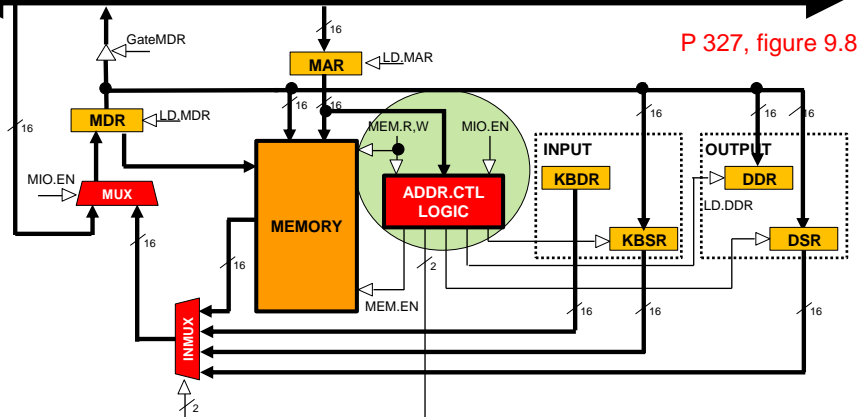
```
LDI R0, KBDR
```

Basic Input Routine



```
POLL    LDI    R0, KBSR  
        BRzp   POLL  
        LDI    R0, KBDR  
  
        ...  
  
KBSR    .FILL  xFE00  
KBDR    .FILL  xFE02
```

Implementation of Memory-mapped I/O

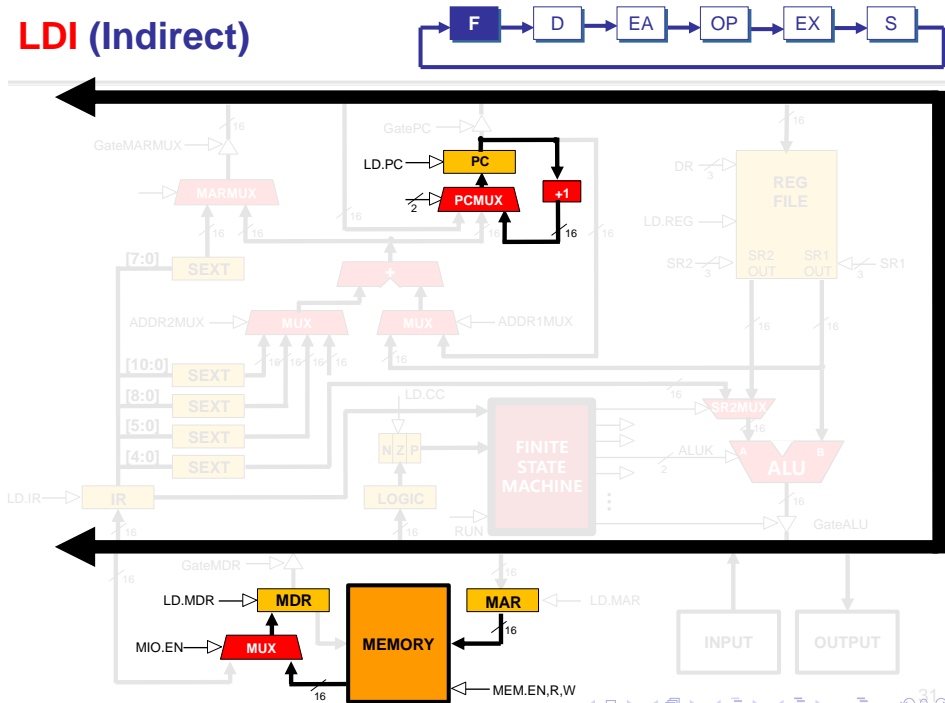


- **MIO.EN** indicates whether a data movement from/to memory or I/O is to take place this clock cycle.
- **MAR** contains the address of the memory location or the memory-mapped address of an I/O device register.
- **R.W** indicates whether a load or a store is to take place.

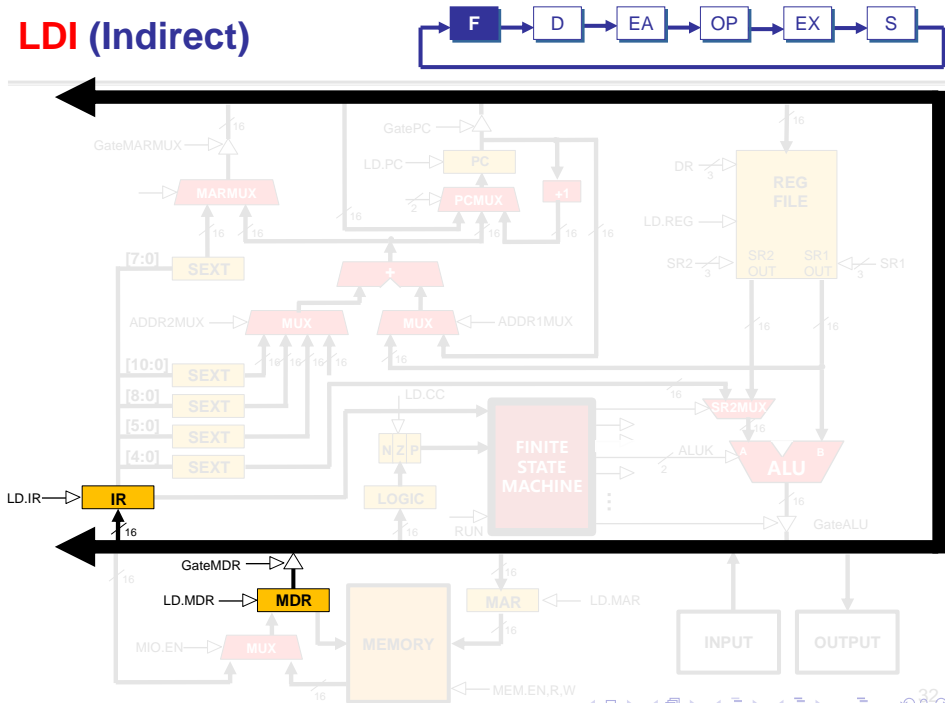
```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



LDI (Indirect)



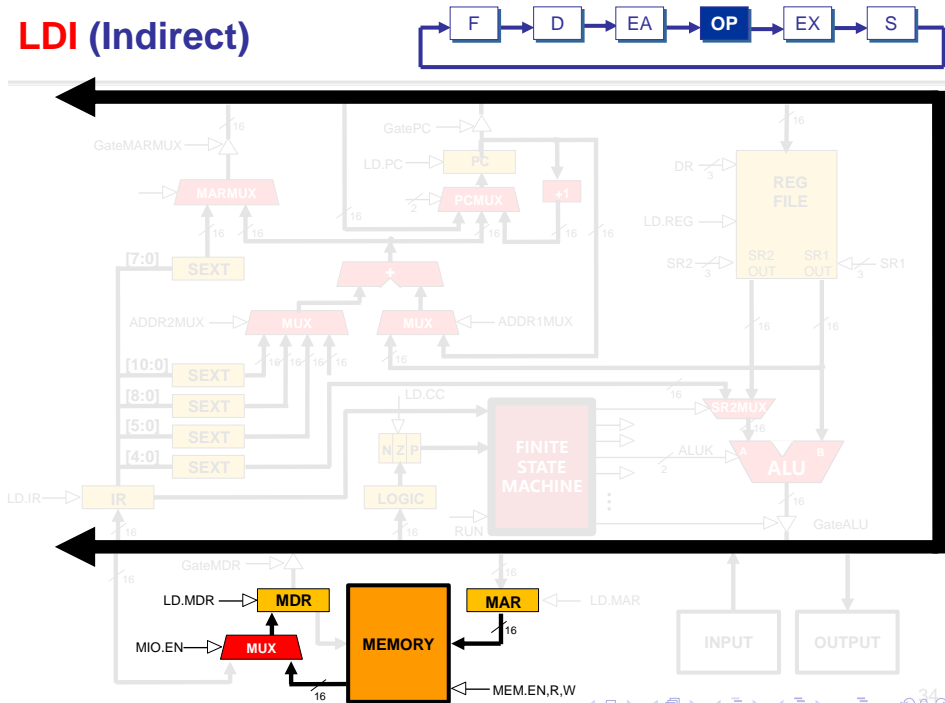
LDI (Indirect)



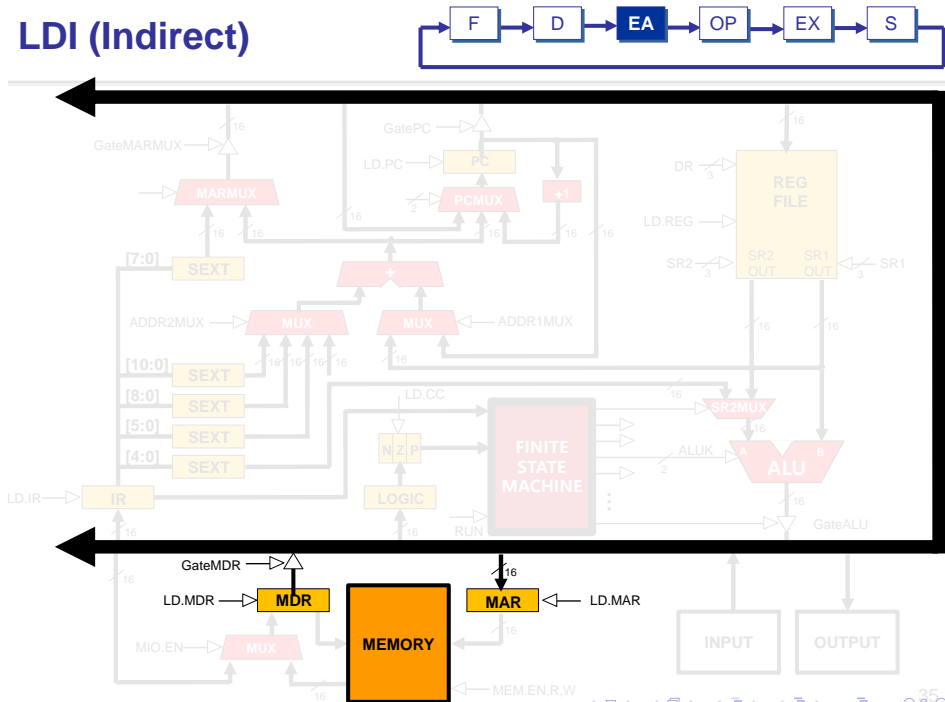
```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP[OP]; OP --> EX[EX]; EX --> S[S]; S --> F;
```



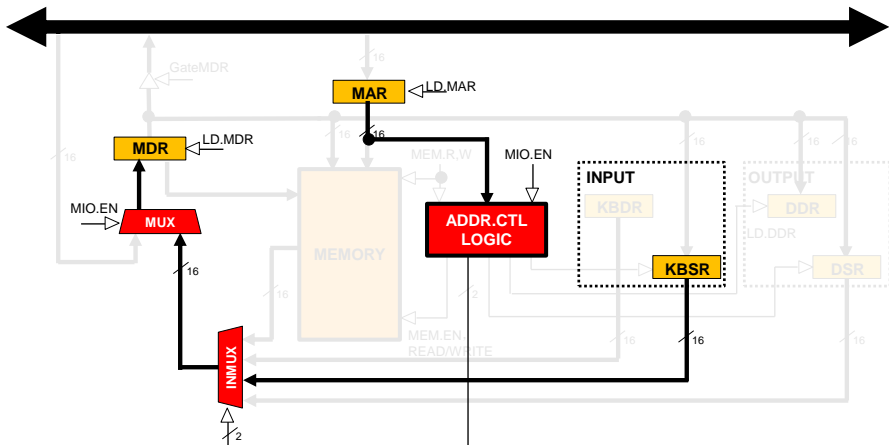
LDI (Indirect)



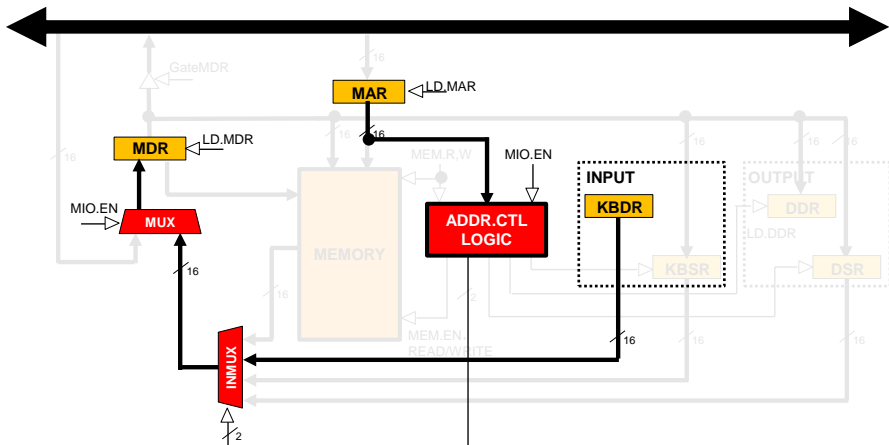
LDI (Indirect)



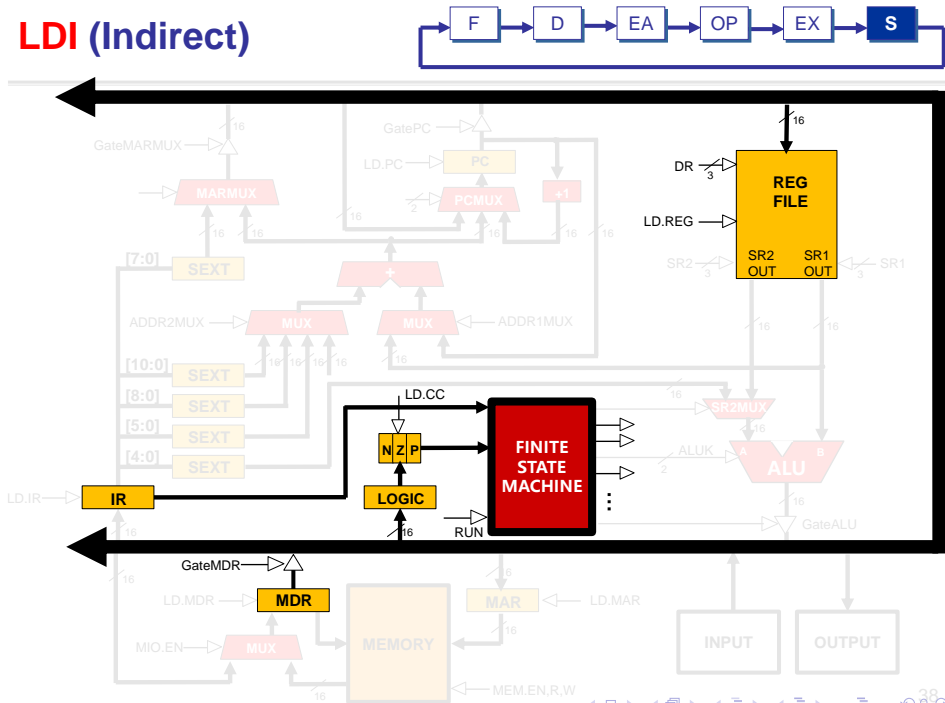
Memory-mapped I/O: **LDI R0, KBSR**



Memory-mapped I/O: **LDI R0, KBDR**



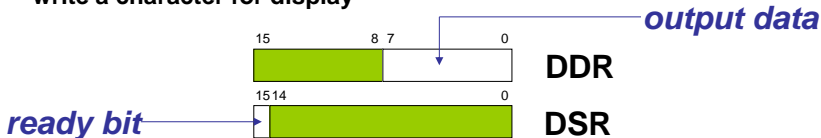
LDI (Indirect)



Example: Output to Screen

■ When Display device is ready to display another character:

- the “ready bit” (DSR[15]) is set to **one**, indicating that processor can write a character for display

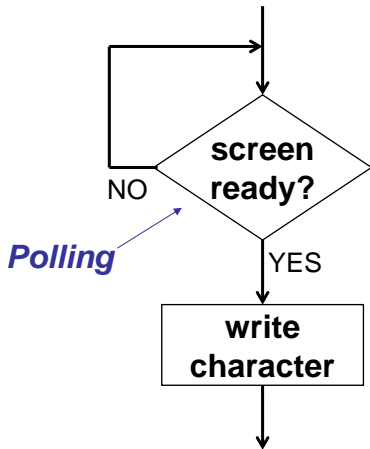


■ When data is written to the Display data register:

- DSR[15] is automatically set to 0
- character in DDR[7:0] is displayed
- any other character data written to DDR is ignored (while DSR[15] is zero)

| Location | I/O Register | Function |
|--------------|--|--|
| xFE04 | Display Status Register (DSR) | Bit [15] is one when device ready to display another char on screen. |
| xFE06 | Display Data Register (DDR) | Character written to bits [7:0] will be displayed on screen. |

Basic Output Routine



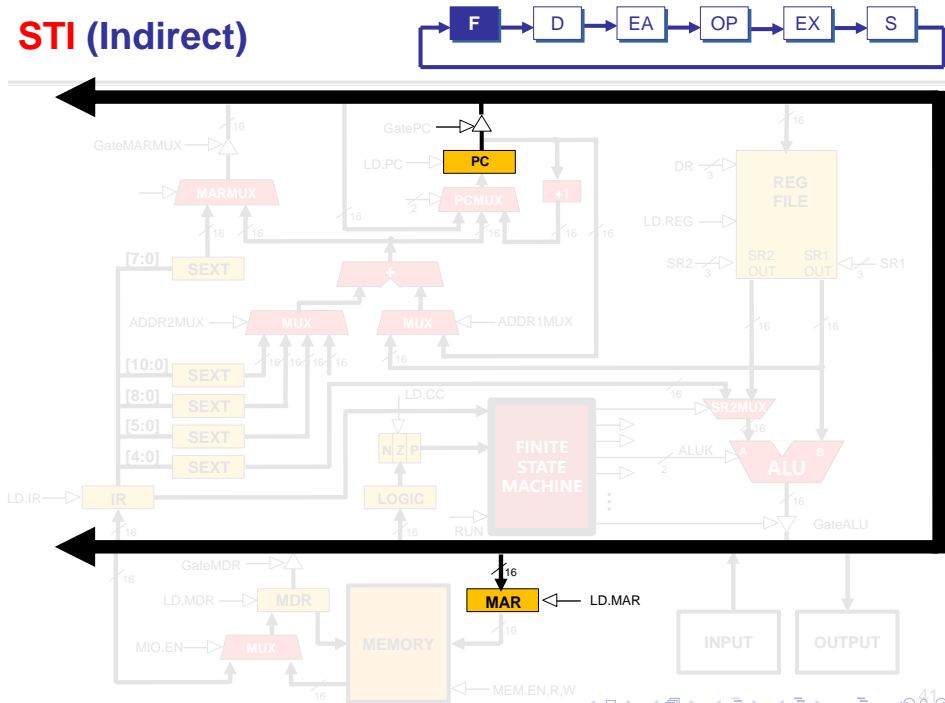
```
POLL    LDI    R1, DSR
        BRzp   POLL
        STI    R0, DDR

        ...

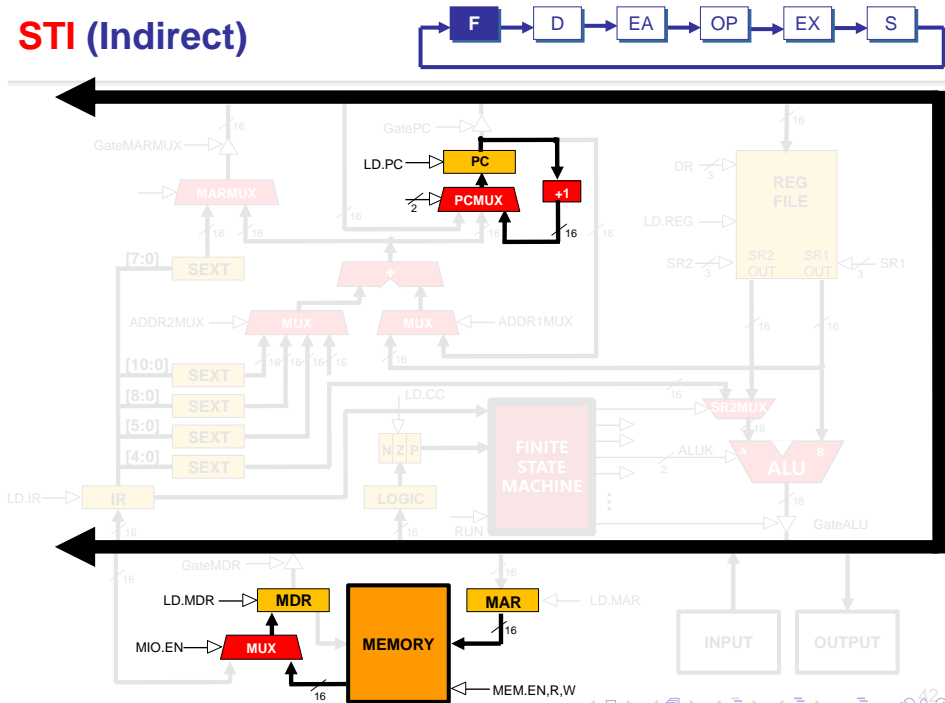
DSR      .FILL  xFE04
DDR      .FILL  xFE06
```

the “ready bit” (DSR[15]) is set to **one**, indicating that processor can write a character for display

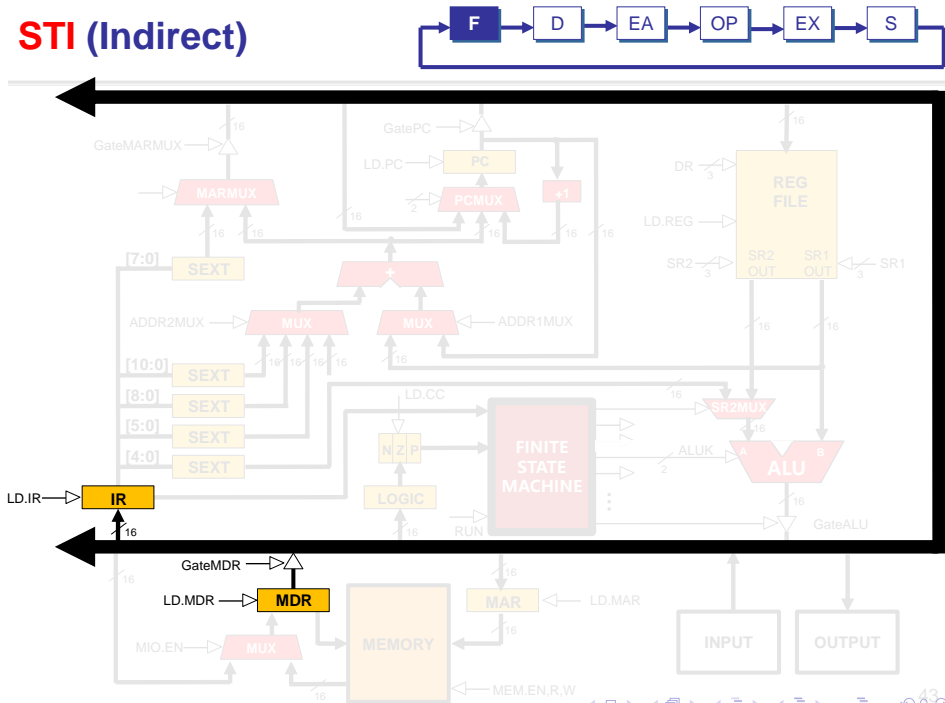
STI (Indirect)



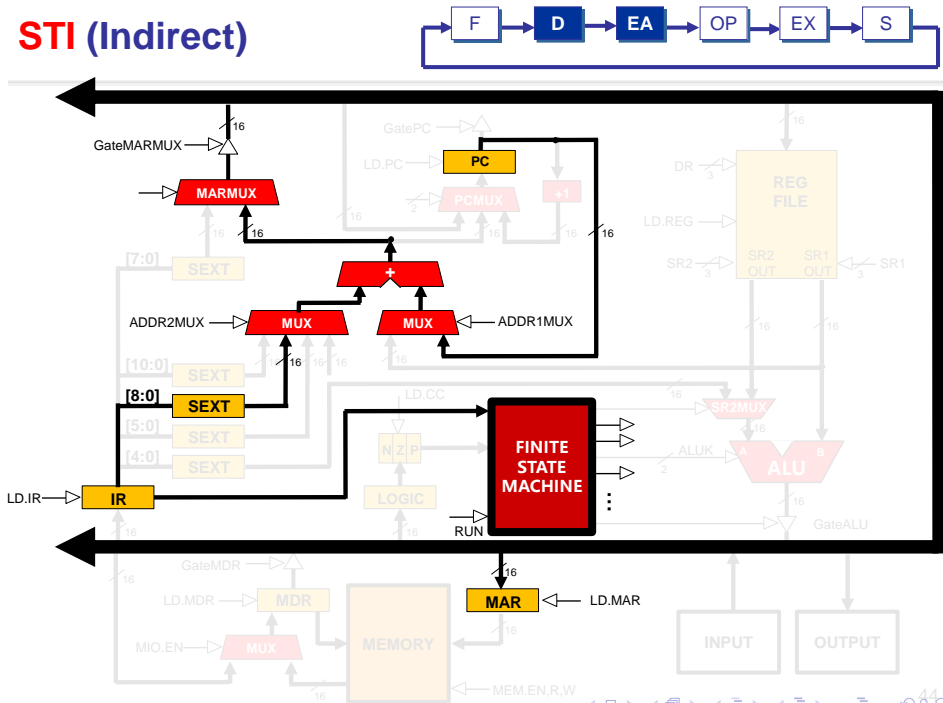
STI (Indirect)



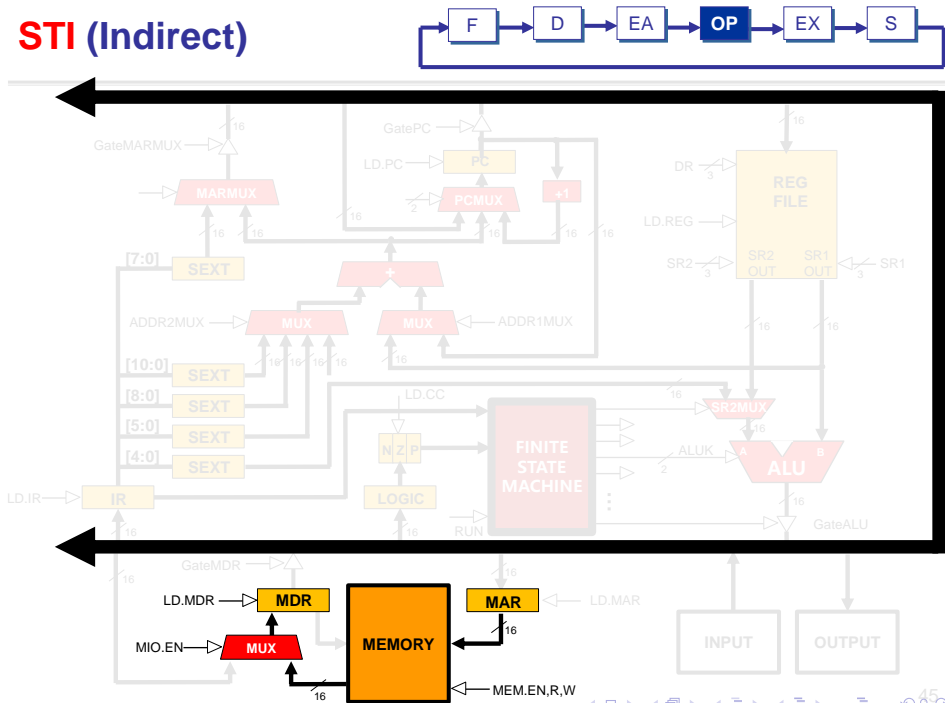
STI (Indirect)



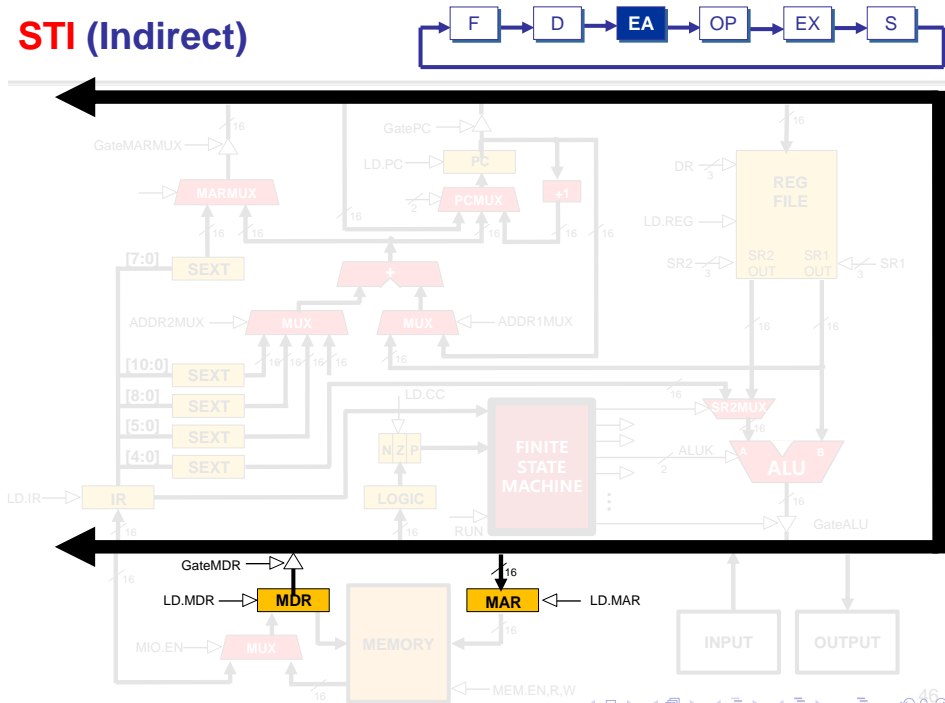
STI (Indirect)



STI (Indirect)



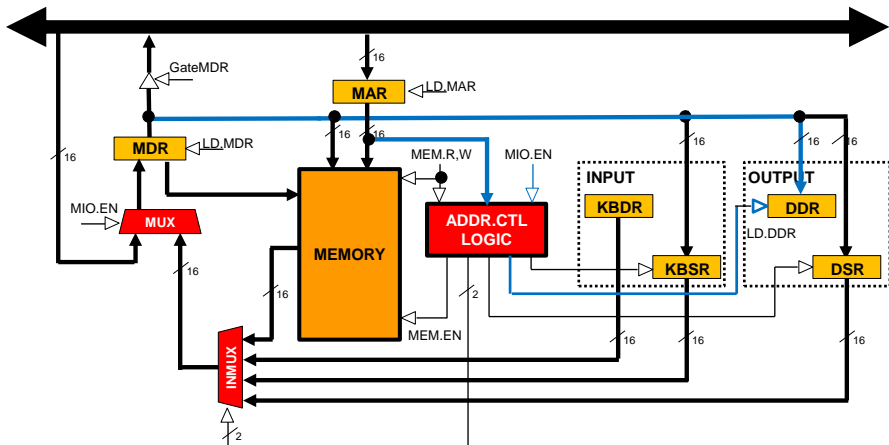
STI (Indirect)



```
graph LR; F[F] --> D[D]; D --> EA[EA]; EA --> OP1[OP]; OP1 --> EX[EX]; EX --> OP2[OP]; OP2 --> F;
```



Memory-mapped I/O: STI R0, DDR ?



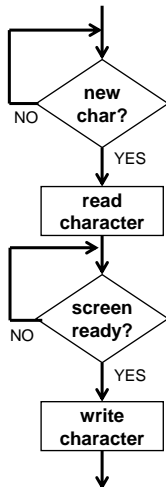
Keyboard Echo Routine

- Usually, input character is also printed to screen.
 - User gets feedback on character typed and knows its OK to type the next character.

```
POLL1    LDI    R0, KBSR
          BRzp   POLL1
          LDI    R0, KBDR
POLL2    LDI    R1, DSR
          BRzp   POLL2
          STI    R0, DDR

          ...

KBSR     .FILL  xFE00
KBDR     .FILL  xFE02
DSR      .FILL  xFE04
DDR      .FILL  xFE06
```





中国科学技术大学
University of Science and Technology of China

计算系统概论A

Introduction to Computing Systems
(CS1002A.02)

Chapter 9-2 Operating System Service Routines

计算机科学与技术学院
School of Computer Science and Technology

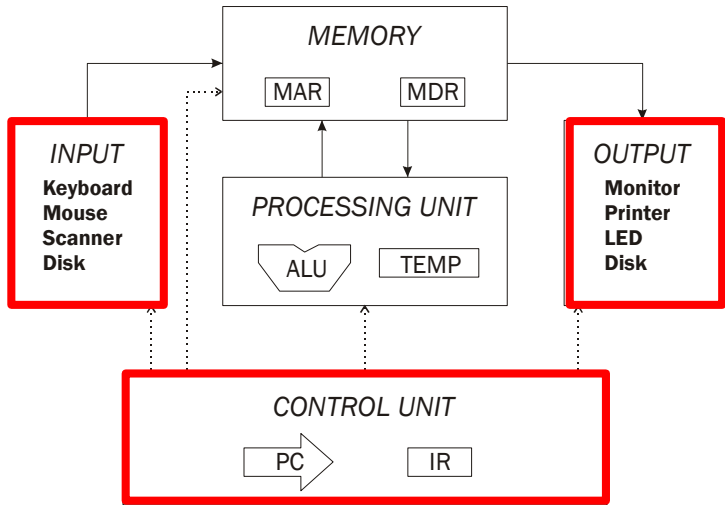


1 Review

2 TRAP Routines

3 Operating System Service Routines

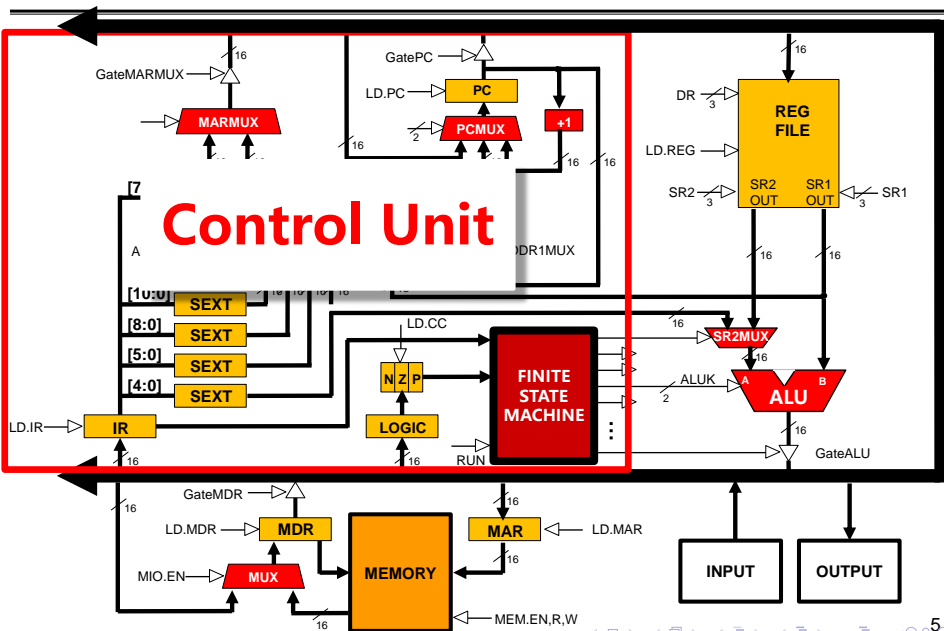
Today: CONTROL in Von Neumann Model



Control Instructions for System Calls

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|----|----|----|----|----|------------|---|-----------|-------------|---|---|---|---|---|---|---|
| BR | 0 | 0 | 0 | 0 | n | z | p | PCOffset9 | | | | | | | | |
| JSR | 0 | 1 | 0 | 0 | 1 | PCOffset11 | | | | | | | | | | |
| JSRR | 0 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JMP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | BaseR | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RET | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| TRAP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | TrapVector8 | | | | | | | |

Today: CONTROL in LC-3 Data Path





1 Review

2 TRAP Routines

3 Operating System Service Routines

System Calls

- **Some operations require specialized knowledge and protection**
 - Understanding IO device registers and how to use them poses a challenge to most application programmers.
 - I/O registers are shared by many programs, and in general it is ill-advised to give user programmers access to these registers.
 - **Solution: *service routines* or *system calls***
 - Low-level, privileged operations performed by operating system
1. User program invokes system call
 2. Operating system code:
 - Saves registers
 - Performs operation
 - Restores registers
 3. Returns control to user program

LC-3 TRAP Mechanism

■ Provides **a set of service routines**

- Part of operating system -- routines start at arbitrary addresses
- Up to 256 routines

■ Requires **a table of starting addresses**

- Stored in memory (x0000 through x00FF)
- Used to associate code with trap number
- Called **Trap Vector Table** (or **System Control Block**)

■ Uses **the TRAP instruction**

- When a user program wishes to have the operating system execute **a specific service routine** on behalf of the user program, and then return control to the user program

■ Uses **a linkage mechanism**

- For returning control to the user program
- Execution resumes immediately after the TRAP instruction
- Using the **RTI instruction**

LC-3 TRAP Routines(*service routines*)

| | | |
|-------|-------|--------------------|
| x0000 | | 预留代码空间 (实用代码空间) |
| : | : | : |
| x0020 | x0400 | 48 |
| x0021 | x0430 | 32(14) |
| x0022 | x0450 | 96 |
| x0023 | x04A0 | 64(42) |
| x0024 | x04E0 | |
| x0025 | x0520 | |
| : | : | : |
| x00FF | | |

The LC-3 Trap
Vector Table

- GETC (TRAP x20)

- Read a single character from keyboard.
- The character is **not echoed** to the console.
- **ASCII** copied into R0 and R0[15:8] is cleared.

- OUT (TRAP x21)

- Write R0[7:0] to the console display.

- PUTS (TRAP x22)

- Write a string of ASCII characters to the console display.
- String address in R0.
- Writing terminates with the occurrence of x0000

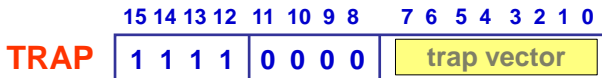
- IN (TRAP x23)

- Print a prompt on the screen and read a single character from the keyboard.
- Character is **echoed** to the console. .
- **ASCII** copied into R0 and R0[15:8] is cleared.

- HALT (TRAP x25)

- Halt execution and print a message on the console.

TRAP Instruction



■ Trap vector

- Identifies which **service routine** the user program wants the operating system to execute on its behalf
- 8-bit trap vector **zero-extended** to form 16-bit address, serves as index into table of service routine addresses

■ Where to go

- Lookup **starting address** from table

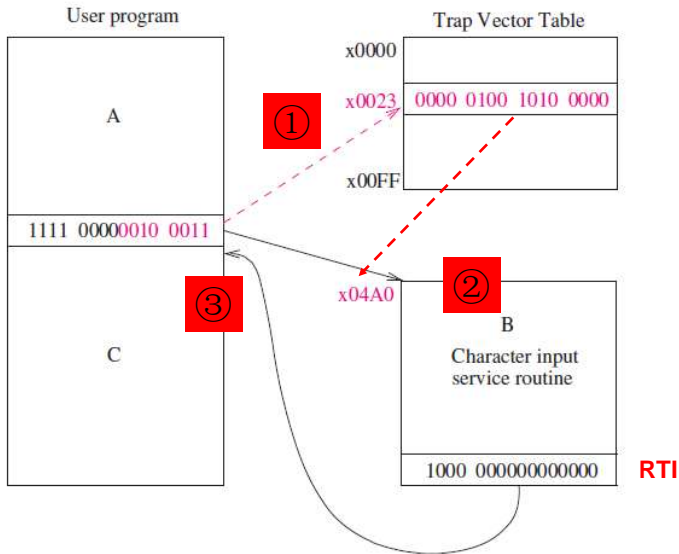
■ Enabling return

- Push the address of next instruction after TRAP together with the PSR to the system stack (**Poped by the RTI instruction**)

| | |
|-------|-------|
| x0000 | |
| : | : |
| x0020 | x0400 |
| x0021 | x0430 |
| x0022 | x0450 |
| x0023 | x04A0 |
| x0024 | x04E0 |
| x0025 | x0520 |
| : | : |
| x00FF | |

The LC-3 Trap Vector Table

TRAP Mechanism Operation



TRAP Mechanism Operation

OLD_PSR=PSR;

if (PSR[15] == 1) { // called by a user program

Saved_USP \leftarrow R6;

R6 \leftarrow **Saved_SSP**;

}

PSR[15]=0; // switch to supervisor mode

Push **OLD_PSR**, **PC+1** on the system stack

PC = mem[ZEXT(trapvect8)]; // set PC

RTI instruction

■ RTI instruction – Return from Trap or Interrupt

RTI

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|--------------|----|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 000000000000 | | | | | | | | | | | |

RTI instruction

if (**PSR**[15] == 1) // user mode

//RTI cannot called by user program

Initiate a privilege mode exception;

else

PC = mem[R6]; // R6 is the SSP, PC is restored

R6 = R6+1;

OLD_PSR = mem[R6]; // **OLD_PSR** is restored

R6 = R6+1; //system stack completes POP before saved PSR
//is restored

PSR = OLD_PSR ; //PSR is restored

if (PSR[15] == 1) // supervisor mode → user mode

saved_SSP=R6;

R6=Saved_USP;

An example

| | | |
|-------|--------------|-------------------------------|
| | .ORIG x3000 | |
| | LD R2,TERM | ; Load -7 |
| | LD R3,ASCII | ; Load ASCII difference |
| AGAIN | TRAP x23 | ; Request keyboard input |
| | ADD R1,R2,R0 | ; Test for terminating |
| | BRz EXIT | ; character |
| | ADD R0,R0,R3 | ; Change to lowercase |
| | TRAP x21 | ; Output to the monitor |
| | BRnzp AGAIN | ; ... and do it again! |
| TERM | .FILL xFFC9 | ; FFC9 is negative of ASCII 7 |
| ASCII | .FILL x0020 | |
| EXIT | TRAP x25 | ; Halt |
| | .END | |



1 Review

2 TRAP Routines

3 Operating System Service Routines

Character Input Service Routine (IN, TRAP x23)

```
01 ; Service Routine for Keyboard Input
02 ;
03     .ORIG x04A0
04 START ST R1,Saver1 ; Save the values in the registers
05         ST R2,Saver2 ; that are used so that they
06         ST R3,Saver3 ; can be restored before RTI
07 ;
08         LD R2,Newline ; Newline: ASCII code for newline
09 L1     LDI R3,DSR      ; Check DDR -- is it free?
0A         BRzp L1
0B         STI R2,DDR    ; Move cursor to new clean line
0C ;
0D         LEA R1,Prompt ; Prompt is starting address
0E         ; of prompt string
0F Loop  LDR R0,R1,#0    ; Get next prompt character
10         BRz Input     ; Check for end of prompt string
11 L2     LDI R3,DSR      ; Check DDR -- is it free?
12         BRzp L2
13         STI R0,DDR     ; Write next char
14         ; prompt string
15         ADD R1,R1,#1    ; Increment prompt pointer
16         BRnzp Loop
17 ;
```

Character Input Service Routine (IN, TRAP x23)

```
18 Input LDI R3,KBSR ; Has a character been typed?
19      BRzp Input
1A      LDI R0,KBDR  ; Load it into R0
1B      L3 LDI R3,DSR
1C      BRzp L3
1D      STI R0,DDR   ; Echo input character
1E                        ; to the monitor
1F      ;
20 L4    LDI R3,DSR
21      BRzp L4
22      STI R2,DDR   ; Move cursor to new clean line
23      LD R1,SaveR1 ; Service routine done, restore
24      LD R2,SaveR2 ; original values in registers.
25      LD R3,SaveR3
26      RTI          ; Return from Trap
27      ;
28 SaveR1 .BLKW 1
29 SaveR2 .BLKW 1
2A SaveR3 .BLKW 1
2B DSR .FILL xFE04
2C DDR .FILL xFE06
2D KBSR .FILL xFE00
2E KBDR .FILL xFE02
2F Newline .FILL x000A ; ASCII code for newline
30 Prompt .STRINGZ "Input a character>"
31      .END
```

A String Output Service Routine (OUT,TRAP x21)

```
01 .ORIG x0420           ; System call starting address
02     ST R1, SaveR1     ; R1 will be used to poll the DSR
03                           ; hardware

04                           ; Write the character
05 TryWrite LDI R1, DSR   ; Get status
06     BRzp TryWrite      ; Bit 15 on says display is ready
07 WriteIt STI R0, DDR    ; Write character
08
09; return from trap
0A Return LD R1, SaveR1   ; Restore registers
0B     RTI                ; Return from trap

0C DSR .FILL xFE04        ; Address of display status register
0D DDR .FILL xFE06        ; Address of display data register
0E SaveR1 .BLKW 1

0F .END
```

A String Output Service Routine (PUTS, TRAP x22)

```
05      .ORIG x0460
06      ST R0, SaveR0 ; Save registers that
07      ST R1, SaveR1 ; are needed by this
08      ST R3, SaveR3 ; trap service routine
0A ; Loop through each character in the array
0C Loop LDR R1, R0, #0 ; Retrieve the character(s)
0D      BRz Return ; If it is 0, done
0E L2    LDI R3, DSR
0F      BRzp L2
10      STI R1, DDR ; Write the character
11      ADD R0, R0, #1 ; Increment pointer
12      BRnzp Loop ; Do it all over again
13 ;
14 ; Return from the request for service call
15 Return LD R3, SaveR3
16      LD R1, SaveR1
17      LD R0, SaveR0
18      RTI
19 ;
1A ; Register locations
1B DSR .FILL xFE04
1C DDR .FILL xFE06
1D SaveR0 .FILL x0000
1E SaveR1 .FILL x0000
1F SaveR3 .FILL x0000
20      .END
```

Halt the machine(HALT,TRAP x25)

```
01 .ORIG x0520                ; Where this routine resides
02     ST R1, SaveR1          ; R1: a temp for MC register
03 ST R0, SaveR0              ; R0 is used as working space
04
05                             ; print message that machine is halting
06
07     LD R0, ASCIINewLine    ;out
08     TRAP x21

09     LEA R0, Message        ;puts
0A     TRAP x22

0B     LD R0, ASCIINewLine    ;out
0C     TRAP x21
0D ;
0E ; clear bit 15 at xFFFE (Master Control Register, MCR )to
stop the machine.
0F ;
10     LDI R1, MCR            ; Load MC register into R1
11     LD R0, MASK            ; R0 = x7FFF
12     AND R0, R1, R0         ; Mask to clear the top bit
13     STI R0, MCR            ; Store R0 into MC register
```

Halt the machine(HALT,TRAP x25)

```
14 ;  
15 ; return from HALT routine.  
16 ; (how can this routine return if the machine is halted  
above?)  
17 ;  
18     LD R1, SaveR1 ; Restore registers  
19     LD R0, SaveR0  
1A     RTI  
1B ;  
1C ; Some constants  
1D ;  
1E ASCIINewLine .FILL x000A  
1F SaveR0 .BLKW 1  
20 SaveR1 .BLKW 1  
21 Message .STRINGZ "Halting the machine."  
22 MCR .FILL xFFFE ; Address of MCR  
23 MASK .FILL x7FFF ; Mask to clear the top bit  
24     .END
```

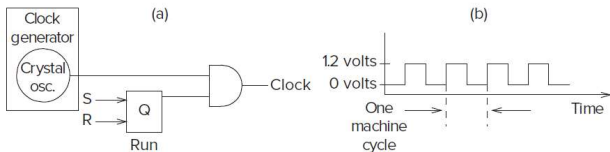


Figure 4.5 The clock circuit and its control.

Using subroutines to implement (IN, TRAP x23)

```
01      .ORIG x04A0
02 START JSR SaveReg
03      LD R2,Newline
04      JSR WriteChar
05      LEA R1,PROMPT
06 ;
07 ;
08 Loop LDR R2,R1,#0 ; Get next prompt char
09      BRz Input
0A      JSR WriteChar
0B      ADD R1,R1,#1
0C      BRnzp Loop
0D ;
.....
1A WriteChar LDI R3,DSR
1B          BRzp WriteChar
1C          STI R2,DDR
1D          RET ; JMP R7 terminates subroutine
1E DSR .FILL xFE04
1F DDR .FILL xFE06
20 ;
.....
```

Using Traps to implement (IN, TRAP x23)

| | | | |
|----------------|-------|--------|----------------------|
| ▶ x0214 | x0000 | 0 | TOUT_R1 .BLKW #1 |
| ▶ x0215 | x000A | 10 | TIN_R7 .BLKW #1 |
| ▶ x0216 | x0033 | 51 | OS_R0 .BLKW #1 |
| ▶ x0217 | x0000 | 0 | OS_R1 .BLKW #1 |
| ▶ x0218 | x0000 | 0 | OS_R2 .BLKW #1 |
| ▶ x0219 | x0000 | 0 | OS_R3 .BLKW #1 |
| ▶ x021A | x000A | 10 | OS_R7 .BLKW #1 |
| ▶ x0232 | x3FE2 | 16354 | TRAP_IN ST R7,TIN_R7 |
| ▶ x0233 | xE03A | -8134 | LEA R0,TRAP_IN_MSG |
| ▶ x0234 | xF022 | -4062 | PUTS |
| ▶ x0235 | xF020 | -4064 | GETC |
| ▶ x0236 | xF021 | -4063 | OUT |
| ▶ x0237 | x31DE | 12766 | ST R0,OS_R0 |
| ▶ x0238 | x5020 | 20512 | AND R0,R0,#0 |
| ▶ x0239 | x102A | 4138 | ADD R0,R0,#10 |
| ▶ x023A | xF021 | -4063 | OUT |
| ▶ x023B | x21DA | 8666 | LD R0,OS_R0 |
| ▶ x023C | x2FD8 | 12248 | LD R7,TIN_R7 |
| ▶ x023D | x8000 | -32768 | RTI |

Data Type Conversion

■ I/O

- Keyboard input routines read ASCII characters (not binary values)
- Console output routines write ASCII ('s' not "x73")

■ Consider this program:

```
TRAP  x23          ; input from keyboard
ADD   R1, R0, #0   ; move to R1
TRAP  x23          ; input from keyboard
ADD   R0, R1, R0   ; add two inputs
TRAP  x21          ; display result
TRAP  x25          ; HALT
```

■ User inputs '2' and '3' -- what happens?

■ Result displayed: 'e'

■ Why?

- ASCII '2' (x32) + ASCII '3' (x33) = ASCII 'e' (x65)

ASCII to Binary

■ Single digit numbers are trivial (subtract x30)

- E.g., '7' is ASCII x37, $x37 - x30 = x7$

■ Input

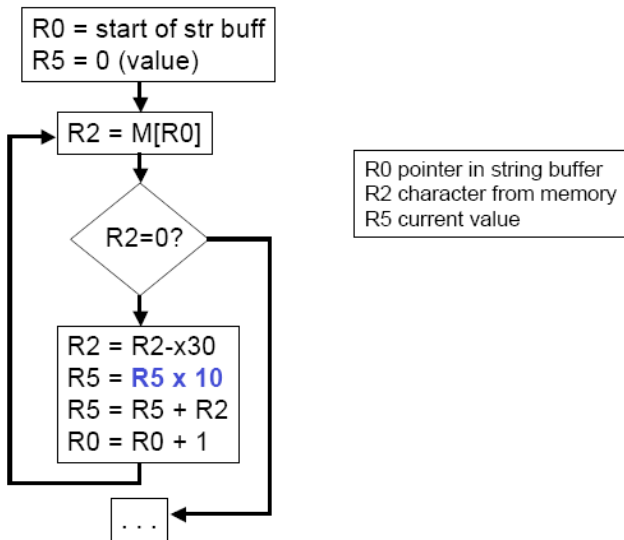
- Assume we've read three ASCII digits (e.g., "259") into a memory buffer

| | |
|-----|-----|
| x32 | '2' |
| x35 | '5' |
| x39 | '9' |
| x0 | |

■ How do we convert this to a *number* we can use?

- Convert first character to digit (subtract x30) and multiply by 100
- Convert second character to digit and multiply by 10
- Convert third character to digit
- Add the three digits together

ASCII to Binary Conversion Algorithm



Multiplication

■ How can we multiply a number by 100?

- Approach 0
 - Use the MUL instruction
- Approach 1
 - Add <number> to itself 10 times
- Approach 2
 - Add 10 to itself <number> times (better if number < 10)
- Approach 3
 - Look it up! Only practical if number of multiplicands is small

Lookup table
in memory

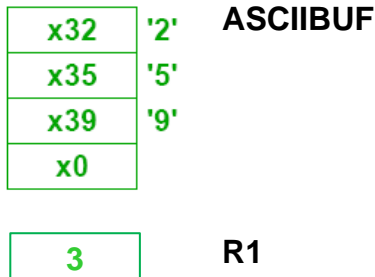
| | |
|-----|------|
| #0 | 0x10 |
| #10 | 1x10 |
| #20 | 2x10 |
| #30 | 3x10 |
| #40 | 4x10 |
| #50 | 5x10 |
| ... | |

Code for Lookup Table

```
; multiply R0 by 100, using lookup table
;
    LEA    R1, Lookup100    ; R1 = table base
    ADD    R1, R1, R0       ; add index (R0)
    LDR    R0, R1, #0       ; load from M[R1]
    ...
Lookup100 .FILL #0           ; entry 0
          .FILL #100        ; entry 1
          .FILL #200        ; entry 2
          .FILL #300        ; entry 3
          .FILL #400        ; entry 4
          .FILL #500        ; entry 5
          .FILL #600        ; entry 6
          .FILL #700        ; entry 7
          .FILL #800        ; entry 8
          .FILL #900        ; entry 9
```

Complete ASCII to Binary Conversion Code

- ASCII "259" to value
259



Complete ASCII to Binary Conversion Code (1 of 3)

; Three-digit buffer at ASCIIBUF.

; R1 tells how many digits to convert.

; Put resulting decimal number in R0.

```
ASCIItoBinary  AND    R0, R0, #0      ;clear result
                ADD    R1, R1, #0      ;test # digits
                BRz     DoneAtoB       ;done if no digits

;

                LD      R3, NegZero     ;R3 = -x30
                LEA     R2, ASCIIBUF    ;ptr to the first digit
                ADD     R2, R2, R1      ;R2 =(R2)+(R1)
                ADD     R2, R2, #-1     ;points to ones digit

;

                LDR     R4, R2, #0      ;load digit
                ADD     R4, R4, R3      ;convert to number
                ADD     R0, R0, R4      ;add ones contribution

;
```

Complete ASCII to Binary Conversion Code(2 of 3)

```
ADD    R1, R1, #-1      ;one less digit
BRz    DoneAtoB          ;done if zero
ADD    R2, R2, #-1      ;points to tens digit
;

LDR     R4, R2, #0        ;load 'tens' digit
ADD     R4, R4, R3        ;convert to number
LEA     R5, Lookup10      ;multiply by 10
ADD     R5, R5, R4
LDR     R4, R5, #0
ADD     R0, R0, R4        ;adds tens contribution to total
;

ADD     R1, R1, #-1      ;one less digit
BRz     DoneAtoB          ;done if zero
ADD     R2, R2, #-1      ;points to hundreds digit
;

LDR     R4, R2, #0        ;load digit
ADD     R4, R4, R3        ;convert to number
LEA     R5, Lookup100     ;multiply by 100
ADD     R5, R5, R4
LDR     R4, R5, #0
ADD     R0, R0, R4        ;adds 100's contrib
```

Complete ASCII to Binary Conversion Code(3 of 3)

```
DoneAtoB      RET
NegZero       .FILL xFFD0    ; -x30
ASCIIIBUF     .BLKW 4
Lookup10      .FILL #0
               .FILL #10
               .FILL #20
               ...
Lookup100     .FILL #0
               .FILL #100
               .FILL #200
               ...
```

Binary to ASCII Conversion

- **Converting a 2's complement binary value to a three-digit decimal number**
 - Resulting characters can be output using OUT
- **Instead of multiplying, we need to **divide by 100** to get hundreds digit.**
 - Why wouldn't we use a lookup table for this problem?
 - Subtract 100 repeatedly from number to divide.
- **First, check whether number is negative.**
 - Write sign character (+ or -) to buffer and make positive.

Binary to ASCII Conversion Code (1 of 3)

```
; R0 is between -999 and +999.  
; Put sign character in ASCIIBUF, followed by three  
; ASCII digit characters.  
BinaryToASCII  LEA R1, ASCIIBUF      ;ptr to result string  
                ADD R0, R0, #0        ;test sign of value  
                BRn NegSign  
                LD  R2, ASCIIplus     ;store '+'  
                STR R2, R1, #0  
                BR  Begin100  
NegSign        LD  R2, ASCIIneg       ;store '-'  
                STR R2, R1, #0  
                NOT R0, R0             ;convert value to pos  
                ADD R0, R0, #1  
Begin100       LD  R2, ASCIIoffset  
                LD  R3, Neg100  
Loop100        ADD R0, R0, R3  
                BRn End100  
                ADD R2, R2, #1         ;add one to digit  
                BR  Loop100
```

Binary to ASCII Conversion Code(2 of 3)

```
End100      STR R2, R1, #1 ;store ASCII 100's digit
            LD  R3, Pos100
            ADD R0, R0, R3 ;restore last subtract
;
            LD  R2, ASCIIoffset
            LD  R3, Neg10
Loop10      ADD R0, R0, R3
            BRn End10
            ADD R2, R2, #1 ;add one to digit
            BR  Loop10
End10       STR R2, R1, #2 ;store ASCII 10's digit
            ADD R0, R0, #10 ;restore last subtract
;
            LD  R2, ASCIIoffset
            ADD R2, R2, R0 ;convert one's digit
            STR R2, R1, #3 ;store one's digit
            RET
;
```

Binary to ASCII Conversion Code(3 of 3)

```
ASCIIplus    .FILL x002B    ;plus sign ASCII code
ASCIIneg     .FILL x002D    ;neg sign ASCII code
ASCIIoffset  .FILL x0030    ;zero's ASCII code
Neg100       .FILL xFF9C    ;-100
Pos100       .FILL x0064    ;100
Neg10        .FILL xFFF6    ;-10
```



中国科学技术大学
University of Science and Technology of China

计算系统概论A

Introduction to Computing Systems
(CS1002A.02)

Chapter 9-3 Interrupt-Driven I/O

计算机科学与技术学院
School of Computer Science and Technology



1 Review

2 Interrupt-Driven I/O

3 Input/Output

What is Interrupt-Driven I/O?

```
Program A is executing instruction n
Program A is executing instruction n+1
Program A is executing instruction n+2
Program A is executing instruction n+3
Program A is executing instruction n+4
.....
.....
.....
```

What is Interrupt-Driven I/O?

Program A is executing instruction n

Program A is executing instruction n+1

Program A is executing instruction n+2

Interrupt!!!

Program A is executing instruction n+3

Program A is executing instruction n+4

.....

.....

.....

What is Interrupt-Driven I/O?

Program A is executing instruction n

Program A is executing instruction n+1

Program A is executing instruction n+2

1: Interrupt signal is detected

1: Program A is put into suspended animation

1: PC is loaded with the starting address of Program B

2: Program B starts satisfying I/O device's needs

2: Program B continues satisfying I/O device's needs

2: Program B continues satisfying I/O device's needs

2: Program B finishes satisfying I/O device's needs

3: Program A is brought back to life

Program A is executing instruction n+3

Program A is executing instruction n+4

.....

.....

.....

Why Have Interrupt-Driven I/O?

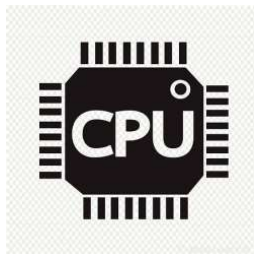
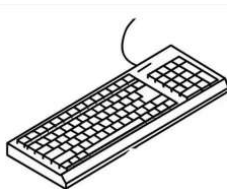
- **Polling** requires the processor to **waste a lot of time** spinning its wheels, re-executing again and again the LDI and BR instructions until the ready bit is set.
- With interrupt-driven I/O, **none of that testing and branching has to go on**. Interrupt-driven I/O allows the processor **to spend its time doing what is hopefully useful work**, executing some other program perhaps, until it is **notified** that some I/O device needs attention.

Why Have Interrupt-Driven I/O? An Example

- Suppose we are asked to write a program that takes a sequence of 100 characters typed on a **keyboard** and **processes** the information contained in **those 100 characters**. We need to perform this process on **1000 consecutive sequences**.
 - Assume the characters are typed at the rate of **80 words/minute**, which corresponds to one character every **0.125 seconds**. So, it would take $100 \cdot 0.125 = \mathbf{12.5 \text{ seconds}}$ to get a 100-character sequence.
 - Assume the processing of the 100-character sequence takes **12.49999 seconds**.
- **Polling:**
 - time for one sequence: $\mathbf{12.5 + 12.49999 = 24.99999 \text{ seconds}}$.
- **Interrupt-driven I/O:**
 - Assume 0.0000001 seconds for each character typed, or **0.00001 seconds** for the entire 100-character sequence.
 - time for one sequence: $\mathbf{0.00001 + 12.49999 = 12.5 \text{ seconds}}$.
- **For 1000 sequences:**
 - Polling vs. Interrupt-driven I/O $\approx \mathbf{6.94h \text{ vs. } 3.47h}$

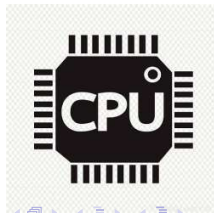
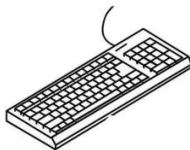
Two Parts to interrupt-driven I/O

- **Part1.** the mechanism that enables an I/O device to interrupt the processor
- **Part2.** the mechanism that handles the interrupt request.



Part I: Causing the Interrupt to Occur

- Several things must be true for an I/O device to actually interrupt the program that is running:
 - C1.** The I/O device must **want service**.
 - C2.** The device must **have the right** to request the service.
 - C3.** The device request must be **more urgent** than what the processor is currently doing.
- If all three elements are present, the processor **stops executing** the program that is running and **takes care of the interrupt**.



Part I: Causing the Interrupt to Occur

■ C1. The I/O device must want service.

- Keyboard: someone has **typed a character**
- Monitor: have successfully **completed** the display of the last character



Part I: Causing the Interrupt to Occur

■ C2. The device must have the right to request the service: the **interrupt enable bit**

- the **interrupt enable bit**, which can be **set or cleared** by the **processor** (usually by the **operating system**).
- In most I/O devices, this **interrupt enable (IE)** bit is part of the device status register.

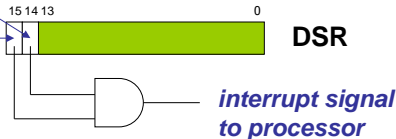
interrupt enable bit

ready bit



interrupt enable bit

ready bit



Part I: Causing the Interrupt to Occur

■ C3. The Urgency of the Request

- To interrupt the running program, the device must have a **higher priority** than the program that is currently running.
- There may be many devices that want to interrupt the processor at a specific time. To succeed, the device must have a **higher priority level** than all other demands for use of the processor.

The Processor Status Register

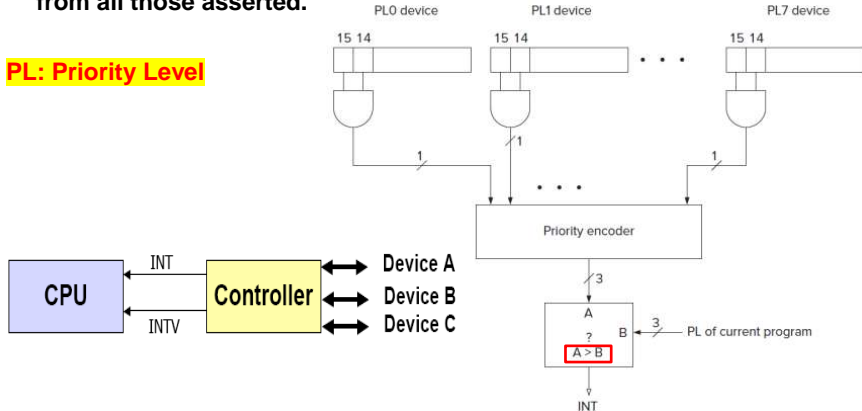


PL7 > PL6 > ... > PL0

Part I: Causing the Interrupt to Occur

- The **INT** signal: To stop the processor from continuing execution of its currently running program and service an interrupt request, the **INT signal must be asserted**.
 - The interrupt request signals are input to a priority encoder, a **combinational logic structure** that selects the highest priority request from all those asserted.

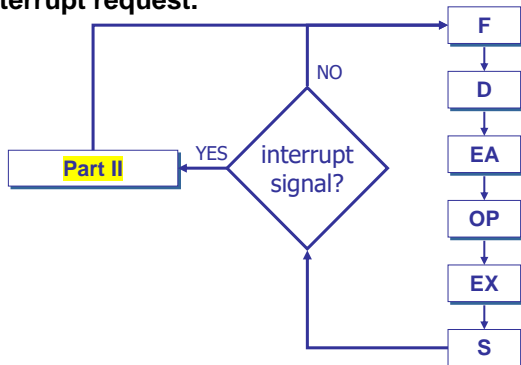
PL: Priority Level



Part I: Causing the Interrupt to Occur

■ The test for INT

- Instead of always going from the last state of one instruction cycle to the first state of the FETCH phase of the next instruction, the next state depends on the INT signal.
- If not asserted, continues with next instruction
- If INT is asserted, then the next state is the first state of **Part II**, handling the interrupt request.



Part II: Handling the Interrupt Request

■ Three stages

1. Initiate the interrupt

1: Interrupt signal is detected

1: Program A is put into suspended animation

1: PC is loaded with the starting address of Program B

2. Service the interrupt

2: Program B starts satisfying I/O device's needs

2: Program B continues satisfying I/O device's needs

2: Program B continues satisfying I/O device's needs

2: Program B finishes satisfying I/O device's needs

3. Return from the interrupt

3: Program A is brought back to life

Part II: Handling the Interrupt Request

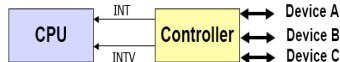
■ Stage1: Initiate the interrupt

- (1) **save the state** of the interrupted program so it can pick up where it left off after the requirements of the interrupt have been completed
 - The state includes the contents of the **memory locations** that are part of the program and the contents of **all the general purpose registers**. It also includes the **PC and PSR**.
 - Assume that the service routine will always save the contents of any general purpose register that it needs before using it. **The only state information the LC-3 saves are the PC and PSR**.
 - The LC-3 saves this state information on the supervisor stack in **the same way** the PC and PSR are saved when a **TRAP instruction** is executed.

Part II: Handling the Interrupt Request

■ Stage1: Initiate the interrupt

- (2) **load the state** of the higher priority interrupting program so it can start satisfying its request.
 - **Interrupt service routines** are **similar** to the **trap service routines**. They are program fragments stored in **system space**.
 - Most processors use the mechanism of **vectored interrupts**. The I/O device transmits to the processor an **eight-bit interrupt vector (INTV)** along with its interrupt request signal and its priority level.
 - The Interrupt Vector Table consists of memory locations x0100 to x01FF, each containing the starting address of an **interrupt service routine (ISR)**.
 - **Trap vector table : x0000 to x00FF**
 - **Interrupt vector table: x0100 to x01FF**
 - The PSR is loaded as follows:
 - Since no instructions in the service routine have yet executed, PSR[2:0] contains no meaningful information. We arbitrarily load it initially with 010.
 - Since **the interrupt service routine runs in privileged mode**, PSR[15] is set to 0.
 - PSR[10:8] is set to the **priority level associated with the interrupt request**.



Part II: Handling the Interrupt Request

■ Stage2: Service the Interrupt

- The PC contains the starting address of the interrupt service routine
- The service routine will execute, and the requirements of the I/O device will be serviced.

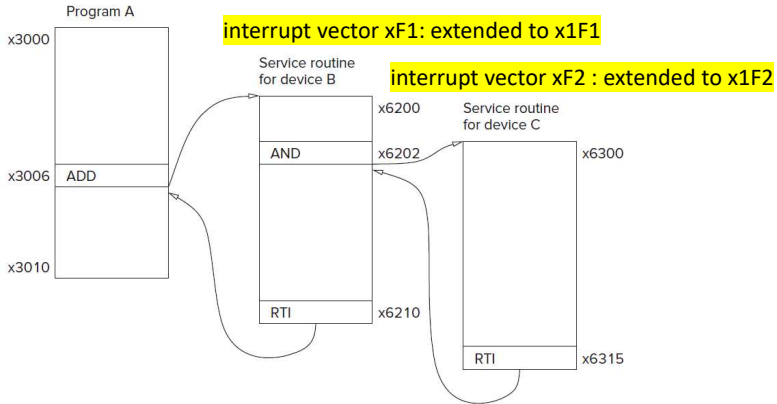
Part II: Handling the Interrupt Request

■ Stage3: Return from the Interrupt

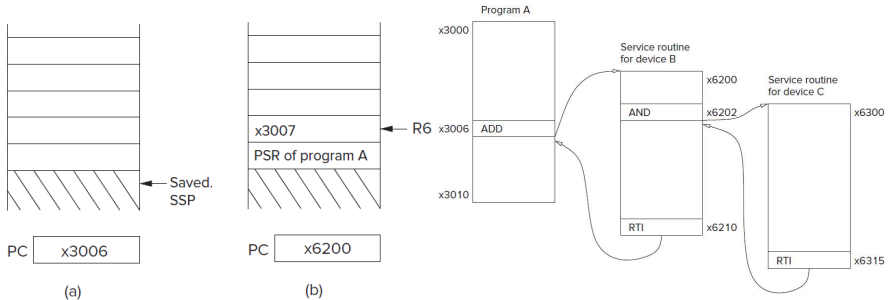
- The **last instruction** in every interrupt service routine is **RTI**, return from trap or interrupt.
 - RTI pops the **PC** and the **PSR** from the supervisor stack and restoring them to their rightful places in the processor.
 - If the privilege level of the interrupted program is **unprivileged**, the **stack pointers must be adjusted**, that is, the Supervisor Stack Pointer saved, and the User Stack Pointer loaded into R6.
 - The PC is restored to the **address of the instruction** that would have been executed next if the program had not been interrupted.

An Example (1)

- Suppose program A is executing when I/O device B, having a PL higher than that of A, requests service.
- During the execution of the service routine for I/O device B, a still more urgent device C requests service



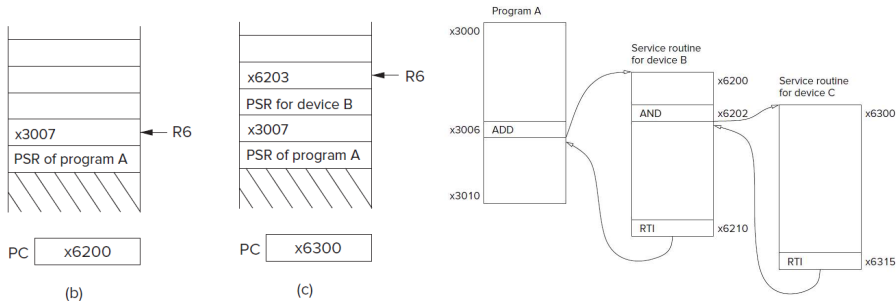
An Example (2)



- 1, the **supervisor stack** and the **PC** before program A fetches the instruction at x3006.
- 2, R6 is pointing to the **current contents of the user stack**, which are not shown!

- 3, The INT signal (caused by an interrupt from device B) is detected at the end of execution of the instruction in x3006.
- 4, R6 → Saved_USP; Save_SSP → R6
- 5, PSR of program A → system stack
- 6, PC+1 → system stack
- 7, The interrupt vector associated with device B is expanded to 16 bits **x01F1**, and the contents of x01F1 (**x6200**) is loaded into the **PC**.

An Example (3)

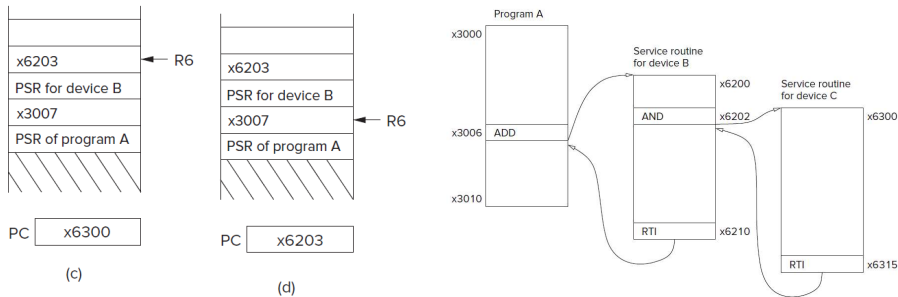


8, The service routine for device B executes until a higher priority interrupt is detected at the end of execution of the instruction at x6202.

9, The **PSR** of the service routine for B, which includes the condition codes produced by the AND instruction at x6202, and the address x6203 (**PC +1**) are pushed on the stack.

10, The interrupt vector associated with device C is expanded to 16 bits (x01F2), and the contents of x01F2 (**x6300**) is loaded into the **PC**.

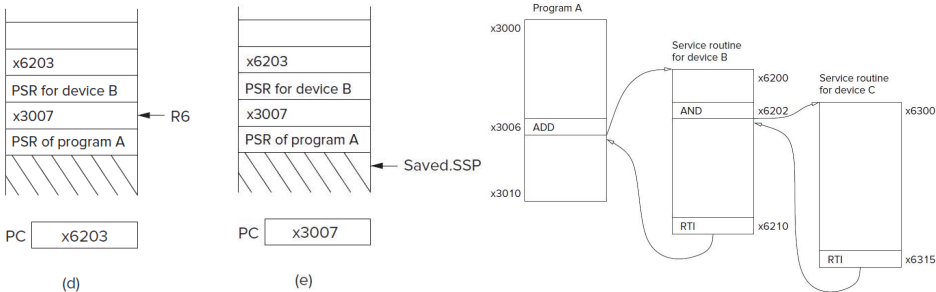
An Example (4)



11, Assume the interrupt service routine for device C executes to completion, finishing with the RTI instruction in x6315.

12, The supervisor stack is popped twice, restoring the **PC** to x6203 and the **PSR** of the service routine for device B, including the condition codes produced by the AND instruction in x6202.

An Example (5)



12, The interrupt service routine for device B resumes execution at x6203 and runs to completion, finishing with the RTI instruction in x6210.

13, The supervisor stack is popped twice, restoring the **PC** to x3007 and the **PSR** of program A, including the condition codes produced by the ADD instruction in x3006.

14, Finally, since program A is in User mode, the contents of **R6** is stored in **Saved_SSP** and **R6** is loaded with the contents of **Saved_USP**.

15, Program A resumes execution with the instruction at x3007.

Interrupts deal with more than I/O devices.

- Any **event** that has a higher priority and is external to the program that is running can interrupt the computer.
 - It does so by supplying its **INT** signal, its **INTV** vector, and its **priority level**.
 - If it is the highest priority event that wishes to interrupt the computer, it does so **in the same way that I/O devices do** as described above.
- **Examples:**
 - **timer interrupt interrupts** the program that is running in order to note the passage of a unit of time.
 - The **machine check interrupt** calls attention to the fact that some part of the computer system is not functioning properly.
 -

Polling Revisited

■ Interrupt Mask

- When set, a processor can ignore INT signal

■ How to implement interrupt mask in LC3?

| | |
|------------|---|
| 09 | LDI R1, PSR |
| 0A | LD R2,INTMASK |
| 0B | AND R2,R1,R2 ; R1=original PSR, R2=PSR with interrupts disabled |
| 0C | |
| 0D POLL | STI R1,PSR ; enable interrupts (if they were enabled to begin) |
| 0E | STI R2,PSR ; disable interrupts |
| 0F | LDI R3,DSR |
| 10 | BRzp POLL ; Poll the DSR |
| 11 | STI R0,DDR ; Store the character into the DDR |
| 12 | STI R1,PSR ; Restore original PSR |
| 1D INTMASK | .FILL xBFFF ; 1011 1111 1111 1111 |
| 1E PSR | .FILL xFFFC |
| 1F DSR | .FILL xFE04 |
| 20 DDR | .FILL xFE06 |

ICS 复习

Important parts

- Appendix A
 - Exclude exception
- Appendix C
 - Exclude exception

About chapter 10

- **Stack machines, Zero-address machine**
 - PPT 8-2
 - Chapter 10.2
- **Simulate a calculator with LC-3**

```
(1)  push    25
(2)  push    17
(3)  add
(4)  push    3
(5)  push    2
(6)  add
(7)  multiply
(8)  pop      E
```

OpMult, which will pop two values from the stack, multiply them, and push the result onto the stack.

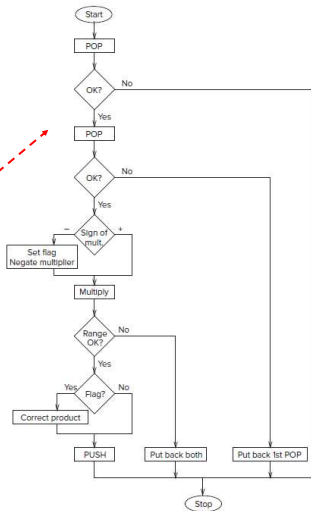


Figure 10.11 Flowchart for the OpMult subroutine.

About chapter 10

- **Data Type Conversion**

- PPT 9-2
- Chapter 10.1

Appendix C :Microarchitecture of the LC-3

- Time is divided into clock cycles.
- The cycle time of a microprocessor is the duration of a clock cycle.
 - A common cycle time for a microprocessor today is 0.33 nanoseconds, which corresponds to 3 billion clock cycles each second. We say that such a microprocessor is operating at a frequency of 3 gigahertz, or 3 GHz.
- We say, “at each instant of time,” but we really mean during each clock cycle.

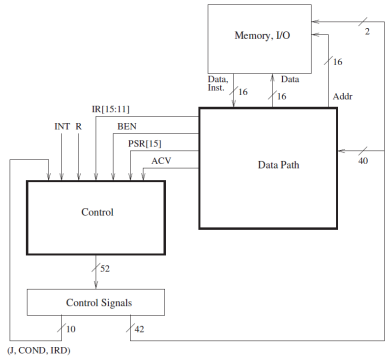


Figure C.1 Microarchitecture of the LC-3, major components.

Appendix C :Microarchitecture of the LC-3

- The control signals needed in the “next” clock cycle depend on the following:
 1. The **control signals** that are present during the current clock cycle.
 2. The LC-3 **instruction** that is being executed.
 3. The **privilege mode** of the program that is executing, and whether the processor has the right to access a particular memory location.
 4. If that LC-3 instruction is a BR, whether the **conditions** for the branch have been **met** (i.e., the state of the relevant condition codes).
 5. Whether or not an external device is requesting that the processor be **interrupted**.
 6. If a memory operation is in progress, whether it is **completing** during this cycle.

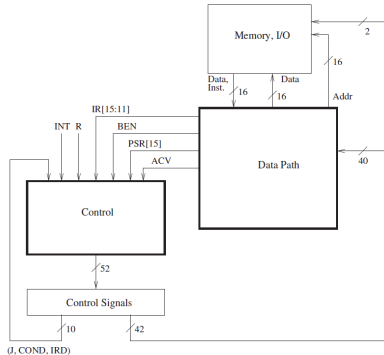


Figure C.1 Microarchitecture of the LC-3, major components.

Appendix C :Microarchitecture of the LC-3

1. J[5:0], COND[2:0], and IRD—ten bits of control signals provided by the current clock cycle.

2. **IR[15:12]**, which identifies the opcode, and **IR[11:11]**, which differentiates JSR from JSRR (i.e., the addressing mode for the target of the subroutine call).

3. PSR[15], bit [15] of the Processor Status Register, which indicates whether the current program is executing with supervisor or user privileges,

4. ACV, a signal that informs the processor that a process operating in User mode is trying to access a location in privileged memory. ACV stands for Access Control Violation. When asserted, it denies the process access to the privileged memory location.

5. BEN to indicate whether or not a BR should be taken.

6. INT to indicate that some external device of higher priority than the executing process requests service.

7. R to indicate the end of a memory operation.

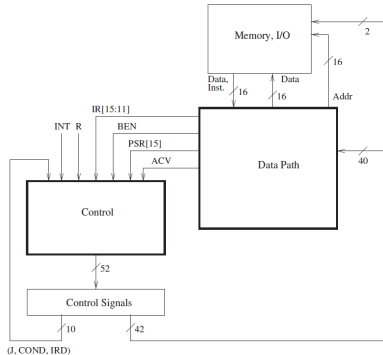


Figure C.1 Microarchitecture of the LC-3, major components

About R

A flash memory chip

K9F2G08U0A-PCB0/PIB0



PIN DESCRIPTION

| Pin Name | Pin Function |
|-------------------------------------|--|
| I/O ₀ ~ I/O ₇ | DATA INPUTS/OUTPUTS The I/O pins are used to input command, address and data, and to output data during read operations. The I/O pins float to high-Z when the chip is deselected or when the outputs are disabled. |
| CLE | COMMAND LATCH ENABLE The CLE input controls the activating path for commands sent to the command register. When active high, commands are latched into the command register through the I/O ports on the rising edge of the WE signal. |
| ALE | ADDRESS LATCH ENABLE The ALE input controls the activating path for address to the internal address registers. Addresses are latched on the rising edge of WE with ALE high. |
| \overline{CE} | CHIP ENABLE The \overline{CE} input is the device selection control. When the device is in the Busy state, \overline{CE} high is ignored, and the device does not return to standby mode in program or erase operation. |
| \overline{RE} | READ ENABLE The \overline{RE} input is the serial data-out control, and when active drives the data onto the I/O bus. Data is valid tREA after the falling edge of \overline{RE} which also increments the internal column address counter by one. |
| \overline{WE} | WRITE ENABLE The \overline{WE} input controls writes to the I/O port. Commands, address and data are latched on the rising edge of the \overline{WE} pulse. |
| \overline{WP} | WRITE PROTECT The \overline{WP} pin provides inadvertent program/erase protection during power transitions. The internal high-voltage generator is reset when the \overline{WP} pin is active low. |
| $\overline{R/B}$ | READY/BUSY OUTPUT The $\overline{R/B}$ output indicates the status of the device operation. When low, it indicates that a program, erase or random read operation is in process and returns to high state upon completion. It is an open drain output and does not float to high-Z condition when the chip is deselected or when outputs are disabled. |
| Vcc | POWER Vcc is the power supply for device. |
| Vss | GROUND |
| N.C. | NO CONNECTION Lead is not internally connected. |

READY/BUSY OUTPUT

The R/B output indicates the status of the device operation. When low, it indicates that a program, erase or random read operation is in process and **returns to high state upon completion.**

Appendix C :Microarchitecture of the LC-3

- During each clock cycle,
 - 42 of these control signals determine the **processing of information in the data path**
 - the other **10** control signals combine with the **10** bits of additional information to determine which set of control signals will be required in the next clock cycle.
- These 52 control signals specify the state of the control structure of the LC-3 microarchitecture

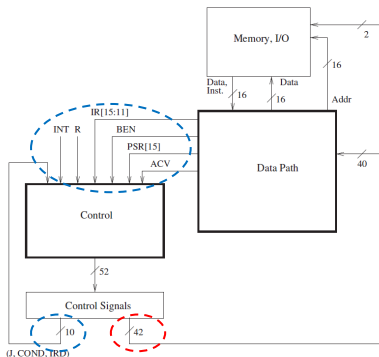


Figure C.1 Microarchitecture of the LC-3, major components.

The state machine

- [INT]
- [ACV]
- set CC

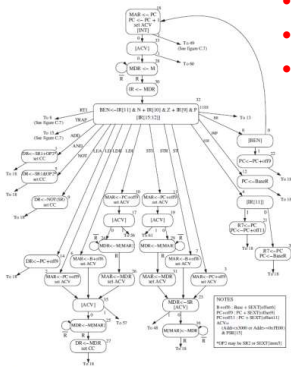


Figure C.2 A state machine for the LC-3.

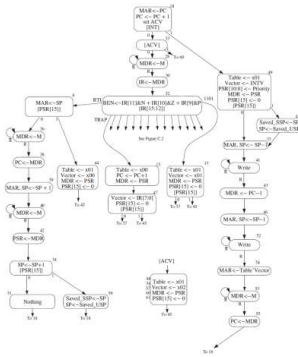


Figure C.7 LC-3 state machine showing interrupt control.



The state machine

- The state machine describes what happens during each **clock cycle** in which the computer is running.
- Each state is **active for exactly one clock cycle** before control passes to the next state.
- Each node in the state machine corresponds to the **activity** that the processor carries out during a single clock cycle.

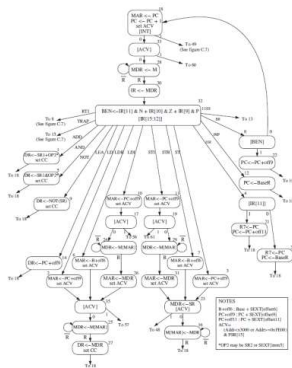
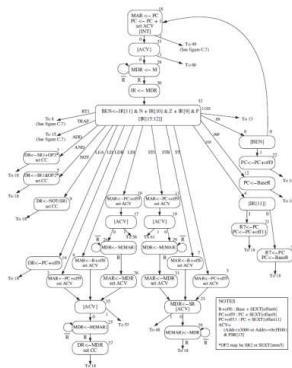


Figure C.2 A state machine for the LC-3.

The state machine

- the **FETCH** phase of the instruction cycle
 - In **state 18**, the **MAR** is loaded with the address contained in **PC**, and the **PC** is **incremented** in preparation for the FETCH of the next LC-3 instruction after the current instruction finishes its instruction cycle.
 - If the content of **MAR** specifies privileged memory, and **PSR[15] = 1**, indicating User mode, the access of the instruction will not be allowed. That would be an **access control violation**, so **ACV** is set.
 - Finally, if there is no interrupt request present (**INT** = 0), the flow passes to **state 33**. Or else, the flow passes to **state 49**.



The state machine

- the **FETCH** phase of the instruction cycle
 - From **state 33**, control passes to **state 60** if the processor is trying to access privileged memory while in User mode, or to **state 28**, if the memory access is allowed, that is, if there is no ACV violation.
 - In **state 28**, since the MAR contains the address of the instruction to be processed, this instruction is read from memory and loaded into the MDR. Since this memory access can **take multiple cycles**, this state continues to execute until a **ready signal** from the memory (**R**) is asserted, indicating that the memory access has completed. Thus, the MDR contains the valid contents of the memory location specified by MAR.
 - The state machine then moves on to **state 30**, where the instruction is loaded into the instruction register (**IR**), completing the **fetch** phase of the **instruction cycle**.

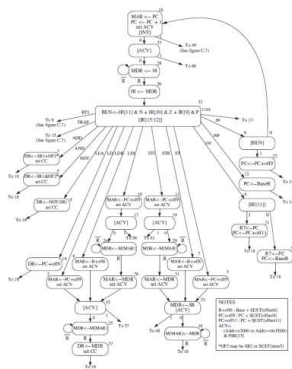


Figure C.2 A state machine for the LC-3.

The state machine

- The state machine then moves to **state 32**, where **DECODE** takes place.
 - there are 16 arcs emanating from state 32, each one corresponding to bits [15:12] of the LC-3 instruction.
 - the arc from **the last state** of each instruction cycle (i.e., the state that completes the processing of that LC-3 instruction) takes us to **state 18**

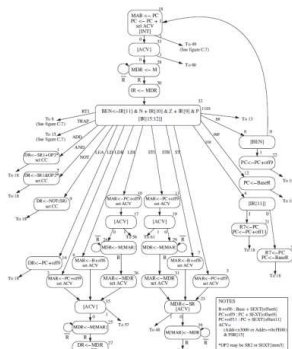


Figure C-2: A state machine for the LC-3.

The data path

- The **data path** consists of **all components** that actually **process the information** during each clock cycle—
 - the **functional units** that operate on the information,
 - the **registers** that store information at the end of one cycle so it will be available for further use in subsequent cycles,
 - and the **buses and wires** that carry information from one point to another in the data path.

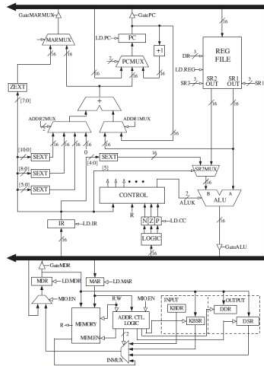


Figure C.3 The LC-3 data path

| Table C.1 Data Path Control Signals | |
|-------------------------------------|---------------|
| Signal Name | Signal Values |
| LD0BEN[1] | NO_LOAD |
| LD0CEN[1] | NO_LOAD |
| LD0EN[1] | NO_LOAD |
| LD0ENEN[1] | NO_LOAD |
| LD0ENEN[2] | NO_LOAD |
| LD0ENEN[3] | NO_LOAD |
| LD0ENEN[4] | NO_LOAD |
| LD0ENEN[5] | NO_LOAD |
| LD0ENEN[6] | NO_LOAD |
| LD0ENEN[7] | NO_LOAD |
| LD0ENEN[8] | NO_LOAD |
| LD0ENEN[9] | NO_LOAD |
| LD0ENEN[10] | NO_LOAD |
| LD0ENEN[11] | NO_LOAD |
| LD0ENEN[12] | NO_LOAD |
| LD0ENEN[13] | NO_LOAD |
| LD0ENEN[14] | NO_LOAD |
| LD0ENEN[15] | NO_LOAD |
| LD0ENEN[16] | NO_LOAD |
| LD0ENEN[17] | NO_LOAD |
| LD0ENEN[18] | NO_LOAD |
| LD0ENEN[19] | NO_LOAD |
| LD0ENEN[20] | NO_LOAD |
| LD0ENEN[21] | NO_LOAD |
| LD0ENEN[22] | NO_LOAD |
| LD0ENEN[23] | NO_LOAD |
| LD0ENEN[24] | NO_LOAD |
| LD0ENEN[25] | NO_LOAD |
| LD0ENEN[26] | NO_LOAD |
| LD0ENEN[27] | NO_LOAD |
| LD0ENEN[28] | NO_LOAD |
| LD0ENEN[29] | NO_LOAD |
| LD0ENEN[30] | NO_LOAD |
| LD0ENEN[31] | NO_LOAD |
| LD0ENEN[32] | NO_LOAD |
| LD0ENEN[33] | NO_LOAD |
| LD0ENEN[34] | NO_LOAD |
| LD0ENEN[35] | NO_LOAD |
| LD0ENEN[36] | NO_LOAD |
| LD0ENEN[37] | NO_LOAD |
| LD0ENEN[38] | NO_LOAD |
| LD0ENEN[39] | NO_LOAD |
| LD0ENEN[40] | NO_LOAD |
| LD0ENEN[41] | NO_LOAD |
| LD0ENEN[42] | NO_LOAD |
| LD0ENEN[43] | NO_LOAD |
| LD0ENEN[44] | NO_LOAD |
| LD0ENEN[45] | NO_LOAD |
| LD0ENEN[46] | NO_LOAD |
| LD0ENEN[47] | NO_LOAD |
| LD0ENEN[48] | NO_LOAD |
| LD0ENEN[49] | NO_LOAD |
| LD0ENEN[50] | NO_LOAD |
| LD0ENEN[51] | NO_LOAD |
| LD0ENEN[52] | NO_LOAD |
| LD0ENEN[53] | NO_LOAD |
| LD0ENEN[54] | NO_LOAD |
| LD0ENEN[55] | NO_LOAD |
| LD0ENEN[56] | NO_LOAD |
| LD0ENEN[57] | NO_LOAD |
| LD0ENEN[58] | NO_LOAD |
| LD0ENEN[59] | NO_LOAD |
| LD0ENEN[60] | NO_LOAD |
| LD0ENEN[61] | NO_LOAD |
| LD0ENEN[62] | NO_LOAD |
| LD0ENEN[63] | NO_LOAD |
| LD0ENEN[64] | NO_LOAD |
| LD0ENEN[65] | NO_LOAD |
| LD0ENEN[66] | NO_LOAD |
| LD0ENEN[67] | NO_LOAD |
| LD0ENEN[68] | NO_LOAD |
| LD0ENEN[69] | NO_LOAD |
| LD0ENEN[70] | NO_LOAD |
| LD0ENEN[71] | NO_LOAD |
| LD0ENEN[72] | NO_LOAD |
| LD0ENEN[73] | NO_LOAD |
| LD0ENEN[74] | NO_LOAD |
| LD0ENEN[75] | NO_LOAD |
| LD0ENEN[76] | NO_LOAD |
| LD0ENEN[77] | NO_LOAD |
| LD0ENEN[78] | NO_LOAD |
| LD0ENEN[79] | NO_LOAD |
| LD0ENEN[80] | NO_LOAD |
| LD0ENEN[81] | NO_LOAD |
| LD0ENEN[82] | NO_LOAD |
| LD0ENEN[83] | NO_LOAD |
| LD0ENEN[84] | NO_LOAD |
| LD0ENEN[85] | NO_LOAD |
| LD0ENEN[86] | NO_LOAD |
| LD0ENEN[87] | NO_LOAD |
| LD0ENEN[88] | NO_LOAD |
| LD0ENEN[89] | NO_LOAD |
| LD0ENEN[90] | NO_LOAD |
| LD0ENEN[91] | NO_LOAD |
| LD0ENEN[92] | NO_LOAD |
| LD0ENEN[93] | NO_LOAD |
| LD0ENEN[94] | NO_LOAD |
| LD0ENEN[95] | NO_LOAD |
| LD0ENEN[96] | NO_LOAD |
| LD0ENEN[97] | NO_LOAD |
| LD0ENEN[98] | NO_LOAD |
| LD0ENEN[99] | NO_LOAD |
| LD0ENEN[100] | NO_LOAD |
| LD0ENEN[101] | NO_LOAD |
| LD0ENEN[102] | NO_LOAD |
| LD0ENEN[103] | NO_LOAD |
| LD0ENEN[104] | NO_LOAD |
| LD0ENEN[105] | NO_LOAD |
| LD0ENEN[106] | NO_LOAD |
| LD0ENEN[107] | NO_LOAD |
| LD0ENEN[108] | NO_LOAD |
| LD0ENEN[109] | NO_LOAD |
| LD0ENEN[110] | NO_LOAD |
| LD0ENEN[111] | NO_LOAD |
| LD0ENEN[112] | NO_LOAD |
| LD0ENEN[113] | NO_LOAD |
| LD0ENEN[114] | NO_LOAD |
| LD0ENEN[115] | NO_LOAD |
| LD0ENEN[116] | NO_LOAD |
| LD0ENEN[117] | NO_LOAD |
| LD0ENEN[118] | NO_LOAD |
| LD0ENEN[119] | NO_LOAD |
| LD0ENEN[120] | NO_LOAD |
| LD0ENEN[121] | NO_LOAD |
| LD0ENEN[122] | NO_LOAD |
| LD0ENEN[123] | NO_LOAD |
| LD0ENEN[124] | NO_LOAD |
| LD0ENEN[125] | NO_LOAD |
| LD0ENEN[126] | NO_LOAD |
| LD0ENEN[127] | NO_LOAD |
| LD0ENEN[128] | NO_LOAD |
| LD0ENEN[129] | NO_LOAD |
| LD0ENEN[130] | NO_LOAD |
| LD0ENEN[131] | NO_LOAD |
| LD0ENEN[132] | NO_LOAD |
| LD0ENEN[133] | NO_LOAD |
| LD0ENEN[134] | NO_LOAD |
| LD0ENEN[135] | NO_LOAD |
| LD0ENEN[136] | NO_LOAD |
| LD0ENEN[137] | NO_LOAD |
| LD0ENEN[138] | NO_LOAD |
| LD0ENEN[139] | NO_LOAD |
| LD0ENEN[140] | NO_LOAD |
| LD0ENEN[141] | NO_LOAD |
| LD0ENEN[142] | NO_LOAD |
| LD0ENEN[143] | NO_LOAD |
| LD0ENEN[144] | NO_LOAD |
| LD0ENEN[145] | NO_LOAD |
| LD0ENEN[146] | NO_LOAD |
| LD0ENEN[147] | NO_LOAD |
| LD0ENEN[148] | NO_LOAD |
| LD0ENEN[149] | NO_LOAD |
| LD0ENEN[150] | NO_LOAD |
| LD0ENEN[151] | NO_LOAD |
| LD0ENEN[152] | NO_LOAD |
| LD0ENEN[153] | NO_LOAD |
| LD0ENEN[154] | NO_LOAD |
| LD0ENEN[155] | NO_LOAD |
| LD0ENEN[156] | NO_LOAD |
| LD0ENEN[157] | NO_LOAD |
| LD0ENEN[158] | NO_LOAD |
| LD0ENEN[159] | NO_LOAD |
| LD0ENEN[160] | NO_LOAD |
| LD0ENEN[161] | NO_LOAD |
| LD0ENEN[162] | NO_LOAD |
| LD0ENEN[163] | NO_LOAD |
| LD0ENEN[164] | NO_LOAD |
| LD0ENEN[165] | NO_LOAD |
| LD0ENEN[166] | NO_LOAD |
| LD0ENEN[167] | NO_LOAD |
| LD0ENEN[168] | NO_LOAD |
| LD0ENEN[169] | NO_LOAD |
| LD0ENEN[170] | NO_LOAD |
| LD0ENEN[171] | NO_LOAD |
| LD0ENEN[172] | NO_LOAD |
| LD0ENEN[173] | NO_LOAD |
| LD0ENEN[174] | NO_LOAD |
| LD0ENEN[175] | NO_LOAD |
| LD0ENEN[176] | NO_LOAD |
| LD0ENEN[177] | NO_LOAD |
| LD0ENEN[178] | NO_LOAD |
| LD0ENEN[179] | NO_LOAD |
| LD0ENEN[180] | NO_LOAD |
| LD0ENEN[181] | NO_LOAD |
| LD0ENEN[182] | NO_LOAD |
| LD0ENEN[183] | NO_LOAD |
| LD0ENEN[184] | NO_LOAD |
| LD0ENEN[185] | NO_LOAD |
| LD0ENEN[186] | NO_LOAD |
| LD0ENEN[187] | NO_LOAD |
| LD0ENEN[188] | NO_LOAD |
| LD0ENEN[189] | NO_LOAD |
| LD0ENEN[190] | NO_LOAD |
| LD0ENEN[191] | NO_LOAD |
| LD0ENEN[192] | NO_LOAD |
| LD0ENEN[193] | NO_LOAD |
| LD0ENEN[194] | NO_LOAD |
| LD0ENEN[195] | NO_LOAD |
| LD0ENEN[196] | NO_LOAD |
| LD0ENEN[197] | NO_LOAD |
| LD0ENEN[198] | NO_LOAD |
| LD0ENEN[199] | NO_LOAD |
| LD0ENEN[200] | NO_LOAD |
| LD0ENEN[201] | NO_LOAD |
| LD0ENEN[202] | NO_LOAD |
| LD0ENEN[203] | NO_LOAD |
| LD0ENEN[204] | NO_LOAD |
| LD0ENEN[205] | NO_LOAD |
| LD0ENEN[206] | NO_LOAD |
| LD0ENEN[207] | NO_LOAD |
| LD0ENEN[208] | NO_LOAD |
| LD0ENEN[209] | NO_LOAD |
| LD0ENEN[210] | NO_LOAD |
| LD0ENEN[211] | NO_LOAD |
| LD0ENEN[212] | NO_LOAD |
| LD0ENEN[213] | NO_LOAD |
| LD0ENEN[214] | NO_LOAD |
| LD0ENEN[215] | NO_LOAD |
| LD0ENEN[216] | NO_LOAD |
| LD0ENEN[217] | NO_LOAD |
| LD0ENEN[218] | NO_LOAD |
| LD0ENEN[219] | NO_LOAD |
| LD0ENEN[220] | NO_LOAD |
| LD0ENEN[221] | NO_LOAD |
| LD0ENEN[222] | NO_LOAD |
| LD0ENEN[223] | NO_LOAD |
| LD0ENEN[224] | NO_LOAD |
| LD0ENEN[225] | NO_LOAD |
| LD0ENEN[226] | NO_LOAD |
| LD0ENEN[227] | NO_LOAD |
| LD0ENEN[228] | NO_LOAD |
| LD0ENEN[229] | NO_LOAD |
| LD0ENEN[230] | NO_LOAD |
| LD0ENEN[231] | NO_LOAD |
| LD0ENEN[232] | NO_LOAD |
| LD0ENEN[233] | NO_LOAD |
| LD0ENEN[234] | NO_LOAD |
| LD0ENEN[235] | NO_LOAD |
| LD0ENEN[236] | NO_LOAD |
| LD0ENEN[237] | NO_LOAD |
| LD0ENEN[238] | NO_LOAD |
| LD0ENEN[239] | NO_LOAD |
| LD0ENEN[240] | NO_LOAD |
| LD0ENEN[241] | NO_LOAD |
| LD0ENEN[242] | NO_LOAD |
| LD0ENEN[243] | NO_LOAD |
| LD0ENEN[244] | NO_LOAD |
| LD0ENEN[245] | NO_LOAD |
| LD0ENEN[246] | NO_LOAD |
| LD0ENEN[247] | NO_LOAD |
| LD0ENEN[248] | NO_LOAD |
| LD0ENEN[249] | NO_LOAD |
| LD0ENEN[250] | NO_LOAD |
| LD0ENEN[251] | NO_LOAD |
| LD0ENEN[252] | NO_LOAD |
| LD0ENEN[253] | NO_LOAD |
| LD0ENEN[254] | NO_LOAD |
| LD0ENEN[255] | NO_LOAD |
| LD0ENEN[256] | NO_LOAD |
| LD0ENEN[257] | NO_LOAD |
| LD0ENEN[258] | NO_LOAD |
| LD0ENEN[259] | NO_LOAD |
| LD0ENEN[260] | NO_LOAD |
| LD0ENEN[261] | NO_LOAD |
| LD0ENEN[262] | NO_LOAD |
| LD0ENEN[263] | NO_LOAD |
| LD0ENEN[264] | NO_LOAD |
| LD0ENEN[265] | NO_LOAD |
| LD0ENEN[266] | NO_LOAD |
| LD0ENEN[267] | NO_LOAD |
| LD0ENEN[268] | NO_LOAD |
| LD0ENEN[269] | NO_LOAD |
| LD0ENEN[270] | NO_LOAD |
| LD0ENEN[271] | NO_LOAD |
| LD0ENEN[272] | NO_LOAD |
| LD0ENEN[273] | NO_LOAD |
| LD0ENEN[274] | NO_LOAD |
| LD0ENEN[275] | NO_LOAD |
| LD0ENEN[276] | NO_LOAD |
| LD0ENEN[277] | NO_LOAD |
| LD0ENEN[278] | NO_LOAD |
| LD0ENEN[279] | NO_LOAD |
| LD0ENEN[280] | NO_LOAD |
| LD0ENEN[281] | NO_LOAD |
| LD0ENEN[282] | NO_LOAD |
| LD0ENEN[283] | NO_LOAD |
| LD0ENEN[284] | NO_LOAD |
| LD0ENEN[285] | NO_LOAD |
| LD0ENEN[286] | NO_LOAD |
| LD0ENEN[287] | NO_LOAD |
| LD0ENEN[288] | NO_LOAD |
| LD0ENEN[289] | NO_LOAD |
| LD0ENEN[290] | NO_LOAD |
| LD0ENEN[291] | NO_LOAD |
| LD0ENEN[292] | NO_LOAD |
| LD0ENEN[293] | NO_LOAD |
| LD0ENEN[294] | NO_LOAD |
| LD0ENEN[295] | NO_LOAD |
| LD0ENEN[296] | NO_LOAD |
| LD0ENEN[297] | NO_LOAD |
| LD0ENEN[298] | NO_LOAD |
| LD0ENEN[299] | NO_LOAD |
| LD0ENEN[300] | NO_LOAD |
| LD0ENEN[301] | NO_LOAD |
| LD0ENEN[302] | NO_LOAD |
| LD0ENEN[303] | NO_LOAD |
| LD0ENEN[304] | NO_LOAD |
| LD0ENEN[305] | NO_LOAD |
| LD0ENEN[306] | NO_LOAD |
| LD0ENEN[307] | NO_LOAD |
| LD0ENEN[308] | NO_LOAD |
| LD0ENEN[309] | NO_LOAD |
| LD0ENEN[310] | NO_LOAD |
| LD0ENEN[311] | NO_LOAD |
| LD0ENEN[312] | NO_LOAD |
| LD0ENEN[313] | NO_LOAD |
| LD0ENEN[314] | NO_LOAD |
| LD0ENEN[315] | NO_LOAD |
| LD0ENEN[316] | NO_LOAD |
| LD0ENEN[317] | NO_LOAD |
| LD0ENEN[318] | NO_LOAD |
| LD0ENEN[319] | NO_LOAD |
| LD0ENEN[320] | NO_LOAD |
| LD0ENEN[321] | NO_LOAD |
| LD0ENEN[322] | NO_LOAD |
| LD0ENEN[323] | NO_LOAD |
| LD0ENEN[324] | NO_LOAD |
| LD0ENEN[325] | NO_LOAD |
| LD0ENEN[326] | NO_LOAD |
| LD0ENEN[327] | NO_LOAD |
| LD0ENEN[328] | NO_LOAD |
| LD0ENEN[329] | NO_LOAD |
| LD0ENEN[330] | NO_LOAD |
| LD0ENEN[331] | NO_LOAD |
| LD0ENEN[332] | NO_LOAD |
| LD0ENEN[333] | NO_LOAD |
| LD0ENEN[334] | NO_LOAD |
| LD0ENEN[335] | NO_LOAD |
| LD0ENEN[336] | NO_LOAD |
| LD0ENEN[337] | NO_LOAD |
| LD0ENEN[338] | NO_LOAD |
| LD0ENEN[339] | NO_LOAD |
| LD0ENEN[340] | NO_LOAD |
| LD0ENEN[341] | NO_LOAD |
| LD0ENEN[342] | NO_LOAD |
| LD0ENEN[343] | NO_LOAD |
| LD0ENEN[344] | NO_LOAD |
| LD0ENEN[345] | NO_LOAD |
| LD0ENEN[346] | NO_LOAD |
| LD0ENEN[347] | NO_LOAD |
| LD0ENEN[348] | NO_LOAD |
| LD0ENEN[349] | NO_LOAD |
| LD0ENEN[350] | NO_LOAD |
| LD0ENEN[351] | NO_LOAD |
| LD0ENEN[352] | NO_LOAD |
| LD0ENEN[353] | NO_LOAD |
| LD0ENEN[354] | NO_LOAD |
| LD0ENEN[355] | NO_LOAD |
| LD0ENEN[356] | NO_LOAD |
| LD0ENEN[357] | NO_LOAD |
| LD0ENEN[358] | NO_LOAD |
| LD0ENEN[359] | NO_LOAD |
| LD0ENEN[360] | NO_LOAD |
| LD0ENEN[361] | NO_LOAD |
| LD0ENEN[362] | NO_LOAD |
| LD0ENEN[363] | NO_LOAD |
| LD0ENEN[364] | NO_LOAD |
| LD0ENEN[365] | NO_LOAD |
| LD0ENEN[366] | NO_LOAD |
| LD0ENEN[367] | NO_LOAD |
| LD0ENEN[368] | NO_LOAD |
| LD0ENEN[369] | NO_LOAD |
| LD0ENEN[370] | NO_LOAD |
| LD0ENEN[371] | NO_LOAD |
| LD0ENEN[372] | NO_LOAD |
| LD0ENEN[373] | NO_LOAD |
| LD0ENEN[374] | NO_LOAD |
| LD0ENEN[375] | NO_LOAD |
| LD0ENEN[376] | NO_LOAD |
| LD0ENEN[377] | NO_LOAD |
| LD0ENEN[378] | NO_LOAD |
| LD0ENEN[379] | NO_LOAD |
| LD0ENEN[380] | NO_LOAD |
| LD0ENEN[381] | NO_LOAD |
| LD0ENEN[382] | NO_LOAD |
| LD0ENEN[383] | NO_LOAD |
| LD0ENEN[384] | NO_LOAD |
| LD0ENEN[385] | NO_LOAD |
| LD0ENEN[386] | NO_LOAD |
| LD0ENEN[387] | NO_LOAD |
| LD0ENEN[388] | NO_LOAD |
| LD0ENEN[389] | NO_LOAD |
| LD0ENEN[390] | NO_LOAD |
| LD0ENEN[391] | NO_LOAD |
| LD0ENEN[392] | NO_LOAD |
| LD0ENEN[393] | NO_LOAD |
| LD0ENEN[394] | NO_LOAD |
| LD0ENEN[395] | NO_LOAD |
| LD0ENEN[396] | NO_LOAD |
| LD0ENEN[397] | NO_LOAD |
| LD0ENEN[398] | NO_LOAD |
| LD0ENEN[399] | NO_LOAD |
| LD0ENEN[400] | NO_LOAD |
| LD0ENEN[401] | NO_LOAD |
| LD0ENEN[402] | NO_LOAD |
| LD0ENEN[403] | NO_LOAD |
| LD0ENEN[404] | NO_LOAD |
| LD0ENEN[405] | NO_LOAD |
| LD0ENEN[406] | NO_LOAD |
| LD0ENEN[407] | NO_LOAD |
| LD0ENEN[408] | NO_LOAD |
| LD0ENEN[409] | NO_LOAD |
| LD0ENEN[410] | NO_LOAD |
| LD0ENEN[411] | NO_LOAD |
| LD0ENEN[412] | NO_LOAD |
| LD0ENEN[413] | NO_LOAD |
| LD0ENEN[414] | NO_LOAD |
| LD0ENEN[415] | NO_LOAD |
| LD0ENEN[416] | NO_LOAD |
| LD0ENEN[417] | NO_LOAD |
| LD0ENEN[418] | NO_LOAD |
| LD0ENEN[419] | NO_LOAD |
| LD0ENEN[420] | NO_LOAD |
| LD0ENEN[421] | NO_LOAD |
| LD0ENEN[422] | NO_LOAD |
| LD0ENEN[423] | NO_LOAD |
| LD0ENEN[424] | NO_LOAD |
| LD0ENEN[425] | NO_LOAD |
| LD0ENEN[426] | NO_LOAD |
| LD0ENEN[427] | NO_LOAD |
| LD0ENEN[428] | NO_LOAD |
| LD0ENEN[429] | NO_LOAD |
| LD0ENEN[430] | NO_LOAD |
| LD0ENEN[431] | NO_LOAD |
| LD0ENEN[432] | NO_LOAD |
| LD0ENEN[433] | NO_LOAD |
| LD0ENEN[434] | NO_LOAD |
| LD0ENEN[435] | NO_LOAD |
| LD0ENEN[436] | NO_LOAD |
| LD0ENEN[437] | NO_LOAD |
| LD0ENEN[438] | NO_LOAD |
| LD0ENEN[439] | NO_LOAD |
| LD0ENEN[440] | NO_LOAD |
| LD0ENEN[441] | NO_LOAD |
| LD0ENEN[442] | NO_LOAD |
| LD0ENEN[443] | NO_LOAD |
| LD0ENEN[444] | NO_LOAD |
| LD0ENEN[445] | NO_LOAD |
| LD0ENEN[446] | NO_LOAD |
| LD0ENEN[447] | NO_LOAD |
| LD0ENEN[448] | NO_LOAD |
| LD0ENEN[449] | NO_LOAD |
| LD0ENEN[450] | NO_LOAD |
| LD0ENEN[451] | NO_LOAD |
| LD0ENEN[452] | NO_LOAD |
| LD0ENEN[453] | NO_LOAD |
| LD0ENEN[454] | NO_LOAD |
| LD0ENEN[455] | NO_LOAD |
| LD0ENEN[456] | NO_LOAD |
| LD0ENEN[457] | NO_LOAD |
| LD0ENEN[458] | NO_LOAD |
| LD0ENEN[459] | NO_LOAD |
| LD0ENEN[460] | NO_LOAD |
| LD0ENEN[461] | NO_LOAD |
| LD0ENEN[462] | NO_LOAD |
| LD0ENEN[463] | NO_LOAD |
| LD0ENEN[464] | NO_LOAD |
| LD0ENEN[465] | NO_LOAD |
| LD0ENEN[466] | NO_LOAD |
| LD0ENEN[467] | NO_LOAD |
| LD0ENEN[468] | NO_LOAD |
| LD0ENEN[469] | NO_LOAD |
| LD0ENEN[470] | NO_LOAD |
| LD0ENEN[471] | NO_LOAD |
| LD0ENEN[472] | NO_LOAD |
| LD0ENEN[473] | NO_LOAD |
| LD0ENEN[474] | NO_LOAD |
| LD0ENEN[475] | NO_LOAD |
| LD0ENEN[476] | NO_LOAD |
| LD0ENEN[477] | NO_LOAD |
| LD0ENEN[478] | NO_LOAD |
| LD0ENEN[479] | NO_LOAD |
| LD0ENEN[480] | NO_LOAD |
| LD0ENEN[481] | NO_LOAD |
| LD0ENEN[482] | NO_LOAD |
| LD0ENEN[483] | NO_LOAD |
| LD0ENEN[484] | NO_LOAD |
| LD0ENEN[485] | NO_LOAD |
| LD0ENEN[486] | NO_LOAD |
| LD0ENEN[487] | NO_LOAD |
| LD0ENEN[488] | NO_LOAD |
| LD0ENEN[489] | NO_LOAD |
| LD0ENEN[490] | NO_LOAD |
| LD0ENEN[491] | NO_LOAD |
| LD0ENEN[492] | NO_LOAD |
| LD0ENEN[493] | NO_LOAD |
| LD0ENEN[494] | NO_LOAD |
| LD0ENEN[495] | NO_LOAD |
| LD0ENEN[496] | NO_LOAD |
| LD0ENEN[497] | NO_LOAD |
| LD0ENEN[498] | NO_LOAD |
| LD0ENEN[499] | NO_LOAD |
| LD0ENEN[500] | NO_LOAD |
| LD0ENEN[501] | NO_LOAD |
| LD0ENEN[502] | NO_LOAD |
| LD0ENEN[503] | NO_LOAD |
| LD0ENEN[504] | NO_LOAD |
| LD0ENEN[505] | NO_LOAD |
| LD0ENEN[506] | NO_LOAD |
| LD0ENEN[507] | NO_LOAD |
| LD0ENEN[508] | NO_LOAD |
| LD0ENEN[509] | NO_LOAD |
| LD0ENEN[510] | NO_LOAD |
| LD0ENEN[511] | NO_LOAD |
| LD0ENEN[512] | NO_LOAD |
| LD0ENEN[513] | NO_LOAD |
| LD0ENEN[514] | NO_LOAD |
| LD0ENEN[515] | NO_LOAD |
| LD0ENEN[516] | NO_LOAD |
| | |

The data path

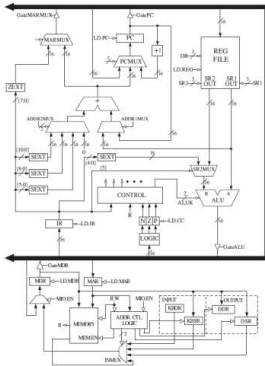


Figure C.3 The LC-3 data path.

- **LD.PC** (p134)

- In order for the PC to be, the finite state machine must assert the PCMUX select lines to choose the output of the box labeled +1 and must also **assert** the LD.PC signal to **load the output of the PCMUX into** the PC at the end of the current cycle.

- **ALUK**

- ALUK consists of two bits, it can have one of four values. Which value it has during any particular clock cycle depends on whether the ALU is required to ADD, AND, NOT, or simply pass one of its inputs to the output during that clock cycle (PASSA).

Additional logic required to provide control signals

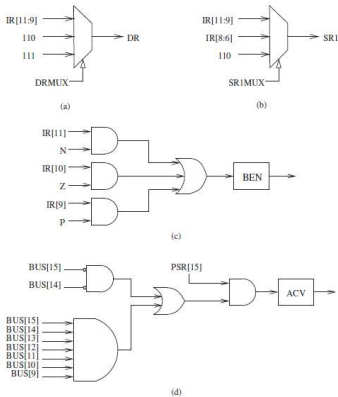
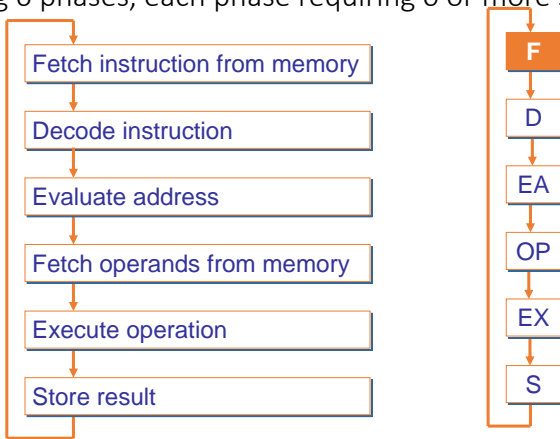


Figure C.6 Additional logic required to provide control signals.

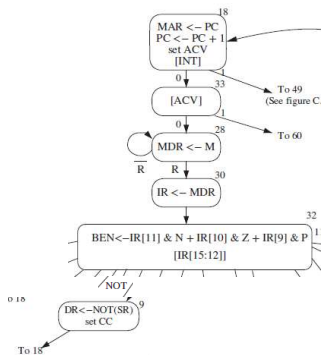
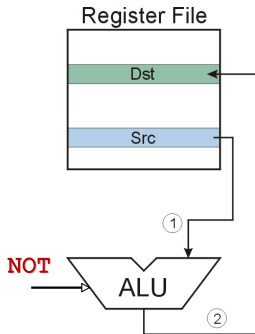
Instruction Cycle (chapter 4.3 & chapter 5.6)

-----including 6 phases, each phase requiring 0 or more steps.



NOT (Register)

| | | | | | | | | | | | | | | | | |
|-----|----|----|----|----|-----|----|---|-----|---|---|---|---|---|---|---|---|
| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| NOT | 1 | 0 | 0 | 1 | Dst | | | Src | | | 1 | 1 | 1 | 1 | 1 | 1 |



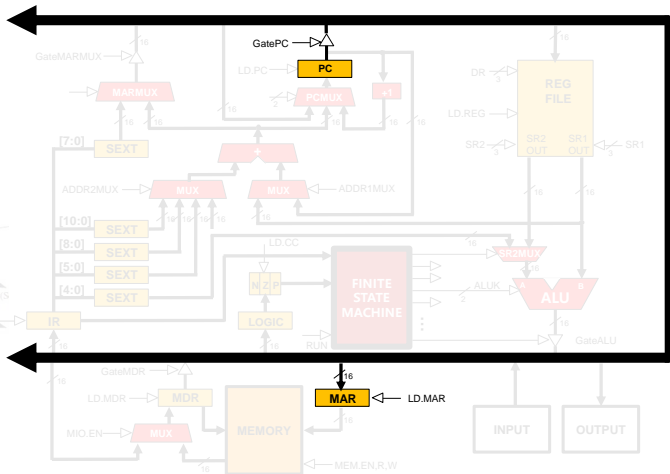
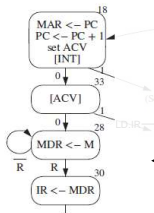
NOT (Register):



State 18

GatePC = YES

LD.MAR = LOAD



NOT (Register):



此处与教材不一致

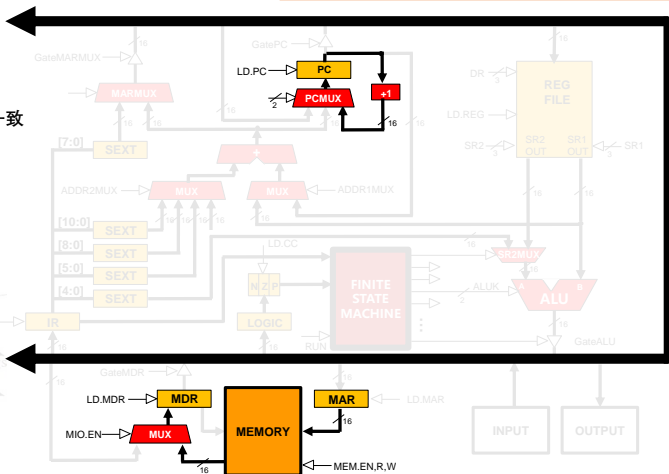
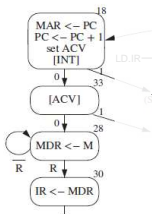
State 28

LD.PC = LOAD

LD.MDR = LOAD

MIO.EN = YES

PCMUX = PC + 1



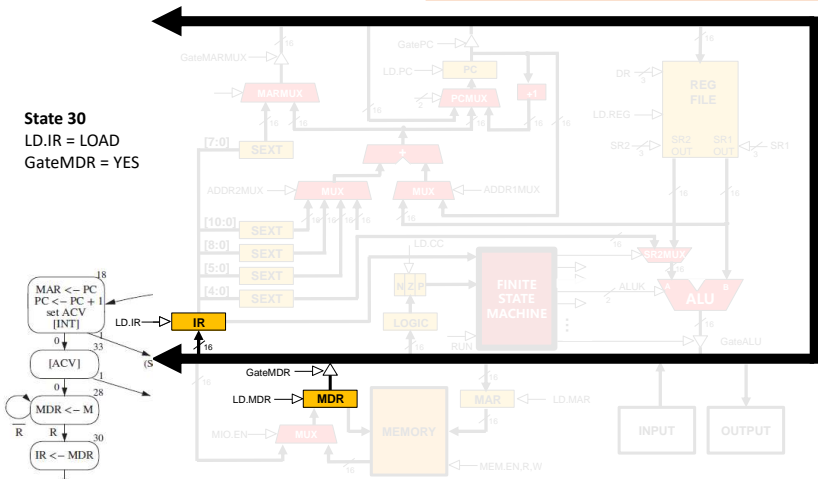
NOT (Register):



State 30

LD.IR = LOAD

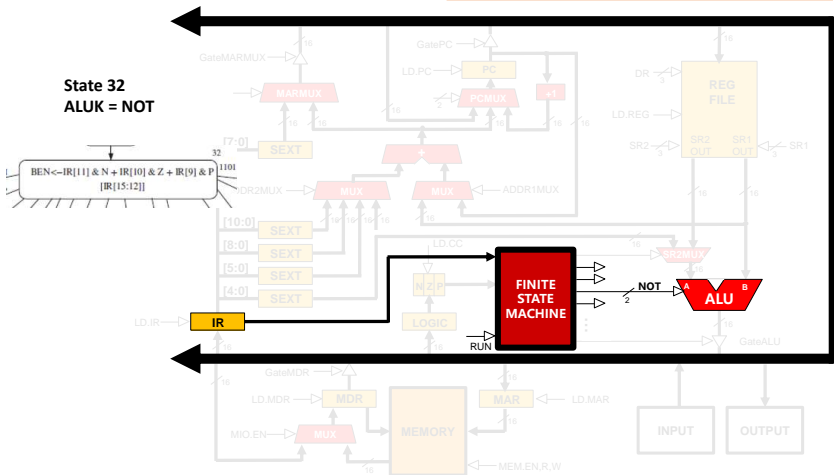
GateMDR = YES



NOT (Register):



State 32
ALUK = NOT



NOT (Register):

参考p131-132

State machine省略了OP and S phases

State 9

SR1MUX = IR[8:6]

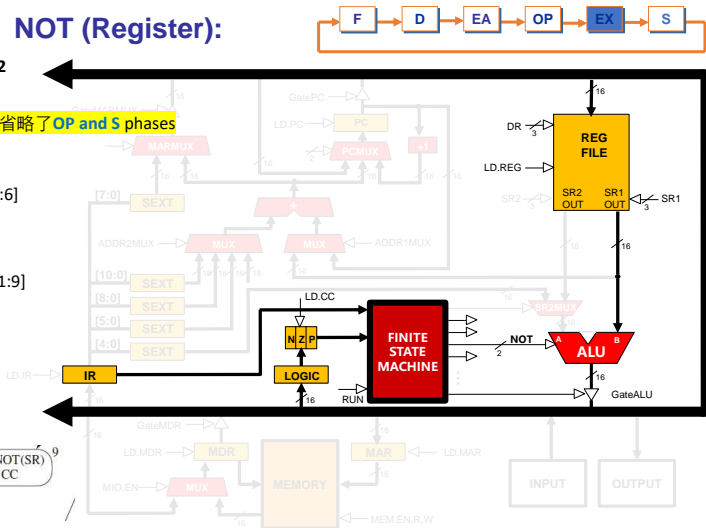
ALUK = NOT

LD.REG=LOAD

GateALU = YES

DRMUX = IR[11:9]

LD.CC = LOAD

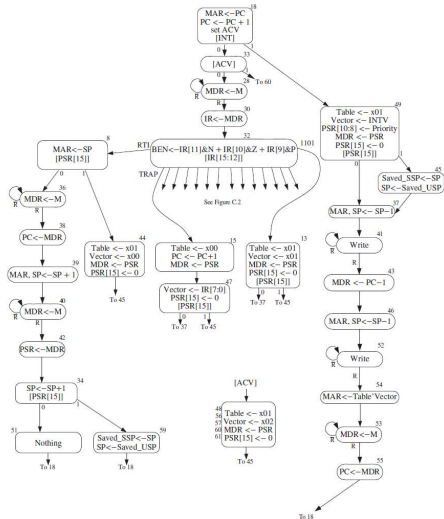


Interrupt and RTI

• Interrupt

- State 18 is the only state in which the processor checks for interrupts
- State 45, switch stack
- State 41, write PSR
- State 52, write PC
- State 53, read new instruction

Table'Vector: concatenating Table and Vector
Table : 0x00, trap
0x01, interrupt



- RTI

- ◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ 🔍 ↺

