

# 计算机组成原理

## Lab0 实验课程概要

2022春季

[zjx@ustc.edu.cn](mailto:zjx@ustc.edu.cn)

# 大纲

- 实验简介
- 实验内容与计划
- 实验成绩评定
- 实验检查标准
- 实验报告要求
- 课程截止时间一览表

# 实验简介

- **实验目标：**设计实现一个真实（非虚拟或仿真，虽简单但较为完整）的计算机硬件系统
- **实验工具：**
  - 软件环境：Vivado 2016.3/2019.1
  - 开发语言：Verilog HDL
  - 硬件环境：Nexys4-DDR开发板或FPGA在线实验平台
- **时间：**周二或周三晚 6:30~9:30（讲课）；  
周三下午14:00~17:00及周四晚 6:30~9:30（答疑）
- **地点：**电三楼406
- **课程资源：**
  - VLAB实验中心：[vlab.ustc.edu.cn](http://vlab.ustc.edu.cn)
  - QQ群：22春-组原实验，793218321

# 实验内容与计划

|     |              |      |
|-----|--------------|------|
| 实验一 | 运算器及其应用      | (1周) |
| 实验二 | 寄存器堆与存储器及其应用 | (1周) |
| 实验三 | CPU测试汇编程序设计  | (1周) |
| 实验四 | 单周期CPU设计     | (3周) |
| 实验五 | 流水线CPU设计     | (3周) |
| 实验六 | 综合设计         | (2周) |

# 实验成绩评定

- 实验总成绩是各次实验成绩的加权求和，每次实验成绩包括检查成绩 (80%) 和报告成绩 (20%)
- 按时完成实验检查和实验报告提交
  - 延迟 $\leq 1$ 周，则最多只能得分80%；若延迟 $\leq 2$ 周，则最多只能得分60%。延迟超过2周不得分
  - 严禁实验代码和实验报告抄袭，否则作零分处理
- 按时且超额完成实验的，视超额部分的创意、检查和报告情况，奖励不超过10分（需教师审定）

# 实验检查标准

- 实验检查内容
  - 实验仿真结果
  - 实验下载后运行结果
  - 回答问题（例如设计思路、解释代码）
  - ... ..
- 实验检查截止时间
  - 规定时长后的周四晚上9:30

**严禁抄袭，否则作零分处理！**

# 实验报告要求

- 实验报告

- 内容包括但不限于逻辑设计（数据通路和状态图）、核心代码、仿真/下载结果、结果分析、实验总结、意见/建议等，设计和测试文件附实验报告后
- Pdf格式，文件名：Labn\_学号\_姓名\_vi，其中n为第几次实验，vi表示版本号，例如，Lab1\_PBxxxxxxxx\_张三\_v1

- 提交实验报告截止日期

- 对应实验检查截止时间延后一周的周四晚上12:00

**严禁抄袭，否则作零分处理！**

# 截止时间一览表

| 实验项目            | 开始时间  | 截止时间   |                 |                |       |
|-----------------|-------|--------|-----------------|----------------|-------|
|                 |       | 检查100% | 检查80%<br>报告100% | 检查60%<br>报告80% | 报告60% |
| 1. 运算器及其应用      | 3月15日 | 3月24日  | 3月31日           | 4月7日           | 4月14日 |
| 2. 寄存器堆与存储器及其应用 | 3月22日 | 3月31日  | 4月7日            | 4月14日          | 4月21日 |
| 3. 汇编应用程序设计     | 3月29日 | 4月7日   | 4月14日           | 4月21日          | 4月28日 |
| 4. 多周期CPU设计     | 4月5日  | 4月28日  | 5月5日            | 5月12日          | 5月19日 |
| 5. 流水线CPU设计     | 4月26日 | 5月19日  | 5月26日           | 6月2日           | 6月9日  |
| 6. 综合设计         | 5月17日 | 6月2日   | 6月9日            | 6月16日          | 6月23日 |



# The End

# 计算机组成原理

## Lab1 运算器及其应用

2022春季

[zjx@ustc.edu.cn](mailto:zjx@ustc.edu.cn)

# 大纲

- 实验目标
- Verilog 语法复习
- FPGAOL实验平台使用
- 实验内容
- 实验步骤

# 实验目标

- 掌握算术逻辑单元 (ALU) 的功能
- 掌握数据通路和控制器的设计方法
- 掌握组合电路和时序电路，以及参数化和结构化的 Verilog描述方法
- 了解查看电路性能和资源使用情况

# Verilog语法复习

## ➤ Verilog描述注意事项

- 推荐使用简单规范的描述方式
- 组合电路
  - 使用assign 或者 always @\* 描述, “=” 赋值;
  - 必须配对使用if...else, case语句赋值完全, 避免出现锁存器;
  - 避免出现反馈! 例如,  $y = y + x$
  - 无需复位, 即组合函数的自变量中无复位信号
- 时序电路
  - 使用always @(posedge clk, posedge rst) 描述, “<=” 赋值
  - 边沿敏感变量表中避免出现除时钟和复位外的其他信号
  - 时钟信号避免出现在语句块内

# Verilog语法复习

## ➤ 变量类型问题

- 使用always语句描述的变量，务必声明为reg类型（声明为reg类型的变量，综合后不一定生成寄存器）；
- 使用assign语句赋值的变量，应声明为wire类型；
- initial或always语句中，未定义直接赋值的变量默认为线网类型的标量（1位wire），向量变量(位宽大于1)必须先定义后使用；
- 同一进程中尽量在一个if或case语句块中对一个变量赋值，否则后边的赋值会覆盖前面的赋值，可能导致逻辑上的问题（特例：组合逻辑描述时，为避免形成锁存器而在开始给变量赋初值）。

# Verilog语法复习

## – 示例：变量类型

```
wire [7:0] a, b;  
reg [7:0] r1, r2, r3;
```

```
always @(*)      // (en, a, b)  
  if (en) r1 = a;  
  else r1 = b;
```

```
always @(en, a)  // @(*)  
  if (en) r2 = a;
```

```
always @(posedge clk)  
  if (en) r3 = a;
```

组合电路：

敏感变量不要遗漏

所有条件分支均有赋值

不能含有反馈，如 $r1=r1+1$

锁存器：避免使用

寄存器：必有触发时钟

# Verilog语法复习

## ➤ 多驱动与多重时钟问题

### — 多驱动问题

- 模块中所有的assign和always块都是并行执行的，不要在多个并行执行体中对同一变量赋值

### — 多重时钟问题

- 不能采用行为描述方式来综合实现多个时钟或多个边沿驱动的触发器

例如：

```
always @(posedge clka, posedge clkb)
```

```
always @(posedge clk, negedge clk)
```



# Verilog语法复习

## ➤ 参数化模块：模块间传递参数

- 模块格式定义如下：

```
module module_name
```

```
#(parameter_list 参数声明)
```

```
(端口声明) ;
```

```
变量声明;
```

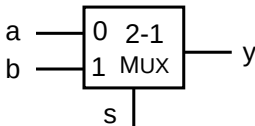
```
逻辑功能描述;
```

```
endmodule
```

# Verilog语法复习

## – 示例1: MUX2

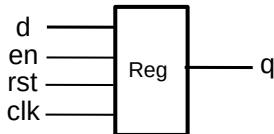
```
module mux2                                //模块名: mux2
  #(parameter MSB = 31,                  //参数声明: 数据最高有效位
    LSB = 0                               //数据最低有效位
  )
  (output [MSB : LSB] y,                  //端口声明: 输出数据
   input [MSB : LSB] a, b,               //两路输入数据
   input s                                //数据选择控制
  );
  assign y = s ? b : a;                  //逻辑功能描述
endmodule
```



# Verilog语法复习

## – 示例2：寄存器

```
module register
  #(parameter WIDTH = 32,
    RST_VALUE = 0)
  (input clk, rst, en,
   input [WIDTH-1 : 0] d,
   output reg [WIDTH-1 : 0] q);
  always @(posedge clk, posedge rst)
    if (rst) q <= RST_VALUE;
    else if (en)
      q <= d;
endmodule
```



- d, q: 输入、输出数据
- clk, rst, en: 时钟、复位、使能

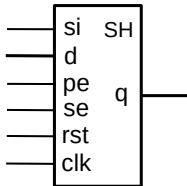
寄存器功能表

| rst | clk | en | q | 功能 |
|-----|-----|----|---|----|
| 1   | x   | x  | 0 | 复位 |
| 0   | ↑   | 1  | d | 置数 |
| 0   | ↑   | 0  | q | 保持 |

# Verilog语法复习

## – 示例3: 移位寄存器

```
module shifter
    #(parameter N = 8,
      RST_VALUE = {N{1'b0}})
    (input clk, rst, pe, se,
     input [N-1: 0] d,
     output reg [N-1: 0] q);
    always @(posedge clk, posedge rst)
        if (rst) q <= RST_VALUE;
        else if (pe) q <= d;
        else if (se) q <= {si, q[N-1: 1]};
endmodule
```



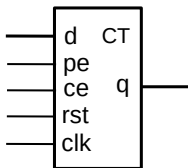
移位寄存器功能表

| rst | clk | pe | se | 功能 |
|-----|-----|----|----|----|
| 1   | x   | x  | x  | 复位 |
| 0   | ↑   | 1  | x  | 置数 |
| 0   | ↑   | 0  | 1  | 右移 |
| 0   | ↑   | 0  | 0  | 保持 |

# Verilog语法复习

## – 示例4：计数器

```
module counter
  #(parameter N = 8,
    RST_VALUE = {N{1'b1}})
  (input clk, rst, pe, ce,
   input [N-1: 0] d,
   output reg [N-1: 0] q);
  always @(posedge clk, posedge rst)
    if (rst) q <= RST_VALUE;
    else if (pe) q <= d;
    else if (ce) q <= q-1;
endmodule
```



递减计数器功能表

| rst | clk | pe | ce | 功能 |
|-----|-----|----|----|----|
| 1   | x   | x  | x  | 复位 |
| 0   | ↑   | 1  | x  | 置数 |
| 0   | ↑   | 0  | 1  | 计数 |
| 0   | ↑   | 0  | 0  | 保持 |

# Verilog语法复习

## ➤ 模块实例化

### – 模块实例化语句格式:

```
module_name #(parameter_map) instance_name (port_map);
```

### – 端口映射方式: 基于位置或者基于名字, 不可混合使用

- 位置映射: 按模块中端口定义的顺序传递

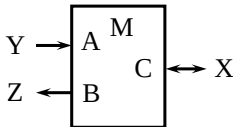
例如: 模块定义为 `module M(A, B, C);`

`M M1(Y, Z, X);` //顺序很重要

- 名字映射: `.PortName (value)`

`M M1(.B(Z), .C(X), .A(Y));` // 顺序无关

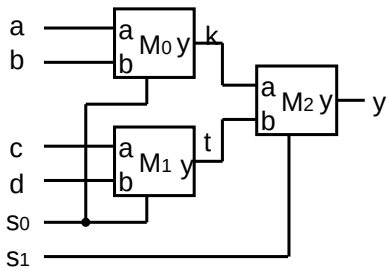
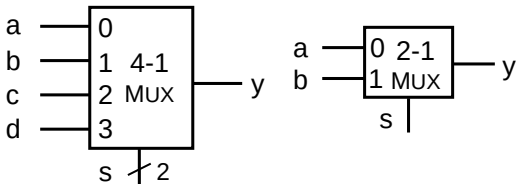
### – 参数的映射方法类似



# Verilog语法复习

## – 示例: MUX4\_8

```
module mux4_8
  (output [7:0] y,
   input [7:0] a, b, c, d,
   input [1:0] s
  );
  wire [7:0] k, t;
  //位置映射
  mux2 #(7, 0) M0 (k, a, b, s[0]);
  mux2 #(7) M1 (t, c, d, s[0]);
  //名字映射
  mux2 #(.MSB(7)) M2 (.s(s[1]), .a(k), .b(t),
    .y(y));
endmodule
```



# Verilog语法复习

## ➤ 仿真时钟

```
reg clk;  
// 时钟周期和个数  
parameter CYCLE = 10, Number = 20;  
  
initial begin  
    clk = 0;  
    repeat (2* Number ) //或者 forever  
        # CYCLE/2 clk = ~ clk;  
end
```

- **initial**和**#**仅用于仿真，不会产生实际硬件电路



# FPGAOL实验平台使用

- 登录平台网站：[fpgaol.ustc.edu.cn](http://fpgaol.ustc.edu.cn)，使用统一身份认证登录，或者直接以游客身份登录

## Login to FPGAOL

Username

Password

login

You can also login with [统一身份认证](#)

可以尝试 [游客访问](#)

# FPGAOL实验平台使用

- 设备获取：点击“acquire”按钮获取一个FPGA结点
- 成功后在下方link栏将显示链接，通过链接进入设备操作界面
  - 默认使用时长10分钟(自动释放结点，点击“release”按钮手动释放)

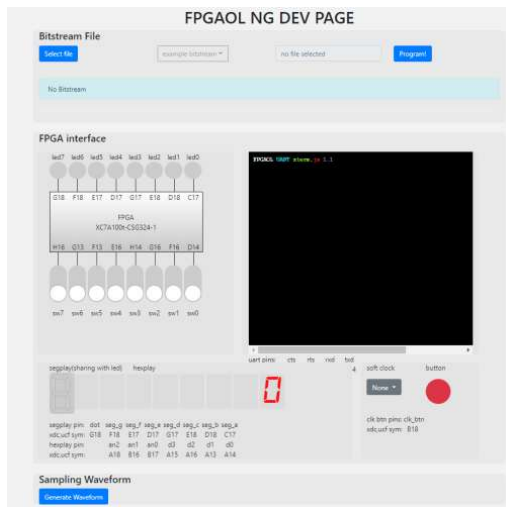
| # | Device Type | Vacant/Total | Manual                             | xdc                         | Use                     |                         |
|---|-------------|--------------|------------------------------------|-----------------------------|-------------------------|-------------------------|
| 1 | FPGAOL 1.0  | 72 / 73      | <a href="#">FPGAOL v1.1 manual</a> | <a href="#">fpgaol1.xdc</a> | <a href="#">acquire</a> | <a href="#">release</a> |
| 2 | FPGAOL 2.0  | 18 / 18      |                                    | <a href="#">fpgaol2.xdc</a> | <a href="#">acquire</a> | <a href="#">release</a> |
| 3 | ZYBO Linaro | 0 / 0        |                                    |                             | <a href="#">acquire</a> | <a href="#">release</a> |

|                 |                                                                                                                                                                                                                               |      |      |
|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|------|
| Device ID       | 216                                                                                                                                                                                                                           | None | None |
| Acquire Time    | March 11, 2022, 5:25 p.m.                                                                                                                                                                                                     | None | None |
| Expiration Time | March 11, 2022, 5:35 p.m.                                                                                                                                                                                                     | None | None |
| Panel Link      | <a href="http://202.38.79.134:12216/?token=MM2DQNLGHFRTCNRRMIZTINBUMJSDKNJZMIYWEZLEMM2DON3CGYYDOO8TGAYGCM8RP0614">http://202.38.79.134:12216/?token=MM2DQNLGHFRTCNRRMIZTINBUMJSDKNJZMIYWEZLEMM2DON3CGYYDOO8TGAYGCM8RP0614</a> | None | None |
| XVC Server      | 202.38.79.134-2                                                                                                                                                                                                               | None | None |

# FPGAOL实验平台使用

- 烧写FPGA: 点击“Select file”按钮，选择需要烧写的bit文件，点击“Program！”

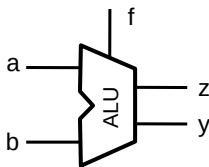


# 实验内容

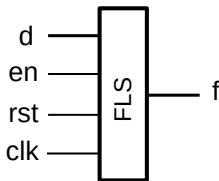
## 1. 算术逻辑单元 (ALU)

1) 32位操作数

2) 6位操作数



## 2. ALU应用：计算斐波那契—卢卡斯数列 (Fibonacci Lucas Series)



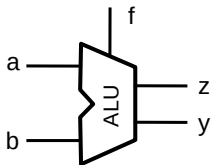
# 实验内容

## 1. 算术逻辑单元 (ALU模块)

### 1) 32位操作数ALU

```
module alu #(parameter WIDTH = 32)  //数据宽度
```

```
(  
    input [WIDTH-1:0] a, b,          //两操作数 (对于减运算, a是被减数)  
    input [2:0] f,                  //操作功能 (加、减、与、或、异或等)  
    output [WIDTH-1:0] y,           //运算结果 (和、差 ...)  
    output z                        //零标志 (运算结果为零, z置1)  
);
```



ALU 模块功能表

| f   | y     | z |
|-----|-------|---|
| 000 | a + b | * |
| 001 | a - b | * |
| 010 | a & b | * |
| 011 | a   b | * |
| 100 | a ^ b | * |
| 其他  | 0     | 1 |

\* 表示根据运算结果设置

# 实验内容

- 查看ALU模块Vivado生成电路和资源使用情况
  - 查看Vivado生成电路
    - RTL电路: Flow Navigator >> RTL Analysis >> Open Elaborated Design >> Schematic
    - 综合电路: Flow Navigator >> Synthesis >> Open Synthesized Design >> Schematic
  - 资源使用情况
    - 综合电路: Flow Navigator >> Synthesis >> Open Synthesized Design >> Report Utilization

# 实验内容

- 查看ALU模块综合电路性能
  - Flow Navigator >> Synthesis >> Open Synthesized Design >> Report Timing Summary

Timing

General Information

Timer Settings

Design Timing Summary

Clock Summary (1)

> Check Timing (12)

▼ Intra-Clock Paths

▼ clk

Setup 7.713 ns (4)

Hold 0.137 ns (4)

Pulse Width 4.500 ns

Timing Summary - timing\_1

| Name   | Slack | Levels | High Fanout | From       | To         | Total Delay |
|--------|-------|--------|-------------|------------|------------|-------------|
| Path 1 | 7.713 | 2      | 3           | A_reg[1]/C | S_reg[3]/D | 2.151       |
| Path 2 | 8.102 | 1      | 4           | B_reg[0]/C | S_reg[2]/D | 1.762       |
| Path 3 | 8.304 | 1      | 4           | B_reg[0]/C | S_reg[1]/D | 1.560       |
| Path 4 | 8.597 | 1      | 4           | A_reg[0]/C | S_reg[0]/D | 1.267       |

# 实验内容

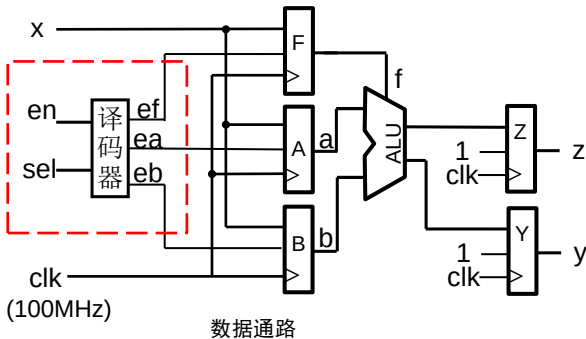
## 2) 6位操作数ALU

### ➤ 外设复用

由于FPGAOL外设资源有限，因此端口需要分时复用：操作数a, b和功能f 复用开关输入x[5:0]。方法：通过sel和en，生成译码电路，将开关输入x[5:0]分时存入寄存器 F(x[2:0])，A(x[5:0])，B(x[5:0])

译码器真值表

| en | sel | ena | enb | ef |
|----|-----|-----|-----|----|
| 1  | 00  | 1   | 0   | 0  |
| 1  | 01  | 0   | 1   | 0  |
| 1  | 10  | 0   | 0   | 1  |
| 1  | 11  | 0   | 0   | 0  |
| 0  | xx  | 0   | 0   | 0  |





# 实验内容

## ➤ ALU模块外设端口分配

```
module alu  
(  
    input clk,  
    input en,  
    input [1:0]sel,  
    input [5:0] x,  
    output [5:0] y,  
    output z  
);
```

外设端口分配表

| 端口  | 外设       |
|-----|----------|
| clk | 100MHz   |
| en  | button   |
| sel | sw[7:6]  |
| x   | sw[5:0]  |
| y   | led[5:0] |
| z   | led[7]   |

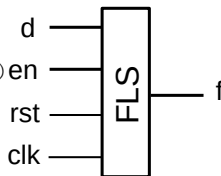
## ➤ALU模块bit文件下载测试

# 实验内容

## 2. ALU应用：计算斐波那契—卢卡斯数列（FLS）

### ➤ FLS模块端口定义

```
module fls (  
    input clk, rst,      //时钟，复位（高电平有效）  
    input en,            //输入输出使能  
    input [6:0] d,       //输入数列初始项  
    output [6:0] f       //输出数列  
);
```



### ➤ FLS模块外设端口分配

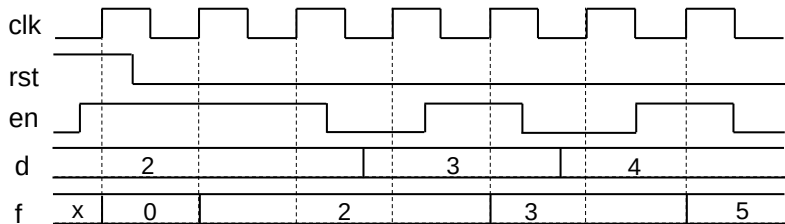
| 端口  | 外设       |
|-----|----------|
| clk | 100MHz   |
| rst | sw7      |
| d   | sw[6:0]  |
| en  | button   |
| f   | led[6:0] |

# 实验内容

## ➤ FLS输入输出逻辑

- 复位rst有效时（高电平）， $f = 0$ ;
- 输出数列的前两项从开关输入，即 $f_0 = d_0, f_1 = d_1$ ;
- 正常工作时， $f_n = f_{n-2} + f_{n-1}$  ( $n > 1$ )
- 模块依次输出 $f = f_0、f_1、f_2、f_3 \dots f_n \dots$

## ➤ FLS模块时序图



# 实验内容

## ➤ FLS逻辑设计要求

- 画出数据通路和有限状态机状态转换图
- 代码描述方式：结构化描述
- 控制器（数据通路中的控制信号）：采用有限状态机（Moore型）

## ➤ FSM描述模式（采用三段式）

// 描述CS

```
always @(posedge clk)
```

```
    if (rst) cs <= S0;    //同步复位
```

```
    else cs <= ns;
```

// 描述NS

```
always @* begin
```

```
    ns = cs;
```

//默认赋值

```
    case (cs)
```

```
        S0: begin
```

```
            .....
```

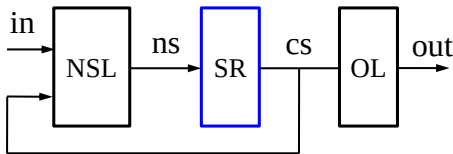
```
        end
```

```
    endcase
```

```
end
```

//描述输出

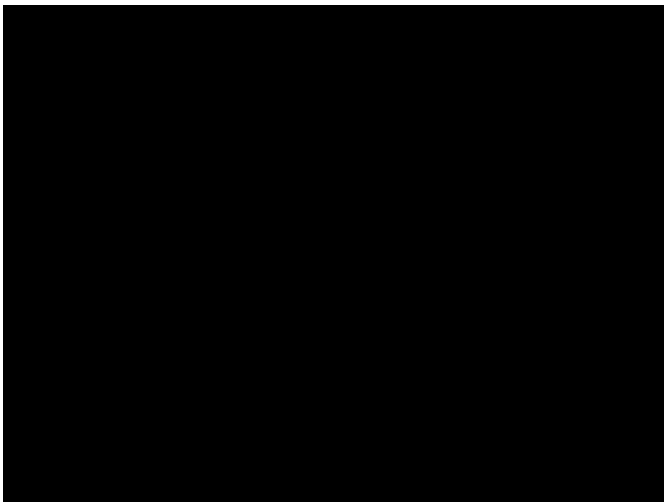
```
assign out = (cs == S0) ? 1'b1:1'b0;
```



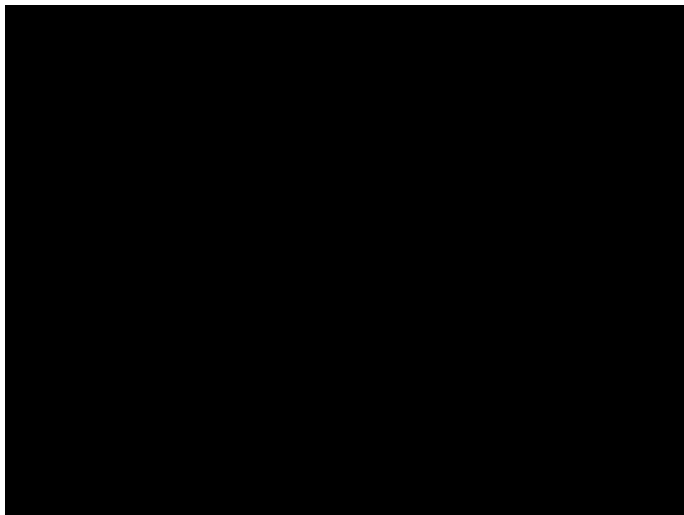
# 实验步骤

1. 完成ALU模块的逻辑设计和仿真
2. 查看32位ALU的RTL和综合电路图，以及综合电路资源和时间性能报告
3. 完成6位ALU的下载测试，并查看RTL电路图，以及实现电路资源和时间性能报告
4. 完成FLS的逻辑设计、仿真和下载测试

# ALU实现效果



# FLS实现效果



# The End



# 计算机组成原理

## Lab2 寄存器堆与存储器及其应用

2022春季

[zjx@ustc.edu.cn](mailto:zjx@ustc.edu.cn)

# 实验目标

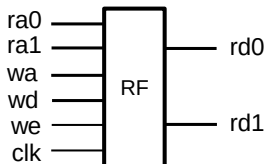
- 掌握寄存器堆（**Register File**）和存储器的功能、时序及其应用
- 熟练掌握数据通路和控制器的设计和描述方法

# 实验内容

## 1. 寄存器堆（Register File）：

- 行为方式参数化描述32 x WIDTH寄存器堆；
- 完成功能仿真

- clk: 时钟
- ra0, rd0: 异步读端口0
- ra1, rd1: 异步读端口1
- wa, wd, we: 同步写端口

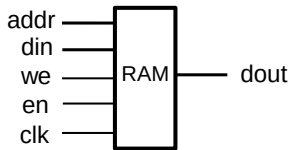


# 实验内容

## 2. RAM存储器:

- IP例化分布式和块式16 x 8位单端口RAM;
- 完成功能仿真和对比

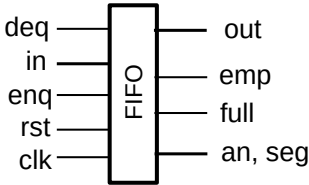
- clk: 时钟
- en: 总使能
- we: 写使能
- addr: 读/写地址
- din: 输入数据
- dout: 输出数据



# 实验内容

## 3. 利用寄存器堆实现FIFO队列:

- enq, deq: 入队列和出队列使能, 假定两者是互斥的, 高电平有效且均要求一次有效仅允许操作一项数据
- in, out: 入/出队列数据
- full, emp: 队列满/空标志, 在满或空时忽略入/出队操作
- an, seg: 数码管控制信号, 显示队列数据
- clk, rst: 时钟, 复位



# 实验内容

## 3. 利用寄存器堆实现FIFO队列：

- 设计FIFO队列电路的数据通路和控制器；
- 结构化方式描述数据通路，**Moore**型FSM描述控制器
- 完成功能仿真
- **FIFO**队列电路下载至FPGA Online中测试

# 寄存器堆

- 也称寄存器文件(Register File)

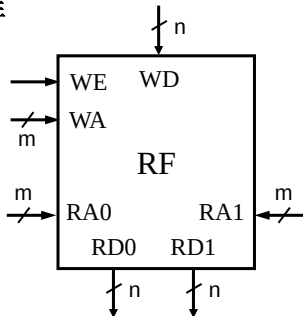
例如，三端口的 $2^m \times n$ 位寄存器堆

## 1个写端口

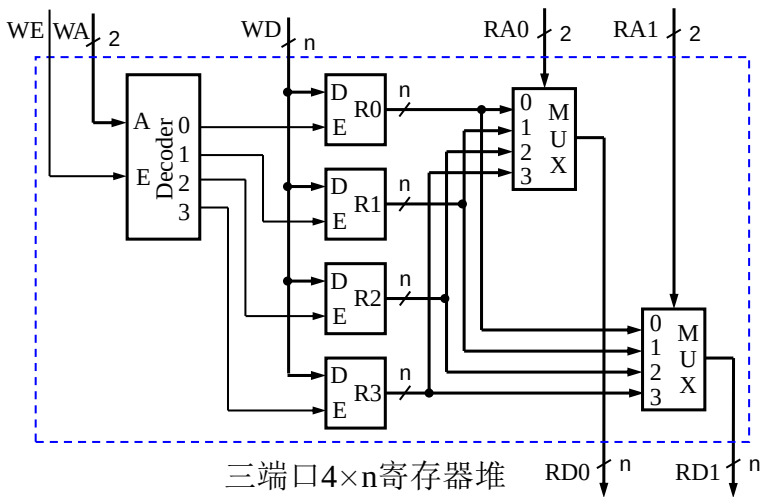
- WA: 写地址
- WD: 写入数据
- WE: 写使能

## 2个读端口

- RA0、RA1: 读地址
- RD0、RD1: 读出数据



# 寄存器堆-结构

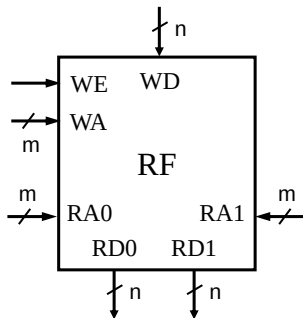




# 寄存器堆-行为描述

// 三端口 $8 \times 4$ 寄存器堆行为描述

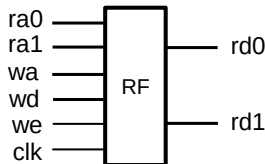
```
wire [2:0] wa, ra0, ra1;  
wire [3:0] wd, rd0, rd1;  
reg [3:0] regfile[0:7];  
  
assign rd0 = regfile[ra0],  
        rd1 = regfile[ra1];  
  
always @ (posedeg clk) begin  
    if (we) regfile[wa] <= wd;  
end
```



# 寄存器堆-端口定义

## // 端口定义

```
module register_file           //32 x WIDTH 寄存器堆
    #(parameter WIDTH = 32)   //数据宽度
    (clk,                      //时钟（上升沿有效）
    input [4:0] ra0,           //读端口 0 地址
    output [WIDTH-1:0] rd0,    //读端口 0 数据
    input [4:0] ra1,           //读端口 1 地址
    output [WIDTH-1:0] rd1,    //读端口 1 数据
    input [4:0] wa,            //写端口地址
    input we,                  //写使能，高电平有效
    input [WIDTH-1:0] wd       //写端口数据
    );
```



# 存储器IP核

- **Vivado**中有存储器IP核可以直接使用
- 两种IP类型：分布式（**Distributed**）、块式（**Block**）存储器
- 定制化方式：**ROM/RAM**、单端口/简单双端口/真正双端口等

# 存储器IP核-生成方式

- **Flow Navigator >> Project Manager >> IP Catalog**
  - Memories & Storage Elements >> RAMs & ROMs >> Distributed Memory Generator
  - 或者 Basic Elements >> Memory Elements >> Distributed Memory Generator
    - Memory config >> Memory Type: [Single Port RAM](#)
    - RST & Initialization >> [Load COE File](#)

同步写端口：a (地址), d (数据), we (写使能), clk

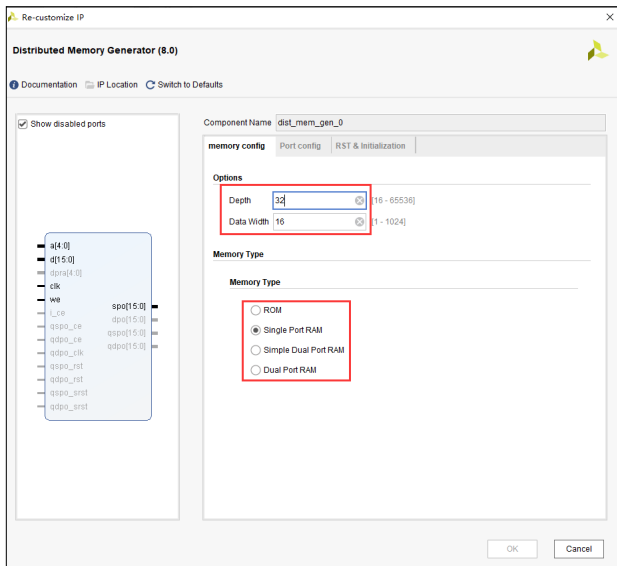
异步读端口：a (地址), spo (数据)

# 存储器IP核-生成方式

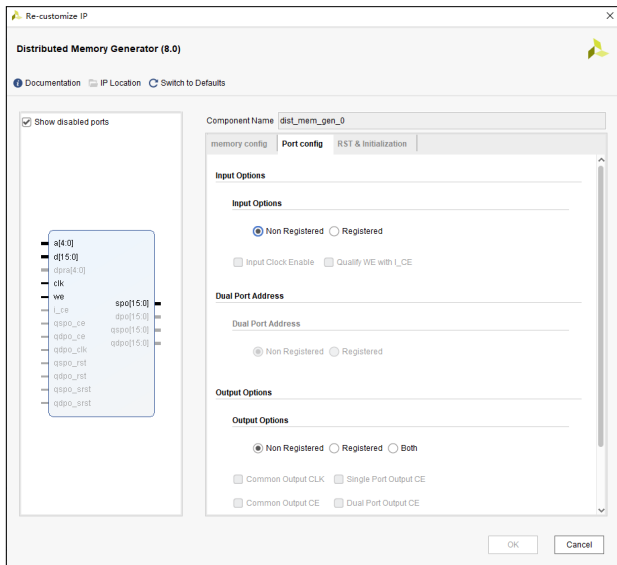
The screenshot shows the Xilinx IP Catalog interface. On the left is the 'Flow Navigator' sidebar with sections: PROJECT MANAGER (Settings, Add Sources, Language Templates, IP Catalog), and IP INTEGRATOR (Create Block Design, Open Block Design, Generate Block Design). The 'IP Catalog' is highlighted with a red box. The main area shows search results for 'mem' (27 matches). The table lists various IP blocks, with 'Block Memory Generator' and 'Distributed Memory Generator' highlighted by red boxes.

| Name                                   | AXI4              | Status     | License  | VLNV                              |
|----------------------------------------|-------------------|------------|----------|-----------------------------------|
| AXI Multi Channel Direct Memory Access | AXI4, AXI4-Stream | Production | Included | xilinx.com:ip:axi_mcdma:1.1       |
| AXI Video Direct Memory Access         | AXI4, AXI4-Stream | Production | Included | xilinx.com:ip:axi_vdma:6.3        |
| SDx Memory Subsystem                   | AXI4              | Production | Included | xilinx.com:ip:sdx_memory_subsys   |
| Basic Elements                         |                   |            |          |                                   |
| Memory Elements                        |                   |            |          |                                   |
| Block Memory Generator                 | AXI4              | Production | Included | xilinx.com:ip:blk_mem_gen:8.4     |
| Distributed Memory Generator           |                   | Production | Included | xilinx.com:ip:dist_mem_gen:8.0    |
| Communication & Networking             |                   |            |          |                                   |
| Ethernet                               |                   |            |          |                                   |
| AXI Direct Memory Access               | AXI4, AXI4-Stream | Production | Included | xilinx.com:ip:axi_dma:7.1         |
| AXI Multi Channel Direct Memory Access | AXI4, AXI4-Stream | Production | Included | xilinx.com:ip:axi_mcdma:1.1       |
| Embedded Processing                    |                   |            |          |                                   |
| AXI Infrastructure                     |                   |            |          |                                   |
| AXI Memory Init                        | AXI4              | Production | Included | xilinx.com:ip:axi_memory_init:1.0 |

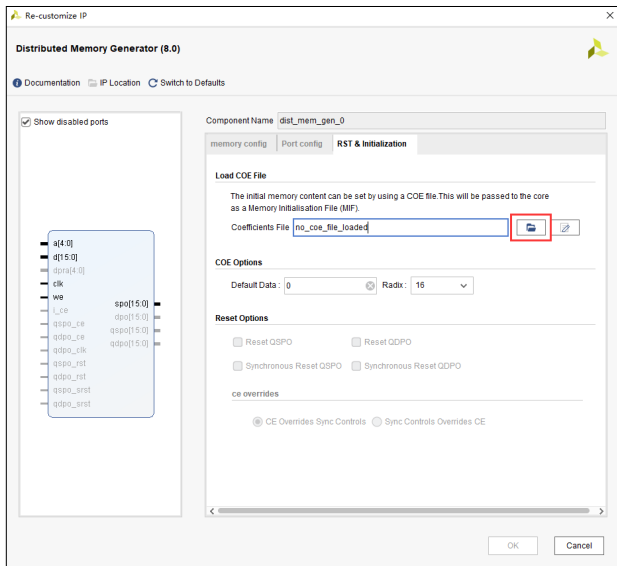
# 分布式存储器 IP



# 分布式存储器 IP

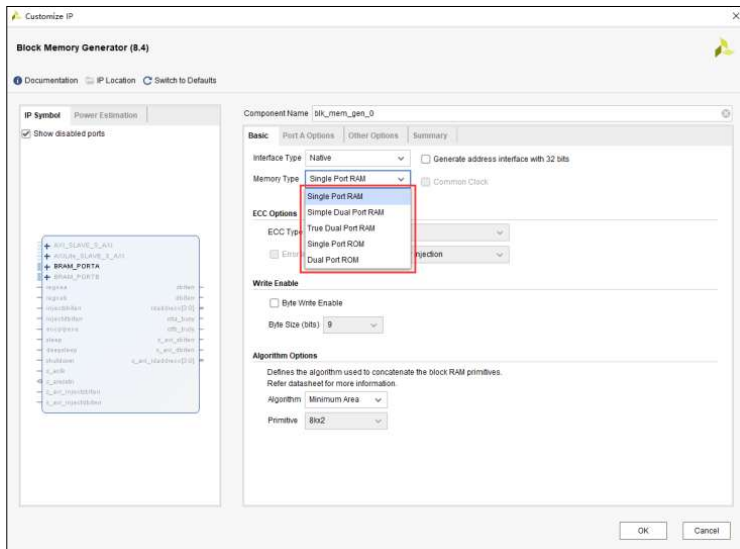


# 分布式存储器 IP

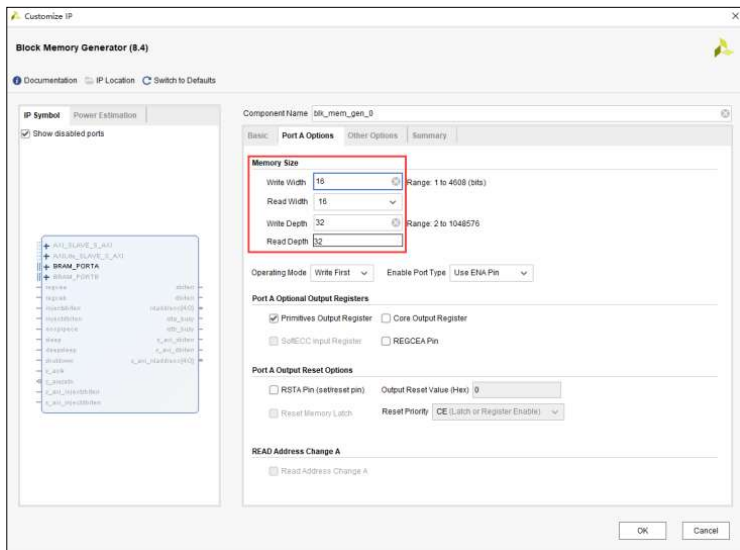




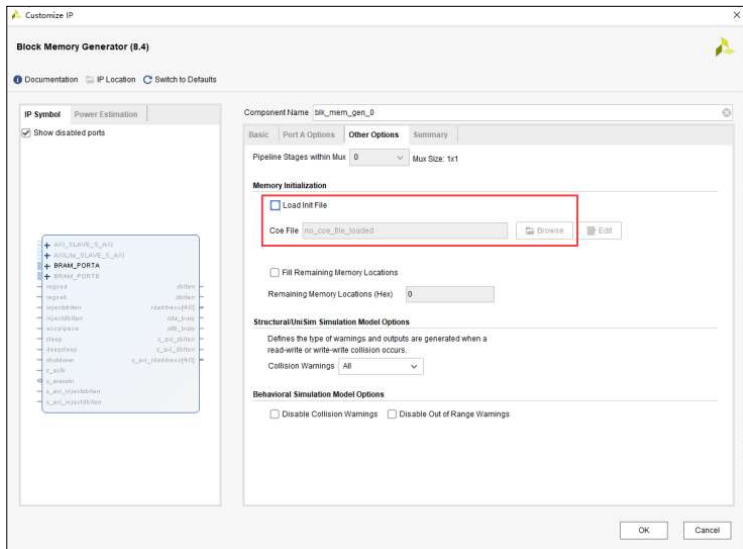
# 块式存储器IP



# 块式存储器 IP

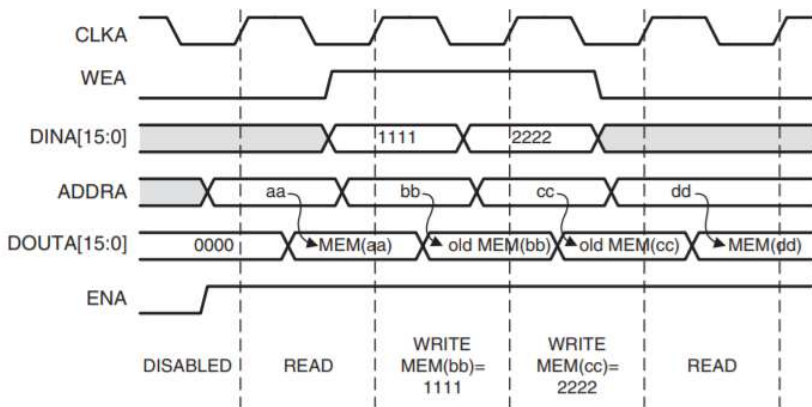


# 块式存储器 IP



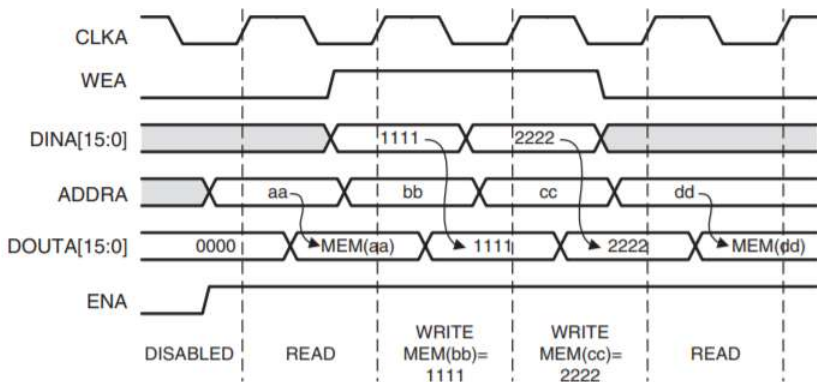
# 存储器IP核-存储器时序（一）

- Read First Mode



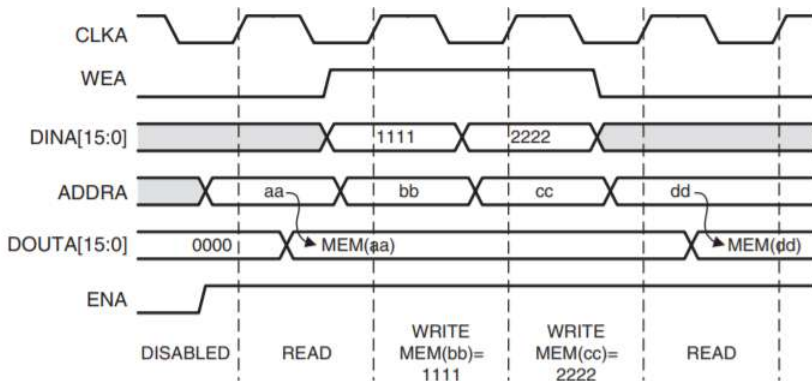
# 存储器IP核-存储器时序（二）

- Write First Mode



# 存储器IP核-存储器时序（三）

- No Change Mode



# 存储器IP核-实例化模板

- **Project Manager – display >> Sources >> IP Sources**

- IP >> dist\_mem\_gen\_0 >> Instantiation Template >> dist\_mem\_gen\_0.vco

```
dist_mem_gen_0 your_instance_name (  
    .a(a),           // input wire [3 : 0] a  
    .d(d),           // input wire [7 : 0] d  
    .clk(clk),       // input wire clk  
    .we(we),         // input wire we  
    .spo(spo)        // output wire [7 : 0] dpo  
);
```

实例化模板

# 存储器IP核-实例化模板

The screenshot displays the Xilinx Vivado interface. On the left, the 'Sources' window shows the project hierarchy. The 'IP (1)' folder is expanded, revealing the 'dist\_mem\_gen\_0 (15)' component. Under its 'Instantiation Template (2)' sub-folder, the file 'dist\_mem\_gen\_0.veo' is highlighted with a red box. The bottom of the Sources window has tabs for 'Hierarchy', 'IP Sources' (which is selected and also highlighted with a red box), 'Libraries', and 'Compile Order'.

On the right, the 'dist\_mem\_gen\_0.veo' file is open in the editor. The code is a Verilog instantiation template. A red box highlights the instantiation block starting at line 57:

```
57 dist_mem_gen_0 your_instance_name (  
58   .a(a),      // input wire [3 : 0] a  
59   .d(d),      // input wire [7 : 0] d  
60   .clk(clk),  // input wire clk  
61   .we(we),    // input wire we  
62   .spo(spo)   // output wire [7 : 0] spo  
63 );
```

The rest of the code in the editor includes comments and a line for the instantiation tag end:

```
46 //  
47 // DO NOT MODIFY THIS FILE.  
48  
49 // IP VLNV: xilinx.com:ip:dist_mem_gen:8.0  
50 // IP Revision: 13  
51  
52 // The following must be inserted into your Verilog file for t  
53 // core to be instantiated. Change the instance name and port  
54 // (in parentheses) to your own signal names.  
55  
56 //----- Begin Cut here for INSTANTIATION Template -----//  
64 // INST_TAG_END ----- End INSTANTIATION Template -----  
65
```



# 存储器IP核-COE文件格式

## // COE文件格式

- **An example COE file:**

; Sample Initialization file for a 32x16 distributed ROM

memory\_initialization\_radix = 16;

memory\_initialization\_vector =

23f4 0721 11ff ABe1 0001 1 0A 0

23f4 0721 11ff ABe1 0001 1 0A 0

23f4 721 11ff ABe1 0001 1 A 0

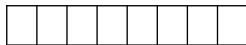
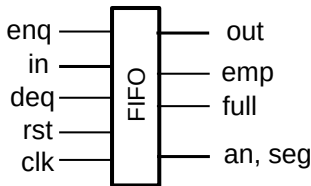
23f4 721 11ff ABe1 0001 1 A 0;

逗号或空格分隔每项  
数据（不允许为负数）

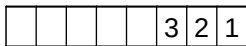
# FIFO队列-功能要求

- 用三端口 $8 \times 4$ 寄存器堆实现最大长度为8的FIFO队列

- **deq, enq**: 出/入队列使能(互斥)，一次有效仅允许操作一项数据
- **out, in**: 出/入队列数据
- **full, emp**: 队列满/空，满/空时忽略入/出队操作
- **an, seg**: 数码管控制信号，显示队列状态



复位(队列空)



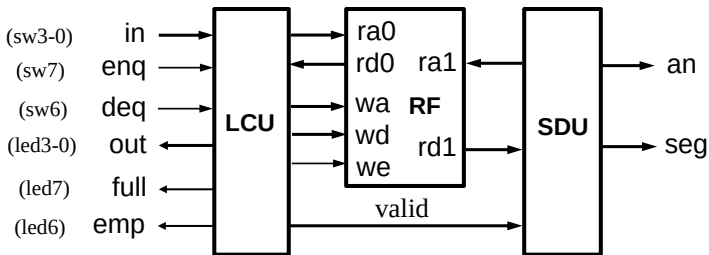
数据1, 2, 3  
依次入队列



数据1出队列

# FIFO队列-逻辑结构

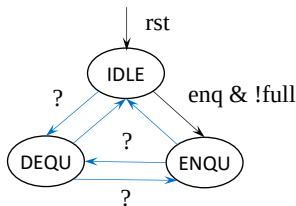
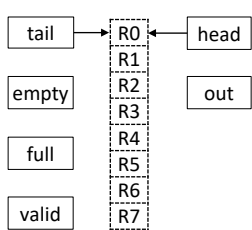
- 队列控制单元 LCU (List Control Unit)
  - 处理出/入队操作，显示队列空/满状态
- 数码管显示单元 SDU (Segment Display Unit)
  - 显示队列数据内容



\* 省略了clk (100MHz) 和 rst (button)

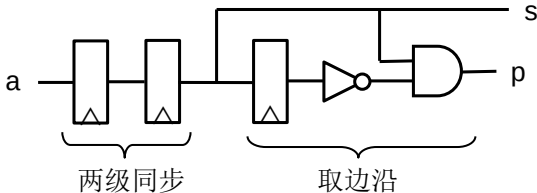
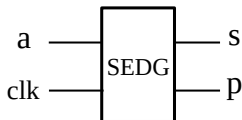
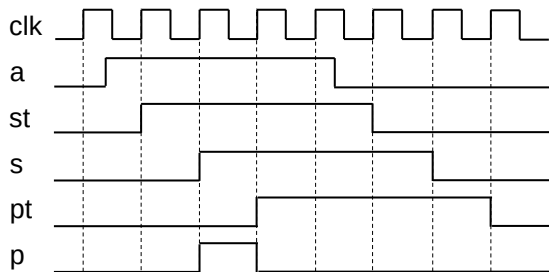
# FIFO队列-控制单元

- **head**: 3位, 队头, 指向出队列位置
- **tail**: 3位, 队尾, 指向入队列位置
- **full, empty**: 各1位, 满和空标志
- **valid**: 8位, 有效标志, 第*i*位对应*R<sub>i</sub>*状态
- **out**: 4位, 出队列数据



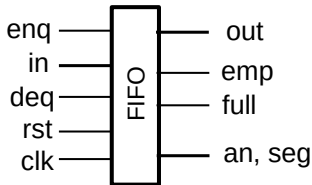
$RF[T] \leftarrow in, V[T] \leftarrow 1,$   
 $T \leftarrow T+1,$   
 $F \leftarrow (T+1) == H,$   
 $E \leftarrow 0;$

# FIFO队列-异步信号同步和取边沿



# FIFO队列-端口定义

```
module fifo (  
    input clk, rst, //时钟（上升沿有效）、同步复位（高电平有效）  
    input enq,      //入队列使能，高电平有效  
    input [3:0] in,  //入队列数据  
    input deq,      //出队列使能，高电平有效  
    output [3:0] out, //出队列数据  
    output [2:0] an,  //数码管选择  
    output [3:0] seg //数码管数据  
);
```



# 实验步骤

1. 行为方式参数化描述寄存器堆，功能仿真
2. IP例化分布式和块式16 x 8位单端口RAM，功能仿真和对比
3. 设计FIFO队列电路的数据通路和控制器，结构化方式描述数据通路，Moore型FSM描述控制器，功能仿真
4. FIFO队列电路下载至FPGA中测试

# The End



# 计算机组成原理

## 实验三 汇编程序设计

2022春季  
zjx@ustc.edu.cn

# 实验大纲

- 实验目标
- 实验环境
- 准备知识
  - ✓ RISC-V汇编指令
  - ✓ Ripes软件简介
  - ✓ RARS软件简介
- 实验内容
- 实验步骤

# 实验目标

- 熟悉RISC-V汇编指令的格式
- 熟悉CPU仿真软件Ripes，理解汇编指令执行的基本原理（数据通路和控制器的协调工作过程）
- 熟悉汇编程序的基本结构，掌握简单汇编程序的设计
- 掌握汇编仿真软件RARS(RISC-V Assembler & Runtime Simulator)的使用方法，会用该软件进行汇编程序的仿真、调试以及生成CPU测试需要的指令和数据文件（COE）
- 理解CPU调试模块PDU的使用方法

# 实验环境

➤ PC 一台

➤ **Ripes:** RISC-V graphical processor simulator

➤ **Rars:** RISC-V Assembler and Runtime Simulator

# 准备知识--RISC-V汇编指令

## 1. RISC-V 32个通用寄存器

| Register | ABI Name | Description                       |
|----------|----------|-----------------------------------|
| x0       | zero     | Hard-wired zero 硬编码 0             |
| x1       | ra       | Return address 返回地址               |
| x2       | sp       | Stack pointer 栈指针                 |
| x3       | gp       | Global pointer 全局指针               |
| x4       | tp       | Thread pointer 线程指针               |
| x5       | t0       | Temporary/alternate link register |
| x6-7     | t1-2     | Temporaries 临时寄存器                 |
| x8       | s0/fp    | Saved register/frame pointer      |
| x9       | s1       | Saved register 保存寄存器              |
| x10-11   | a0-1     | Function arguments/return values  |
| x12-17   | a2-7     | Function arguments 函数参数           |
| x18-27   | s2-11    | Saved registers 保存寄存器             |
| x28-31   | t3-6     | Temporaries 临时寄存器                 |

RISC-V整数寄存器的汇编助记符

# 准备知识--RISC-V汇编指令

## 2. RV32I指令类型（RISC-V基本32位整数指令集）

### 1) 运算类

- 算术: `add`, `addi`, `sub`, `lui`, `auipc`
- 逻辑: `and`, `or`, `xor`
- 移位(shift): `sll`, `srl`, `sra`
- 比较(set if less than): `slt`, `sltu`

| Category   | Name                    | Fmt | RV32I Base |              |
|------------|-------------------------|-----|------------|--------------|
| Shifts     | Shift Left Logical      | R   | SLL        | rd,rs1,rs2   |
|            | Shift Left Log. Imm.    | I   | SLLI       | rd,rs1,shamt |
|            | Shift Right Logical     | R   | SRL        | rd,rs1,rs2   |
|            | Shift Right Log. Imm.   | I   | SRLI       | rd,rs1,shamt |
|            | Shift Right Arithmetic  | R   | SRA        | rd,rs1,rs2   |
|            | Shift Right Arith. Imm. | I   | SRAI       | rd,rs1,shamt |
| Arithmetic | ADD                     | R   | ADD        | rd,rs1,rs2   |
|            | ADD Immediate           | I   | ADDI       | rd,rs1,imm   |
|            | SUBtract                | R   | SUB        | rd,rs1,rs2   |
|            | Load Upper Imm          | U   | LUI        | rd,imm       |
|            | Add Upper Imm to PC     | U   | AUIPC      | rd,imm       |
| Logical    | XOR                     | R   | XOR        | rd,rs1,rs2   |
|            | XOR Immediate           | I   | XORI       | rd,rs1,imm   |
|            | OR                      | R   | OR         | rd,rs1,rs2   |
|            | OR Immediate            | I   | ORI        | rd,rs1,imm   |
|            | AND                     | R   | AND        | rd,rs1,rs2   |
|            | AND Immediate           | I   | ANDI       | rd,rs1,imm   |
| Compare    | Set <                   | R   | SLT        | rd,rs1,rs2   |
|            | Set < Immediate         | I   | SLTI       | rd,rs1,imm   |
|            | Set < Unsigned          | R   | SLTU       | rd,rs1,rs2   |
|            | Set < Imm Unsigned      | I   | SLTIU      | rd,rs1,imm   |

# 准备知识--RISC-V汇编指令

## 2)访存类

- 加载(load): **lw**,  
lb, lh, lbu, lhu
- 存储(store): **sw**,  
sb, sh

## 3)转移类

- 分支(branch):  
**beq**, bne, **blt**, bge,  
bltu, bgeu
- 跳转(jump): **jal**,  
**jalr**

| Category    | Name                 | Fmt | RV32I Base       |
|-------------|----------------------|-----|------------------|
| Branches    | Branch =             | B   | BEQ rs1,rs2,imm  |
|             | Branch ≠             | B   | BNE rs1,rs2,imm  |
|             | Branch <             | B   | BLT rs1,rs2,imm  |
|             | Branch ≥             | B   | BGE rs1,rs2,imm  |
|             | Branch < Unsigned    | B   | BLTU rs1,rs2,imm |
|             | Branch ≥ Unsigned    | B   | BGEU rs1,rs2,imm |
| Jump & Link | J&L                  | J   | JAL rd,imm       |
|             | Jump & Link Register | I   | JALR rd,rs1,imm  |
| Loads       | Load Byte            | I   | LB rd,rs1,imm    |
|             | Load Halfword        | I   | LH rd,rs1,imm    |
|             | Load Byte Unsigned   | I   | LBU rd,rs1,imm   |
|             | Load Half Unsigned   | I   | LHU rd,rs1,imm   |
|             | Load Word            | I   | LW rd,rs1,imm    |
| Stores      | Store Byte           | S   | SB rs1,rs2,imm   |
|             | Store Halfword       | S   | SH rs1,rs2,imm   |
|             | Store Word           | S   | SW rs1,rs2,imm   |

# 准备知识--RISC-V汇编指令

## 1) 运算指令

➤ add rd, rs1, rs2                      #  $x[rd] = x[rs1] + x[rs2]$

| 31      | 25 24 | 20 19 | 15 14        | 12 11 | 7 6    | 0 |
|---------|-------|-------|--------------|-------|--------|---|
| funct7  | rs2   | rs1   | funct3       | rd    | opcode |   |
| 7       | 5     | 5     | 3            | 5     | 7      |   |
| 0000000 | src2  | src1  | ADD/SLT/SLTU | dest  | OP     |   |
| 0000000 | src2  | src1  | AND/OR/XOR   | dest  | OP     |   |
| 0000000 | src2  | src1  | SLL/SRL      | dest  | OP     |   |
| 0100000 | src2  | src1  | SUB/SRA      | dest  | OP     |   |

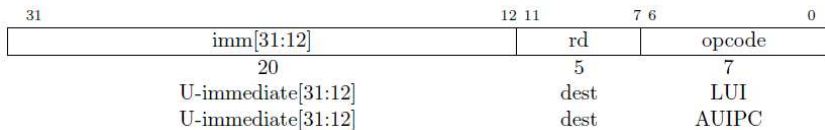
➤ addi rd, rs1, imm                      #  $x[rd] = x[rs1] + sext(imm)$

| 31                | 20 19 | 15 14         | 12 11 | 7 6    | 0 |
|-------------------|-------|---------------|-------|--------|---|
| imm[11:0]         | rs1   | funct3        | rd    | opcode |   |
| 12                | 5     | 3             | 5     | 7      |   |
| I-immediate[11:0] | src   | ADDI/SLTI[U]  | dest  | OP-IMM |   |
| I-immediate[11:0] | src   | ANDI/ORI/XORI | dest  | OP-IMM |   |



# 准备知识--RISC-V汇编指令

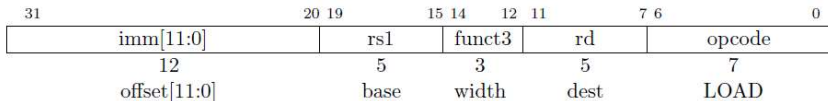
- lui rd, imm      #  $x[rd] = \text{sext}(\text{imm}[31:12] \ll 12)$
- auipc rd, imm    #  $x[rd] = \text{pc} + \text{sext}(\text{imm}[31:12] \ll 12)$



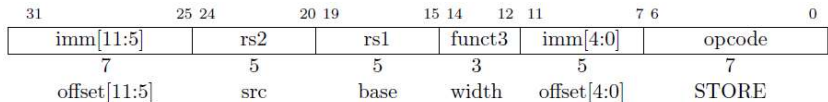
# 准备知识--RISC-V汇编指令

## 2) 访存指令

➤ `lw rd, offset(rs1)`     $\# x[rd] = M[x[rs1] + sext(offset)]$



➤ `sw rs2, offset(rs1)`     $\# M[x[rs1]+sext(offset)] = x[rs2]$



# 准备知识--RISC-V汇编指令

## 3) 分支指令

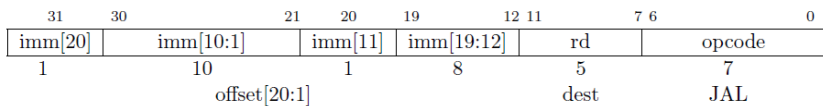
- beq rs1, rs2, offset # if (rs1 == rs2) pc += sext(offset)
- blt rs1, rs2, offset # if (rs1 < rs2) pc += sext(offset)

| 31              | 30        | 25   | 24   | 20      | 19             | 15      | 14     | 12 | 11 | 8 | 7 | 6 | 0 |
|-----------------|-----------|------|------|---------|----------------|---------|--------|----|----|---|---|---|---|
| imm[12]         | imm[10:5] | rs2  | rs1  | funct3  | imm[4:1]       | imm[11] | opcode |    |    |   |   |   |   |
| 1               | 6         | 5    | 5    | 3       | 4              | 1       | 7      |    |    |   |   |   |   |
| offset[12,10:5] |           | src2 | src1 | BEQ/BNE | offset[11,4:1] |         | BRANCH |    |    |   |   |   |   |
| offset[12,10:5] |           | src2 | src1 | BLT[U]  | offset[11,4:1] |         | BRANCH |    |    |   |   |   |   |
| offset[12,10:5] |           | src2 | src1 | BGE[U]  | offset[11,4:1] |         | BRANCH |    |    |   |   |   |   |

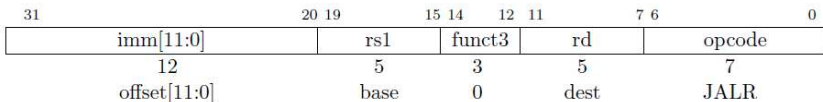
# 准备知识--RISC-V汇编指令

## 4) 跳转指令

➤ jal rd, offset      #  $x[rd] = pc+4$ ;  $pc += sext(offset)$



➤ jalr rd, offset(rs1)    #  $t = pc+4$ ;  
 $pc = (x[rs1] + sext(offset)) \& \sim 1$ ;  $x[rd] = t$



# 准备知识--RISC-V汇编指令

## 3. 汇编指示和伪指令

### ➤ 汇编指示符 (Assembler directives)

- .data, .text
- .word, .half, .byte, .string
- .eqv
- .align
- .....

### ➤ 伪指令

- li, la, mv
- nop, not, neg
- j, jr, call, ret
- .....

Example:

```
.eqv CONSTANT, 0xdeadbeef
```

```
li a0, CONSTANT
```

```
# lui a0,0xdeadc
```

```
# addi a0,a0,0xfffffeef
```

| Text Segment             |            |             |                         |                      |
|--------------------------|------------|-------------|-------------------------|----------------------|
| Bkpt                     | Address    | Code        | Basic                   | Source               |
| <input type="checkbox"/> | 0x00000000 | 0xdead537   | lui x10,0xffffdead      | 4: li a0, 0xdeadbeef |
| <input type="checkbox"/> | 0x00000004 | 0xeeef50513 | addi x10,x10,0xfffffeef |                      |

| Registers |        |            | Floating Point |        |            | Control and Status |        |            |
|-----------|--------|------------|----------------|--------|------------|--------------------|--------|------------|
| Name      | Number | Value      | Name           | Number | Value      | Name               | Number | Value      |
| zero      | 0      | 0x00000000 | zero           | 0      | 0x00000000 | zero               | 0      | 0x00000000 |
| ra        | 1      | 0x00000000 | ra             | 1      | 0x00000000 | ra                 | 1      | 0x00000000 |
| sp        | 2      | 0x00003ffc | sp             | 2      | 0x00003ffc | sp                 | 2      | 0x00003ffc |
| sp        | 3      | 0x00001800 | sp             | 3      | 0x00001800 | sp                 | 3      | 0x00001800 |
| tp        | 4      | 0x00000000 | tp             | 4      | 0x00000000 | tp                 | 4      | 0x00000000 |
| t0        | 5      | 0x00000000 | t0             | 5      | 0x00000000 | t0                 | 5      | 0x00000000 |
| t1        | 6      | 0x00000000 | t1             | 6      | 0x00000000 | t1                 | 6      | 0x00000000 |
| t2        | 7      | 0x00000000 | t2             | 7      | 0x00000000 | t2                 | 7      | 0x00000000 |
| s0        | 8      | 0x00000000 | s0             | 8      | 0x00000000 | s0                 | 8      | 0x00000000 |
| s1        | 9      | 0x00000000 | s1             | 9      | 0x00000000 | s1                 | 9      | 0x00000000 |
| a0        | 10     | 0xdead0000 | a0             | 10     | 0xdeadbeef | a0                 | 10     | 0xdeadbeef |
| a1        | 11     | 0x00000000 | a1             | 11     | 0x00000000 | a1                 | 11     | 0x00000000 |
| a2        | 12     | 0x00000000 | a2             | 12     | 0x00000000 | a2                 | 12     | 0x00000000 |

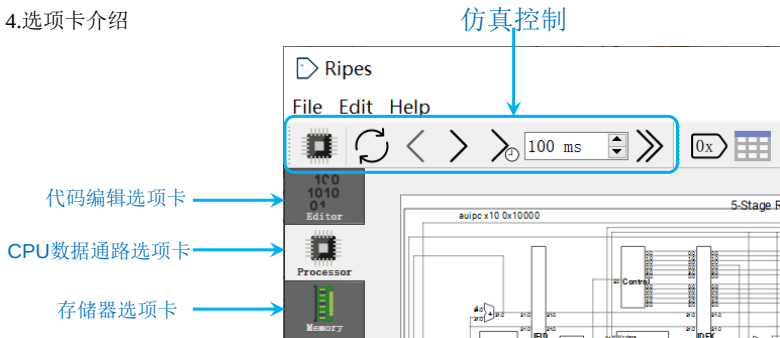
# 准备知识—Ripes软件简介

1. Ripes is a graphical processor simulator and assembly code editor built for the RISC-V instruction set architecture, suitable for teaching how assembly level code is executed on various microarchitectures.

2. 软件下载链接: <https://github.com/mortbopet/Ripes/releases>

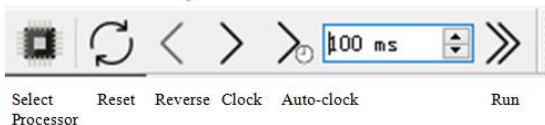
3. 软件简介: <https://github.com/mortbopet/Ripes/wiki/Ripes-Introduction>

4. 选项卡介绍

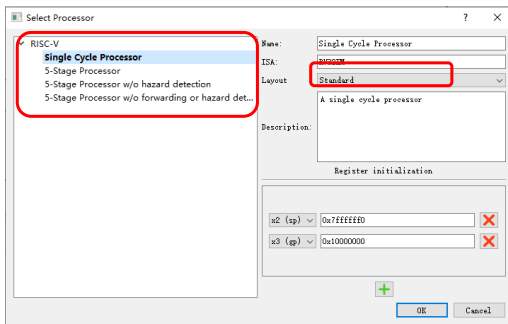


# 准备知识—Ripes软件简介

## ► 仿真控制



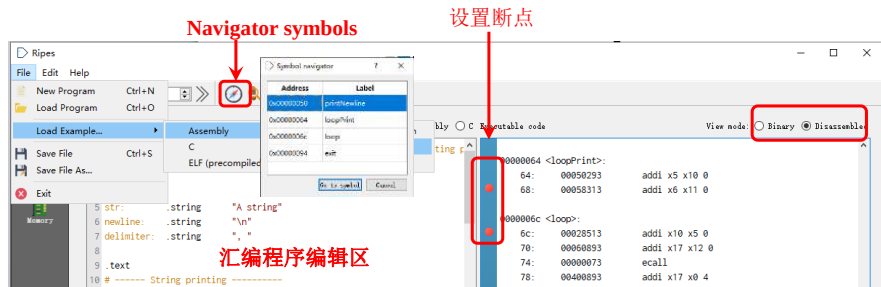
## ✓ Select Processor



- Standard:** Control components and signals are omitted.
- Extended:** Control components and signals are visible as well as wire bit-widths.

# 准备知识—Ripes软件简介

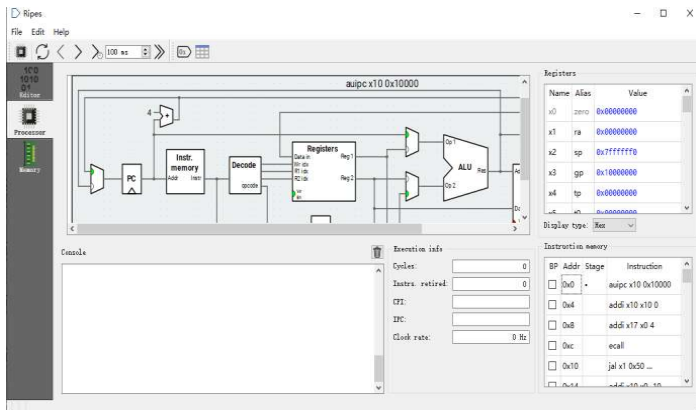
## ➤ 代码编辑选项卡





# 准备知识—Ripes软件简介

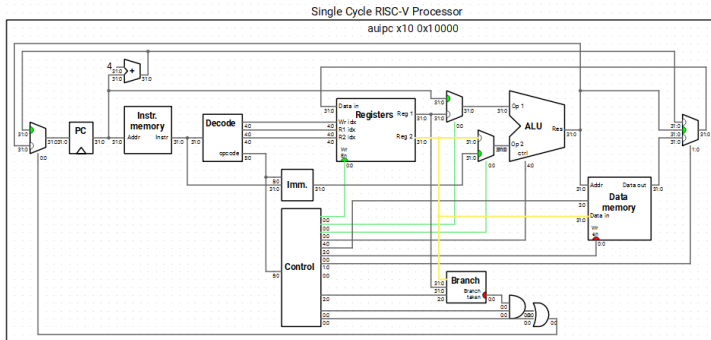
## ➤ CPU数据通路选项卡



- ✓ Registers(can modify), Instruction memory(BP, PC, stage, instruction), execution info, console

# 准备知识—Ripes软件简介

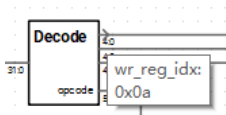
## ✓ 单周期CPU数据通路



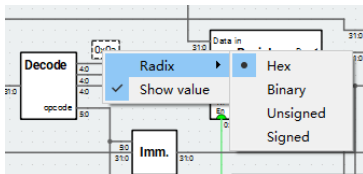
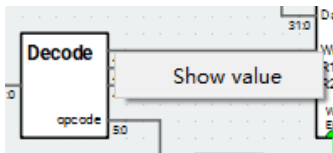
- Green dot in Multiplexers input : signal selected
- Boolean (1-bit signals: high (1) when a wire is green, low (0) when a wire is grey.
- Other signals: when modified, will briefly flash green
- processor view zoom: ctrl+scroll
- Clicking a wire highlights the entirety of the wire: yellow

# 准备知识—Ripes软件简介

- Hover over any port in the processor view: display the name and value of the port

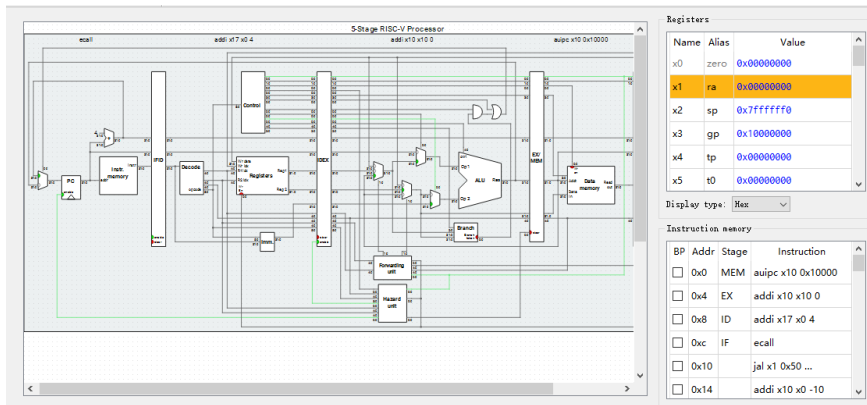


- right click on any port and press "show value" to display its label.
- If a port's value label has been made visible, it is possible to change the radix of the displayed value through right-clicking the port label.



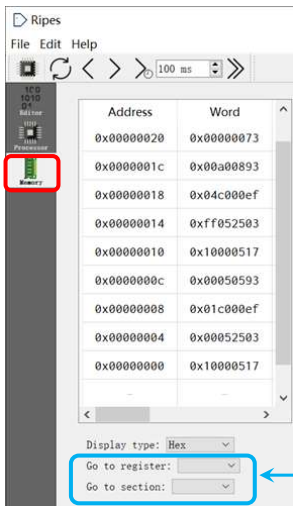
# 准备知识—Ripes软件简介

✓ 流水线CPU数据通路



# 准备知识—Ripes软件简介

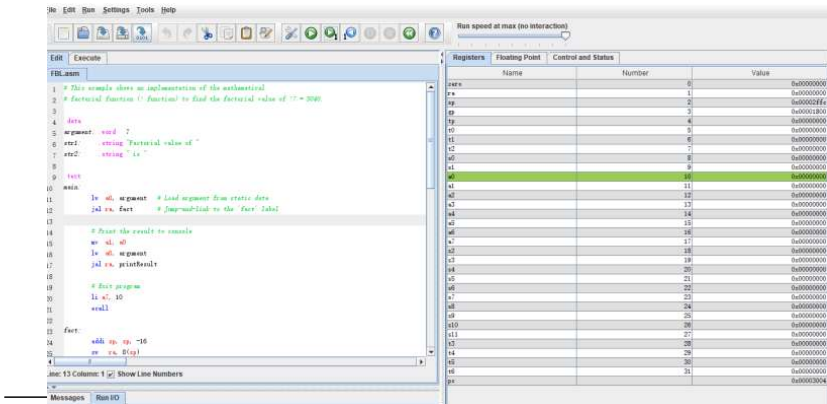
## ➤ 存储器选项卡



存储器定位

# 准备知识—Rars软件简介

1. RARS is written in Java and requires at least Release 1.8 of the Java SE Java Runtime Environment (JRE) to work. It is distributed as an executable JAR file.
2. 软件简介: <https://github.com/TheThirdOne/rars/issues>
3. 界面介绍



# 准备知识—Rars软件简介

## 4. 存储器设置

Settings>>Memory Configuration>>compact, data at Address0>>Apply and Close

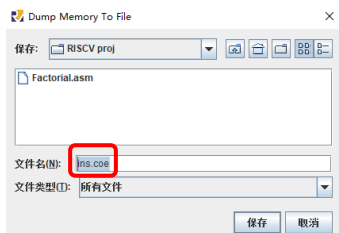
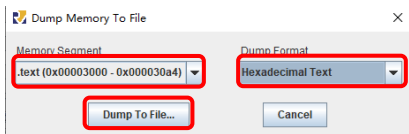
The screenshot displays the RARS MIPS simulator interface. The 'Text Segment' is visible, showing assembly code with addresses and instructions. The 'Data Segment' is also visible, showing memory addresses and values. The 'Registers' window is open, showing the state of various registers, including 'ra' (return address) and 'sp' (stack pointer). The 'sp' register is highlighted in green, indicating it is the current stack pointer. The 'ra' register is also highlighted in green, indicating it is the return address. The 'Registers' window shows the following registers and their values:

| Register | Value      |
|----------|------------|
| zero     | 0x00000000 |
| ra       | 0x00000000 |
| sp       | 0x00000000 |
| fp       | 0x00000000 |
| tp       | 0x00000000 |
| t0       | 0x00000000 |
| t1       | 0x00000000 |
| t2       | 0x00000000 |
| t3       | 0x00000000 |
| t4       | 0x00000000 |
| t5       | 0x00000000 |
| t6       | 0x00000000 |
| t7       | 0x00000000 |
| t8       | 0x00000000 |
| t9       | 0x00000000 |
| s0       | 0x00000000 |
| s1       | 0x00000000 |
| s2       | 0x00000000 |
| s3       | 0x00000000 |
| s4       | 0x00000000 |
| s5       | 0x00000000 |
| s6       | 0x00000000 |
| s7       | 0x00000000 |
| s8       | 0x00000000 |
| s9       | 0x00000000 |
| a0       | 0x00000000 |
| a1       | 0x00000000 |
| a2       | 0x00000000 |
| a3       | 0x00000000 |
| a4       | 0x00000000 |
| a5       | 0x00000000 |
| a6       | 0x00000000 |
| a7       | 0x00000000 |
| x2       | 0x00000000 |
| x3       | 0x00000000 |
| x4       | 0x00000000 |
| x5       | 0x00000000 |
| x6       | 0x00000000 |
| x7       | 0x00000000 |
| x8       | 0x00000000 |
| x9       | 0x00000000 |
| x10      | 0x00000000 |
| x11      | 0x00000000 |
| x12      | 0x00000000 |
| x13      | 0x00000000 |
| x14      | 0x00000000 |
| x15      | 0x00000000 |
| x16      | 0x00000000 |
| x17      | 0x00000000 |
| x18      | 0x00000000 |
| x19      | 0x00000000 |
| x20      | 0x00000000 |
| x21      | 0x00000000 |
| x22      | 0x00000000 |
| x23      | 0x00000000 |
| x24      | 0x00000000 |
| x25      | 0x00000000 |
| x26      | 0x00000000 |
| x27      | 0x00000000 |
| x28      | 0x00000000 |
| x29      | 0x00000000 |
| x30      | 0x00000000 |
| x31      | 0x00000000 |
| pc       | 0x00000000 |

# 准备知识—Rars软件简介

## 6.代码段段机器码(16进制)导出

File>>Dump Memory To File，按下图设置完毕后，选择文件保存路径，命名为ins.coe。

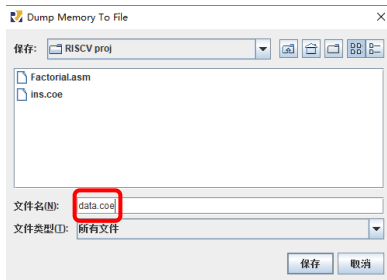
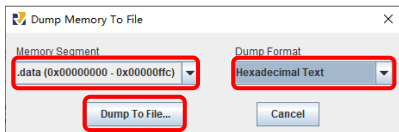




# 准备知识—Rars软件简介

## 7.数据段机器码(16进制)导出

File>>Dump Memory To File，按下图设置完毕后，选择文件保存路径，命名为data.coe。



8. 采用记事本分别打开生成的ins.coe和data.coe，在文档的最开始加上以下语句后保存：  
memory\_initialization\_radix = 16;  
memory\_initialization\_vector =

# 实验内容

## 1.理解并仿真RIPES示例汇编程序

加载Ripes示例汇编程序 (Console Printing)→选择单周期CPU数据通路→  
单步执行程序→观察数据通路控制信号和寄存器内容的变化

## 2.设计汇编程序，验证6条指令功能

Rars软件设计汇编程序→单步运行程序→人工检查→生成COE文件

- ✓ sw, lw
- ✓ add, addi
- ✓ beq, jal

备注：通过查看数据存储器 and 32个通用寄存器来实现人工检查

# 实验内容

示例:

.data

out: .word 0xff      #led, 初始全亮

in: .word 0                #switch

.text

la a0, out                #仿真需要

sw x0, 0(a0)              #test sw: 全灭led

addi t0, x0, 0xff      #test addi: 全亮led

sw t0, 0(a0)

lw t0, 4(a0)              #test lw: 由switch设置led

sw t0, 0(a0)

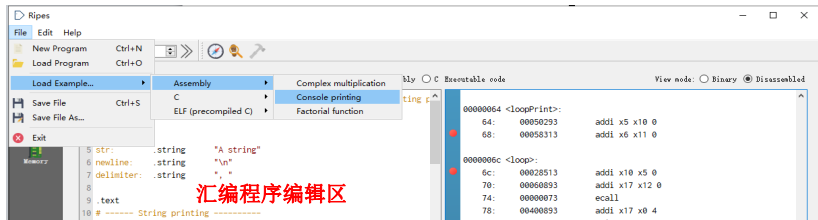
... ..

## 3. 设计汇编程序，计算斐波那契—卢卡斯数列

- 数列前两项：1，2
- 数据输出方式：数据存储器

# 实验步骤

## 1. 理解并仿真RIPES示例汇编程序 (Console Printing)



## 2. Rars软件设计汇编程序，实现人工检查6条指令功能，并生成COE文件

- ✓ sw, lw
- ✓ add, addi
- ✓ beq, jal

## 3. Rars软件设计汇编程序，实现计算斐波那契—卢卡斯数列（数列前两项为1，2），并生成COE文件

# The End

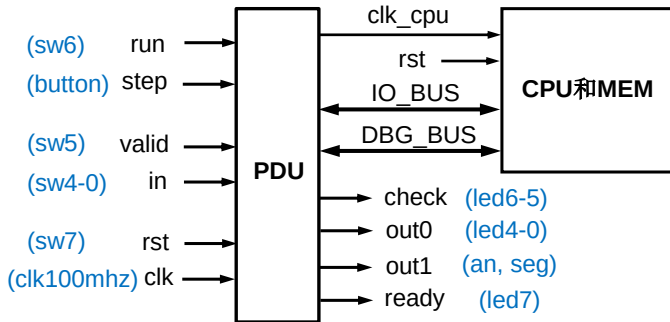
# 实验四 单周期CPU设计

# 实验目标

- 理解CPU的结构和工作原理
- 掌握单周期CPU的设计和调试方法
- 熟练掌握数据通路和控制器的设计和描述方法

# 实验内容

- 设计实现单周期RISC-V CPU，可执行以下**10条**指令
  - add, addi, **sub**, **auipc**, lw, sw, beq, **blt**, jal, **jalr**





# 运算指令

- `add rd, rs1, rs2`       $\# x[rd] = x[rs1] + x[rs2]$

| 31      | 25 24 | 20 19 | 15 14        | 12 11  | 7 6 | 0      |
|---------|-------|-------|--------------|--------|-----|--------|
| funct7  |       | rs2   | rs1          | funct3 | rd  | opcode |
| 7       | 5     | 5     | 3            | 5      | 7   |        |
| 0000000 | src2  | src1  | ADD/SLT/SLTU | dest   | OP  |        |
| 0000000 | src2  | src1  | AND/OR/XOR   | dest   | OP  |        |
| 0000000 | src2  | src1  | SLL/SRL      | dest   | OP  |        |
| 0100000 | src2  | src1  | SUB/SRA      | dest   | OP  |        |

- `addi rd, rs1, imm`       $\# x[rd] = x[rs1] + sext(imm)$

| 31                | 20 19 | 15 14         | 12 11  | 7 6    | 0      |
|-------------------|-------|---------------|--------|--------|--------|
| imm[11:0]         |       | rs1           | funct3 | rd     | opcode |
| 12                | 5     | 3             | 5      | 7      |        |
| I-immediate[11:0] | src   | ADDI/SLTI[U]  | dest   | OP-IMM |        |
| I-immediate[11:0] | src   | ANDI/ORI/XORI | dest   | OP-IMM |        |

# 运算指令

- `sub rd, rs1, rs2`       $\# x[rd] = x[rs1] - x[rs2]$

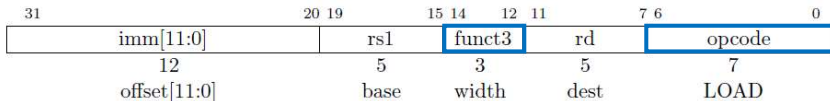
| 31      | 25 24 | 20 19 | 15 14        | 12 11  | 7 6 | 0      |
|---------|-------|-------|--------------|--------|-----|--------|
| funct7  |       | rs2   | rs1          | funct3 | rd  | opcode |
| 7       | 5     | 5     | 3            | 5      | 7   |        |
| 0000000 | src2  | src1  | ADD/SLT/SLTU | dest   | OP  |        |
| 0000000 | src2  | src1  | AND/OR/XOR   | dest   | OP  |        |
| 0000000 | src2  | src1  | SLL/SRL      | dest   | OP  |        |
| 0100000 | src2  | src1  | SUB/SRA      | dest   | OP  |        |

- `auipc rd, imm`       $\# x[rd] = pc + sext(imm[31:12] \ll 12)$

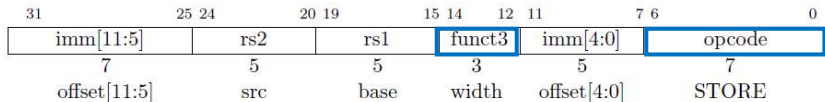
| 31                 | 12 11 | 7 6   | 0      |
|--------------------|-------|-------|--------|
| imm[31:12]         |       | rd    | opcode |
| 20                 | 5     | 7     |        |
| U-immediate[31:12] | dest  | LUI   |        |
| U-immediate[31:12] | dest  | AUIPC |        |

# 访存指令

- `lw rd, offset(rs1)`     $\# x[rd] = M[x[rs1] + sext(offset)]$

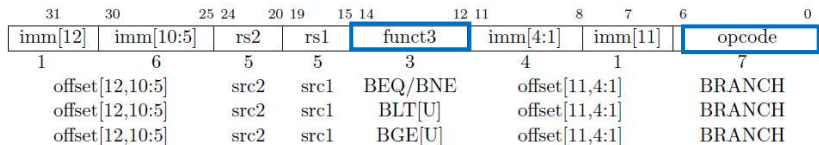


- `sw rs2, offset(rs1)`     $\# M[x[rs1] + sext(offset)] = x[rs2]$



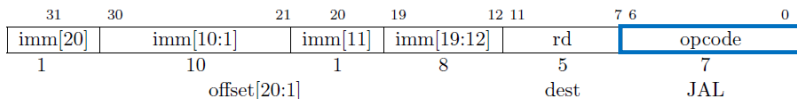
# 分支指令

- beq rs1, rs2, offset # if (rs1 == rs2) pc += sext(offset)
- blt rs1, rs2, offset # if (rs1 <<sub>s</sub> rs2) pc += sext(offset)

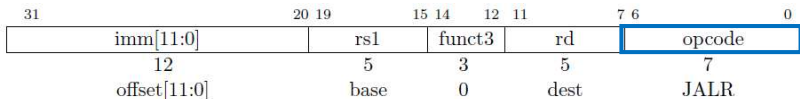


# 跳转指令

- jal rd, offset      #  $x[\text{rd}] = \text{pc} + 4; \text{pc} += \text{sext}(\text{offset})$



- jalr rd, offset(rs1)    #  $t = \text{pc} + 4;$   
 $\text{pc} = (x[\text{rs1}] + \text{sext}(\text{offset})) \& \sim 1; x[\text{rd}] = t$



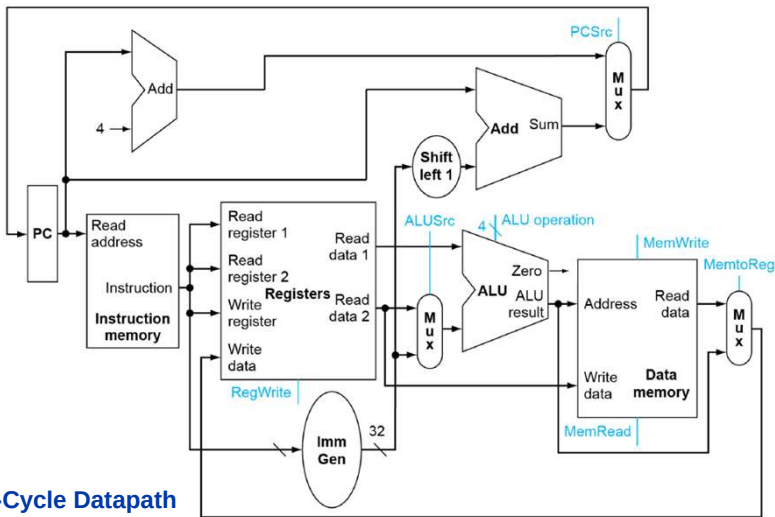
# RV32I 指令编码

|           |       |     |     |    |         |       |
|-----------|-------|-----|-----|----|---------|-------|
| imm[11:0] |       | rs1 | 000 | rd | 0010011 | ADDI  |
| imm[11:0] |       | rs1 | 010 | rd | 0010011 | SLTI  |
| imm[11:0] |       | rs1 | 011 | rd | 0010011 | SLTIU |
| imm[11:0] |       | rs1 | 100 | rd | 0010011 | XORI  |
| imm[11:0] |       | rs1 | 110 | rd | 0010011 | ORI   |
| imm[11:0] |       | rs1 | 111 | rd | 0010011 | ANDI  |
| 0000000   | shamt | rs1 | 001 | rd | 0010011 | LLI   |
| 0000000   | shamt | rs1 | 101 | rd | 0010011 | SRLI  |
| 0100000   | shamt | rs1 | 101 | rd | 0010011 | SRAI  |
| 0000000   | rs2   | rs1 | 000 | rd | 0110011 | ADD   |
| 0100000   | rs2   | rs1 | 000 | rd | 0110011 | SUB   |
| 0000000   | rs2   | rs1 | 001 | rd | 0110011 | SLL   |
| 0000000   | rs2   | rs1 | 010 | rd | 0110011 | SLT   |
| 0000000   | rs2   | rs1 | 011 | rd | 0110011 | SLTU  |
| 0000000   | rs2   | rs1 | 100 | rd | 0110011 | XOR   |
| 0000000   | rs2   | rs1 | 101 | rd | 0110011 | SRL   |
| 0100000   | rs2   | rs1 | 101 | rd | 0110011 | SRA   |
| 0000000   | rs2   | rs1 | 110 | rd | 0110011 | OR    |
| 0000000   | rs2   | rs1 | 111 | rd | 0110011 | AND   |

# RV32I 指令编码

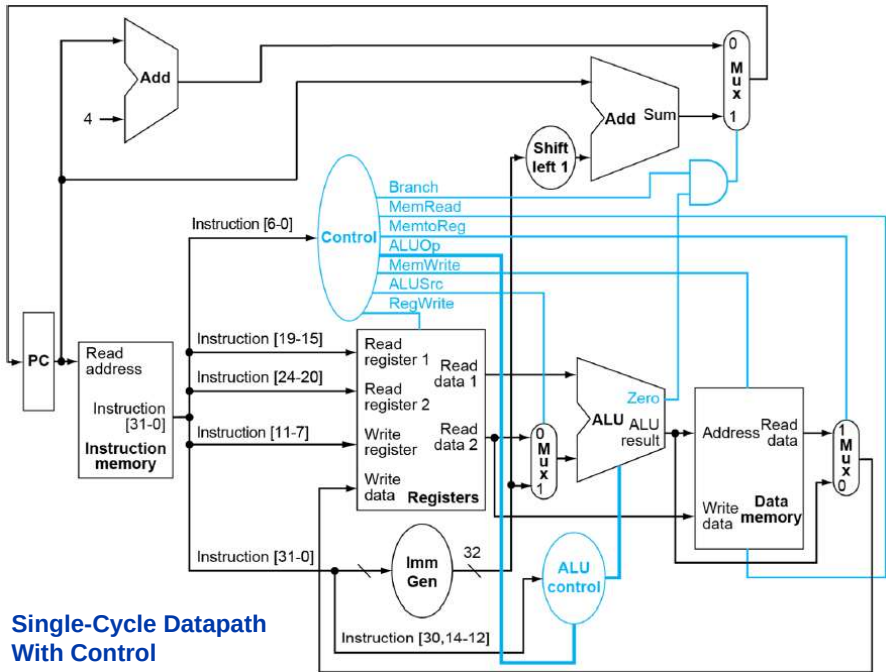
|                       |     |     |     |             |         |       |
|-----------------------|-----|-----|-----|-------------|---------|-------|
| imm[31:12]            |     |     |     | rd          | 0110111 | LUI   |
| imm[31:12]            |     |     |     | rd          | 0010111 | AUIPC |
| imm[20 10:1 11 19:12] |     |     |     | rd          | 1101111 | JAL   |
| imm[11:0]             |     | rs1 | 000 | rd          | 1100111 | JALR  |
| imm[12 10:5]          | rs2 | rs1 | 000 | imm[4:1 11] | 1100011 | BEQ   |
| imm[12 10:5]          | rs2 | rs1 | 001 | imm[4:1 11] | 1100011 | BNE   |
| imm[12 10:5]          | rs2 | rs1 | 100 | imm[4:1 11] | 1100011 | BLT   |
| imm[12 10:5]          | rs2 | rs1 | 101 | imm[4:1 11] | 1100011 | BGE   |
| imm[12 10:5]          | rs2 | rs1 | 110 | imm[4:1 11] | 1100011 | BLTU  |
| imm[12 10:5]          | rs2 | rs1 | 111 | imm[4:1 11] | 1100011 | BGEU  |
| imm[11:0]             |     | rs1 | 000 | rd          | 0000011 | LB    |
| imm[11:0]             |     | rs1 | 001 | rd          | 0000011 | LH    |
| imm[11:0]             |     | rs1 | 010 | rd          | 0000011 | LW    |
| imm[11:0]             |     | rs1 | 100 | rd          | 0000011 | LBU   |
| imm[11:0]             |     | rs1 | 101 | rd          | 0000011 | LHU   |
| imm[11:5]             | rs2 | rs1 | 000 | imm[4:0]    | 0100011 | SB    |
| imm[11:5]             | rs2 | rs1 | 001 | imm[4:0]    | 0100011 | SH    |
| imm[11:5]             | rs2 | rs1 | 010 | imm[4:0]    | 0100011 | SW    |

# 单周期CPU数据通路



Single-Cycle Datapath

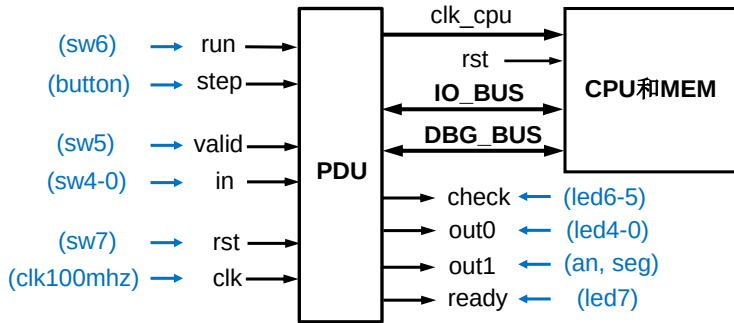




# 处理器调试单元

- **PDU (Processor Debug Unit)**

- 控制CPU的运行方式: **run = 1** 连续运行, **0** 单步运行
- 管理外设 (开关sw、指示灯led、数码管an & seg)
- 显示运行结果和数据通路状态



# IO\_BUS信号

- CPU运行时访问开关(sw)、指示灯(led)和数码管(an, seg)
  - io\_addr: I/O外设的地址
  - io\_din: CPU接收来自输入缓冲寄存器 (IBR) 的sw输入数据
  - io\_dout: CPU向led和seg输出的数据
  - io\_we: CPU向led和seg输出时的使能信号, 利用该信号将io\_dout存入输出缓冲寄存器 (OBR), 再经数码管显示电路将其显示在数码管 (an, seg)

# DBG\_BUS信号

- 调试时将存储器和寄存器堆内容，以及CPU数据通路状态信息导出显示
  - m\_rf\_addr: 存储器(MEM)或寄存器堆(RF)的调试读口地址
  - rf\_data: 从RF读取的数据
  - m\_data: 从MEM读取的数据
  - pc: PC的内容

# CPU运行方式

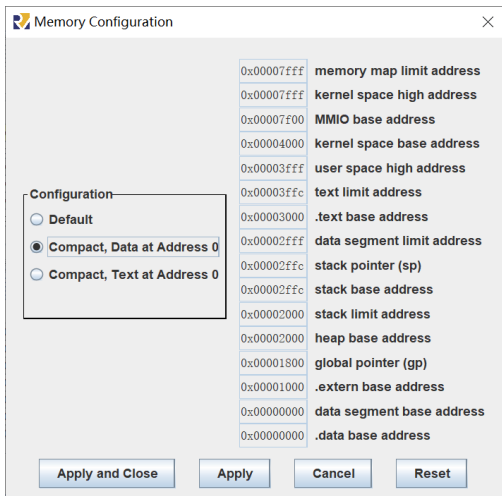
- **run = 1: 连续运行**
  - PDU向CPU输出连续时钟信号clk\_cpu
  - CPU通过I/O\_BUS访问外设
  - 输入端口: in, valid
  - 输出端口: out0, out1, ready
- **run = 0: 单步运行（每次执行一条指令）**
  - 每按动step一次, PDU产生一个周期的clk\_cpu
  - 执行外设输入指令前, 应先设置好valid或in后, 再按动step
  - 执行任何指令后, led和数码管(an, seg)显示当前程序运行结果
  - 随后可以通过改变valid和in查看寄存器堆、存储器和PC的内容

# 外设使用说明表

| sw  |     |       |         | button | led   |       |         | an/seg   | 说明     |
|-----|-----|-------|---------|--------|-------|-------|---------|----------|--------|
| 7   | 6   | 5     | 4~0     |        | 7     | 6~5   | 4~0     |          |        |
| rst | run | valid | in      | step   | ready | check | out0    | out1     |        |
| ↑   | -   | -     | -       | -      | 1     | 00    | 0x1f    | 0x12...8 | 复位     |
| x   | 1   | valid | in      | -      | ready | 00    | out0    | out1     | 连续运行   |
|     | 0   | valid | in      | ↑      | ready | 00    | out0    | out1     | 单步运行   |
|     |     | ↑↓    | addr_rf | x      | 0     | 01    | addr_rf | data_rf  | 查看寄存器堆 |
|     |     |       | addr_m  |        | 0     | 10    | addr_m  | data_m   | 查看存储器  |
|     |     |       | -       |        | 0     | 11    | 0       | PC       | 查看PC   |

# RARS存储器配置

- 紧凑且数据地址为0
  - 0x0000\_0000 ~ 0x0000\_2ffff
- 代码地址：
  - 0x0000\_3000 ~ 0x0000\_3ffc



# 实验存储器配置

- 指令存储器(256x32bit)地址: 0x0000\_3000 ~ 0x0000\_33ff
- 数据存储器(256x32bit)地址: 0x0000\_0000 ~ 0x0000\_03ff
- 外设端口地址: 0x0000\_0400 ~ 0x0000\_07ff

存储器映射外设端口地址表

| 存储器地址       | I/O_addr | 输出端口名 | 输入端口名 | 外设      |
|-------------|----------|-------|-------|---------|
| 0x0000_0400 | 0        | out0  | -     | led4-0  |
| 0x0000_0404 | 1        | ready | -     | led7    |
| 0x0000_0408 | 2        | out1  | -     | an, seg |
| 0x0000_040C | 3        | -     | in    | sw4-0   |
| 0x0000_0410 | 4        | -     | valid | sw5     |



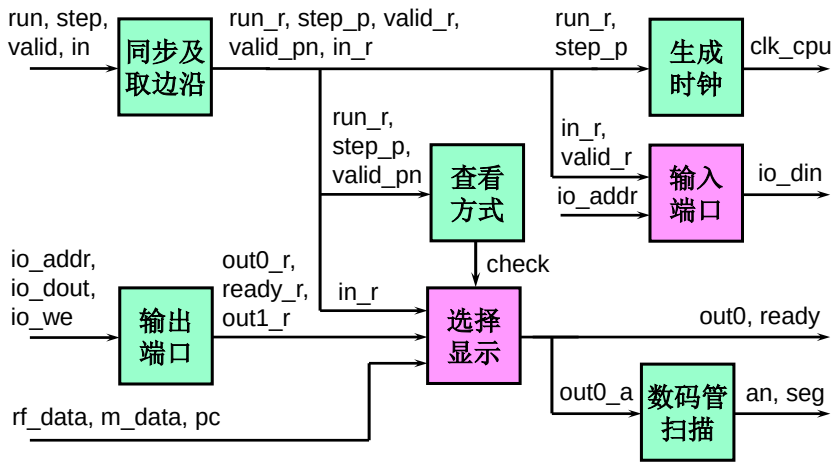
# CPU模块接口

```
module cpu (  
    input clk,  
    input rst,  
  
    //IO_BUS  
    output [7:0] io_addr,    //led和seg的地址  
    output [31:0] io_dout,   //输出led和seg的数据  
    output io_we,           //输出led和seg数据时的使能信号  
    input [31:0] io_din,     //来自sw的输入数据  
  
    //Debug_BUS  
    input [7:0] m_rf_addr,   //存储器(MEM)或寄存器堆(RF)的调试读口地址  
    output [31:0] rf_data,   //从RF读取的数据  
    output [31:0] m_data,    //从MEM读取的数据  
    output [31:0] pc         //PC的内容  
);
```

# PDU模块接口

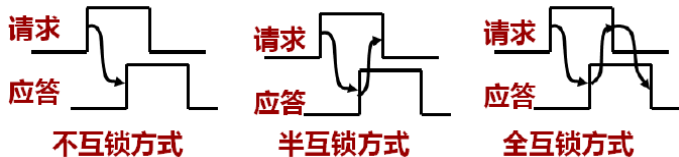
|                                                                                                                                                                                                                                                                                                                                     |                                                                                                                                                                                                                                                                                                                                                                                       |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>module pdu (<br/>    input clk,<br/>    input rst,<br/><br/>    //选择CPU工作方式<br/>    input run,<br/>    input step,<br/>    output clk_cpu,<br/><br/>    //输入sw的端口<br/>    input valid,<br/>    input [4:0] in,<br/><br/>    //输出led和seg的端口<br/>    output [1:0] check, //led6-5:查看类型<br/>    output [4:0] out0, //led4-0</pre> | <pre>    output [2:0] an, //8个数码管<br/>    output [3:0] seg,<br/>    output ready, //led7<br/><br/>    //IO_BUS<br/>    input [7:0] io_addr,<br/>    input [31:0] io_dout,<br/>    input io_we,<br/>    output [31:0] io_din,<br/><br/>    //Debug_BUS<br/>    output [7:0] m_rf_addr,<br/>    input [31:0] rf_data,<br/>    input [31:0] m_data,<br/>    input [31:0] pc<br/>);</pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

# PDU逻辑结构图

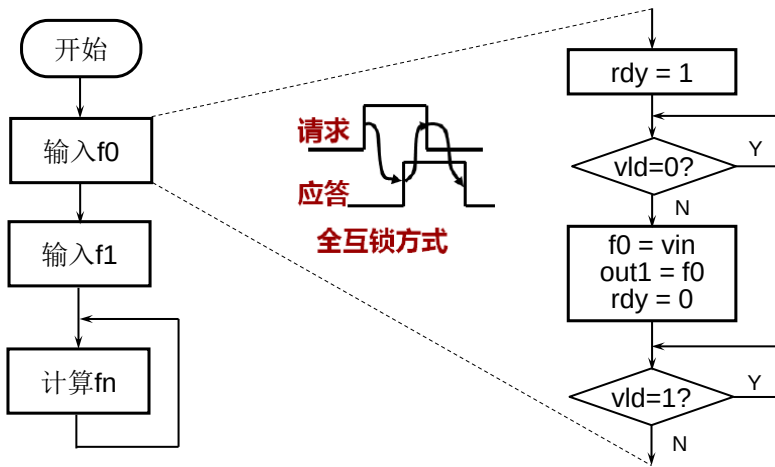


# 应用程序测试（选做）

- 设计汇编程序：计算斐波那契—卢卡斯数列
  - 依次输入数列开始两项（设置in后按动valid确认），并输出至数码管（out1）上显示
  - 计算数列后续项，并输出至数码管上显示，按动valid继续
- 通过握手信号，协调CPU和外设的数据传输过程
  - ready: CPU准备好，指示外设提供数据
  - valid: 外设指示数据（in）有效



# 应用程序流程图（选做）



# 实验步骤

1. 修改Lab2寄存器堆模块，增加1个用于调试的读端口，且使其r0内容恒定为0
2. 结构化描述单周期CPU，并进行功能仿真
  - 假定指令存储器和数据存储器（增加一个读端口用于调试）均使用分布式存储器，容量均为256x32位，使用Lab3实验内容2生成的COE文件(将测试指令由6条增加至10条)初始化
3. 将CPU和PDU下载至FPGA中测试，使用Lab3实验内容3生成的COE文件对指令存储器和数据存储器初始化

# The End

# 计算机组成原理

## 实验五 流水线CPU设计

2022春季

**zjx@ustc.edu.cn**

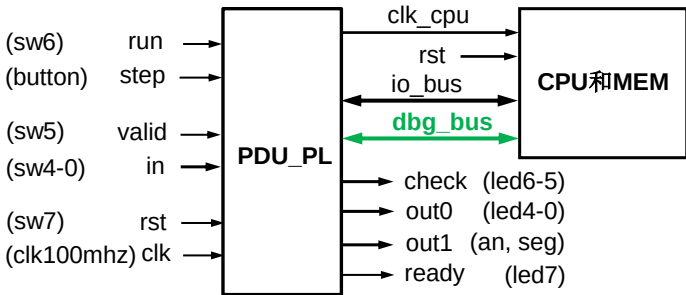


# 实验目标

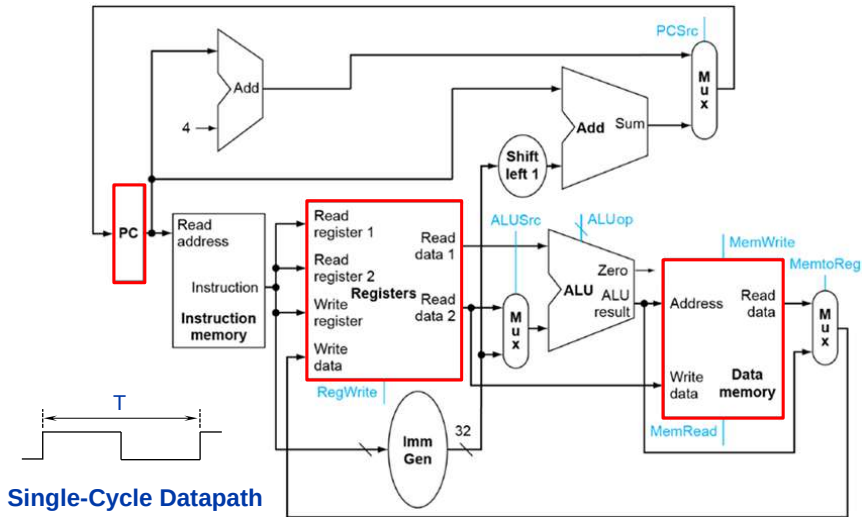
- 理解流水线CPU的结构和工作原理
- 掌握流水线CPU的设计和调试方法，特别是流水线中数据相关和控制相关的处理
- 熟练掌握数据通路和控制器的设计和描述方法

# 实验内容

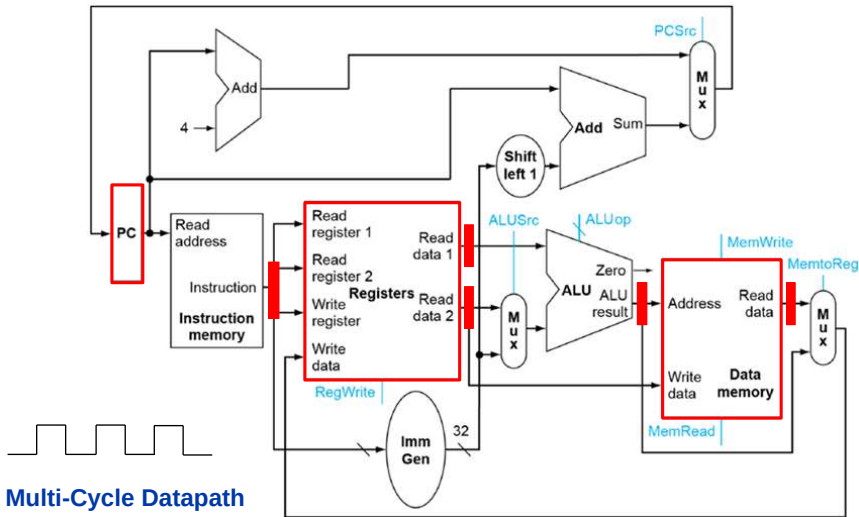
- 设计实现5级流水线的RISC-V CPU
  - 能够执行6条指令: **add, addi, lw, sw, beq, jal**
  - 指令存储器和数据存储器均使用分布式存储器, 容量均为256x32位, 数据存储器地址为0x0000\_0000 ~ 0x0000\_2fff, 指令存储器地址为0x0000\_3000 ~ 0x0000\_3ffc。
  - 寄存器堆和数据存储器均增加一个读端口用于调试



# 单周期CPU数据通路

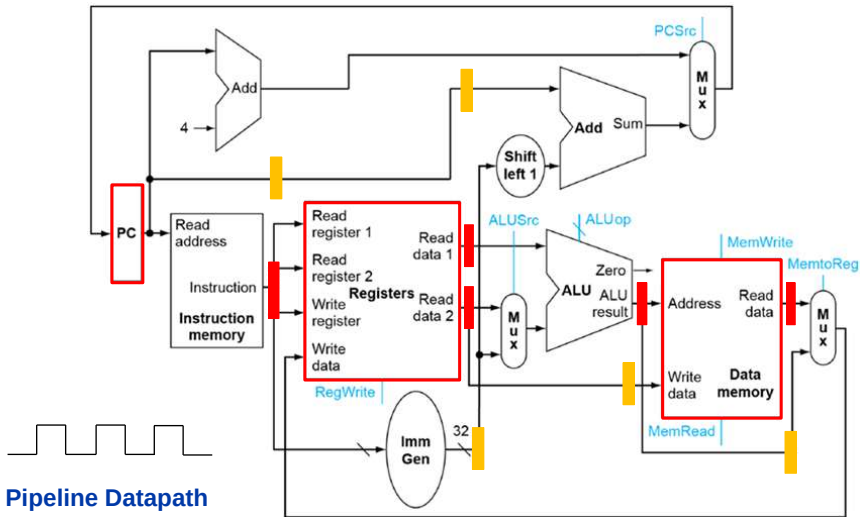


# 多周期CPU数据通路

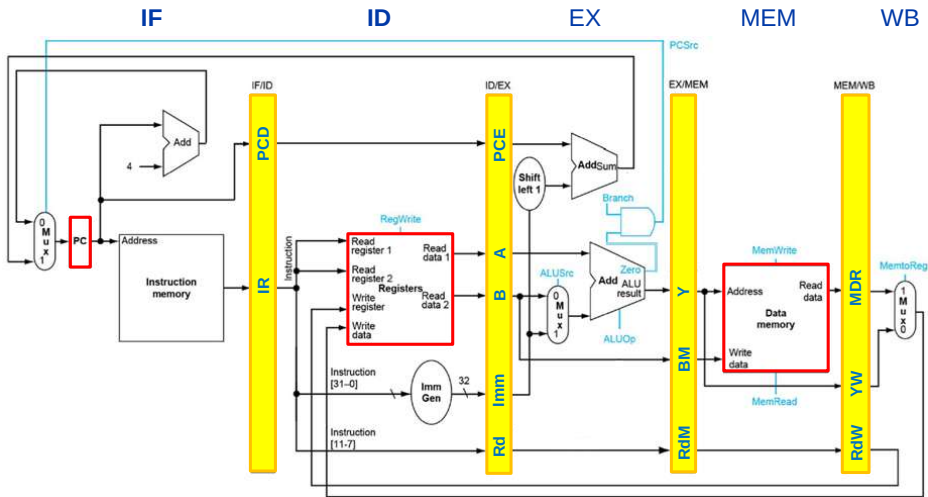


Multi-Cycle Datapath

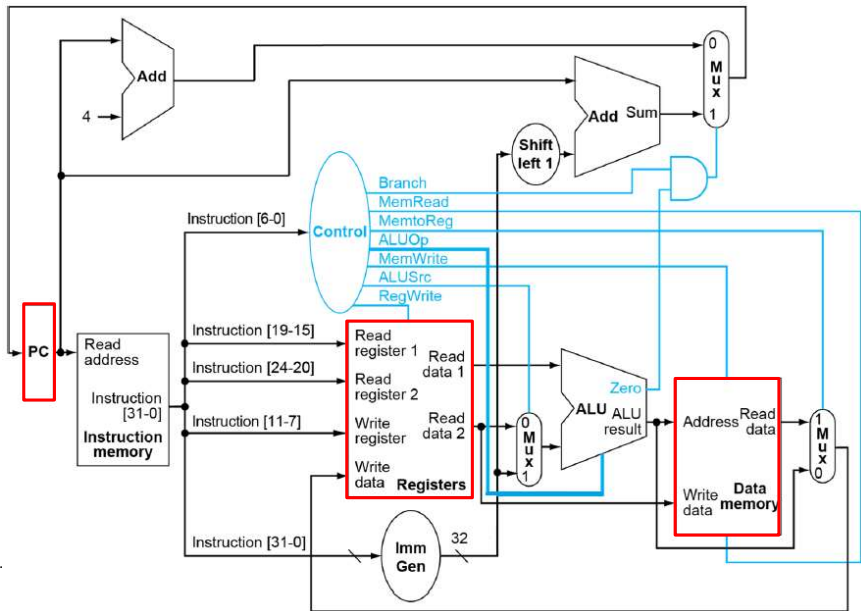
# 流水线CPU数据通路



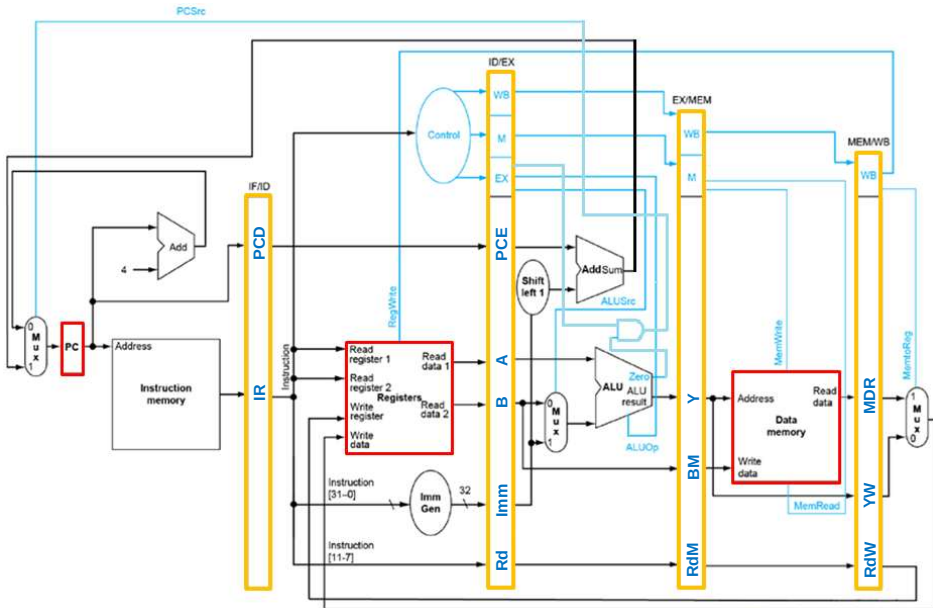
# 流水线CPU数据通路



# 单周期CPU数据通路+控制器



# 流水线CPU数据通路+控制器

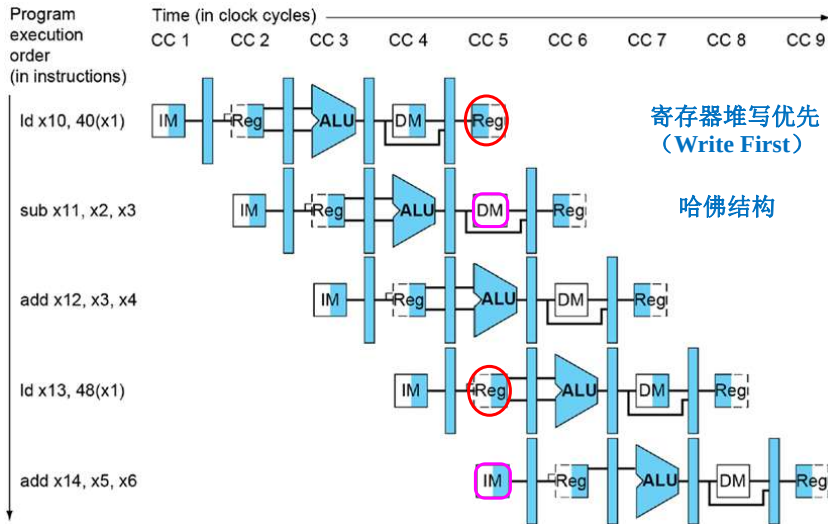




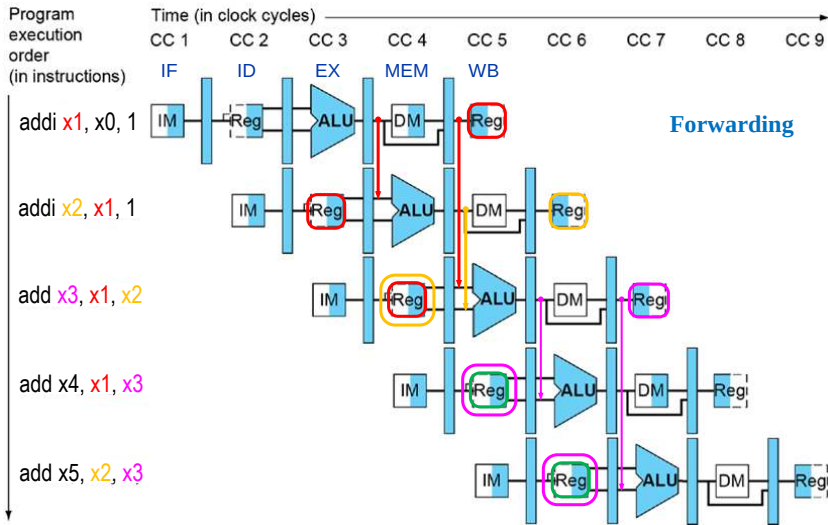
# 流水线相关及其处理

- **结构相关：当多条指令执行时竞争使用同一资源时**
  - 存储器相关处理：哈佛结构（指令和数据存储器分开）
  - 寄存器堆相关处理：同一寄存器读写时，写优先（Write First）
- **数据相关：当一条指令需要等待前面指令的执行结果时**
  - 数据定向（Forwarding）：将执行结果提前传递至之前流水段
  - 加载-使用相关（Load-use hazard）：阻止紧随Load已进入流水线的指令流动（Stall），向后续流水段插入空操作（Bubble）
- **控制相关：当遇到转移指令且不能继续顺序执行时**
  - 清除（Flush）紧随转移指令已进入流水线的指令
  - 从转移目标处取指令后执行

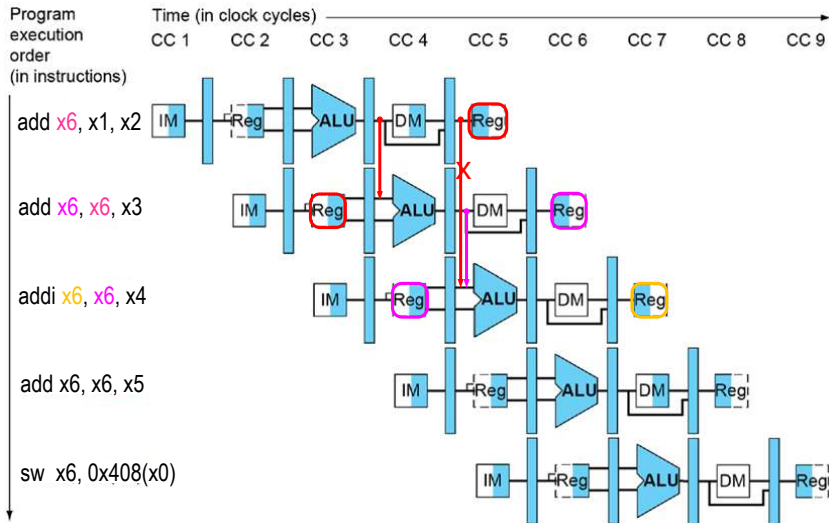
# 流水线相关：结构相关



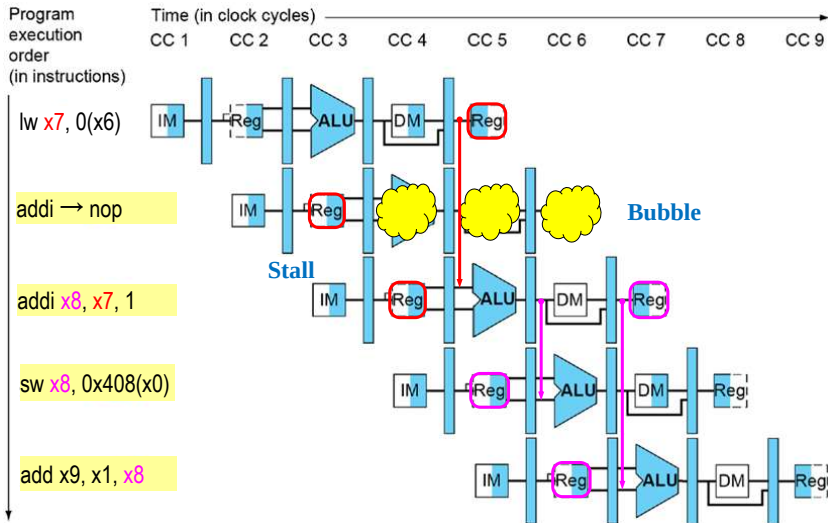
# 流水线相关：数据相关



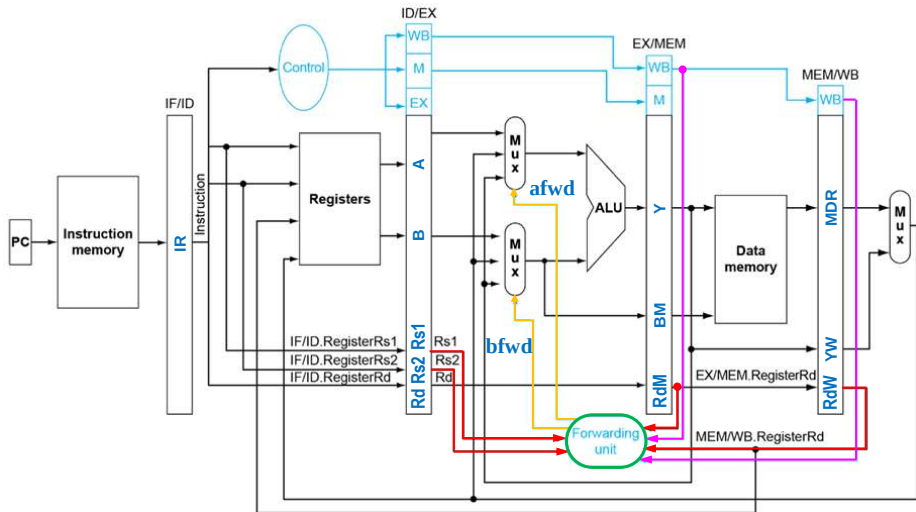
# 流水线相关：数据相关 (续1)



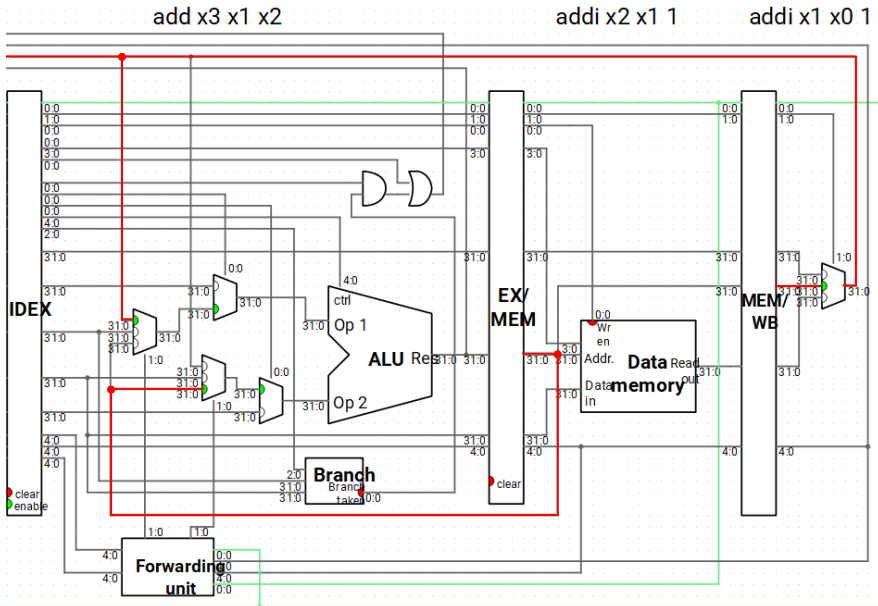
# 数据相关 (续2)



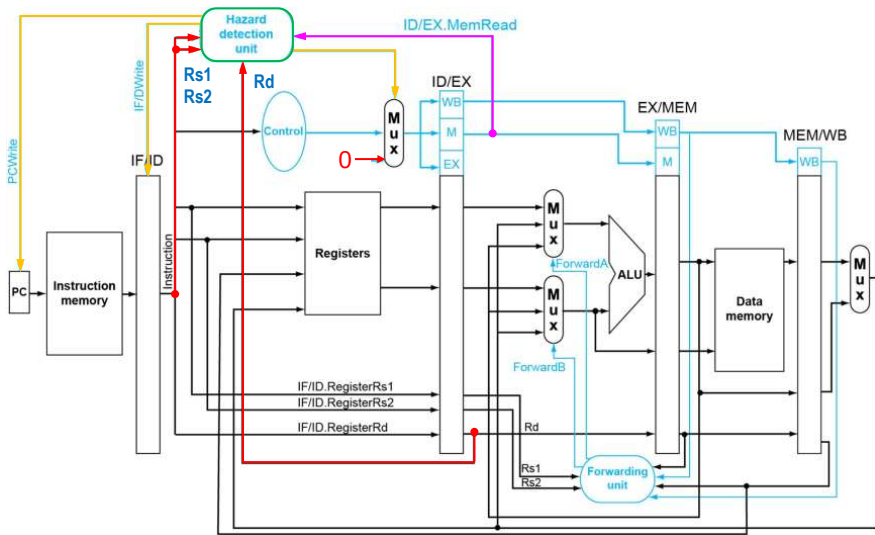
# Forwarding



# Forwarding: Ripes

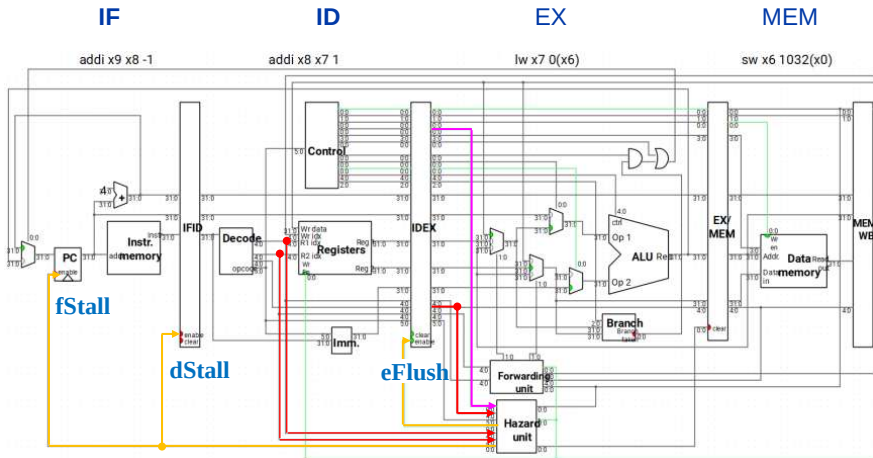


# Load-Use Hazard

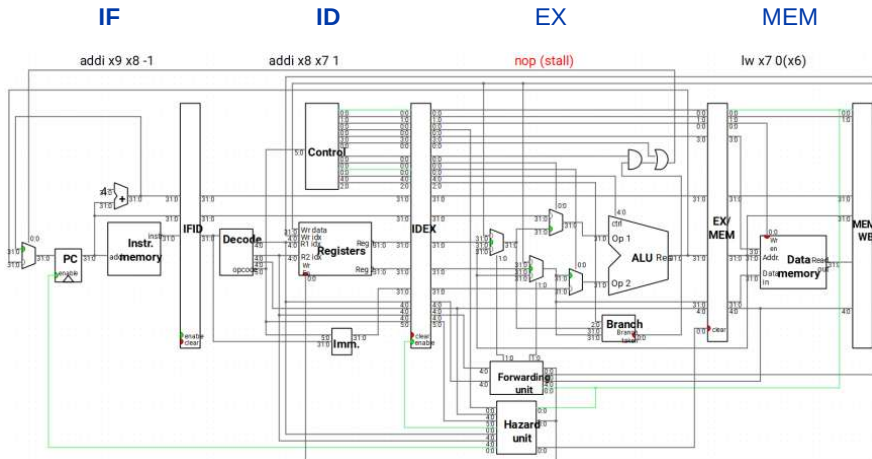




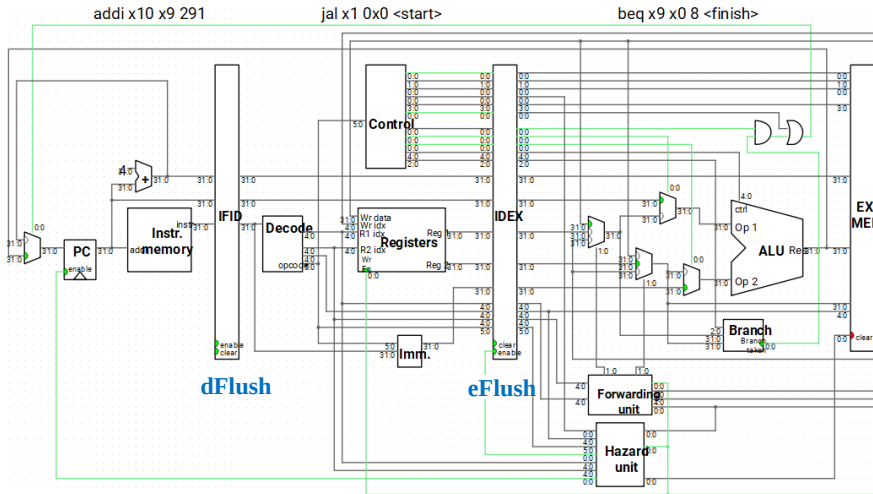
# Load-Use Hazard: Ripes



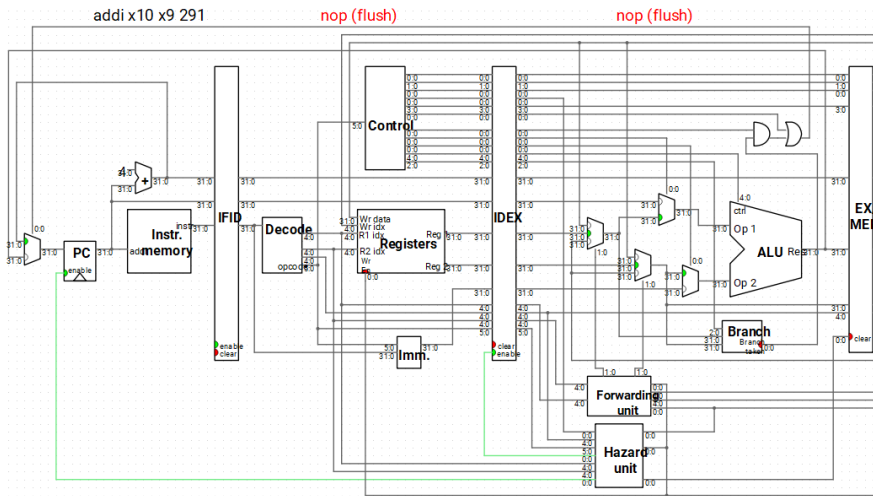
# Load-Use Hazard: Ripes



# Branch Hazard: Ripes

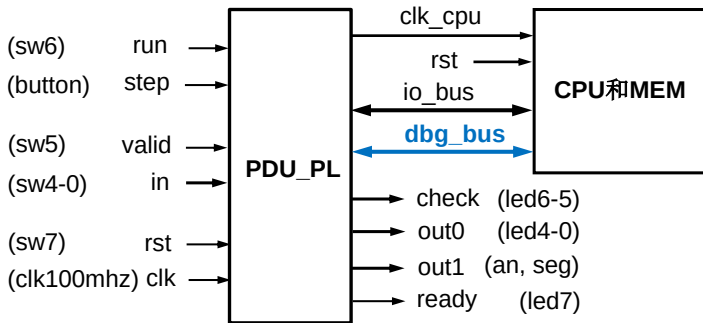


# Branch Hazard: Ripes



# 流水线处理器调试单元

- **PDU\_PL (Processor Debug Unit for Pipe-Line)**
  - 控制CPU的运行方式: **run = 1** 连续运行, **0** 单步运行
  - 管理外设 (开关sw、指示灯led、数码管seg), 显示CPU运行结果和数据通路状态



# IO\_BUS信号

- CPU运行时访问外设的总线（地址、数据和控制）信号
  - io\_addr: I/O外设的地址
  - io\_din: CPU接收来自外设sw的输入数据
  - io\_dout: CPU向外设led和seg输出的数据
  - io\_we: CPU向外设led和seg输出时的使能信号

存储器映射外设的端口地址表

| 存储器地址       | I/O_addr | 输出端口名 | 输入端口名 | 外设      |
|-------------|----------|-------|-------|---------|
| 0x0000_0400 | 0        | out0  | -     | led4-0  |
| 0x0000_0404 | 1        | ready | -     | led7    |
| 0x0000_0408 | 2        | out1  | -     | an, seg |
| 0x0000_040C | 3        | -     | in    | sw4-0   |
| 0x0000_0410 | 4        | -     | valid | sw5     |

# DBG\_BUS信号

- 调试时将寄存器堆和数据存储器内容，以及流水段寄存器内容导出显示
  - m\_rf\_addr: 存储器(MEM)或寄存器堆(RF)的调试读口地址
  - rf\_data: 从RF读出的数据
  - m\_data: 从MEM读出的数据
  - 流水段寄存器
    - PC/IF/ID: pcin, pc, pcd, ir
    - ID/EX: pce, a, b, imm, rd, ctrl
    - EX/MEM: y, bm, rdm, ctrlm
    - MEM/WB: yw, mdr, rdw, ctrlw

**Ctrl:**

|        |        |        |        |    |    |       |    |    |    |       |    |    |       |        |    |
|--------|--------|--------|--------|----|----|-------|----|----|----|-------|----|----|-------|--------|----|
| 31     | 30     | 29     | 28     | 27 | 26 | 25    | 24 | 23 | 22 | 21    | 20 | 19 | 18    | 17     | 16 |
| fstall | dstall | dflush | eflush | 0  | 0  | a_fwd |    | 0  | 0  | b_fwd |    | 0  | rf_wr | wb_sel |    |

|    |    |      |      |    |    |     |    |   |   |       |       |        |   |   |   |
|----|----|------|------|----|----|-----|----|---|---|-------|-------|--------|---|---|---|
| 15 | 14 | 13   | 12   | 11 | 10 | 9   | 8  | 7 | 6 | 5     | 4     | 3      | 2 | 1 | 0 |
| 0  | 0  | m_rd | m_wr | 0  | 0  | jal | br | 0 | 0 | a_sel | b_sel | alu_op |   |   |   |

# CPU运行方式

- **run = 1: 连续运行**
  - PDU向CPU输出连续时钟信号clk\_cpu
  - CPU通过I/O\_BUS访问外设
  - 输入端口: in, valid
  - 输出端口: out0, out1, ready
- **run = 0: 单步运行（每次执行一条指令）**
  - 每按动step一次, PDU产生一个周期的clk\_cpu
  - 执行外设输入指令前, 应先设置好valid或in后, 再按动step
  - 执行任何指令后, led和数码管(an, seg)显示当前程序运行结果
  - 随后可以通过改变valid和in查看寄存器堆、存储器和**流水段寄存器(Pipe-Line Register, PLR)**的内容



# 外设使用说明表

- 利用vld和in选择需要查看的信息，chk指示out0和out1上显示信息的类型

| sw  |     |     |      |     |      | btn  | led |     |       | seg      | 说明       |
|-----|-----|-----|------|-----|------|------|-----|-----|-------|----------|----------|
| 7   | 6   | 5   | 4~2  | 1   | 0    |      | 7   | 6~5 | 4~0   |          |          |
| rst | run | vld | in   |     |      | step | rdy | chk | out0  | out1     |          |
|     |     |     | ah_m | pre | next |      |     |     |       |          |          |
| ↑   | -   | -   | -    | -   | -    | -    | 1   | 00  | 0x1F  | 0x12..78 | 复位       |
| x   | 1   | vld | in   |     |      | -    | rdy | 00  | out0  | out1     | 连续运行     |
|     | 0   | vld | in   |     |      | ↑    | rdy | 00  | out0  | out1     | 单步运行     |
|     |     | ↑↓  | -    | ↑↓  | ↑↓   | x    | rdy | 01  | a_rf  | d_rf     | 查看寄存器堆   |
|     |     |     | ah_m |     |      |      |     | 10  | al_m  | d_m      | 查看存储器    |
|     |     |     | -    |     |      |      |     | 11  | a_plr | d_plr    | 查看流水段寄存器 |

# 查看寄存器堆和数据存储器

- 寄存器堆

- 利用vld (sw5)切换chk (led6~5) = 01
- 使用pre (sw1)和next (sw0)修改寄存器堆调试读端口地址a\_rf (led4~0), out1(seg)显示读出的数据

- 数据存储器

- 利用vld (sw5)切换chk (led6~5) = 10
- 通过sw4~2设置数据存储器调试读端口高位地址ah\_m (sw4~2)
- 使用pre (sw1)和next (sw0)修改数据存储器调试读端口低位地址al\_m (led4~0), out1(seg)显示读出的数据

# 查看流水段寄存器

- 利用vld (sw5)切换chk (led6~5) = 11
- 使用pre (sw1)和next (sw0)分别修改流水段寄存器地址ah\_plr和al\_plr, out1 (seg)显示对应寄存器数据(d\_plr)

| ah_plr<br>(led4~3) | al_plr<br>(led2~0) | d_plr<br>(seg) | ah_plr<br>(led4~3) | al_plr<br>(led2~0) | d_plr<br>(seg) |
|--------------------|--------------------|----------------|--------------------|--------------------|----------------|
| 00<br>(PC/IF/ID)   | x00                | pc             | 10<br>(EX/MEM)     | x00                | y              |
|                    | x01                | pcd            |                    | x01                | bm             |
|                    | x10                | ir             |                    | x10                | rdm            |
|                    | x11                | pcin           |                    | x11                | ctrlm          |
| 01<br>(ID/EX)      | 000                | pce            | 11<br>(MEM/WB)     | x00                | yw             |
|                    | 001                | a              |                    | x01                | mdr            |
|                    | 010                | b              |                    | x10                | rdw            |
|                    | 011                | imm            |                    | x11                | ctrlw          |
|                    | 100                | rd             |                    |                    |                |
|                    | 101                | ctrl           |                    |                    |                |

# CPU\_PL模块接口

```
module cpu_pl (  
    input clk,  
    input rst,  
  
    //IO_BUS  
    output [7:0] io_addr,    //led和seg的地址  
    output [31:0] io_dout,   //输出led和seg的数据  
    output io_we,           //输出led和seg数据时的使能信号  
    input [31:0] io_din,     //来自sw的输入数据  
  
    //Debug_BUS  
    input [7:0] m_rf_addr,   //存储器(MEM)或寄存器堆(RF)的调试读口地址  
    output [31:0] rf_data,   //从RF读取的数据  
    output [31:0] m_data,    //从MEM读取的数据  
  
    //PC/IF/ID 流水段寄存器
```

# CPU\_PL模块接口(续)

```
output [31:0] pc,  
output [31:0] pcd,  
output [31:0] ir,  
output [31:0] pcin,
```

// ID/EX 流水段寄存器

```
output [31:0] pce,  
output [31:0] a,  
output [31:0] b,  
output [31:0] imm,  
output [4:0] rd,  
output [31:0] ctrl,
```

// EX/MEM 流水段寄存器

```
output [31:0] y,
```

```
output [31:0] bm,  
output [4:0] rdm,  
output [31:0] ctrlm,
```

// MEM/WB 流水段寄存器

```
output [31:0] yw,  
output [31:0] mdr,  
output [4:0] rdw,  
output [31:0] ctrlw  
);
```

# PDU\_PL模块接口

|                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                               |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>module pdu_pl (<br/>    input clk,<br/>    input rst,<br/><br/>    //选择CPU工作方式<br/>    input run,<br/>    input step,<br/>    output clk_cpu,<br/><br/>    //输入sw的端口<br/>    input valid,<br/>    input [4:0] in,<br/><br/>    //输出led和seg的端口<br/>    output [1:0] check, //led6-5:查看类型<br/>    output [4:0] out0,  //led4-0</pre> | <pre>    output [2:0] an,    //8个数码管<br/>    output [3:0] seg,<br/>    output ready,      //led7<br/><br/>    //IO_BUS<br/>    input [7:0] io_addr,<br/>    input [31:0] io_dout,<br/>    input io_we,<br/>    output [31:0] io_din,<br/><br/>    //Debug_BUS<br/>    output [7:0] m_rf_addr,<br/>    input [31:0] rf_data,<br/>    input [31:0] m_data,<br/><br/>    //PC/IF/ID 流水段寄存器</pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

# PDU\_PL模块接口(续)

```
input [31:0] pc,  
input [31:0] pcd,      //pc_id  
input [31:0] ir,       //inst_id  
input [31:0] pcin,  
  
//ID/EX 流水段寄存器  
input [31:0] pce,      //pc_ex  
input [31:0] a,        //rf_rdata1_ex  
input [31:0] b,        //rf_rdata2_ex  
input [31:0] imm,      //imm_gen_out_ex  
input [4:0] rd,        //rf_waddr_ex  
input [31:0] ctrl,  
  
//EX/MEM 流水段寄存器  
input [31:0] y,        //alu_out_mem
```

```
input [31:0] bm,       // reg_b_mem  
input [4:0] rdm,       // rf_waddr_mem  
input [31:0] ctrlm,  
  
//MEM/WB 流水段寄存器  
input [31:0] yw,       //alu_out_wb  
input [31:0] mdr,      //Mem_rdata_wb  
input [4:0] rdw,       //rf_waddr_wb  
input [31:0] ctrlw  
);
```

# 实验步骤

1. 修改Lab4寄存器堆模块，使其满足写优先(Write First)，即在对同一寄存器读写时，写数据可立即从读数据输出

2.设计完整的流水线CPU：

- 1) 设计无数据和控制相关处理的流水线CPU，将CPU和PDU连接，加载no\_hazard\_test.coe至指令存储器（自测）。
- 2) 设计仅有数据相关处理的流水线CPU，将CPU和PDU连接，加载data\_hazard\_test.coe至指令存储器（自测）。
- 3) 设计有数据和控制相关处理的流水线CPU，将CPU和PDU连接，分别加载hazard\_test.coe 和fib\_test.coe至指令存储器（最终检查）。



# The End

# 实验六 综合设计

2022春季

[zjx@ustc.edu.cn](mailto:zjx@ustc.edu.cn)

# 实验目标

- 理解计算机硬件系统的组成结构和工作原理
- 掌握软硬件综合系统的设计和调试方法

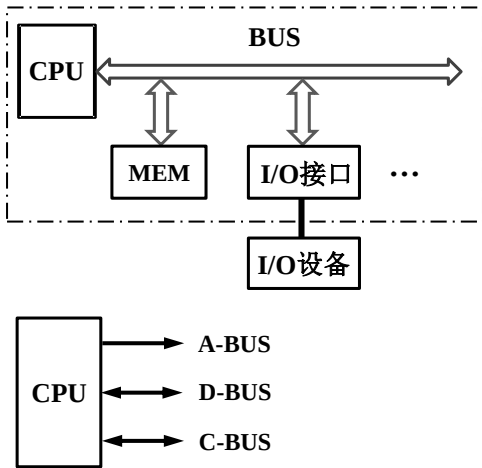
# 实验内容

- 完成一个简单计算机硬件系统及其应用程序的设计
  - CPU: 采用Lab5设计的, 或经过改进设计的 (例如: 扩充指令、中断处理、转移预测、Cache等)
  - 存储器: 指令/数据存储器均采用例化存储器
  - 外设: 拨动/按钮开关、指示灯、数码管、串口通信、定时/计数器、键盘、鼠标、VGA显示等
  - 应用程序: 求最大/小值、排序、流水灯、串口通信、画图、文本编辑等

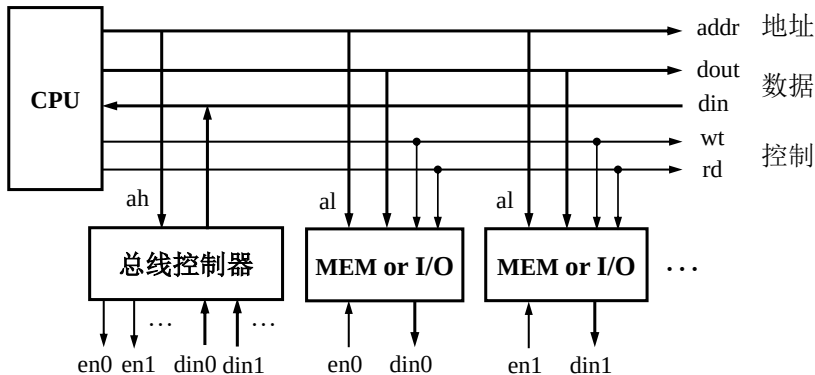
注意: 1. 侧重点可以在CPU的改进, 也可以在外设和应用程序  
2. 若改进CPU, 需要设计相应的测试程序

# 计算机硬件系统

- 单周期/流水线 CPU
- ROM和RAM
- 总线 BUS
  - 地址总线 A-BUS
  - 数据总线 D-BUS
  - 控制总线 C-BUS
- I/O接口
  - 并行开关和LED
  - 异步串行通信
  - 定时/计数器
  - VGA显示器
  - .....



# 总线的一种简单实现



# 实验检查

- 检查完整系统下载至FPGA后的运行情况

# The End