计算机组成原理 16 位流水 CPU 实验报告

小组编号 402
计 24 田博 2012011346
计 24 裘捷中 2012011345
计 24 柯均洁 2012011335

2014 年 12 月 22 日

Contents

1 实验成果综述

实验使用了 VHDL 语言在 FPGA 上编程实现包含 25 条标准指令和 5 条扩展指令的 16 位类似 MIPS 指令集的 CPU，支持多周期以及 5 级流水线。

为了解决结构冲突、数据冲突、控制冲突，使用了 Forward Unit、Hazard Detection 暂停流水等技术，提高流水线的效率。

扩展功能包括通过串口与 PC 机进行通信，从而运行监控程序，并在 PC 端控制 CPU 运行的指令（配合 CPU 运行的 kernel 程序）。其余功能还包括一个 VGA 显示，来同步显示当前的处理器状态，包括 PC、寄存器、指令等，从而方便 CPU 的调试。

同时实验还开发了一系列 Python 脚本，用于自动生成 entity 表和管脚绑定信息，减少开发的工作量，避免粗心引起的错误，极大的提高了开发的效率。

项目可以通过 https://github.com/dxmtb/CPU 访问，提供了所有的源代码和测试程序，以及生成代码的脚本。bits 文件夹包含了最终的 CPU 的 bit 文件，包含三个版本，cpu-12.5-final.bit 是 12.5MHz 的 CPU，cpu-click-final.bit 是单步版本，cpu-vga.bit 是单步并在 VGA 显示 CPU 状态的版本。

实验开发主要使用了 XILINX ISE 软件，开发版使用老师提供的 THINPAD，包括 FPGA、UART、RAM 等部件。

2 分工介绍

- 田博：CPU 顶层结构（连线），控制器，阶段锁存器，VGA 显示模块，RAM 和串口模块，寄存器组；
- 裘捷中：数据通路和控制信号设计，Forward Unit，Hazard Detector，RAM 和串口模块，ALU；
- 柯均洁：初版数据通路和控制信号，多路选择器，大部分实验报告。

3 关键设计介绍

总体上来看，CPU 分五级流水，就可以分为五个部分，下面结合实验中的一些困难详细介绍一些关键技术。

3.1 访存与串口

开发板为我们提供了两个 RAM，其中 RAM1 与串口共享总线。我们的指令和数据都需要用到 RAM，并且这种都要支持修改和读取操作。在最初的设计中，指令独占 RAM2，而数据和串口共享 RAM2，并且只实现了指令的读取操作，这样就避免了结构冲突。

之后发现监控程序需要修改指令内存，于是考虑到把指令和数据分开的动机已经消失，就把让串口独享 RAM1 的总线，从而简化串口的实现，而指令和数据都使用 RAM2。这样就无法避免结构冲突，即如果需要写内存或者读取除当前指令以外的数据，都会与当前读取指令的操作冲突。我们的做法是暂停流水。具体是在 Hazard Detection 中检测 MEM 阶段有操作，并且这个操作不是针对串口，就暂停 PC 和锁存器。

另一个难点是，传统的实现中内存操作往往要多个周期，这会拖慢 CPU 的速度。我们采用“双边沿”的方法有效的在一个时钟周期内完成所有访存操作。其实并没有检测两个沿，具体时序是：

a. 在上升沿，锁存器给出了 MEM 阶段的控制信号；
b. 判断时钟为 1 时，就重置 RAM 信号，即置各个控制信号为 1；
c. 在时钟的下降沿，所有控制信号均已就绪，RAM 信号已重置，可以根据当前操作给出新的控制信号；

d. RAM 的时钟较快，在控制时钟的下降沿不就，RAM 的时钟上升沿，执行了 MEM 指令；

e. 在时钟的上升沿，新的 RAM 数据（如果是读取的话）被送入锁存器，进入新的循环。

3.2 VGA 扩展

VGA 输出 800x600 的视频信号，通过对 50MHz 时钟进行分频来达到需要的工作频率。为了简化，我们仅仅使用两种颜色，黑色和白色。每一个 16 位的 CPU 状态，例如 PC 或者寄存器，每一个 bit 占一个 40x40 的一个方框，白色表示 1，黑色表示 0.

VGADisplayer 向 CoreDisplayer 模块传输当前刷新的坐标，通过判断坐标的对应位置来判断显示哪个变量，并且判断当前位的值。

3.3 代码自动生成

代码自动生成包括生成 entity 表、portmap 表、signal 表、控制信号生成。

entity 表比较简单，只要枚举每个 VHDL 文件，将相关关键字替换即可。

对于 portmap，由于我们并不知道它的输入和输出是什么，所以比较棘手。我们可以观察到，每个 portmap 的输出必定是到一个 signal，这样我们可以通过固定的明明规范，例如 entity 名下划线输出信号名来生成相应的 signal，并绑定到那个 signal。这样，我们只需要填充每个 portmap 的输入即可，并且通过名字就可以很容易判断 signal 的来源，大大减少了错误的可能。

对于控制信号，我们在代码开始对输出的 variable 赋值 NULL 语句的控制信号。每个控制信号和其取值有着规范的命名：控制信号的取值均为枚举类型，值为控制信号名下划线信号取值名。将控制信号表导出为 csv，就方便了脚本进行分析和自动代码生成。

4 实验结果

4.1 测试程序运行结果

CPU 时钟：12.5M 执行时间（详细运行情况请见 result1.MOV）

- 测试程序 1 运行时间：12.029s
- 测试程序 2 运行时间：18.072s

- 测试程序 3 运行时间；14.015s
- 测试程序 4 运行时间：26.021s
- 测试程序 5 运行时间：14.021s

4.2 扩展指令运行结果

程序的主要作用是往 R0 load 一个立即数，然后把 R0 的值放到 R1，再比较 R0 和 R1，如果两个相等，则往串口输出字母 E 运行情况（详细运行情况请见 result2.MOV）。

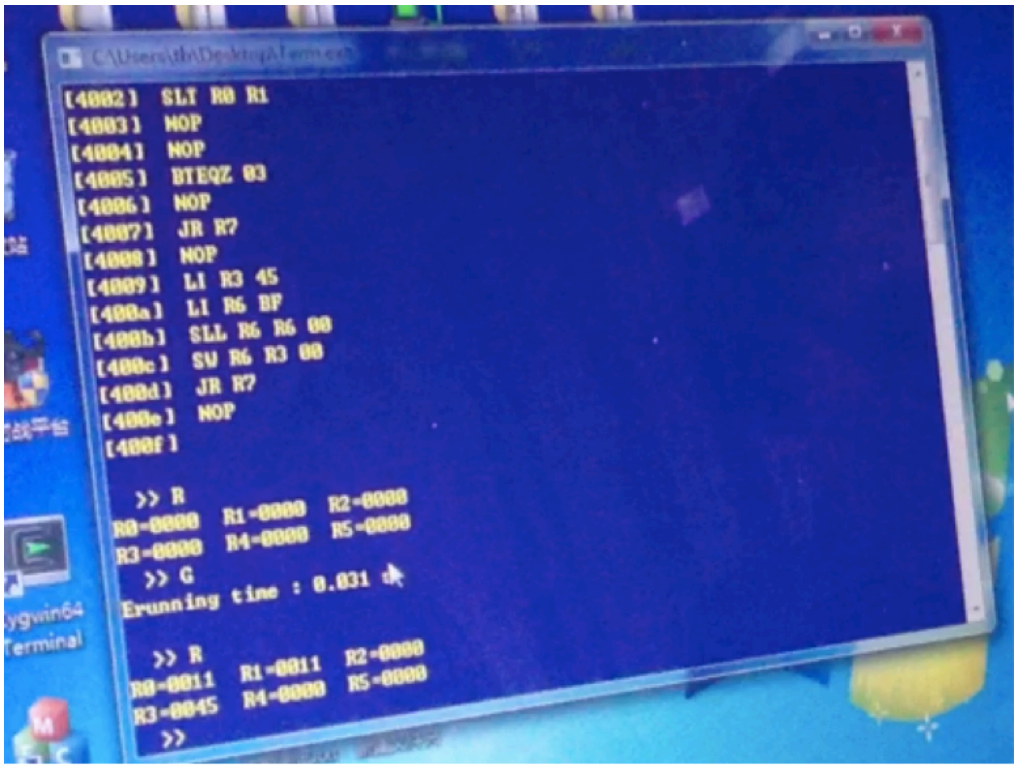

Figure 1: 扩展指令测试

如图 1所示，寄存器值发生了变化而且输出了字母 E，因此程序成功运行

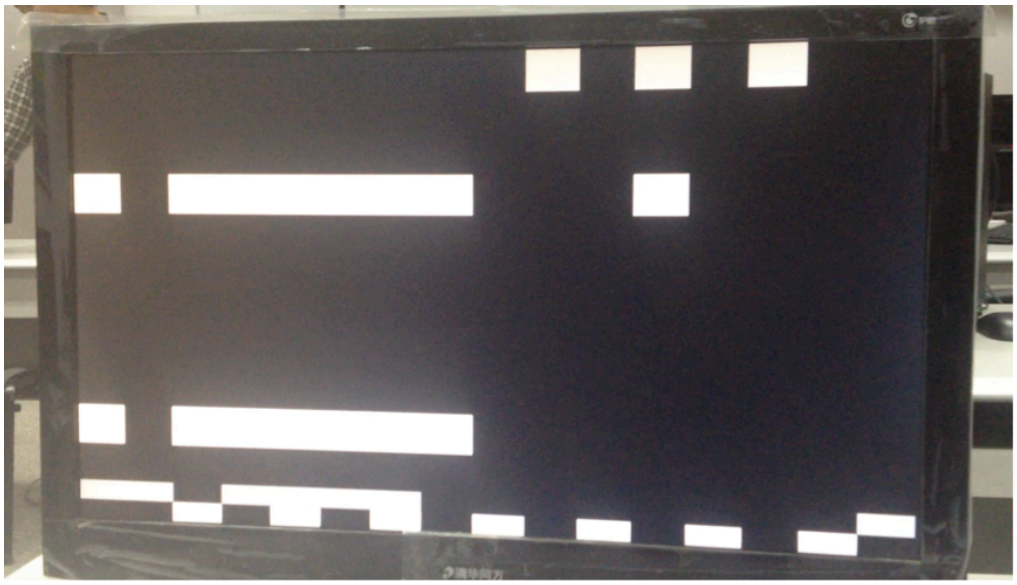

Figure 2: VGA 扩展

4.3 VGA 扩展运行结果

我们用 VGA 模块在显示屏上显示寄存器信息如图 2所示，从上到下依次为：PC、SP、IH、0-7 号寄存器、当前运行指令的指令通过按动按钮，我们可以观察到白块的移动，观察到寄存器值的变化，从而方便了程序的调试详细运行情况请见 result1.MOV

5 实验总结

通过这次实验，我们对 CPU 的各种细节有了更加充分的了解。更多的是，我们理解了团队合作的重要性：一个人做不到的事，有了其它人的帮助就可以做到。

我们由于数据同路的问题，直到 DDL 前一周的周五才开始编写代码，经过通宵和最后 38 小时的连续奋战，在检查当天的晚上 8 点终于完美运行所有功能，这与团队每个人的努力都分不开的。

实际上，我们最后一天想要尽快完成，在混乱不堪的内存串口模块中小修小补，直到前去检查也没有修补完成。到了检查现场，反而觉得超脱了急功近利的心情，静下心来对模块进行了重写，最后奇迹般的发现这个模块完美运行。这让我们体会到关键时刻保持清醒与镇定，摆脱急躁有多么重要。

这次大实验，所有的代码都是我们独立完成，尽管相对一些靠学长的组多花了许多时间，也少完成了一些扩展功能，但我们收获了人生的经验和感悟，这也是除了知识以外这个实验带给我们的最大收获。

6 详细设计与实现

6.1 数据通路和控制信号设计图图

见 Figure 3 和 Table 1。

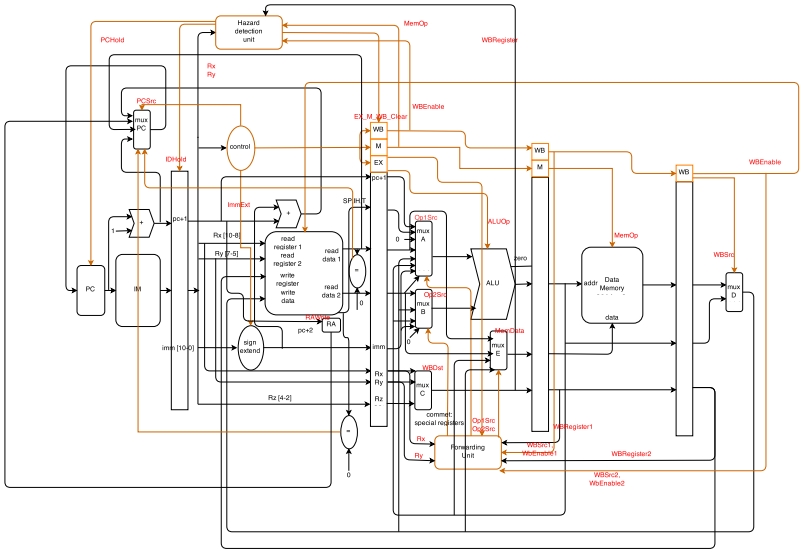

Figure 3: 数据通路和控制信号设计图

6.2 IF 阶段

6.2.1 MUX_PC

输入信号

- PCSrc：控制 MUX_PC 来选择下一个 PC 的取值，是顺序执行的 PC+1 或者是新的 PC 地址。取值情况为：

Table 1: 控制信号列表 (* 表示任意)

	PCSrc	ImmExt	RAWrite	Op1Src	Op2Src	WBDst	ALUOp	MemData	MemOp	WBSrc	WBEnable
NOP	PC+1	*	No	*	*	*	*	*	No	*	If WBDst!= * then Yes
ADDIU		(Sign,7-0)		Rx	Imm	Rx	Plus			ALURes	
ADDIU3		(Sign,3-0)		Rx	Imm	Ry	Plus			ALURes	
ADDSP		(Sign,7-0)		SP	Imm	SP	Plus			ALURes	
ADDU				Rx	Ry	Rz	Plus			ALURes	
AND				Rx	Ry	Rx	And			ALURes	
B	Branch-True	(Sign,10-0)									
BEQZ	Branch-Rx-0	(Sign,7-0)									
BNEZ	Branch-Rx-1	(Sign,7-0)									
BTEQZ	Branch-T-0	(Sign,7-0)									
CMP				Rx	Ry	T	Sub			ALURes	
JR	Rx										
LI		(Zero,7-0)		Imm	0	Rx	Or			ALURes	
LW		(Sign,4-0)		Rx	Imm	Ry	Plus		Read	Mem	
LW_SP		(Sign,7-0)		SP	Imm	Rx	Plus		Read	Mem	
MFIH				IH	0	Rx	Or			ALURes	
MFPC				PC+1	0	Rx	Or			ALURes	
MTIH				Rx	0	IH	Or			ALURes	
MTSP				Rx	0	SP	Or			ALURes	
OR				Rx	Ry	Rx	Or			ALURes	
SLL		(Shift,4-2)		Ry	Imm	Rx	SLL			ALURes	
SRA		(Shift,4-2)		Ry	Imm	Rx	SRA			ALURes	
SUBU				Rx	Ry	Rz	Sub			ALURes	
SW		(Sign,4-0)		Rx	Imm		Plus	Ry	Write		
SW_SP		(Sign,7-0)		SP	Imm		Plus	Rx	Write		
JRRA	RA										
JALR	Rx		Yes								
CMPI		(Sign,7-0)		Rx	Imm	T	Sub			ALURes	
MOVE				Ry	0	Rx	Or			ALURes	
SLT				Rx	Ry	T	If_Less			ALURes	

  - PC1：等于 PC+1，即顺序执行
  - B：跳转到新地址 Branch
  - RA：跳转地址来自 RA 寄存器
  - Rx：跳转地址来自 Rx 寄存器
  - Rx_0：R[x]=0 时跳转到新地址 Branch
  - Rx_1：R[x]!=0 时跳转新地址 Branch
  - T_0：T 寄存器 =0 时跳转新地址 Branch

- PCHold：
  - PC 是否保持，由 Hazard Detect Unit 控制。若 PCHold 为 1，则保持原有的 PC 不变；若 PCHold 为 0，则选择计算结果作为新的 PC

输出信号

- Ret：根据输入和控制信号输出相应的 PC 值

6.2.2 RamHandler 指令单元

由于在我们的实现中，指令内存和数据内存使用的时同一块 RAM(RAM2)，所以对于 RAM 的操作我们统一用 RamHandler 这个 entity 来封装实现。为保证说明时的一致性，我们会在**??**详细对这一模块进行描述。

6.3 ID 阶段

6.3.1 Controller 控制单元

单元描述

- 进行指令译码，产生之后阶段所需要的控制信号
- 默认初始化为 NOP 指令，即不写寄存器、不读内存

输入信号

- INS：16 位指令

输出信号

IFRegs IF 阶段的控制信号，控制指令的跳转，由以下具体信号组成：

- PCSrc：选择哪个地址作为下一个 PC，取值详见 6.2.1

IDRegs ID 阶段的控制信号，由以下具体信号组成：

- ImmExt：立即数扩展的形式
- RAWrite：是否要写 RA 寄存器

EXRegs EX 阶段的控制信号，由以下具体信号组成：

- Op1Src：ALU 操作数 1 的来源，MUX_A 选择信号
- Op2Src：ALU 操作数 2 的来源，MUX_B 选择信号
- WBDst：写回的目标寄存器。
- ALUOp：ALU 的操作码。
- MemData：读内存的地址来源寄存器

MRegs MEM 阶段的控制信号，由以下具体信号组成：

- MemOp：内存的读取，取值为：
  - None：非访存操作
  - Read：读取内存数据
  - Write：写内存数据

WBRegs WB 阶段的控制信号，由以下具体信号组成：

- WBSrc：写回寄存器数据来源于 ALU 输出还是内存读取输出
- WBEnable：是否写回

6.3.2 RegisterGroup 寄存器组

单元描述

维护寄存器组 将编程可见的 8 个通用寄存器（编号 0-7）以及 T 寄存器（编号 8）、IH 寄存器（编号 9）、SP 寄存器（编号 10）封装到一起。RA 寄存器单独存放。

寄存器值的写入 在一个 CPU 周期的上升沿写，写入的寄存器编号为 write_reg，由写回阶段的写回目的地信号 WB_Register_out_data.WBDst 指定。

寄存器值的读取 读取的寄存器号由 read_reg1 和 read_reg2 指定。特殊寄存器由专门的输出信号 regIH_out、regSP_out、regT_out 读出

输入信号

- clk：CPU 时钟
- write_enable：控制寄存器堆是否写入
- read_reg1、read_reg2：要读取的两个寄存器编号
- write_reg：要写入的寄存器编号
- write_data：要写入的值

输出信号

- reg1_data、reg2_data：读取的寄存器数据
- reg_0_out 到 reg7_out、regIH_out、regSP_out、regT_out：8 个通用寄存器及 Rsp、Rih 寄存器的数据

6.3.3 RA 寄存器

单元描述 由于 RA 寄存器用于特殊的 JALR 和 JRRA 指令，因此单独维护 RA 寄存器的读写。

输入信号

- clk：CPU 时钟周期
- RAWrite：是否写 RA 寄存器
- write_data：用于写 RA 寄存器的数据，PC+1

输出信号

- RA：当前 RA 寄存器的值

实现细节 由于写 RA 时只需要写入 RPC，所以将输入的写数据 write_data 取为 PC+1，若需要写 RA 则将 write_data 加 1 得到 RPC 写入 RA 寄存器

6.3.4 SignExtend / ZeroExtend 符号/零扩展单元

单元描述 计算不同指令符号扩展或零扩展之后的 16 位立即数

输入信号

- imm_in：输入的需要扩展的立即数（11 位）
- ImmExt：由于不同指令存放立即数的长度和扩展类型是不同的，因此需要该控制信号标明立即数的位数及需要扩展的类型取值类型为
  - ImmExt_Sign_7 立即数为 7-0 位，符号扩展
  - ImmExt_Sign_3 立即数为 3-0 位，符号扩展
  - ImmExt_Sign_10 立即数为 10-0 位，符号扩展

- ImmExt_Sign_4 立即数为 4-0 位，符号扩展
- ImmExt_Shift_4_2 立即数为 4-2 位（相当于右移）
- ImmExt_Zero_7 立即数为 7-0 位，零扩展

输出信号

- imm_out：16 位扩展立即数

6.3.5 Forwarding Unit 转发单元

单元描述 利用数据旁路技术来解决部分数据冒险

输入信号

- Rx, Ry
- Op1Src：ALU 操作数来源，MUX_A 选择信号
- Op2Src：ALU 操作数来源，MUX_B 选择信号
- wbsrc：当前指令要写回的结果来源于 ALU 的输出或访存结果
- wbregister1：EX_MEM 阶段的那条指令写回的寄存器
- wbregister2：MEM_WB 阶段的那条指令写回的寄存器
- wbenable1：EX_MEM 阶段的那条指令是否写寄存器
- wbenable2：MEM_WB 阶段的那条指令是否写寄存器
- memdata：写内存的数据来源

输出信号

- forwarda、forwardb、forwarde：MUX_A、MUX_B、MUX_E 是否使用 forward 数据及 forward 数据的来源，类型均为 ForwardType，取值为：
  - Forward_None：不需要使用转发数据
  - Forward_From_M_WB：转发数据来自 MEM_WB 阶段寄存器，也就是上一条指令的 ALU 计算结果
  - Forward_From_WB：转发数据来自上一个 MEM 阶段得到的准备写回的数据，即访存所得数据

实现细节

- 转发数据的来源有两个，一个是上一个阶段 ALU 的计算结果，另外一个是访存的结果。选择类型通过 ForwardType 的类型来指定
- 转发 ALU 计算结果的条件：
  - 写回寄存器编号 wbregister1 与 rx/ry/SP_index/IH_index 相同
  - 对应的写回使能 wbenable1 置 1
  - 当前指令写回的数据来源于 ALU 的计算结果
- 转发访存结果的条件：
  - 转发 ALU 计算结果的调节不满足
  - 写回寄存器编号 wbregister2 与 rx/ry/SP_index/IH_index 相同
  - 对应的写回使能 wbenable2 置 1
- 有些指令写回寄存器的值不是两个数操作后的运算结果，而是零扩展、某些寄存器值等等，可以将这种情况看做一个数和 0 相加，这样写回寄存器的值也可以看做 ALURes 来统一处理

6.3.6 Hazard Detector 冒险测试单元

单元描述

- 解决访存指令 LW 引起的数据冒险，若下一条指令 Ex 阶段需要使用 LW 指令的访存数据，则需要暂停流水，也就是插入 NOP。
- 此外，由于指令和数据均存放在 RAM2 中，所以访问数据时也需要暂停流水

输入信号

- wbenable：是否写回
- memop：ex 阶段指令访存的操作，取/读/不操作
- memop2：mem 阶段指令访存的操作，取/读/不操作
- idrx、idry：id 阶段的 rx 和 ry
- wbregister：写回的寄存器
- MemAddr：访问的内存地址

输出信号

- exmwbclear：是否清空 EX，MEM，WB 阶段的信号
- Idhold：是否保持 ID 阶段指令
- Pchold：是否保持 PC

实现细节

LW 引起的数据冒险暂停流水线作用条件

- 需要写回寄存器，即 wbenable 置 1
- memop==MemOp_read，即这是一条 LW 指令
- idrx 或 idry 与写回寄存器 wbregister 相等

MEM 阶段访问 RAM2 暂停流水线作用条件 由于我们把 IM 和 DM 都放在了 RAM2 上，所以一旦涉及到 RAM2 的读写操作，就无法正常读出下一条指令，因此必须暂定流水线一个周期。具体调节为：

- MEM 阶段有读写 RAM2 操作
- 访存地址不是串口相关地址

6.4 EX 阶段

6.4.1 ALU 模块

单元描述 提供运算结果

输入信号

- Op1：ALU 操作数 1
- Op2：ALU 操作数 2
- ALUOp：ALU 的运算类型控制信号，取值为：
  - ALUOp_Plus：Op1 + Op2
  - ALUOp_And：Op1 and Op2

- ALUOp_Sub：Op1 - Op2
- ALUOp_Or：Op1 or Op2
- ALUOp_SLL：Op1 逻辑左移 Op2
- ALUOp_SRA：Op1 算术右移 Op2
- ALUOp_If_Less：若 Op1 < Op2 则置为 1，否则置为 0

输出信号

- ALURes：运算结果

6.4.2 MUX_A/MUX_B：ALU 操作选择器

MUX_A

单元描述 选择数据作为 ALU 的第一个操作数

输入信号

- 由 EX 阶段锁存器给出的的 Op1src，指定操作数来源取值为
  - Rx：R[x]
  - Ry：R[y]
  - SP：SP 寄存器
  - IH：IH 寄存器
  - Imm：16 位立即数
  - PC1：PC+1
- 由 Forward Unit 产生的 ForwardA，指定是否使用转发数据及转发数据来源。
- Rx, Ry SP, IH：分别表示相应寄存器的值
- Imm：立即数
- PC1：PC+1 的值
- Data_From_WB：从 WB 阶段写回的数据
- Data_From_M_WB：从 MEM 阶段写回的数据

输出信号 根据 ForwardA 的值以及控制信号 Op1src 选择对应的输入值作为该多路选择器的输出。

MUX_B

单元描述 选择数据作为 ALU 的第二个操作数

输入信号

- 由 EX 阶段锁存器给出的的 Op2src，指定操作数来源，取值为
  - Ry：R[y]
  - Imm：16 位立即数
  - 0：零
- Ry：R[y]
- Imm：立即数
- Data_From_WB：从 WB 阶段写回的数据
- Data_From_M_WB：从 MEM 阶段写回的数据
- 由 Forward Unit 产生的 ForwardB，指定是否使用转发数据及转发数据来源。

输出信号 根据 ForwardB 的值以及控制信号 Op2src 选择对应的输入值作为该多路选择器的输出。

6.4.3 MUX_C：写回寄存器编号选择器

单元描述 选择写回寄存器的来源，由 WBDst 控制

输入信号

- Rx, Ry, Rz
- EX 阶段锁存器给出的 WBDst 信号，取值为：
  - Rx：写回 rx 字段的寄存器
  - Ry：写回 ry 字段的寄存器

  - Rz：写回 rz 字段的寄存器
  - SP：写回 SP 寄存器
  - IH：写回 IH 寄存器
  - T：写回 T 寄存器

输出信号　根据局 WBDst 选择对应的输入值作为该多路选择器的输出。

6.4.4　MUX_E：访存地址来源选择器

单元描述　选择访存地址的来源寄存器

输入信号

- Rx, Ry
- Data_From_WB：从 WB 阶段写回的数据
- Data_From_M_WB：从 MEM 阶段写回的数据
- 由 EX 阶段锁存器给出的 MemData 信号，取值为：
  - Rx：来源于 rx 字段的寄存器
  - Ry：来源于 ry 字段的寄存器
- Forward Unit 产生的 ForwardE，指定是否使用转发数据及转发数据来源

输出信号　根据 ForwardE 的值以及控制信号 MemData 选择对应的输入值作为该多路选择器的输出。

6.5　MEM 阶段

见关键设计介绍。

6.6　WB 阶段

6.6.1　MUX_D：写回数据来源选择器

单元描述　选择访写回数据来源于 ALU 输出或者是访存结果

输入信号

- WB 阶段锁存器给出的 WBSrc 信号，取值为
  - ALURes：写回数据来源于 ALU 的计算结果
  - Mem：写回数据来源于访存读取的数据
- ALURes：来自 ALU 的计算结果
- Mem：来自访存的结果

输出信号 根据 WBSrc 信号选择对应的写回数据