计算机组织与系统结构实习报告 Lab 2.2

学号：1500012863

姓名：杨程旭

大班教师：程旭

Part I：RISC-V 多周期模拟器

1. **基于实现的RISC-V ISA，给出指令各阶段的寄存器传输级描述 （10分）。每类指令举1-2个例子即可。**

R型指令：

sub rd, rs1, rs2

共分4个阶段

取指：inst[31:0] <- memory[PC], PC <- PC +4

译码：opcode = 0x33, func3 = 0x7, func7 = 0x00, raddr1 = inst[19:15], raddr2 = inst[24:20], waddr = inst[11:7]

执行：ALUop = 3(R型减操作), ALUout = REG[raddr1] - REG[raddr2]

写回：RegWrite = 1, wdata = ALUout

I型指令：

1. andi rd, rs1, imm

共分4个阶段

取指：inst[31:0] <- memory[PC], PC <- PC +4

译码：opcode = 0x13, func3 = 0x7, raddr1 = inst[19:15], waddr = inst[11:7], imm[11:0] = inst[31:20]

执行：ALUop = 23(I型与操作), ALUout = REG[raddr1] & imm

写回：RegWrite = 1, wdata = ALUout

(2) lb rd, offset(rs1)

共分5阶段

取指：inst[31:0] <- memory[PC], PC <- PC +4

译码：opcode = 0x03, func3 = 0x3, raddr1 = inst[19:15], waddr = inst[11:7], imm[11:0] = inst[31:20]

执行：ALUop = 13(计算内存地址的操作), ALUout = REG[raddr1] + imm

访存：MemRead = 1, val = memory[ALUout]

写回：RegWrite = 1, wdata = val

S型指令：

sd rs2, offset(rs1)

共分4阶段

取指：inst[31:0] <- memory[PC], PC <- PC +4

译码：opcode = 0x23, func3 = 0x3, raddr1 = inst[19:15], raddr2 = inst[24:20], imm[11:5] = inst[31:25], imm[4:0] = inst[11:7]

执行：ALUop = 30(计算写入地址的操作), ALUout = REG[raddr1] + imm

访存：MemWrite = 1, memory[ALUout] = REG[raddr2]

SB型指令：

beq rs1, rs2, offset

共分3阶段

取指：inst[31:0] <- memory[PC], PC <- PC +4

译码：opcode = 0x63, func3 = 0x0, raddr1 = inst[19:15], raddr2 = inst[24:20], imm[12] = inst[31], imm[11] = inst[7], imm[10:5] = inst[30:25], imm[4:1] = inst[11:8], imm[0] = 0

执行：ALUop = 31, if(REG[raddr1] == REG[raddr2]) PC = old\_PC + imm

U型指令：

lui rd, offset

共4个阶段

取指：inst[31:0] <- memory[PC], PC <- PC +4

译码：opcode = 0x37,waddr = inst[11:7], imm = inst[31:12]

执行：ALUop = 36(左移12位的操作), ALUout = imm << 12

写回：RegWrite = 1, wdata = ALUout

UJ型指令：

jal rd, imm

共分4阶段

取指：inst[31:0] <- memory[PC]

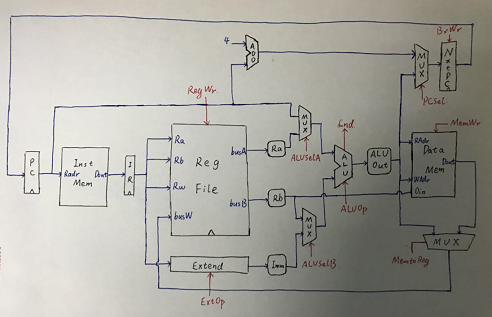
译码：opcode = 0x6f, waddr = inst[11:7], imm[20] = inst[31], imm[19:12] = inst[19:12], imm[11] = inst[20], imm[10:1] = inst[30:21], imm[0] = 0

执行：ALUop = 37, ALUout = temp\_PC + 4, PC = temp\_PC + imm

写回：RegWrite = 1, wdata = ALUout

1. **基于以上分析，给出多周期处理器的数据通路图和控制信号产生逻辑。不限形式，手绘也可。**

数据通路图如下：（红色为控制信号）



控制信号产生逻辑：

由于指令数较多（将近50个），很难画出有限状态图，因此只能大致描述一下每个信号的产生逻辑：

RegWr：只有写回阶段且需要写回寄存器时为1

ExtOp：当需要符号扩展时为1，经过译码后利用硬布线的方式可以得到该信号。

ALUselA：选择Ra或PC，译码阶段结束后产生。

ALUselB：选择Rb或扩展后的立即数，译码阶段结束后产生。

ALUop：译码阶段结束后产生，控制ALU做的操作。

Cnd：ALU计算后随结果一起产生。

PCSel：Cnd产生后相应产生，当遇到转移指令且Cnd为1时有效。

BrWr：跳转指令的执行阶段有效。

MemtoReg：处于写回阶段时用于选择写入ALUout还是Memout，译码阶段后产生。

MemWr：只有访存阶段且需要写内存时为1

1. **运行测试程序，给出动态执行的指令数。**
2. **运行测试程序，给出多周期处理器的执行周期数，并计算平均CPI。**

|  |  |  |  |
| --- | --- | --- | --- |
| **测试程序** | **动态指令数** | **执行周期数** | **CPI** |
| **test1** | **28** | **122** | **4.357143** |
| **test2** | **28** | **122** | **4.357143** |
| **test3** | **117** | **494** | **4.222222** |
| **test4** | **162** | **689** | **4.253086** |
| **test5** | **118** | **693** | **5.872881** |
| **test6** | **68** | **483** | **7.102941** |
| **test7** | **132** | **559** | **4.234848** |
| **test8** | **60** | **251** | **4.183333** |
| **test9** | **33** | **182** | **5.515152** |
| **test10** | **23** | **140** | **6.086957** |

对结果进行简单分析：

1和2：由于多周期没有数据冒险，1和2的差别在于顺序的不同，因此cpi一样是很容易理解的。

3和4：4相比3多了5次sum+=result[i],指令数增加，执行周期数增加，又由于多了一些访存操作(load),CPI略有上升。

5和6：5存在访存，6没有访存，而且二者都是除法，CPI较高。

7和8：8是7的循环展开，所以8的指令数和周期数都较低。

9和10：9存在函数调用而10没有，因此10的指令数和周期数都较低，同时9多出来的指令的周期数相比除法指令的周期数都低，因此平均下来9的CPI比10低。

Part II：RISC-V 流水线模拟器

1. **基于实现的RISC-V ISA，给定流水线处理器的阶段划分，并简单介绍各阶段的工作。**

按照标准的5级流水线划分：

取指：主要工作一是从指令存储器中取指令，将指令的各个位域存入中间寄存器IF\_ID。二是对下一条指令进行预测。（预测会在Part III详述）

译码：主要工作一是对指令进行译码并从寄存器堆中取得寄存器的值或者从指令中取立即数的值并进行相应的扩展，存入中间寄存器ID\_EX。二是根据指令的opcode、func3、func7产生这条指令所需要的控制信号（使用硬布线方式），存入中间寄存器ID\_EX。

执行：从中间寄存器ID\_EX中选择合适的值并根据控制信号（ALUop）做指定的运算，把结果存入中间寄存器EX\_MEM。

访存：根据MemWr信号对存储器进行写入或者读出的操作。读出的结果会写到中间寄存器MEM\_WB。

写回：根据中间寄存器MEM\_WB中的控制信号，选择合适的值（来自内存或ALUout）写入到寄存器堆中正确的寄存器内。

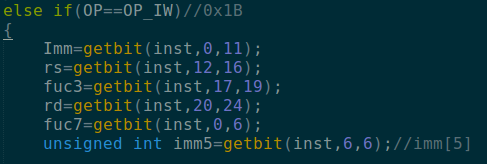
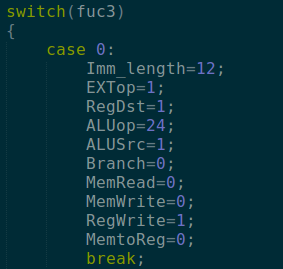
1. **给出流水线处理器的数据通路图和控制信号产生逻辑。不限形式，手绘也可。**

数据通路图见下下页。

控制信号的产生逻辑：

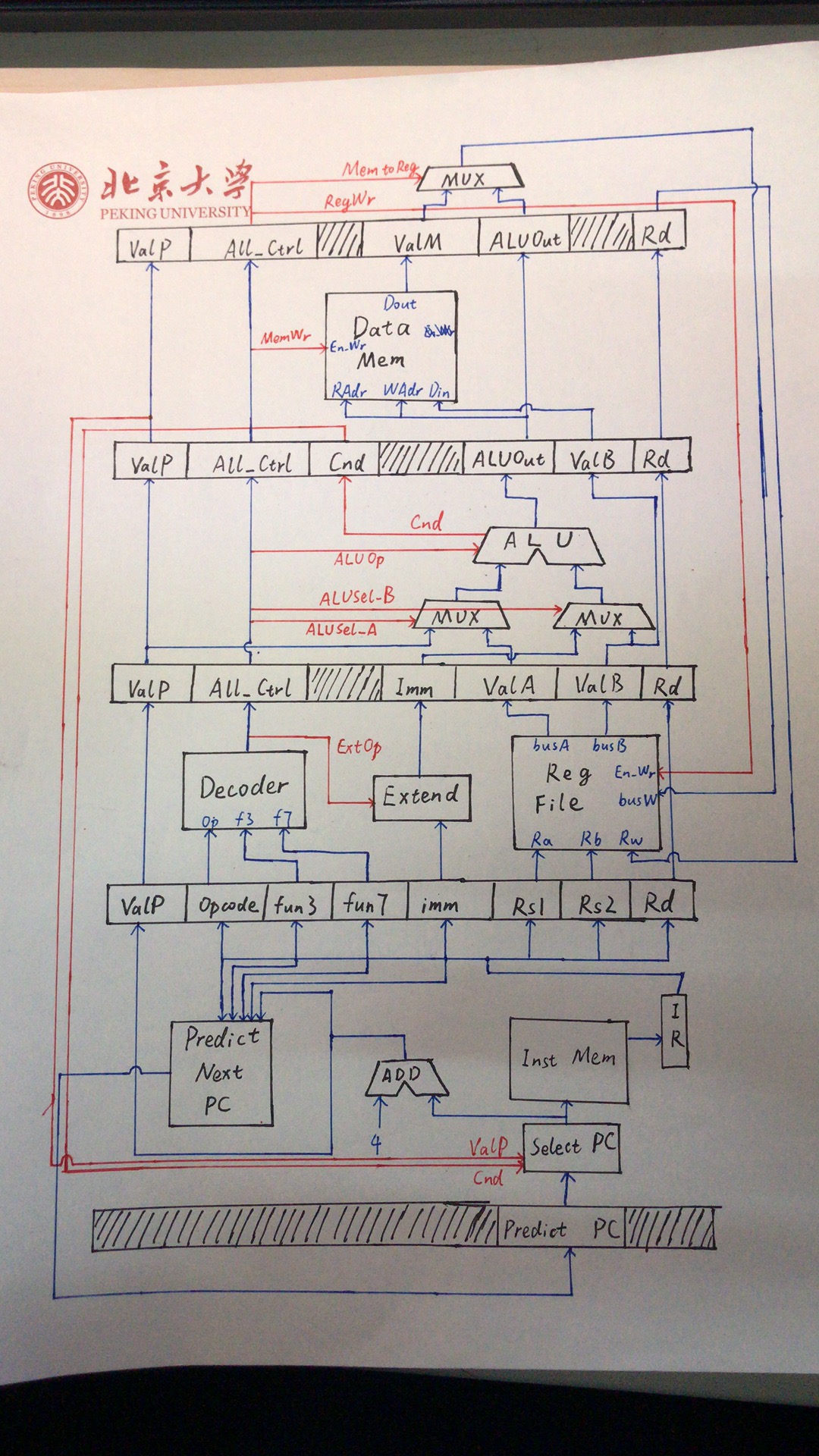
由于流水线与多周期和单周期的区别，指令的控制信号也要利用中间寄存器传下去。正如之前所说，任何一个指令，只要opcode、func3、func7确定了，那么这个指令所有周期的控制信号就确定了，因此在译码阶段，根据上述三个位域，采用硬布线的方式（decoder模块）产生所有的控制信号（在数据通路图中红线标识）。

每个控制信号与多周期的含义一致。完整的产生逻辑过于庞大，这里举一个例子进行说明：如下图所示

在译码阶段得到这是一条addiw指令，然后之间产生这个指令所需要的所有的控制逻辑，比方说EXTop=1表示需要符号扩展，MemRead=0表示不需要读内存，MemtoReg=0表示选择ALUout作为写回数据。

以上信号全部来自于代码模板中的Reg\_def.h，不排除与其他人相似甚至一样的可能。



1. **请简要描述该流水线中会产生的各种冒险，每类均需举例说明。**

数据冒险：

1. add a0, a1, a2

add a3, a0, a1

这种情况下，第二条指令在译码阶段需要得到a0的值，但是此时上一条指令要写回a0但是还处于执行阶段，没有写回，此时出现数据冒险，这类可以通过数据前递来解决。

1. lw a0, 18(sp)

add a3,a0,a1

这种情况下，同样存在数据冒险，此外，利用数据前递是不能解决的，一个周期的等待是不能避免的，因为a0正确的值只有在MEM阶段才能得到。

转移预测的分支冒险：

1. beq a1, a2, 0x50

这种情况下，只有在这条指令执行到执行阶段才能知道是否真正需要跳转，此时有两条指已经被取出来进入流水线了（一条在取指阶段，一条在译码阶段），因此存在分支预测冒险。为了解决这一冒险，减少转移预测错误，实现了利用2位历史信息的动态分支预测，将在Part III详述。

1. **运行测试程序，给出流水线处理器的执行周期数，并计算平均CPI。**

|  |  |  |  |
| --- | --- | --- | --- |
| **测试程序** | **动态指令数** | **执行周期数** | **CPI** |
| **test1** | **29** | **122** | **4.357143** |
| **test2** | **28** | **122** | **4.357143** |
| **test3** | **117** | **494** | **4.222222** |
| **test4** | **162** | **689** | **4.253086** |
| **test5** | **118** | **693** | **5.872881** |
| **test6** | **68** | **483** | **7.102941** |
| **test7** | **132** | **559** | **4.234848** |
| **test8** | **60** | **251** | **4.183333** |
| **test9** | **33** | **182** | **5.515152** |
| **test10** | **23** | **140** | **6.086957** |

1. **请对该流水线处理器中因不同类型的冒险而发生的停顿进行统计，并打印数据和分析。**

Part III：其他加分