班级中序号：101

南京航空航天大学

计 算 机 组 成 原 理 课 程 设 计

题 目 五级流水线处理器设计

学生姓名 李裕卓

学 号 162110320

学 院 16院

专 业 计算机科学与技术

班 级 1621103

课程老师 施慧彬

二〇二三年 六月

目录

[第一章 课程设计说明 4](#_Toc138166926)

[1.1指令实现 4](#_Toc138166927)

[1.2完成情况 4](#_Toc138166928)

[1.3流水线设计说明 4](#_Toc138166929)

[第二章 模块化和层次化设计说明 5](#_Toc138166930)

[2.1总电路设计图 5](#_Toc138166931)

[2.2流水线设计图 5](#_Toc138166932)

[2.3 层次化设计 6](#_Toc138166933)

[2.4 设计说明 6](#_Toc138166934)

[第三章 具体模块定义 7](#_Toc138166935)

[3.1 fetch 7](#_Toc138166936)

[3.2 IFIDReg 9](#_Toc138166937)

[3.3 decode 9](#_Toc138166938)

[3.4 IDEXReg 12](#_Toc138166939)

[3.5 exe 12](#_Toc138166940)

[3.6EXMEMReg 15](#_Toc138166941)

[3.7 mem 15](#_Toc138166942)

[3.8 MEMWRReg 16](#_Toc138166943)

[3.9 wb 16](#_Toc138166944)

[3.10 forwading 16](#_Toc138166945)

[3.11 load\_use 18](#_Toc138166946)

[3.12 control\_hazar 19](#_Toc138166947)

[3.13 pipeline\_cpu 20](#_Toc138166948)

[第四章 五段流水线CPU的调试 21](#_Toc138166949)

[4.1 具体verilog代码 21](#_Toc138166950)

[4.1.1 与单周期一致的代码略 21](#_Toc138166951)

[4.1.2 IFIDREG 21](#_Toc138166952)

[4.1.3 IDEXReg 22](#_Toc138166953)

[4.1.4 EXMEMReg 25](#_Toc138166954)

[4.1.5 MEMWRReg 28](#_Toc138166955)

[4.1.6 forwarding 30](#_Toc138166956)

[4.1.7 load\_use 31](#_Toc138166957)

[4.1.8 control\_hazar 32](#_Toc138166958)

[4.2 ModelSim仿真 33](#_Toc138166959)

[4.2.1 运行结果：波形图 33](#_Toc138166960)

[4.2.2 运行结果：打印信息 33](#_Toc138166961)

[4.2.3 运行结果：写出txt 34](#_Toc138166962)

[第五章 心得体会 34](#_Toc138166963)

# 第一章 课程设计说明

1.1指令实现

实现指令如下：

addu, subu, slt, and, nor, or, xor, sll, srl, sltu, jalr, jr, sllv, sra, srav, srlv, addiu, beq, bne, lw, sw, lui, slti, sltiu, bgez, bgtz, blez, bltz, lb, lbu, sb, andi, ori, xori, j, jal.

共36条指令。

1.2完成情况

1. 实现了36条指令，使用转发技术解决数据冲突，使用阻塞解决了load-use型冲突，解决了分支和跳转指令引起的控制冒险。成功运行测试指令。

2. 采用五级流水线设计

1.3流水线设计说明

处理了一条指令的执行过程被分成五个阶段，每个阶段由相应的功能部件完成。如果将各阶段看成相应的流水段，则指令的执行过程就构成了一条指令流水线。

（1）Ifetch（取指）：取指令并根据信号计算下一条指令地址。

（2）Reg/Dex（取数和译码）：根据指令取出寄存器中数据，并生成控制信号。

（3）Exec（执行）：根据指令进行计算。

（4）Mem（读写存储器）：根据指令读出或写入数据。

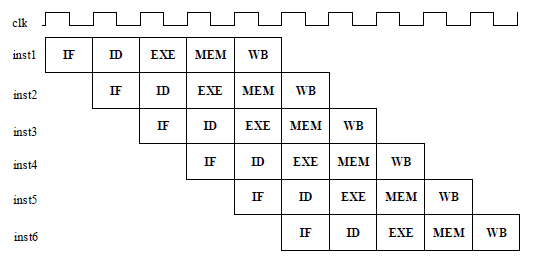
 （5）Wr（写寄存器）：根据指令将数据写入相应寄存器。

图1.1 流水线示意图

# 第二章 模块化和层次化设计说明

2.1总电路设计图

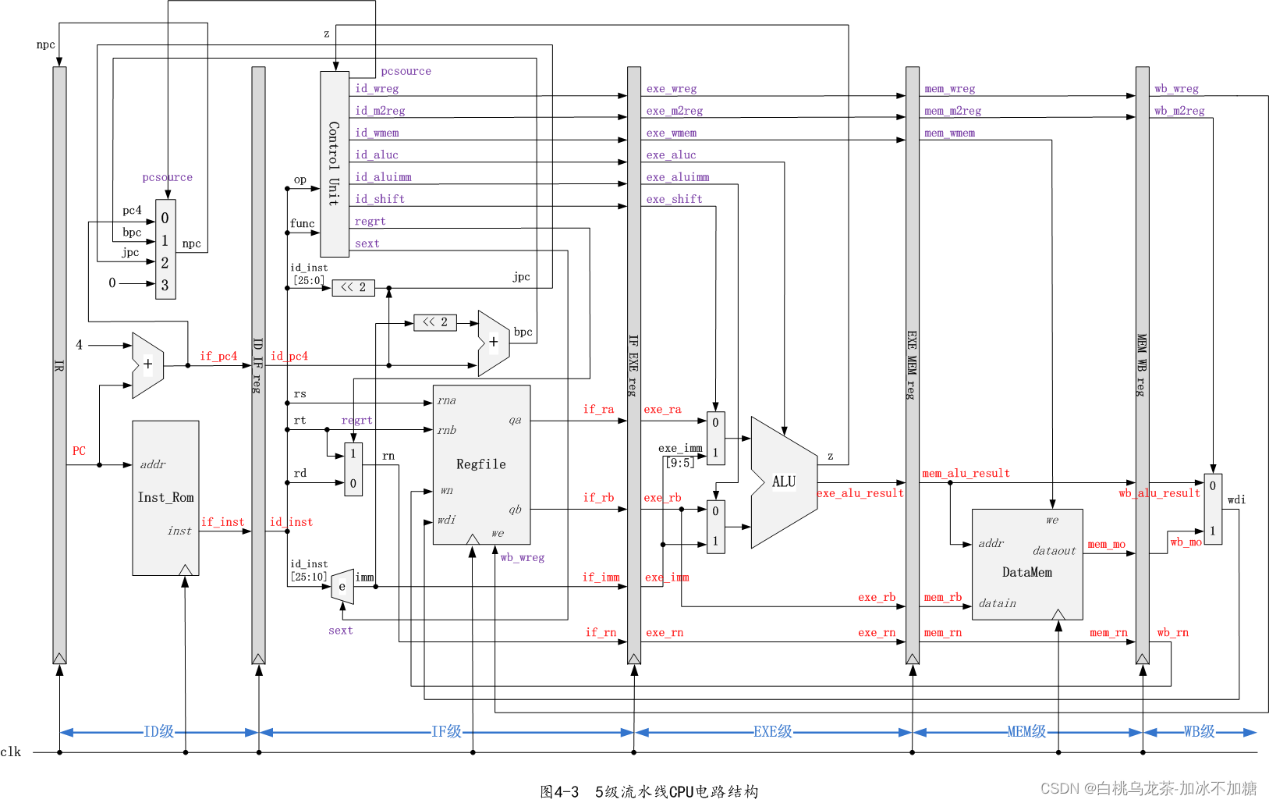
****

图2.1 总电路设计图

每个流水段传递所需信号和数据以及后续流水段所需信号和数据，大部分功能部件与单周期类似。增添forwading部件用于转发，增添load-use部件用于解决loaduse型数据冒险，增添control\_hazar部件用于解决控制冒险。

2.2流水线设计图

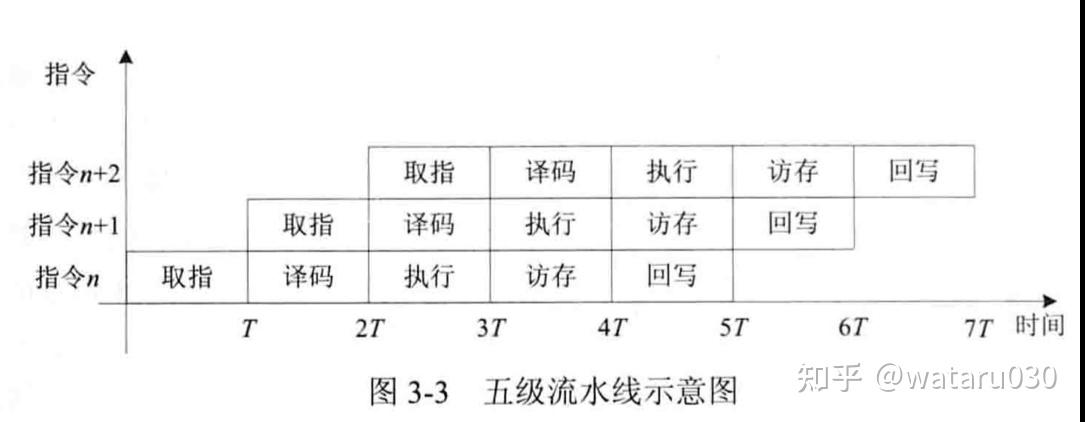


图2.2 流水线设计图

2.3 层次化设计

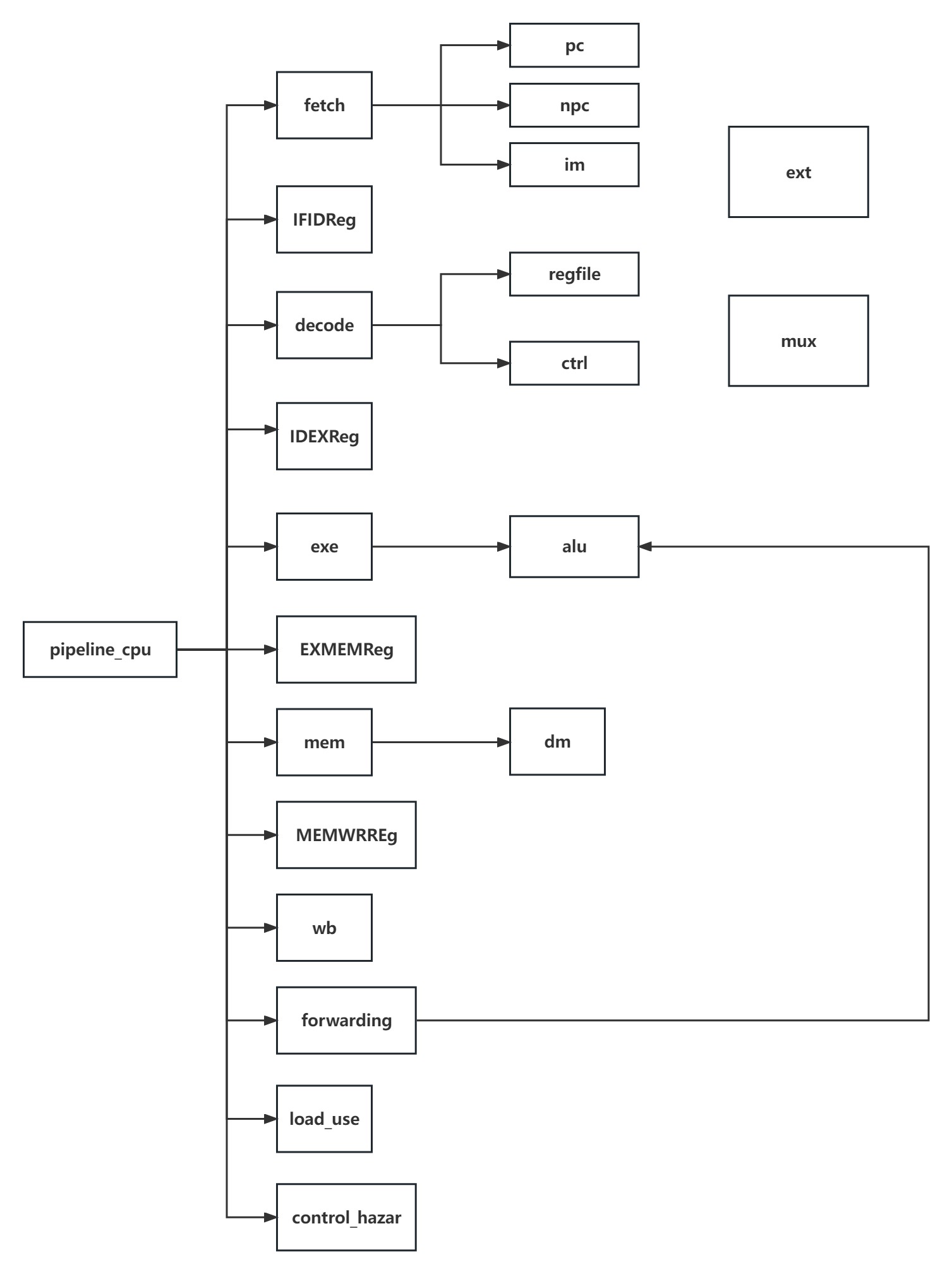


图2.3 流水线层次图

2.4 设计说明

采用了模块化和层次化的电路设计方法，根据五级流水线处理器的结构特征，将其分为五个阶段，每个阶段对应一个模块，分别命名为 fetch, decode, exe, mem, wb。同时，为每个流水段寄存器也设计了一个模块，分别为 IFIDReg, IDEXReg, EXMEMReg, MEMWRReg。此外，为了解决转发、数据冒险、控制冒险等问题，还将各个部件集成到了三个模块中，分别为 forwarding, load\_use, control\_hazar。这些模块构成了 cpu 的主要部分，通过 pipeline\_cpu 将它们连接起来。另外，还有一些辅助的小部件，它们都被整合到了主要模块中。

# 第三章 具体模块定义

3.1 fetch

1. 模块介绍

运行流水线ifetch阶段，主要功能是输出当前指令的地址和获取下一条指令的地址

2 功能和输入输出

接受信号给子模块使用，同时把结果输出给段寄存器。

(a)Input：clk, bubble<1:0>, branch<5:0>, j<3:0>, zero, sml, imm16<15:0>, target<25:0>, jumpRigister<31:0>, pcbefore<31:0>.

(b)inst\_id, PC

3 嵌套模块介绍

(1) PC和NPC

（a）基本描述：计算并输出下一条指令地址

（b）模块接口

|  |  |
| --- | --- |
| Input | 描述 |
| Clk | 时钟信号 |
| [1:0] bubble | 阻塞信号 |
| [5:0]branch | 条件分支指令信号，判断是哪种类型的条件转移 |
| [5:0]j | 转移指令信号，判断是哪种类型的转移指令 |
| zero | 相等信号，用于辨别转移条件是否满足 |
| small | 小于信号，用于辨别转移条件是否满足 |
| imm16 | 16位立即数 |
| target | 跳转信号的目标地址 |
| jumpRigister | 从寄存器读出的地址 |
| Output | 描述 |
| [31:0]pcAdd | PC+4 |
| [31:0]PC | 输出的指令地址 |

（c）功能定义

|  |  |
| --- | --- |
| 功能名称 | 功能描述 |
| 输出指令地址 | 时钟信号到来时，将 NPC 赋给 PC |
| 计算下一条指令地址 | 时钟信号到来时，由当前指令计算下一条指令地址 |
| 输出PC | 输出PC+4，以便于存储到第31号寄存器中 |

(2) im

(a) 基本描述：储存指令，根据输入地址输出指令

(b) 接口定义：

|  |  |
| --- | --- |
| Input | 描述 |
| [31:0] address | 指令地址 |
| Output | 描述 |
| [31:0] instruction | 指令 |

(c) 功能定义

|  |  |
| --- | --- |
| 功能名称 | 功能描述 |
| 载入指令 | 初始化载入 code.txt 中的指令 |
| 输出指令 | 根据当前指令地址，输出指令 |

3.2 IFIDReg

1 模块介绍

该模块是一个流水段寄存器，时钟信号下降沿时把信号传给下一流水段寄存器

2 功能和输入输出

接受信号，时钟信号上升沿时把信号传给下一流水段或寄存器。若 bubble

信号不为 0，说明有阻塞，不会把信号传给下一流水段或寄存器

(a)Input：clk, bubble, PC\_in, inst\_in

(b)Output: inst, PC

3.3 decode

1 模块介绍

该模块运行流水线的 Reg/Dec 阶段，主要功能是取数和译码

2 功能和输入输出

接受信号给子模块使用，同时把结果输出给段寄存器。分解指令的功能，把指令分为 op, rs, rt, rd, shamt, func, offset, imm16, target 给子模块使用或输出给后面阶段使用。并解析指令生成控制信号。

(a)Input：clk, inst, busW, RegWr\_in,

(b)Output: imm16, busA, busB , rs , rt , rd , shf , target ,branch<5:0>,J<3:0> , RegDst , ALUSrc , MemtoReg , RegWr , MemWr , ExtOp , ALUCtr<4:0>

3 嵌套模块介绍

（1）regfile

(a) 基本功能描述：寄存器堆实现32个寄存器，由两个输入的读地址分别输出两个寄存器内容；由一个写地址和写数据，以及写入信号实现寄存器的写入。

(b) 接口定义：

|  |  |
| --- | --- |
| Input | 描述 |
| clk | 时钟信号 |
| reset | 清零信号 |
| we | 写使能信号 |
| readAddress1 | 读出地址1 |
| readAddress2 | 读出地址2 |
| writeAddress | 写入地址 |
| writeData | 写入的数据 |
| Output | 描述 |
| readData1 | 读出的数据1 |
| readData2 | 读出的数据2 |

(c) 功能定义：

|  |  |
| --- | --- |
| 功能名称 | 功能描述 |
| 读寄存器数据 | 由两个读出地址，读出两个寄存器数据 |
| 向寄存器写入数据 | 由写入地址和写入数据，向指定寄存器写入数据 |

（2） ctrl

(a) 基本描述：根据op与func字段生成控制信号 。

(b) 接口定义：

|  |  |
| --- | --- |
| Input | 描述 |
| [5:0] op | 输入的指令操作码 |
| [5:0] func | 输入的指令识别码（R型） |
| clk | 时钟信号 |
| Output | 描述 |
| [5:0] branch | 条件分支信号（beq，bne，bgez，bgtz，blez，bltz） |
| [3:0] j | 转移信号（jump，jal，jr，jalr） |
| link | 是否保存下一条指令到$ra |
| lb | 是否要读入一字节（有符号扩展） |
| lbu | 是否要读入一字节（无符号扩展） |
| sb | 存储一字节数据 |
| useShamt | 是否使用Shamt |
| regDst | 使用Rd还是Rt中作为寄存器写入地址 |
| mem2Reg | 使用alu结果还是存储器中的内容作为写入寄存器中的内容 |
| regWr | 允许写入寄存器 |
| memWr | 允许写入存储器 |
| extOp | 是否使用有符号扩展 |
| rtype | 指令是否是R型指令 |
| aluSrc | 选择rt地址中的内容还是扩展后的直接数 |
| [4:0] aluCtr | Alu操作码 |

(c) 功能定义：

|  |  |
| --- | --- |
| 功能名称 | 功能描述 |
| 输出控制信号 | 根据op和func输出各种控制信号 |

3.4 IDEXReg

1 模块介绍：

该模块是一个流水段寄存器，储存控制信号和数据，时钟信号下降沿时把信号，数据传给下一流水段。

2 功能和输入输出：

接受信号，时钟信号下降沿时把信号传给下一流水段或寄存器。若 bubble 信号不为 0，说明有阻塞，不会把信号传给下一流水段或寄存器。

(a)Input:clk, regReset, bubble, [5:0]branch\_id, [3:0], j\_id, regDst\_id, aluSrc\_id, link\_id, lw\_id, lb\_id, lbu\_id, sb\_id, useshamt\_id, [4:0]aluCtr\_id, mem2Reg\_id, regWr\_id, memWr\_id, extOp\_id, rtype\_id, [31:0]busA\_id, Reg[Rs][31:0]busB\_id, Reg[Rt], [4:0]shamt\_id, [15:0]imm16\_id, imm16[25:0]jumpAddress\_id, [4:0]rs\_id, [4:0]rt\_id, [4:0]rd\_id, [31:0]instru\_idAddress,

(b)output:[5:0]branch\_ex, [3:0]j\_ex, Dst\_ex, aluSrc\_ex, link\_ex, lw\_ex, lb\_ex, lbu\_ex, sb\_ex, useshamt\_ex, [4:0]aluCtr\_ex, mem2\_ex, Wr\_ex, memWr\_ex, extOp\_ex, rtype\_ex, [31:0]busA\_ex, //[Rs][31:0]busB\_ex, [Rt][4:0]shamt\_ex, [15:0]imm16\_ex, [25:0]jumpAddress\_ex, [4:0]rs\_ex, [4:0]rt\_ex, [4:0]rd\_ex, [31:0]instru\_exAddress

3.5 exe

1 模块介绍：

该模块运行流水线的 Exec 阶段，主要功能是计算地址和两数运算结果。

2 功能和输入输出：

接受信号给子模块使用，同时把结果输出给段寄存器。把后几个阶段的数据和结果传进来，由 forwarding 模块提供转发信号，选择要进入 ALU 操作的数据，处理控制冒险。此阶段也选择I型指令和R型指令的目的寄存器。.

3 嵌套模块介绍：

（1）ALU

（a）基本描述：实现各种运算操作。

（b）接口定义：

|  |  |
| --- | --- |
| Input | 描述 |
| [31:0] a | 操作数a |
| [31:0] b | 操作数b |
| [4:0] op | 操作类型 |
| Output | 描述 |
| [31:0] out | 计算结果 |
| zero | 计算结果是否为0 |
| small | 操作数a是否小于操作数 |
| overflow | 是否溢出 |

（c）功能定义：

|  |  |
| --- | --- |
| 功能名称 | 功能描述 |
| 输出计算结果 | 由操作类型，输出a和b计算结果 |
| 输出zero | 输出计算结果是否为0 |
| 输出small | 输出a是否小于b |
| 输出overflow | 输出是否溢出 |

（2）MUX

（a）基本描述：需要有二路，三路，四路选择器，以二路为例。

（b）接口定义：

|  |  |
| --- | --- |
| Input | 描述 |
| [len-1:0] a | 输入a |
| [len-1:0] b | 输入b |
| Select | 选择控制信号 |
| Output | 描述 |
| [len-1:0] out | MUX选择结果 |

（c）功能定义：

|  |  |
| --- | --- |
| 功能名称 | 功能描述 |
| 输出选择结果 | 由select信号输出选择结果 |

（3）EXT

（a）基本描述：将输入的len长度的数据扩展成32位的数据。

（b）接口定义：

|  |  |
| --- | --- |
| Input | 描述 |
| a | 输入的长为len的数据 |
| sign\_ext | 是否是符号扩展 |
| Output | 描述 |
| b | 输出的32位的数据 |

（c）功能定义：

|  |  |
| --- | --- |
| 功能名称 | 功能描述 |
| 输出扩展后的数据 | 将输入数据扩展成32位 |

3.6EXMEMReg

1 模块介绍：

该模块是一个流水段寄存器，储存控制信号和数据，时钟信号上升沿时把信号，数据传给下一流水段。

2 功能和输入输出

(a) Input: [5:0]branch\_ex, [3:0]j\_ex, mem2Reg\_ex, regWr\_ex, memWr\_ex, lbu\_ex, lb\_ex, sb\_ex, link\_ex, overflow\_ex, zero\_ex, sml\_ex, [31:0]busA\_ex, [31:0]aluResult\_ex, [31:0]busB\_ex, [15:0]imm16\_ex, [25:0]jumpAddress\_ex, [4:0]rt\_ex, [4:0]rw\_ex, [31:0]instru\_exAddress.

(b) Output: [5:0] branch\_mem, [3:0] j\_mem, mem2Reg\_mem, Wr\_mem, memWr\_mem, lb\_mem, lbu\_mem, sb\_mem, link\_mem, overflow\_mem, zero\_mem, sml\_mem, [31:0]aluResult\_mem, [15:0]imm16\_mem, [25:0]jumpAddress\_mem, [31:0]busB\_mem, [31:0]busA\_mem, [4:0]rt\_mem, [4:0]rw\_mem, [31:0]instru\_memAddress

3.7 mem

1 模块介绍：该模块运行流水线的 MEM 阶段，主要功能是从主存中读写数值。

2 功能：接受信号给子模块使用，同时把结果输出给段寄存器。

3 嵌套模块介绍：

（1） dm

（a）基本描述：进行数据向内存中的读写。

（b）接口定义：

|  |  |
| --- | --- |
| Input | 描述 |
| clk | 时钟信号 |
| wren | 写使能信号 |
| sb | 存储一个字节的标志 |
| lb | 读入一个字节的标志（有符号扩展） |
| lbu | 读入一个字节的标志（无符号扩展） |
| [31:0] address | 读/写数据的地址 |
| [31:0] writeData | 写入的数据 |
| Output | 描述 |
| [31:0] readData | 读出的数据 |

（c）功能定义：

|  |  |
| --- | --- |
| 功能名称 | 功能描述 |
| 读出数据 | 按地址读出存储器中数据 |
| 写入数据 | 按地址向存储器中写数据 |

3.8 MEMWRReg

1 模块介绍：

该模块是一个位于 MEM 和 WB 阶段间的流水段寄存器，储存控制信号和数据，时钟信号上升沿时把信号，数据传给下一流水段或寄存器。

2 功能和输入输出：

接受信号，时钟信号上升沿时把信号传给下一流水段或寄存器。遇到阻塞会清零。

(a)Input: clk, regReset, mem2Reg\_mem, regWr\_mem, link\_mem, overflow\_mem, [31:0]aluResult\_mem, [31:0]data\_in, [4:0]rw\_mem, [31:0]instru\_memAddress.

(b)Output: mem2Reg\_wr, Wr\_wr, link\_wr, overflow\_wr, [31:0]aluResult\_wr, [31:0]data\_out, [4:0]rw\_wr, [31:0]instru\_wrAddress.

3.9 wb

1 模块介绍：

该模块运行流水段的写回阶段，来自ALU的结果和主存的结果经过选择，传到decode 阶段的寄存器里面，写使能信号也将传输过去。

2 功能和输入输出：

(a) Input：Dout, Result, MemtoReg。

(b) Output: Din

3.10 forwading

1 模块介绍

该模块处理数据冒险的转发，如相差一条指令和两条指令的，写入寄存器的数据被下一条指令读出。

2 接口定义

|  |  |
| --- | --- |
| Input | 描述 |
| rw\_mem | mem 段的rw |
| rw\_wr | wr段的rw |
| rs\_ex | ex段的rs |
| rt\_ex | ex段的rt |
| aluSrc\_ex | ex段的aluSrc |
| regWr\_mem | mem段的regWr |
| regWr\_wr | wr段的regWr |
| mem2Reg\_wr | wr段的mem2Reg |
| bubble | 是否有阻塞，用于解决loaduse |
| Output | 描述 |
| aluAchoose | AluA端最终选择 |
| aluBchoose | AluB端最终选择 |

3 功能定义：

|  |  |
| --- | --- |
| 条件 | 功能描述 |
| ALUSrc\_A = 0 | A ：busA或者shamt |
| ALUSrc\_A = 1 | A：mem段的 aluResult |
| ALUSrc\_A = 2 | A：wr段的 aluResult |
| ALUSrc\_A = 3 | A：数据存储器的读出数据 |
| ALUSrc\_B = 0 | B ：busB或者立即数 |
| ALUSrc\_B = 1 | B：mem段的 aluResult |
| ALUSrc\_B = 2 | B：wr段的 aluResult |
| ALUSrc\_B = 3 | B：数据存储器的读出数据 |

3.11 load\_use

1 模块介绍：

用于处理lw，lb，lbu所引起的数据冒险，阻塞一个周期再转发。

2 接口定义：

|  |  |
| --- | --- |
| Input | 描述 |
| clk | 时钟信号 |
| lw\_ex | ex段的lw信号 |
| lb\_ex | ex段的lb信号 |
| lbu\_ex | ex段的lbu信号 |
| rs\_id | ID 阶段的 rs |
| rt\_id | ID 阶段的 rt |
| rt\_ex | EXE 阶段的 rt |
| usebranch\_mem | 分支跳转是否成功 |
| J\_mem | 跳转指令类型 |
| bubble | 先前的阻塞 |
| Output | 描述 |
| bubble\_new | 新的气泡 |

3 功能定义：

|  |  |
| --- | --- |
| 条件 | 描述 |
| 发生 lw 冒险 | bubble = 1，阻塞一次 |
| 发送 lb 冒险 | bubble = 1，阻塞一次 |
| 发生 lbu 冒险 | bubble = 1，阻塞一次 |
| bubble = 1 | bubble = 0，只阻塞一次 |
| j ！= 0 | 上一条指令是 j或jal或jalr或jr指令，需要跳转 地址，则冒险不需要处理，bubble = 0。 |
| usebranch=1 | 上一条指令成功条件转移，冒险不处理，bubble=0. |

3.12 control\_hazar

1 模块介绍

该模块处理跳转指令和分支指令的控制冒险。

2 接口定义

|  |  |
| --- | --- |
| Input | 描述 |
| clk | 时钟信号 |
| usebranch\_mem | 分支跳转是否成功 |
| J\_mem | 跳转指令类型 |
| Output | 描述 |
| regReset | 寄存段清零 |

3 功能定义

|  |  |
| --- | --- |
| 条件 | 描述 |
| j ！= 0 | 是 j或jal或jalr或jr指令，需要跳转 地址，令regReset=1 |
| usebranch=1 | 指令成功条件转移，令regReset=1 |
| 其他条件 | 令regReset=0 |

3.13 pipeline\_cpu

1 模块介绍

该模块将所有部件相连，组成mips流水线处理器。

2 接口定义：

|  |  |
| --- | --- |
| Input | 描述 |
| clk | 时钟信号 |
| Rst | 清零信号 |
| instru\_if | 给if流水段的指令 |
| data\_in | 从存储器独处的数据 |
| output | 描述 |
| instru\_ifAddress | If段的指令地址 |
| wren | 存储器写使能 |
| sb | 给存储器：是否是sb指令 |
| lb | 给存储器：是否是lb指令 |
| lbu | 给存储器：是否是lbu指令 |
| dataAddress | 存储器读写地址 |
| writeData | 存储器读写数据 |

3 功能定义：

按照“code.txt”文件中的指令执行。

# 第四章 五段流水线CPU的调试

4.1 具体verilog代码

4.1.1 与单周期一致的代码略

4.1.2 IFIDREG

// IF—ID段寄存器

module IFIDReg\_320

(

    input                   clk,

    input[1:0]                   bubble,

    input[31:0]           instru\_if,

    input[31:0] instru\_ifAddress,

    input regReset,

    output reg[31:0]          instru\_id,

    output reg[31:0] instru\_idAddress

    );

    always @(negedge clk) begin

        if(bubble==2'b00&&regReset==0) begin

            instru\_id=instru\_if;

            instru\_idAddress=instru\_ifAddress;

        end

    end

    always @(posedge regReset) begin //上升沿时说明要重整id

        if(bubble==2'b00) begin

            instru\_id=instru\_if;

            instru\_idAddress=instru\_ifAddress;

        end

    end

endmodule

4.1.3 IDEXReg

module IDEXReg\_320(

    input   clk,                //时钟信号输入

    input   regReset,

    input   bubble,              //阻塞信号输入

    //控制信号输入

    input[5:0]   branch\_id,

    input[3:0]   j\_id,

    input   regDst\_id,

    input   aluSrc\_id,

    input   link\_id,

    input lw\_id,

    input   lb\_id,

    input   lbu\_id,

    input   sb\_id,

    input   useshamt\_id,

    input[4:0] aluCtr\_id,

    input   mem2Reg\_id,

    input   regWr\_id,

    input   memWr\_id,

    input   extOp\_id,

    input   rtype\_id,

    //数据输入

    input[31:0]   busA\_id,            //Reg[Rs]

    input[31:0]   busB\_id,            //Reg[Rt]

    input[4:0] shamt\_id,

    input[15:0]   imm16\_id,           //输入imm16

    input[25:0]   jumpAddress\_id,

    input[4:0]    rs\_id,

    input[4:0]    rt\_id,

    input[4:0]    rd\_id,

    input[31:0]   instru\_idAddress,

    //控制信号输出

    output reg[5:0]  branch\_ex,

    output reg[3:0]  j\_ex,

    output reg  regDst\_ex,

    output reg  aluSrc\_ex,

    output reg  link\_ex,

    output reg lw\_ex,

    output reg  lb\_ex,

    output reg  lbu\_ex,

    output reg  sb\_ex,

    output reg  useshamt\_ex,

    output reg[4:0]  aluCtr\_ex,

    output reg  mem2Reg\_ex,

    output reg  regWr\_ex,

    output reg  memWr\_ex,

    output reg  extOp\_ex,

    output reg  rtype\_ex,

    //数据输出

    output reg[31:0]   busA\_ex,            //Reg[Rs]

    output reg[31:0]   busB\_ex,            //Reg[Rt]

    output reg[4:0]    shamt\_ex,

    output reg[15:0]   imm16\_ex,

    output reg[25:0]   jumpAddress\_ex,

    output reg[4:0]    rs\_ex,

    output reg[4:0]    rt\_ex,

    output reg[4:0]    rd\_ex,

    output reg[31:0]   instru\_exAddress

);

    initial begin

        branch\_ex=0;

        j\_ex=0;

        regDst\_ex=0;

        aluSrc\_ex=0;

        aluCtr\_ex=0;

        mem2Reg\_ex=0;

        regWr\_ex=0;

        memWr\_ex=0;

        extOp\_ex=0;

        rtype\_ex=0;

        //数据输出

        busA\_ex=0;           //Reg[Rs]

        busB\_ex=0;            //Reg[Rt]

        imm16\_ex=0;

        jumpAddress\_ex=0;

        rt\_ex=0;

        rd\_ex=0;

        instru\_exAddress=0;

    end

    always @(negedge clk) begin

        if(bubble==2'b00) begin

            // 控制信号保存

            branch\_ex=branch\_id;

            j\_ex=j\_id;

            regDst\_ex=regDst\_id;

            aluSrc\_ex=aluSrc\_id;

            link\_ex = link\_id;

            lw\_ex=lw\_id;

            lb\_ex=lb\_id;

            lbu\_ex=lbu\_id;

            sb\_ex=sb\_id;

            useshamt\_ex=useshamt\_id;

            aluCtr\_ex=aluCtr\_id;

            mem2Reg\_ex=mem2Reg\_id;

            regWr\_ex=regWr\_id;

            memWr\_ex=memWr\_id;

            extOp\_ex=extOp\_id;

            lw\_ex=lw\_id;

            //rtype\_ex=rtype\_id;

            // 数据信号保存

            busA\_ex=busA\_id;

            busB\_ex=busB\_id;

            imm16\_ex=imm16\_id;

            shamt\_ex=shamt\_id;

            jumpAddress\_ex=jumpAddress\_id;

            rs\_ex=rs\_id;

            rt\_ex=rt\_id;

            rd\_ex=rd\_id;

            instru\_exAddress=instru\_idAddress;

        end

        if((bubble!=2'b00)||regReset) begin

            branch\_ex=0;

            j\_ex=0;

            regDst\_ex=0;

            aluSrc\_ex=0;

            aluCtr\_ex=0;

            mem2Reg\_ex=0;

            regWr\_ex=0;

            memWr\_ex=0;

            extOp\_ex=0;

            lw\_ex=0;

            //rtype\_ex=rtype\_id;

            // 数据信号保存

            busA\_ex=0;

            busB\_ex=0;

            imm16\_ex=0;

            rs\_ex=0;

            rt\_ex=0;

            instru\_exAddress=0;

            rd\_ex=0;

        end

    end

endmodule

4.1.4 EXMEMReg

module EXMEMReg\_320(

    input           clk,                //时钟信号输入

    input    regReset,

    //控制信号输入

    input[5:0]  branch\_ex,

    input[3:0]  j\_ex,

    input mem2Reg\_ex,//选择busw时要用

    input regWr\_ex, //写寄存器要用

    input memWr\_ex, //mem用

    input lbu\_ex,

    input lb\_ex,

    input sb\_ex,

    input link\_ex,

    //input  rtype\_ex,

    //数据输入

    input   overflow\_ex,//给pc

    input   zero\_ex,//给pc

    input   sml\_ex,

    input [31:0] busA\_ex,

    input[31:0]  aluResult\_ex,//mem address

    input[31:0]  busB\_ex,//mem datain

    input[15:0]  imm16\_ex,

    input[25:0]  jumpAddress\_ex,

    input[4:0]   rt\_ex,

    input[4:0]   rw\_ex,

    input[31:0]  instru\_exAddress,

    //控制信号输出

    output reg [5:0] branch\_mem,

    output reg [3:0] j\_mem,

    output reg  mem2Reg\_mem,

    output reg  regWr\_mem,

    output reg  memWr\_mem,

    output reg lb\_mem,

    output reg lbu\_mem,

    output reg sb\_mem,

    output reg link\_mem,//jal指令没有给rt设置传到之后有用

    output reg overflow\_mem,

    output reg zero\_mem,

    output reg sml\_mem,

    //数据输出

    output reg[31:0] aluResult\_mem,

    output reg[15:0] imm16\_mem,

    output reg[25:0] jumpAddress\_mem,

    output reg[31:0] busB\_mem,

    output reg[31:0] busA\_mem,

    output reg[4:0]  rt\_mem,

    output reg[4:0]  rw\_mem,

    output reg[31:0] instru\_memAddress//写给31号寄存器要用到之前的指令地址

);

initial begin

    branch\_mem=0;

    j\_mem=0;

    mem2Reg\_mem=0;

    regWr\_mem=0;

    memWr\_mem=0;

    overflow\_mem=0;

    zero\_mem=0;

    aluResult\_mem=32'd0;

    imm16\_mem=16'd0;

    jumpAddress\_mem=26'd0;

    busB\_mem=32'd0;

    rw\_mem=5'd0;

    instru\_memAddress=0;

end

    always @(posedge clk) begin //遇到控制冒险什么的，在当前周期后半部分解决掉，不能拖到下个周期

        if(~regReset) begin

            branch\_mem=branch\_ex;

            j\_mem=j\_ex;

            mem2Reg\_mem=mem2Reg\_ex;

            regWr\_mem=regWr\_ex;

            memWr\_mem=memWr\_ex;

            //rtype\_mem=rtype\_id;

            overflow\_mem=overflow\_ex;

            zero\_mem=zero\_ex;

            sml\_mem=sml\_ex;

            aluResult\_mem=aluResult\_ex;

            busB\_mem=busB\_ex;

            rw\_mem=rw\_ex;

            rt\_mem=rt\_ex;

            instru\_memAddress=instru\_exAddress;

            jumpAddress\_mem=jumpAddress\_ex;

            imm16\_mem=imm16\_ex;

            lb\_mem=lb\_ex;

            lbu\_mem=lbu\_ex;

            sb\_mem=sb\_ex;

            link\_mem=link\_ex;

            busA\_mem=busA\_ex;

        end

    end

    always @(posedge clk) begin

        if(regReset) begin

            branch\_mem=0;

            j\_mem=0;

            mem2Reg\_mem=0;

            regWr\_mem=0;

            memWr\_mem=0;

            //rtype\_mem=rtype\_id;

            overflow\_mem=0;

            zero\_mem=0;

            sml\_mem=0;

            busB\_mem=0;

            rw\_mem=0;

            jumpAddress\_mem=0;

            imm16\_mem=0;

            rt\_mem=0;

        end

    end

endmodule

4.1.5 MEMWRReg

module MEMWRReg\_320(

    input clk,

    input regReset,//是否出现跳转，出现了就全清零

    //控制信号输入

    input mem2Reg\_mem,

    input regWr\_mem,

    input link\_mem,

    input overflow\_mem,

    //input           rtype\_mem,

    //数据输入

    input[31:0] aluResult\_mem,

    input[31:0] data\_in,

    input[4:0]  rw\_mem,

    input[31:0] instru\_memAddress,

    //控制信号输出

    output reg mem2Reg\_wr,

    output reg regWr\_wr,

    output reg link\_wr,

    //output reg      rtype\_wr

    //数据输出

    output reg overflow\_wr,

    output reg[31:0] aluResult\_wr,

    output reg[31:0] data\_out,

    output reg[4:0]  rw\_wr,

    output reg[31:0] instru\_wrAddress

);

initial begin

    mem2Reg\_wr=0;

    regWr\_wr=0;

    overflow\_wr=0;

    aluResult\_wr=0;

    data\_out=0;

    rw\_wr=0;

end

    always @(negedge clk) begin

        if(~regReset) begin

            overflow\_wr=overflow\_mem;

            mem2Reg\_wr=mem2Reg\_mem;

            //rtype\_wr=rtype\_mem;

            link\_wr=link\_mem;

            regWr\_wr=regWr\_mem;

            aluResult\_wr=aluResult\_mem;

            data\_out=data\_in;

            rw\_wr=rw\_mem;

            instru\_wrAddress=instru\_memAddress;

        end

        else begin

            regWr\_wr=0;

        end

    end

endmodule

4.1.6 forwarding

//数据转发控制单元

module Forwarding\_Unit\_302(

    input[4:0]     rw\_mem,

    input[4:0]     rw\_wr,

    input[4:0]     rs\_ex,

    input[4:0]     rt\_ex,

    input   aluSrc\_ex,

    input   regWr\_mem,

    input   regWr\_wr,

    input   mem2Reg\_wr,

    input[1:0] bubble,

    output reg[1:0]   aluAchoose,

    output reg[1:0]   aluBchoose

);

    always @(\*) begin

    //转发给ALU A端

        // 相邻依赖

        if(rw\_mem == rs\_ex && regWr\_mem && rw\_mem != 5'd0 ) begin

            aluAchoose = 2'b01;

        end

        //隔一条指令

        else if(rw\_wr == rs\_ex && regWr\_wr && rw\_wr != 5'd0 ) begin

            aluAchoose = 2'b10;

        end

        // 无需转发

        else begin

            aluAchoose = 2'b00;

        end

    //aluA lw之后两条使用了lw的寄存器

        if ((rw\_wr == rs\_ex && regWr\_wr && rw\_wr != 5'd0 )&&mem2Reg\_wr) begin

            aluAchoose = 2'b11;

        end

    //ALU B端

        // 相邻指令

        if (aluSrc\_ex==0) begin

            if(rw\_mem == rt\_ex && regWr\_mem && rw\_mem != 5'd0 ) begin

                aluBchoose = 2'b01;

                //$display("regWrMem:%d",regWr\_mem);

            end

            // 隔一条指令

            else if(rw\_wr == rt\_ex && regWr\_wr && rw\_wr != 5'd0 ) begin

                aluBchoose= 2'b10;

            end

            //无需转发

            else begin

                aluBchoose = 2'b00;

            end

        end

        else begin

            aluBchoose=2'b00;

        end

    //ALUB load-use冒险，只要有阻塞并且符合相邻冒险，就说明是loaduse冒险

        if(bubble&&(rw\_mem == rt\_ex && regWr\_mem && rw\_mem != 5'd0 )) begin

                aluBchoose = 2'b11;

        end

    end

endmodule

4.1.7 load\_use

module load\_use320(

    input clk,

    input lw\_ex,

    input lb\_ex,

    input lbu\_ex,

    input[4:0] rs\_id,

    input[4:0] rt\_id,

    input [4:0] rt\_ex,

    input [5:0] usebranch\_mem,

    input [3:0] j\_mem,

    input[1:0] bubble,

    output reg[1:0] bubble\_new

);

initial begin

    bubble\_new=0;

end

always@(negedge clk) begin

    if(bubble>0) bubble\_new=bubble-1;//bubble大于1就减少

    else if(lw\_ex && (rt\_ex==rs\_id || rt\_ex == rt\_id)) begin //发生了loaduse冲突

        if (~(j\_mem || usebranch\_mem)) begin

            bubble\_new=1;

        end

    end

end

endmodule

4.1.8 control\_hazar

module Control\_hazar320(

    input clk,

    input [3:0] j\_mem,

    input  usebranch\_mem,

    input[5:0] branch\_mem,

    input predict,

    output reg regReset

);

initial begin

    regReset=0;

end

always @(negedge clk) begin

    if(regReset==0&&(j\_mem || branch\_mem&&((usebranch\_mem&&predict==0) || (usebranch\_mem==0&&predict)) ) ) begin //j型 或者 预测不跳转但是跳转了 或者 预测跳转但是没有跳转

        regReset=1;

        $display("handle jump and branch contrl");

    end

    else begin

        regReset=0;

    end

end

endmodule

4.1.9 predict

module predict\_320(

    input clk,

    input [5:0] branch\_mem,

    input usebranch\_mem,

    input predict,

    output reg predict\_new

);

initial begin

    predict\_new=0;

end

always@(negedge clk) begin

    if(usebranch\_mem) predict\_new=1;

    else if(branch\_mem&&usebranch\_mem==0) predict\_new=0;

end

endmodule

4.2.0 PC

`timescale 1ns / 1ns

module PC\_320(

input clk,

input [1:0] bubble,

input branch\_id,

input [15:0]imm16\_id,

input usebranch\_mem,

input regReset,

input [5:0] branch\_mem,

input [3:0] j\_mem,

input zero\_mem,

input sml\_mem,

input [15:0] imm16\_mem,

input [25:0] target\_mem,

input [31:0] jumpRigister,

input [31:0] pcbefore,

input predict,

output reg [31:0] pc

);

//wire [31:0] pcOut;

//PC\_320 pc(clk,pc,pcOut);

//InstructionMemory\_320 inmem(pc,insturction);

wire [31:0] imm32\_id;

wire [31:0] imm32\_mem;

wire [31:0] pcAdd;

EXT\_320 ext16\_mem(imm16\_mem,1'b1,imm32\_mem);

EXT\_320 ext16\_id(imm16\_id,1'b1,imm32\_id);

initial begin

    pc=32'd0;

end

assign pcAdd=pc+4;

//计算下一条指令地址

always @(negedge clk && bubble==0 ) begin

    //pcAdd = pc + 32'd4;

    //$display("%b",useBranch);

    if(j\_mem!=0) begin//j跳转

        if(j\_mem[3]) begin

            pc={pcbefore[31:28],target\_mem[25:0],2'b00};

            $display("jump goto %h",pc);

        end

        else if(j\_mem[2]) begin

            pc={pcbefore[31:28],target\_mem[25:0],2'b00};

            $display("jal goto %h",pc);

        end

        else if(j\_mem[1]) begin

            pc=jumpRigister;

            $display("jr goto %h",pc);

        end

        else if(j\_mem[0]) begin

            pc=jumpRigister;

            $display("jalr goto %h",pc);

        end

    end

    //预判正确，继续预测

    else begin

        if(predict&&branch\_id) begin //预测要跳转:跳转信号为1且确实有跳转指令

            pc = pc + (imm32\_id<<2) ;

            $display("--------predict goto %h----------",pc);

        end

        else begin//预测不要跳转

            pc = pcAdd;

            if(branch\_id) begin

                $display("--------predict goto %h----------",pc);

            end

        end

    end

end

always @(posedge regReset && branch\_mem) begin //错误了并且真有跳转指令，排除j型引起的

        //处理预判错误的

        //$display("!!!%d %d %d %d",pc,pcbefore,imm32\_mem,pcbefore + (imm32\_mem<<2));

    if(pc-32'd36 !=pcbefore + (imm32\_mem<<2)&&usebranch\_mem ) begin //需要跳转并且预测错了需要更新

        pc = pcbefore + (imm32\_mem<<2);

        $display("handle wrong predict,should goto: %h",pc);

    end

    else begin

        pc = pcbefore+4;

        $display("handle wrong predict,should goto: %h",pc);

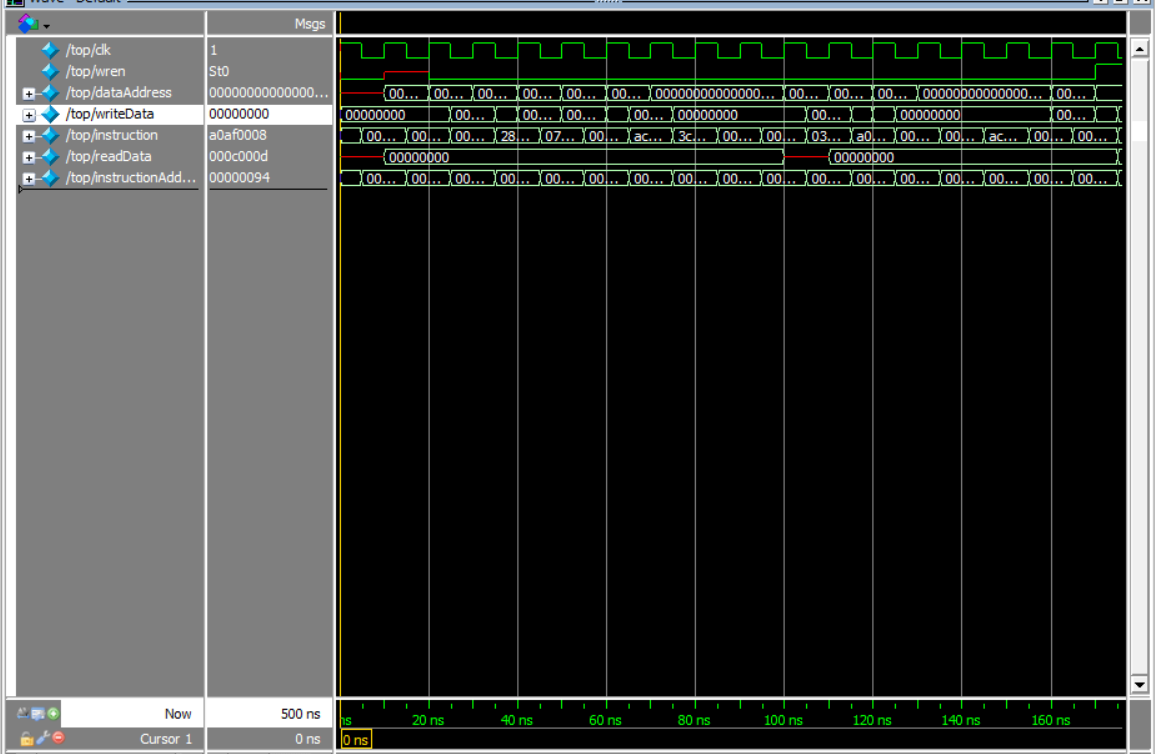
    end

end

endmodule

4.2 ModelSim仿真

4.2.1 运行结果：波形图



4.2.2 运行结果：打印信息

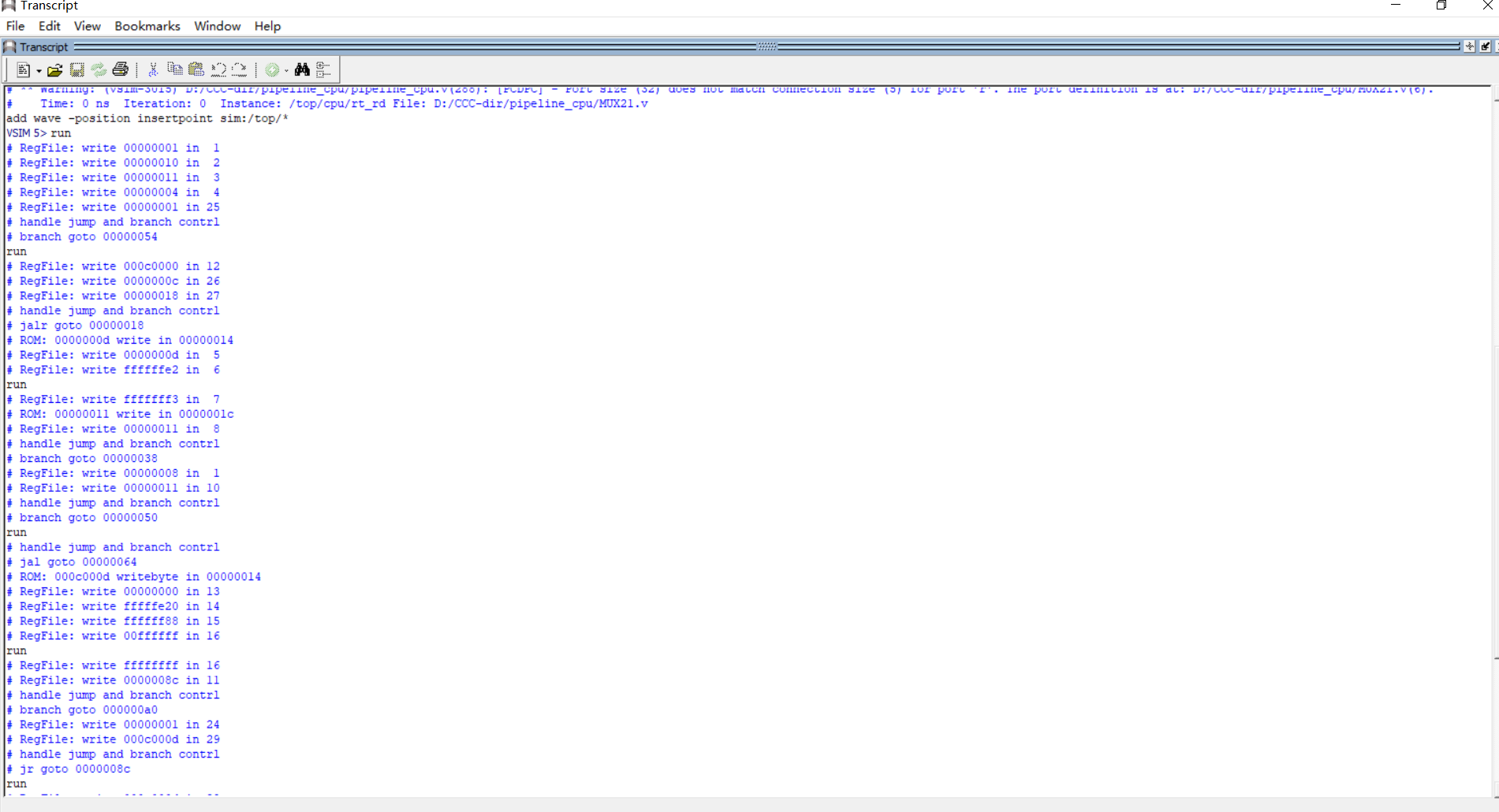
当发生下列情况是，程序会打印信息：

1.寄存器发生写入；

2.存储器发生写入；

3.发生条件转移指令；

4.发生跳转指令；



经比对，所有指令正确运行。

4.2.3 运行结果：写出txt

在“pipeline-cpu-out.txt”写出了相关信息，经比对，指令正确运行。

# 第五章 心得体会

流水线CPU的制作并没有像单周期那样轻松，一开始的无从下手，调试时的摸不着头脑，指令乱跳转的心急如焚……

直到最后抓住了流水线的特征，即分段传输信号，让不同部件可以同时工作。各种问题令人难以置信的迎刃而解，甚至处理后面的转发时，都没有耗费太大力气，所以我认为问题最关键抓住不同阶段的控制信号和数据是做什么的这件事。有的数据和信号是下个阶段马上要使用的，有的是要继续传递的，理清了这些，流水线的绝大部分问题都是可以耗费时间来攻克的。

我在写完单周期CPU以后觉得流水线也不过如此，妄图一口吃成大胖子，直接写完36条。遗憾的是完全做不到。之后我先完成了11条指令的流水线，没有转发，没有控制冒险，只能运行一条指令，再慢慢添加转发，解决load-use，解决分支和跳转带来的控制冒险。

惊喜的是当解决了11条以后，36条可以十分迅速地解决，甚至耗时不会超过五个小时，这一是因为单周期经历过11条扩展到36条的步骤，打下了基础；二是因为对流水线的理解有了提高。

单周期时我还是使用记事本写代码，到了流水线实在无法忍受没有关键词提示和代码高亮了，换了vscode。所以工欲善其事必先利其器。

最难的一个模块是分支预测，最后写出来的是一位预测位，效果还是不错的，就是光调试这个分支预测就花费了大半个下午，直到那天晚上才大功告成，宣告课设的结束。那天见不到月亮，我从巡天楼往外走，不由想到这也是最后一次在天目湖校区长时间的写代码和调试。呵，还是蛮有意思的。

总而言之，单周期和流水线全部写完了。这次课设还是收获蛮大的，包括如何沉下心来理解代码和调整bug。同时，做完的比较早，在解答他人的疑惑的过程中，也加深了自己的理解。做出个简单的cpu还是很有自豪感的。