**计算机组成课程设计**

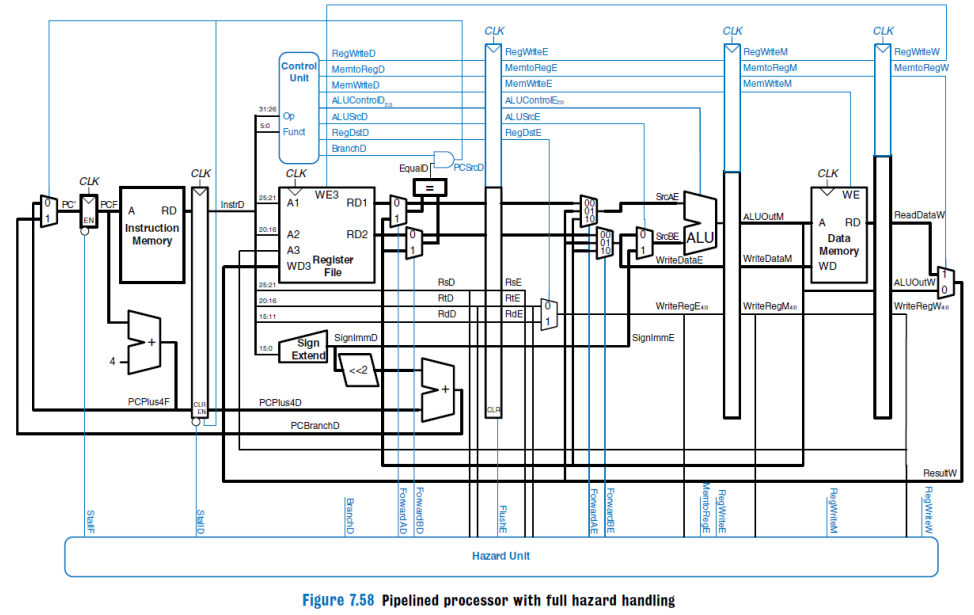
P5 课下测试 – Verilog流水线

周美廷-76066002

\*由于本人是留学生，随最终文档依然使用中文写但为了本人无需翻来覆去看英文版和中文版，于是将文档写成两种语言

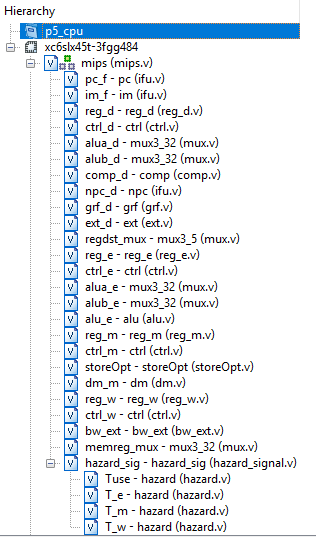
# **作业概括 (Homework Summary)**

在上次实验的基础上，使用Verilog语言设计一个流水线处理器。以下是流水线CPU的设计图。



**图1 流水线CPU设计图**

以下是本人在Verilog ISE中实现出来的顶层代码结构。



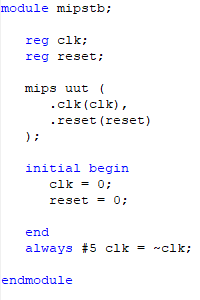
**图2 Verilog中的代码结构**

从**图 1**可看到mips.v是顶层代码，其负责连线任务，把相关模块连起来，自己只有两个输入。

**表1 mips.v规格**

|  |  |  |
| --- | --- | --- |
| 功能名称 | 方向 | 功能描述 |
| clk | I | 时钟信号 |
| reset | I | 复位信号 |

于是，testbench中只需要驱动clk和reset：



**图3 CPU的testbench代码**

# **模块规格 (Module Specifications)**

这种单周期CPU应采用模块化设计，因此它包含多个具有不同用途的模块。

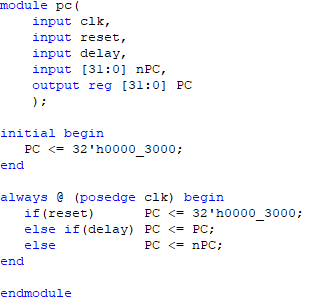
This single-cycle CPU is meant to be modularly designed and therefore it consists of several modules with different purposes.

## **IFU（取指令单元 / Instruction Fetch Unit）**

IFU模块分程3个主要部分，以下是详细细节：

The IFU module is divided into 3 main modules: PC (Program Counter), IM (Instruction Memory), and nPC (PC Destination).

### **PC (Program Counter)**



**图4 ifu.v中的PC模块**

**表2 ifu.v - PC规格**

|  |  |  |
| --- | --- | --- |
| 功能名称 | 方向 | 功能描述 |
| clk | I | 时钟信号 |
| reset | I | 复位信号 |
| delay | I | CPU暂停信号 |
| nPC [31:0] | I | 目标地址 |
| PC [31:0] | O | 当前程序地址 |

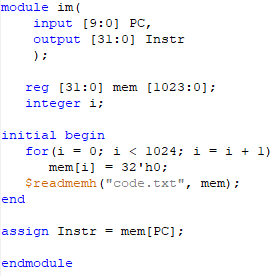
设计说明/Design Details:

* 复位后，PC指向0x0000\_3000，此处为第一条指令的地址。（After reset, PC will point to 0x0000\_3000 which is also the first instruction address）

模块流程/Design Workflow:

* PC’s starting address is 0x0000\_3000, which means if reset is valid, PC will point to this address
* If the CPU has to be paused (delay signal is TRUE), PC will point to itself, meaning that it won’t point to the next command address which will cause the CPU to not execute the next command until the CPU isn’t paused
* Otherwise PC will just go to the next destination (nPC).

### **IM**



**图5 ifu.v中的IM模块**

**表3 ifu.v - IM规格**

|  |  |  |
| --- | --- | --- |
| 功能名称 | 方向 | 功能描述 |
| PC [9:0] | I | PC 的第 [11:2] 地址 |
| Instr [31:0] | O | 当前指令 |

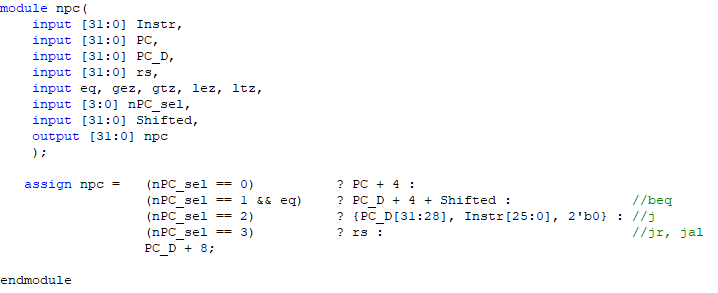
设计说明/Design Details:

* IM容量为4KB（32bit×1024字）（The capacity of IM is 4 KB or 32bit x 1024 word）
* 采用$readmemh指令来完成相应的功能 （This module is using $readmemh function to do required actions）

模块流程/Design Workflow:

* Set an initial value for the entire Instruction Memory as 0, meaning there are no instructions at start.
* Read the instructions from **code.txt** using **$readmemh** and save it to the IM

### **nPC**



**图6 ifu.v中的nPC模块**

**表4 ifu.v - nPC规格**

|  |  |  |
| --- | --- | --- |
| 功能名称 | 方向 | 功能描述 |
| Instr [31:0] | I | 当前指令 |
| PC [31:0] | I | F Level的PC地址 |
| PC\_D [31:0] | I | D Level的PC地址 |
| rs [31:0] | I | jr指令用来当跳转地址 |
| eq | I | 两个比较数据是否相等的信号 |
| nPC\_sel | I | 跳转指令的信号 |
| Shifted [31:0] | I | beq用的有符号移 |
| nPC [31:0] | O | 下一个目标地址 |

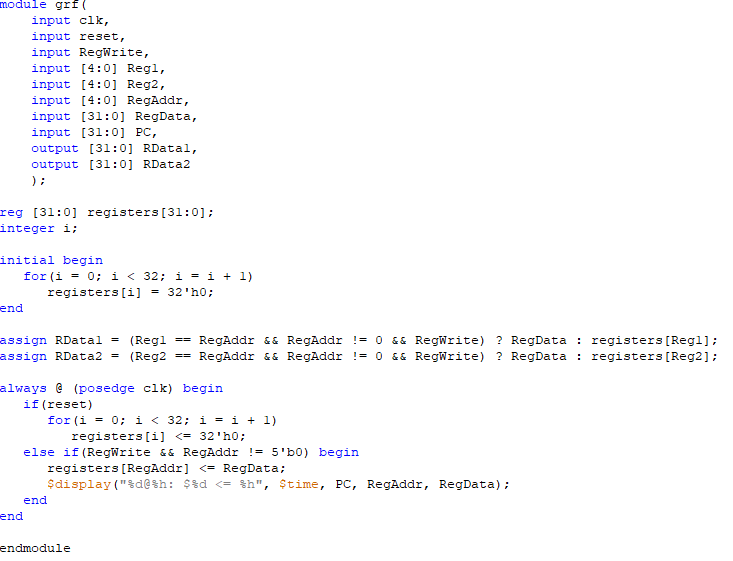
设计说明/Design Details:

* 这设计中有四个跳转指令会影响到PC的目标地址，**beq, jal, jr**, **j**（There are 4 jump instructions that can affect the PC address destination, **beq, jal, jr, j**）

模块流程/Design Workflow:

* Define the actions of different jump conditions, by default it’s the normal PC+4
* 2’b01 = beq | 2’b10 = jal/j | 2’b11 = jr

## **GRF（通用寄存器组/General Register File）**



**图7 grf.v模块**

**表5 grf.v规格**

|  |  |  |
| --- | --- | --- |
| 功能名称 | 方向 | 功能描述 |
| clk | I | 时钟信号 |
| reset | I | 复位信号 |
| RegWrite | I | 写控制信号 |
| Reg1 [4:0] | I | 读寄存器地址1 |
| Reg2 [4:0] | I | 读寄存器地址2 |
| RegAddr [4:0] | I | 目标写寄存器地址 |
| RegData [31:0] | I | 写入数据 |
| PC [31:0] | I | 当前PC地址 |
| RData1 [31:0] | O | 32位数据1 |
| RData2 [31:0] | O | 32位数据2 |

设计说明/Design Details:

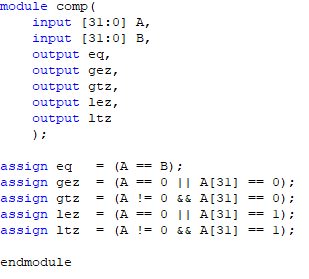
* 容量为**4KB** (32bit/word×1024word)。（The capacity of this module is 4KB which means 32bit/word×1024word）
* 0 号寄存器的值始终保持为 0。其他寄存器初始值均为 0，无需专门设置（The value of Register No. 0 is always 0. The other registers have an initial value of 0 so they don’t require special settings）
* 每个时钟上升沿到来时若要写入数据(即写使能信号为1且非reset时)则输出写入的位置及写入的值，格式为 : (At clock posedge, always display some data information using the following command)

**$display("%d@%h: $%d <= %h", $time, PC, RegAddr, RegData);**

模块流程/Design Workflow:

* In total, there are 5 different inputs in this GRF module:
* The first two inputs are taken from the instruction code generated by IFU (Address no. 25-21 (rs/base/offset) and 20-16 (rt))
* The third one is either Address no 20-16 (rt) in case of **ori, lw, sw, beq, lui** or 15-11 (rd) in case of **addu** and **subu**
* The fourth one is the RegWrite signal (this will be TRUE if the current instruction isn’t **sw, jr, j** nor **beq**)
* The fifth one is Register Data (RegData)
* At clock posedge, if RegWrite signal is TRUE and RegAddr isn’t 0, then write the RegData input to the corresponding register, after that display the information wanted.
* For RData1 and RData2, if it’s a write command (RegWrite == True), then both will be assigned the value of the write data (RegData), otherwise just output the data that is to be read.

## **Comp（数据比较单元/Data Comparator）**

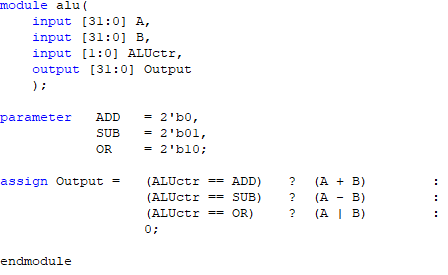


**图8 comp.v模块**

**表6 comp.v规格**

|  |  |  |
| --- | --- | --- |
| 功能名称 | 方向 | 功能描述 |
| A[31:0] | I | 输入数据A |
| B[31:0] | I | 输入数据B |
| eq | O | A == B |
| gez | O | A >= 0 |
| gtz | O | A > 0 |
| lez | O | A <= 0 |
| ltz | O | A < 0 |

## **ALU（算术逻辑单元/Arithmetic Logic Unit）**



**图9 alu.v模块**

**表7 alu.v规格**

|  |  |  |
| --- | --- | --- |
| 功能名称 | 方向 | 功能描述 |
| A[31:0] | I | 输入数据A |
| B[31:0] | I | 输入数据B |
| ALUctr[1:0] | I | ALU控制信号  00：加法运算  01：减法运算  10：或运算  11：比较 |
| Output[31:0] | O | 输出结果32位 |

ALU is the most common module, this module is meant to perform arithmetic operations such as addition, subtraction, comparation, etc.

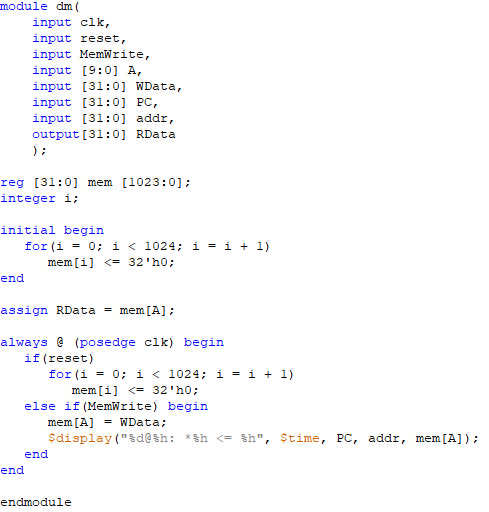
设计说明/Design Details:

* 提供 32 位加、减、或运算及大小比较功能（Provides 32-bit addition, subtraction, OR operation and size comparison）
* 可以不支持溢出（不检测溢出）（Overflow may not be supported (no overflow detected)）
* ALU 的输出不允许直接作为转发输入，只能是 ALUOutM (The Output of ALU module shouldn’t be a direct input, it should be brought to the M level register first and come out as ALUOutM (or in my case, it’s called AO\_M))

模块流程/Design Workflow:

* According to the instruction that’s through ALUctr input (this instruction is generated by the Controller module), the module will perform corresponding operation. The **ori** instruction will cause ALU to perform OR operation (10), the **subu** will cause ALU to perform subtraction operation (01), and the other instructions will simply cause ALU to perform addition operation.
* At the same time, both data inputs A and B will be checked, if the values are equal, then the Zero signal will be TRUE. The value of A and B will be decided based on what level it is currently on, and the signal of the forward module.

## **DM（数据存储器/Data Memory）**



**图10 dm.v模块**

**表8 dm.v规格**

|  |  |  |
| --- | --- | --- |
| 功能名称 | 方向 | 功能描述 |
| clk | I | 时钟信号 |
| reset | I | 复位信号 |
| MemWrite | I | 读写控制信号 |
| A [9:0] | I | MemAddr地址，为ALU结果的[11:2] |
| WData [31:0] | I | 输入数据32位 |
| PC [31:0] | I | 当前PC地址 |
| Addr [31:0] | I | 当前DM指定的地址 |
| RData [31:0] | O | 输出结果32位 |

The DM module is meant to save data generated from previous operations.

设计说明/Design Details:

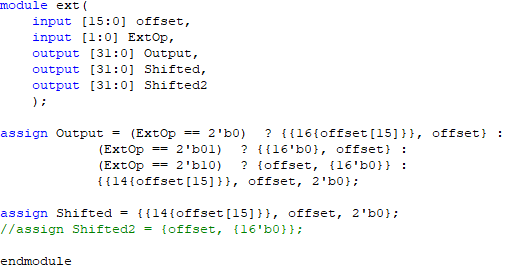
* DM容量为4KB（32bit×1024字）（The capacity of DM is 4 KB or 32bit x 1024 word）
* 每个时钟上升沿到来时若要写入数据(即写使能信号为1且非reset时)则输出写入的位置及写入的值，格式为 : (At clock posedge, always display some data information using the following command)

**$display("%d@%h: \*%h <= %h", $time, PC, addr, mem[A]);**

模块流程/Design Workflow:

* This is just a simple module to store data to the Data Memory, whether the CPU will write the data is decided by the MIPS instruction sent to MemWrite. If the instruction is **sw**, MemWrite will be TRUE, therefore storing the data to the DM.
* If MemWrite is FALSE, the module would simply read a data from the current pointed address.

## **EXT（Extender）**



**图11 ext.v模块**

**表8 ext.v规格**

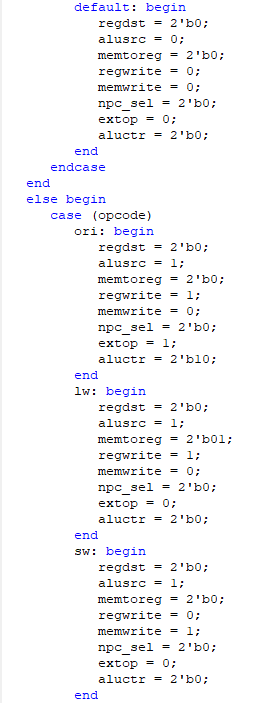
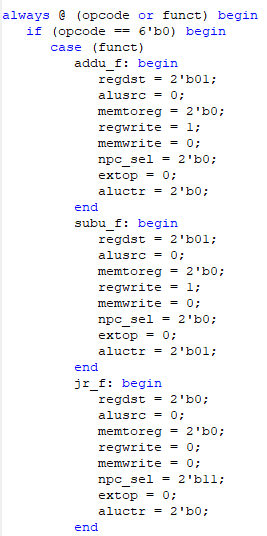
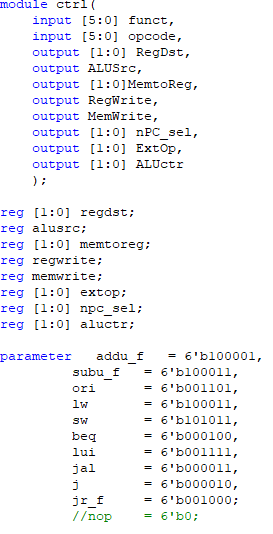
|  |  |  |
| --- | --- | --- |
| 功能名称 | 方向 | 功能描述 |
| offset [15:0] | I | 将被extended的地址 |
| extop | I | Extender功能信号 |
| Output [31:0] | O | 基本extender输出 |
| Shifted [31:0] | O | 已被移的地址输出 |

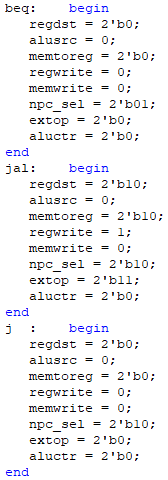
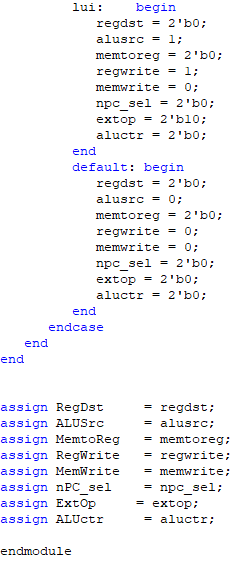
设计说明/Design Details:

模块流程/Design Workflow:

* If extop is 0, thus FALSE, then the module will perform **signed-extend**, if it’s 1 it will perform **zero-extend**, if it’s 2 it will perform a simple **zero-shift**, meaning that the first 16bit data will be shifted to the upper half while the lower half will then be filled with 0s
* Shifted is used for beq destination jump address (sign\_extend (offset||02))

## **Controller（控制器）**



**图12 控制器模块**

**表9 控制器规格**

|  |  |  |
| --- | --- | --- |
| 功能名称 | 方向 | 功能描述 |
| funct [5:0] | I | Function 6位 |
| opcode [5:0] | I | Opcode 6位 |
| RegDst | O | 写地址控制 |
| ALUSrc | O | 控制ALU |
| MemtoReg [1:0] | O | DM读控制 |
| RegWrite | O | GRF读写控制 |
| MemWrite | O | DM写控制 |
| nPC\_sel [1:0] | O | 跳转指令标志 |
| ExtOp [1:0] | O | Ext扩展控制 |
| ALUctr[1:0] | O | ALU运算操作控制 |

模块流程/Design Workflow:

* First, define all the instruction codes set as a static constant, in Verilog this kind of code has a type called **parameter**
* The input of this module is the 5-0 and 31-26 address bit of the instruction code generated from IFU. While most MIPS instruction opcode are located in the 31-26 bit of the instruction code, it works differently for **addu** and **subu**. The opcode for both **addu** and **subu** are 000000 but they got function code in the 5-0 bit of the instruction code. The complete code is listed in the table below.
* RegDst means that the register that will be used is the rd register, this is used by **addu** and **subu** while the other instructions will just use the rt register.
* MemtoReg literally means loading data from the memory to the register. Of course, this signal will be true if the instruction code is a loading instruction such as **lw** or **lui**
* RegWrite signal is to let the CPU write the data into a register. This will be used by instructions that cause changes to data so that it needs to be written (saved) in the register such as **lui, ori, lw, addu, subu, jal**
* MemWrite is a signal to let the CPU write the data into memory. This is only used by instructions that will store data such as **sw**
* The nPC\_sel signal basically is just a bool to see if it’s a branch instruction. In this project there are three branch instructions **beq, jal, jr, j**, therefore it’s a 2 bit signal. This signal will then be used in the IFU module to decide whether to use PC+4 or PC+X or X for the address movements

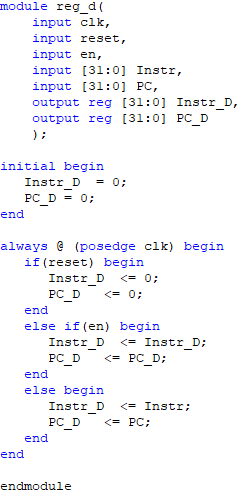
**表10 控制器指令真值表**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| funct | 100001 | 100011 | n/a | | | | | | | 001000 | 000000 |
| opcode | 000000 | 000000 | 001101 | 100011 | 101011 | 000100 | 001111 | 000011 | 000010 | 000000 | 000000 |
|  | addu | subu | ori | lw | sw | beq | lui | jal | j | jr | nop |
| RegDst | 01 | 01 | 00 | 00 | 00 | 00 | 00 | 10 | 00 | 00 | x |
| ALUSrc | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | x |
| MemtoReg[1:0] | 00 | 00 | 00 | 01 | xx | xx | 10 | 11 | 00 | 00 | xx |
| RegWrite | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| MemWrite | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| nPC\_sel | 00 | 00 | 00 | 00 | 00 | 01 | 00 | 10 | 10 | 11 | 0 |
| ExtOp | x | x | 1 | 0 | 0 | x | 0 | x | 00 | x | x |
| ALUctr[1:0] | Add | Subtract | Or | Add | Add | xx | xx | xx | xx | xx | xx |

# **数据通路（Data Path）**

Since this is a pipeline-CPU, the process of the CPU will be divided into 5 big parts: F, D, E, M, W. These represents different operations:

## **F Level**

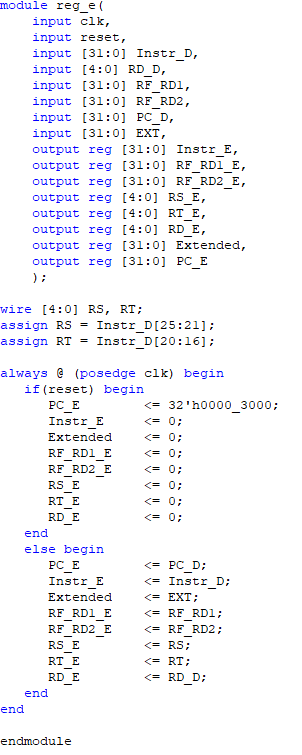


**图13 reg\_d代码**

### **Included Modules**

* PC : Decide whether the CPU should read the next instruction or not, if yes, which
* IM : Read and save command instructions
* Reg\_D : Prepare register values for the D Level

## **D Level**

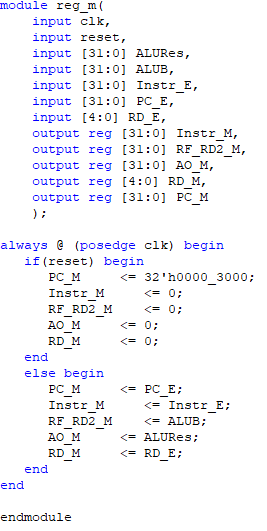


**图14 reg\_d代码**

### **Included Modules**

* CTRL : Get the controller signals to use in the D level
* NPC : Decide if it’s a jump instruction and decide the next instruction address
* GRF : Write data to register if needed so
* EXT : Extend data
* Mux : Multiplexer to decide which data will be compared for the **zero** signal
* ALU : Compare data if they equal and then set the **zero** signal
* Reg\_E : Prepare register values for the E Level

## **E Level**

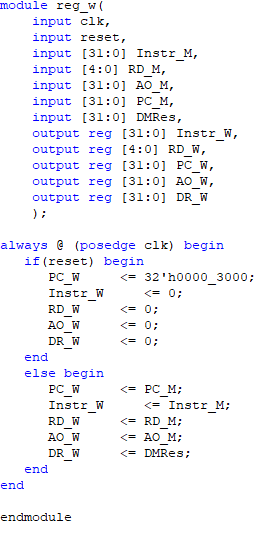


**图15 reg\_m代码**

### **Included Modules**

* CTRL : Get the controller signals to use in the E level
* Mux : Multiplexer to decide which data will be used for arithmetic operations
* Mux2 : Multiplexer to decide which data will be used for the RegAddr for use when it reaches the GRF module in the D Level
* ALU : Execute arithmetic operations
* Reg\_M : Prepare register values for the M Level

## **M Level**



**图16 reg\_w代码**

### **Included Modules**

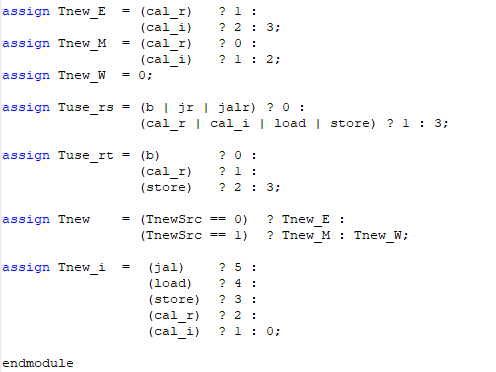
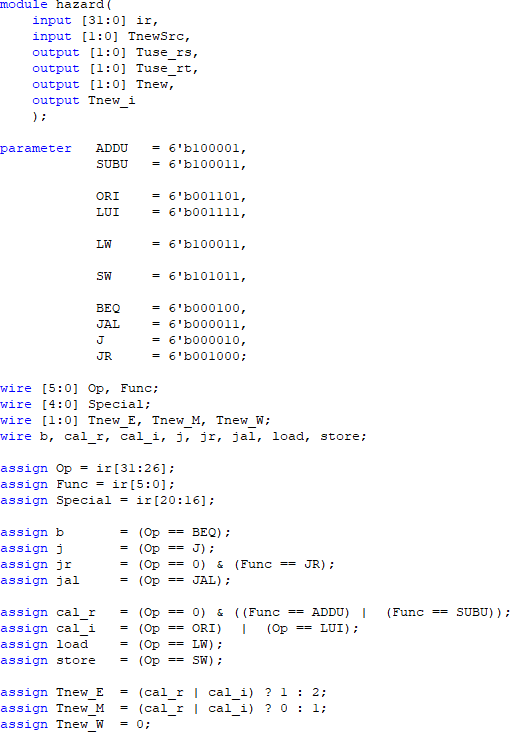
* CTRL : Get the controller signals to use in the M level
* DM : Writes data to memory if needed so
* Reg\_W : Prepare register values for the W Level

## **W Level**

### **Included Modules**

* CTRL : Get the controller signals to use in the W level
* Mux : Multiplexer to decide which data will be used to be written in the register in the GRF module of the D Level

## **Hazard Control**



**图17 hazard.v模块**

首先，我们把各种指令根据它们对寄存器堆的读取行为进行分类，同一类指令的工作流程是相似的。

* Cal\_r类（R-R，寄存器与寄存器进行计算）：add, addu, subu等。
* Cal\_i类（R\_I，寄存器与常数进行计算）：addi, lui, ori等。
* Beq类（通过CMP比较器判断跳转）：beq, bne, bgez等。
* Load类（读取内存的值）：lw, lb, lbu, lh等。
* Save类（保存值到内存）：sw, sb, sh等。
* J类的四条指令各自都有微妙差别，推荐四条指令分开处理。

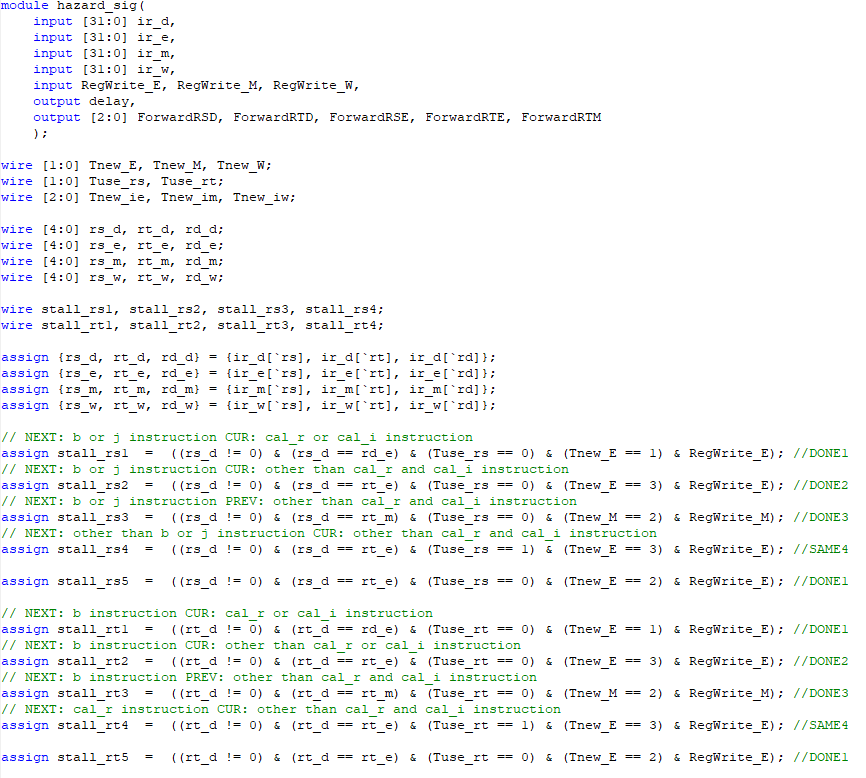
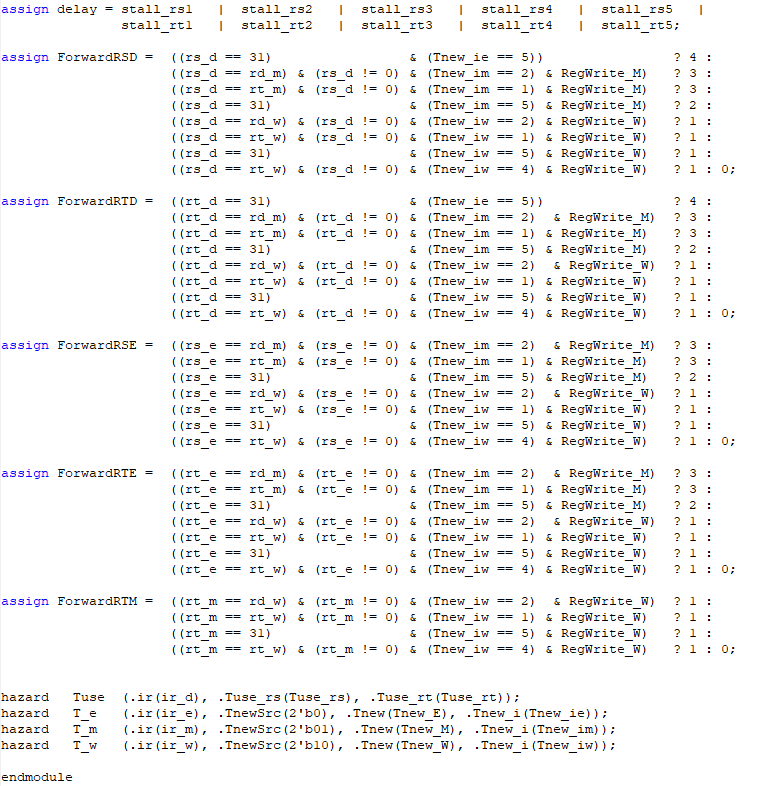
其中，只有cal\_r, cal\_i, load三类指令和jal, jalr会产生结果。

简单来说，就是遇到需要读取寄存器堆的指令时，暂停信号会变TRUE，让CPU暂停以下，先不要执行下一个指令而先让整个CPU走到最后的W级，读取数据然后在继续执行。

当判定需要暂停的时候，我们需要执行三个操作：

* 冻结PC，不让PC的值改变
* 让ID/EX流水级寄存器清零，等于插入了一个NOP指令
* 冻结IF/ID流水级寄存器，不让它的值改变。

转发的原理，是后面的指令需要用到前面指令的运算结果，而后面指令在执行时，前面指令的运算结果还没有来得及写入寄存器堆。则此时，我们增加若干条数据通路，把前面指令的运算结果直接传递回来，达到不用暂停也能规避风险的目的。

**图18 hazard\_sig.v模块**

**具体实现过程：**

* 把指令分成几个分类：cal\_r, cal\_i, b, j, jr, jal, jalr, load, store
* cal\_r 指令都是用rd而cal\_i和load使用rt，判断冲突时，需要针对正确的指令和寄存器，不能大致判断（并且寄存器当时并非为0，说明确实发生有效冲突）
* 定义Tuse\_rs和Tuse\_rt，两个分别是代表rs和rt寄存器是否要被新的指令使用。对于Tuse\_rs，若新的指令是cal\_r | cal\_i | load | store那就是会使用rs

对于Tuse\_rt，若新的指令是cal\_r信号为1，store信号为2

* 然后定义几个暂停的信号，本人的程序中有8种：

NEXT: D Level, CUR: E Level, PREV: M Level

对于rs寄存器的情况

* Stall\_rs1: NEXT: b | j CUR: cal\_r
* Stall\_rs2: NEXT: b | j CUR: ~(cal\_r | cal\_i)
* Stall\_rs3: NEXT: b | j PREV: ~(cal\_r | cal\_i)
* Stall\_rs4: NEXT: ~(b | j) CUR: ~(cal\_r | cal\_i)
* Stall\_rs5: NEXT: b | j CUR: cal\_i

对于rt寄存器的情况

* Stall\_rt1: NEXT: b CUR: cal\_r
* Stall\_rt2: NEXT: b CUR: ~(cal\_r | cal\_i)
* Stall\_rt3: NEXT: b PREV: ~(cal\_r | cal\_i)
* Stall\_rt4: NEXT: cal\_r CUR: ~(cal\_r | cal\_i)
* Stall\_rt5: NEXT: b CUR: cal\_i
* 之所以把Stall\_rs1和Stall\_rs5以及Stall\_rt1和Stall\_rt5分开，是因为cal\_r和cal\_i使用不同寄存器的缘故
* 发现以上10种情况中的任何一个都要发出暂停的信号，意味着需要先让那些需要把数据写到寄存器上的指令把数据写完再继续执行下一个指令，以免发生冲突。

转发系统使用Mux部件来进行的。

## **The Data Path Details**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **部件** | **输入** | **lw** | **sw** | **addu** | **subu** | **ori** | **beq** | **j** | **jal** | **jr** | **lui** |
| PC |  |  |  |  |  |  |  |  |  |  |  |
| IM |  | PC | PC | PC | PC | PC | PC | PC | PC | PC | PC |
|  |  |  |  |  |  |  |  |  |  |  |  |
| IR\_D |  | IM | IM | IM | IM | IM | IM | IM | IM | IM | IM |
| PC\_D |  |  | | | | | PC4 | PC4 | PC4 | PC4 | PC4 |
|  |  |  |  |  |  |  |  |  |  |  |  |
| NPC | PC |  | | | | | PC\_D | PC\_D | PC\_D | PC\_D | PC\_D |
| Instr |  | | | | | IR\_D | IR\_D | IR\_D | IR\_D | IR\_D |
| GRF | Reg1 | IR\_D [rs] | IR\_D [rs] | IR\_D [rs] | IR\_D [rs] | IR\_D [rs] | IR\_D [rs] |  | | IR\_D [rs] | IR\_D [rs] |
| Reg2 |  | | IR\_D [rt] | IR\_D [rt] |  | IR\_D [rt] |  | | | |
| EXT |  | IR\_D [16] | IR\_D [16] |  | | IR\_D [16] | IR\_D [16] |  | | | IR\_D [16] |
| CMP | D1 |  | | | | | RD1 |  | | | |
| D2 |  | | | | | RD2 |  | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |
| Instr\_E |  | IR\_D | IR\_D | IR\_D | IR\_D | IR\_D | IR\_D | IR\_D | IR\_D | IR\_D | IR\_D |
| PC\_E |  |  | | | | | PC\_D | PC\_D | PC\_D | PC\_D | PC\_D |
| RS\_E |  | RD1 | RD1 | RD1 | RD1 | RD1 | RD1 |  | | | |
| RT\_E |  |  | RD2 | RD2 | RD2 | RD2 | RD2 |  | | | |
| EXTE |  | EXT | EXT |  | | EXT | EXT |  | | | EXT |
|  |  |  |  |  |  |  |  |  |  |  |  |
| ALU | A | RS\_E | RS\_E | RS\_E | RS\_E | RS\_E | RS\_E |  | | | |
| B | EXTE | EXTE | RT\_E | RT\_E | EXTE | RT\_E |  | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |
| Instr\_M |  | IR\_E | IR\_E | IR\_E | IR\_E | IR\_E | IR\_E | IR\_E | IR\_E | IR\_E | IR\_E |
| PC\_M |  |  | | | | PC\_E | PC\_E | PC\_E | PC\_E | PC\_E | PC\_E |
| AO\_M |  | ALU | ALU | ALU | ALU | ALU |  | | | |  |
| RT\_M |  |  | RT\_E |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| DM | A | AO\_M | AO\_M |  | | | | | | | |
| WD |  | RD\_M |  | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |
| Instr\_W |  | IR\_M |  | IR\_M | IR\_M | IR\_M |  | | | | IR\_M |
| PC\_W |  |  | | | | | PC\_W | PC\_W | PC\_W | PC\_W | PC\_W |
| AO\_W |  |  | | AO\_M | AO\_M | AO\_M |  | | | | |
| DR\_W |  | DM |  | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |
| GRF | RegA | IR\_W [rt] |  | IR\_W [rd] | IR\_W [rd] | IR\_W [rt] |  | | | | IR\_W [rt] |
| RegD | DR\_W |  | AO\_W | AO\_W | AO\_W |  | | | | |

# **测试程序说明 (Testing Program)**

测试指令集：

.data

testarr: .space 8

# addu, subu, ori, lui, lw, sw, beq, jal, jr, j, nop

.text

ori $gp, $zero, 0x0000

ori $sp, $zero, 0x0000

ori $at, $zero, 0x3456

addu $at, $at, $at

lw $at, 0x0004

sw $at, 0x0004

lui $v0, 0x7878

subu $v1, $v0, $at

lui $a1, 0x1234

ori $a0, $zero, 0x0005

nop

lui $v0, 0x8723

nop

ori $a3, $v1, 0x0404

nop

lui $t0, 0x7777

addu $at, $at, $v0

lw $at, 0x0000

ori $t0, $t0, 0xffff

addu $t1, $at, $v1

subu $zero, $zero, $t0

ori $zero, $zero, 0x1100

lw $t2, 0x0000

addu $t2, $a3, $t2

ori $at, $a3, 0x1010

sw $at, 0x0000

ori $t0, $zero, 0x0000

ori $t1, $zero, 0x0001

ori $t2, $zero, 0x0001

addu $t0, $t0, $t2

subu $zero, $0, $t0

nop

lui $t0, 0x1234

ori $t1, $t0, 0x0001

lui $t0, 0x0000

ori $t1, $t0, 0x0000

nop

nop

addu $t2, $t1, 3

subu $t3, $t1, $t2

nop

sw $t3, testarr($zero)

beq $t3, $t2, any

lw $t3, testarr($zero)

nop

any: ori $t0, $t0, 0x1010

jal procedure

lui $t0, 0x5678

jal end

procedure:

ori $t1, $t0, 0x0009

jr $ra

end:

机器码：

341c0000

341d0000

34013456

00210821

8c010004

ac010004

3c027878

00411823

3c051234

34040005

00000000

3c028723

00000000

34670404

00000000

3c087777

00220821

8c010000

3508ffff

00234821

00080023

34001100

8c0a0000

00ea5021

34e11010

ac010000

34080000

34090001

340a0001

010a4021

00080023

00000000

3c081234

35090001

3c080000

35090000

00000000

00000000

3c010000

34210003

01215021

012a5823

00000000

ac0b0000

116a0002

8c0b0000

00000000

35081010

0c000c33

3c085678

0c000c35

35090009

03e00008

# **思考题**

1. 在本实验中你遇到了哪些不同指令组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？请有条理的罗列出来。(非常重要)

以下是测试过程中考虑到有可能发生冲突的指令组合。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No. | 测试类型 | 前序指令 | 冲突位置 | 冲突寄存器 | 测试用例 |
| 1 | R-M-RS | addu | MEM | rs | addu $t0, $t0, $t2  subu $zero, $0, $t0 |
| 2 | R-M-RT | addu | MEM | rt | addu $t2, $t1, 3  subu $t3, $t1, $t2 |
| 3 | R-W-RS | addu | MEM | rs | addu $at, $at, $v0  ~~ori $t0, $t0, 0xffff~~  addu $t1, $at, $v1 |
| 4 | R-W-RT | addu | MEM | rt | addu $at, $at, $at  ~~lw $at, 0x0004~~  ~~sw $at, 0x0004~~  ~~lui $v0, 0x7878~~  subu $v1, $v0, $at |
| 5 | I-M-RS | ori | MEM | rs | ori $at, $zero, 0x3456  addu $at, $at, $at |
| 6 | I-M-RT | lui | MEM | rt | lui $t0, 0x7777  ori $t0, $t0, 0xffff |
| 7 | I-W-RS | ori | MEM | rs | ori $zero, $zero, 0x1100  ~~addu $t2, $a3, $a2~~  ori $t0, $zero, 0x0000 |
| 8 | I-W-RT | lui | MEM | rt | lui $t0, 0x7777  ~~ori $t0, $t0, 0xffff~~  subu $zero, $zero, $t0 |
| 9 | ~~LD-M-RS~~ | ~~lw~~ | ~~MEM~~ | ~~rs~~ |  |
| 10 | LD-M-RT | lw | MEM | rt | lw $t2, 0x0000  addu $t2, $a3, $t2 |
| 11 | LD-W-RS | lw | MEM | rs | lw $at, $v0  ~~ori $t0, $t0, 0xffff~~  addu $t1, $at, $v1 |
| 12 | LD-W-RT | lw | MEM | rt | lw $at, 0x0004  ~~sw $at, 0x0004~~  ~~lui $v0, 0x7878~~  subu $v1, $v0, $at |

**解决方案：**

* 把指令分成几个分类：cal\_r, cal\_i, b, j, jr, jal, jalr, load, store
* 定义Tuse\_rs和Tuse\_rt，两个分别是代表rs和rt寄存器是否要被新的指令使用。对于Tuse\_rs，若新的指令是cal\_r | cal\_i | load | store那就是会使用rs

对于Tuse\_rt，若新的指令是cal\_r信号为1，store信号为2

* 然后定义几个暂停的信号，本人的程序中有8种：

NEXT: D Level, CUR: E Level, PREV: M Level

对于rs寄存器的情况

* Stall\_rs1: NEXT: b | j CUR: cal\_r
* Stall\_rs2: NEXT: b | j CUR: ~(cal\_r | cal\_i)
* Stall\_rs3: NEXT: b | j PREV: ~(cal\_r | cal\_i)
* Stall\_rs4: NEXT: ~(b | j) CUR: ~(cal\_r | cal\_i)
* Stall\_rs5: NEXT: b | j CUR: cal\_i

对于rt寄存器的情况

* Stall\_rt1: NEXT: b CUR: cal\_r
* Stall\_rt2: NEXT: b CUR: ~(cal\_r | cal\_i)
* Stall\_rt3: NEXT: b PREV: ~(cal\_r | cal\_i)
* Stall\_rt4: NEXT: cal\_r CUR: ~(cal\_r | cal\_i)
* Stall\_rt5: NEXT: b CUR: cal\_i
* 之所以把Stall\_rs1和Stall\_rs5以及Stall\_rt1和Stall\_rt5分开，是因为cal\_r和cal\_i使用不同寄存器的缘故
* 发现以上10种情况中的任何一个都要发出暂停的信号，意味着需要先让那些需要把数据写到寄存器上的指令把数据写完再继续执行下一个指令，以免发生冲突。