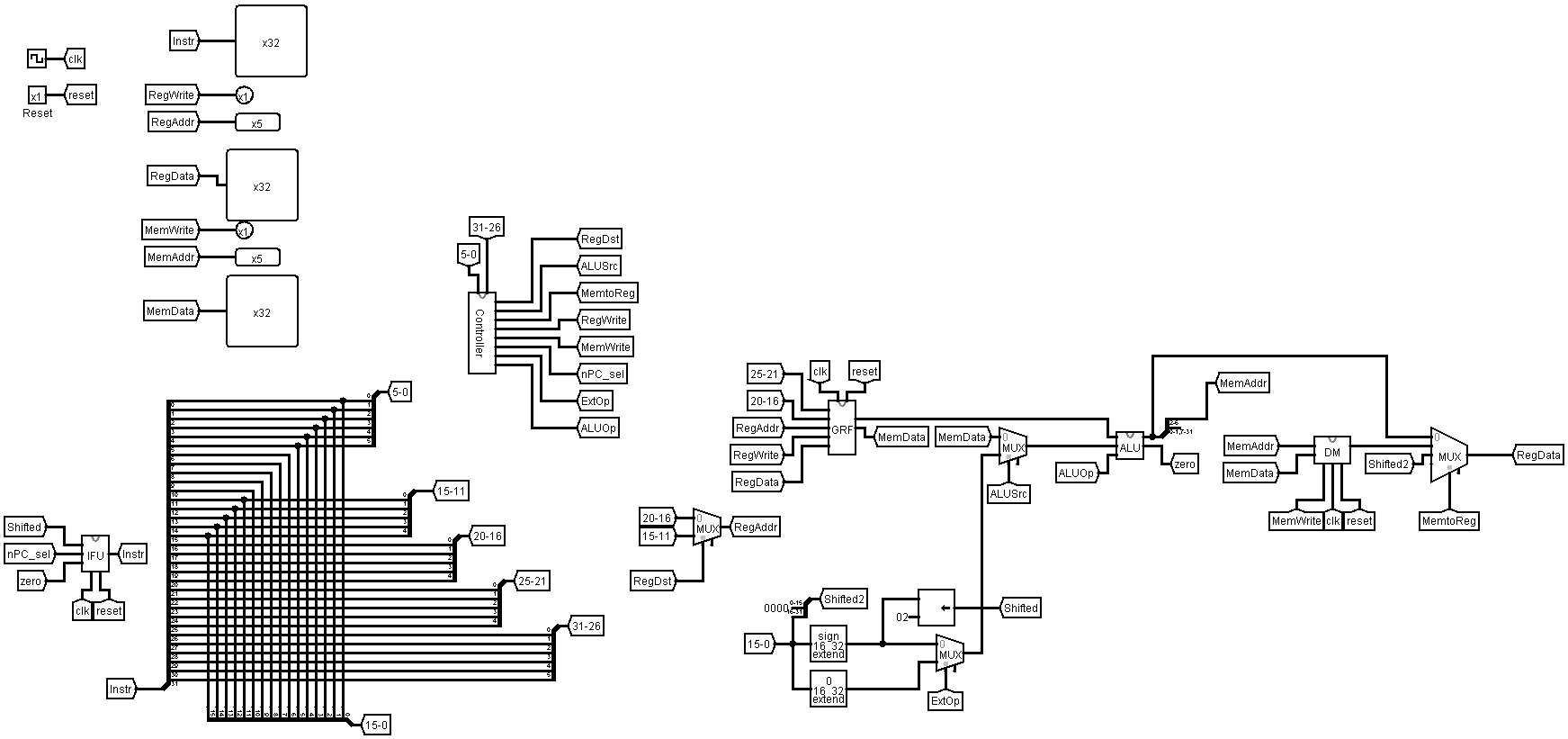
**计算机组成课程设计**

P4 课下测试 – Verilog单周期

周美廷-76066002

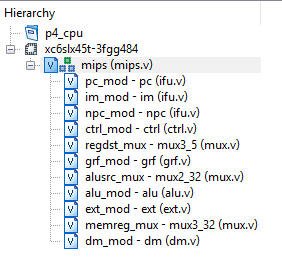
\*由于本人是留学生，随最终文档依然使用中文写但为了本人无需翻来覆去看英文版和中文版，于是将文档写成两种语言

# **顶层设计 (The Basic Idea)**

****以下图片便是本人在Logisim设计出的CPU（Below is the CPU design that I implemented in Logisim）

**图1 Logisim中的顶层设计**

以下是本人在Verilog ISE中按照Logisim中的设计实现出来的顶层代码（mips.v）。



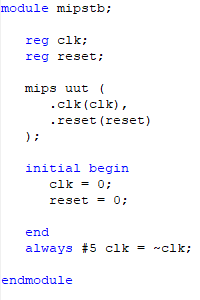
**图2 Verilog中的代码结构**

从**图 2**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。

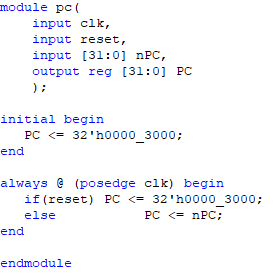
This single-cycle CPU is meant to be modularly designed and therefore it consists of several modules with different purposes.

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

与logisim中的设计有所不同，verilog的设计中本人把IFU模块分程3个主要部分，以下是详细细节：

A little bit different from the design in Logisim, in this design, 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 |
| nPC [31:0] | I | 目标地址 |
| PC [31:0] | O | 当前程序地址 |

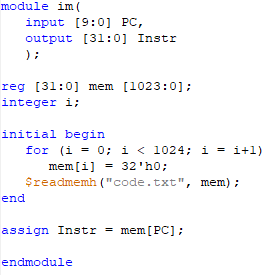
设计说明/Design Details:

* 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, 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 |

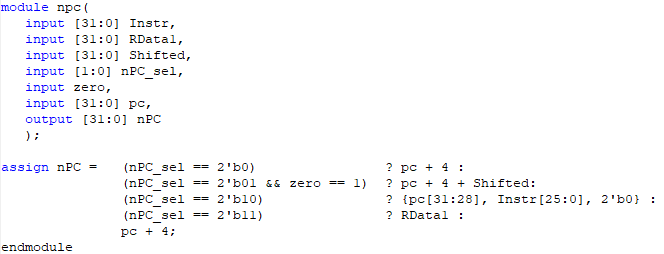
设计说明/Design Details:

* 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 |
| RData1 [31:0] | I | jr指令用来当跳转地址 |
| Shifted [31:0] | I | beq用的有符号移 |
| nPC\_sel | I | 跳转指令的信号 |
| zero | I | 两个比较数据是否为零的信号 |
| PC [31:0] | I | 当前的PC地址 |
| nPC [31:0] | O | 下一个目标地址 |

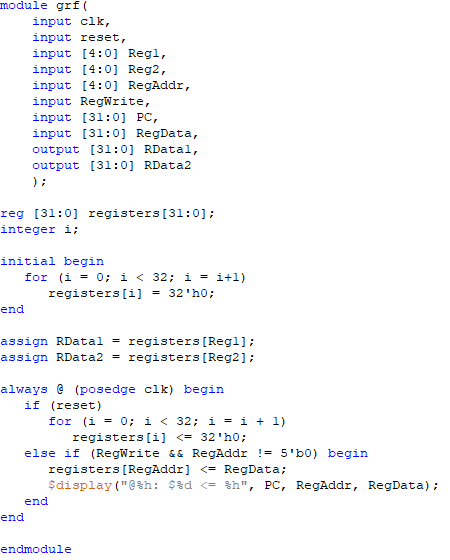
设计说明/Design Details:

* 这设计中有三个跳转指令会影响到PC的目标地址，**beq, jal,** jr（There are 3 jump instructions that can affect the PC address destination, **beq, jal, jr**）

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 | 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 |
| 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 |
| RegWrite | I | 写控制信号 |
| PC [31:0] | I | 当前PC地址 |
| RegData [31:0] | I | 写入数据 |
| RData1 [31:0] | O | 32位数据1 |
| RData2 [31:0] | O | 32位数据2 |

设计说明/Design Details:

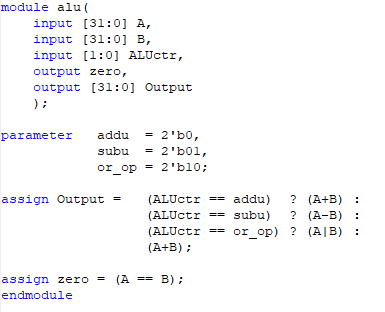
* 具有32个32位的寄存器（This module has 32 32bit registers）
* 0 号寄存器的值始终保持为 0。其他寄存器初始值均为 0，无需专门设置（The value of Register No. 0 is always 0. The other registers have an initial value of 0 so they don’t require special settings）
* 每个时钟上升沿到来时若要写入数据(即写使能信号为1且非reset时)则输出写入的位置及写入的值，格式为 : (At clock posedge, always display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

**$display("@%h: $%d <= %h", 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** 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.

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



**图8 alu.v模块**

**表6 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 |
| zero | O | 计算结果是否为0 |
| 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.

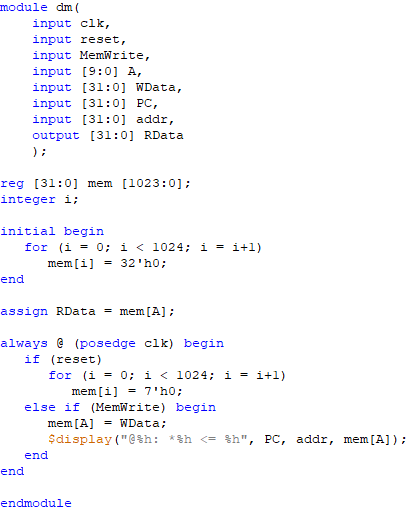
设计说明/Design Details:

* 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)）

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

## **DM（数据存储器/Data Memory）**



**图9 dm.v模块**

**表7 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:

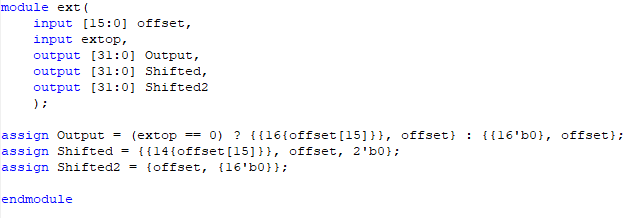
* DM容量为4KB（32bit×1024字）（The capacity of DM is 4 KB or 32bit x 1024 word）
* 每个时钟上升沿到来时若要写入数据(即写使能信号为1且非reset时)则输出写入的位置及写入的值，格式为 : (At clock posedge, always display some data information using the following command)

**$display("@%h: \*%h <= %h", 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.
* For later uses, the data will be chosen by the input A, this input is generated by the ALU module, which means if the operation doesn’t include any data that were stored to the RAM, the output of this module would be 0

## **EXT（Extender）**



**图10 ext.v模块**

**表7 ext.v规格**

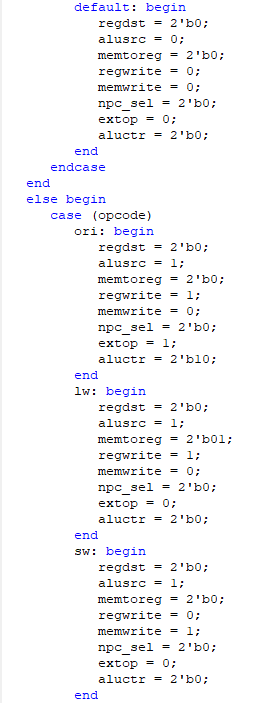
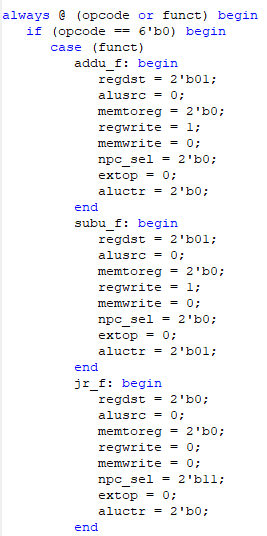
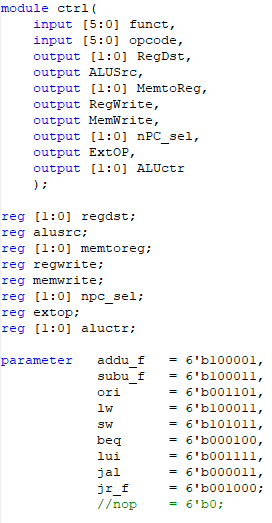
|  |  |  |
| --- | --- | --- |
| 功能名称 | 方向 | 功能描述 |
| offset [15:0] | I | 将被extended的地址 |
| extop | I | Extender功能信号 |
| Output [31:0] | O | 基本extender输出 |
| Shifted [31:0] | O | 已被移的地址输出1 |
| Shifted2 [31:0] | O | 已被移的地址输出2 |

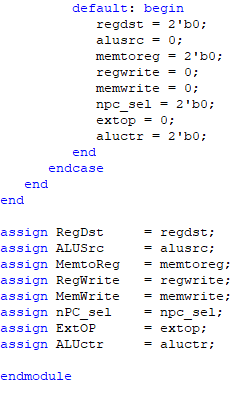
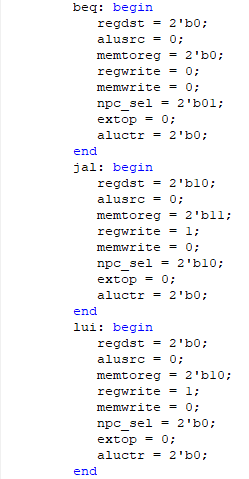
设计说明/Design Details:

模块流程/Design Workflow:

* If extop is 0, thus FALSE, then the module will perform **signed-extend**, otherwise it will perform **zero-extend**
* Shifted is used for beq destination jump address (sign\_extend (offset||02))
* Shifted2 is 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

## **Controller（控制器）**





**图11 控制器模块**

**表8 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 | 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**, 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9 控制器指令真值表**

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

# **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, 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

nop

ori $a3, $v1, 0x0404

nop

lui $t0, 0x7777

ori $t0, $t0, 0xffff

subu $zero, $zero, $t0

ori $zero, $zero, 0x1100

addu $t2, $a3, $a2

ori $t0, $zero, 0x0000

ori $t1, $zero, 0x0001

ori $t2, $zero, 0x0001

addu $t0, $t0, $t2

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

00000000

34670404

00000000

3c087777

3508ffff

00080023

34001100

00e65021

34080000

34090001

340a0001

010a4021

00000000

3c081234

35090001

3c080000

35090000

00000000

00000000

3c010000

34210003

01215021

012a5823

00000000

ac0b0000

116a0002

8c0b0000

00000000

35081010

0c000c2b

3c085678

0c000c2d

35090009

03e00008

# **思考题**

## **(L0.T2)**

1. 根据你的理解，在下面给出的DM的输入示例中，地址信号addr位数为什么是[11:2]而不是[9:0]？这个addr信号又是从哪里来的？



因为数据大小为4字节，地址中的 [1:0] 自然可以忽略掉，因为两位数最高是11，也就是3，而我们的数据是4

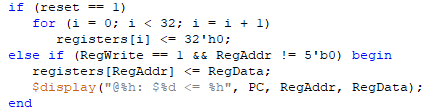
1. 在相应的部件中，reset的优先级比其他控制信号（不包括clk信号）都要高，且相应的设计都是同步复位。清零信号reset是针对哪些部件进行清零复位操作？这些部件为什么需要清零？

清零指令是用来清零DM中的数据，以及GRF中的寄存器数据。复位的指望效果是整个程序从头开始（PC指向第一个指令），而默认时，全部寄存器或存储器的初始值是0，若程序已经走到一半然后从头开始却没清零那些已存下来的数据整个程序计算结果都会有误的。

## **(L0.T4)**

1. 列举出用Verilog语言设计控制器的几种编码方式（至少三种），并给出代码示例。

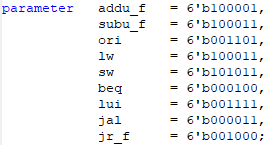
* If-else (或case)



* Assign语句



* 宏定义



1. 根据你所列举的编码方式，说明他们的优缺点。

* If-else

优点：可以参入比较复杂的条件

缺点：如果只是一个简单的条件转换if-else会比较麻烦

* Case

优点：用起来比较干净简易

缺点：不能参入复杂的条件

* Assign

优点：

缺点：

* Parameter

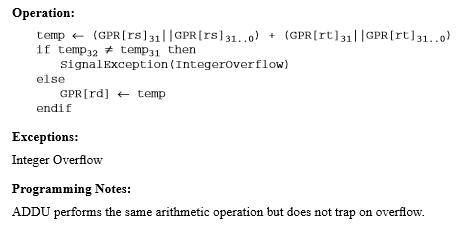
优点：可以统一一次性定义判断值

缺点：

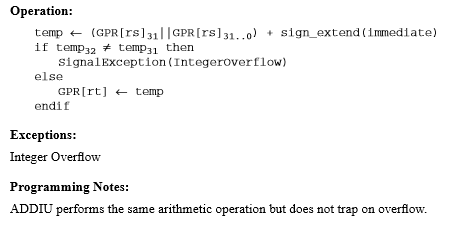
## **(L0.T5)**

1. C语言是一种弱类型程序设计语言。C语言中不对计算结果溢出进行处理，这意味着C语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持C语言，MIPS指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi与addiu是等价的，add与addu是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的Operation部分 。

Add:



Addi:



上面的两个指令说明中已经说明了

if temp32 temp31 then

SignalException(IntegerOverflow)

else

GPR[rt] 🠠 temp

endif

也就是说，在add和addi支持溢出的情况下，如果发现计算中有溢出，就会拦住这溢出，若没溢出就正常地把计算结果存下来。addu和addiu本身都不支持溢出，也不会发现溢出，其他操作都与add和addi一样，若add和addi中没有溢出识别那他们的操作基本都是一模一样，结果自然是等价。

1. 根据自己的设计说明单周期处理器的优缺点。

优点：设计起来相当比较简单，因为它就只有单周期的处理，并不需要考虑太多情况，全部数据会同时被使用

缺点：若程序比较大设计的工作量比较复杂因为很多数据有可能会混乱，而且因为全部数据同时使用会导致资源浪费

1. 简要说明jal、jr和堆栈的关系。