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

1. 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）

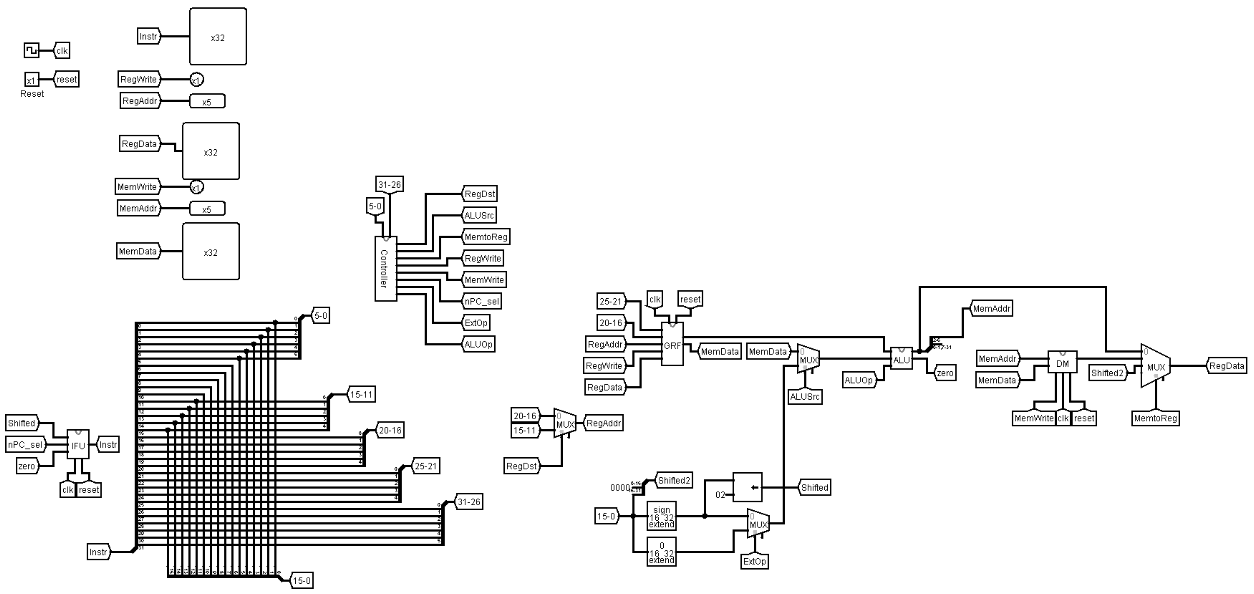
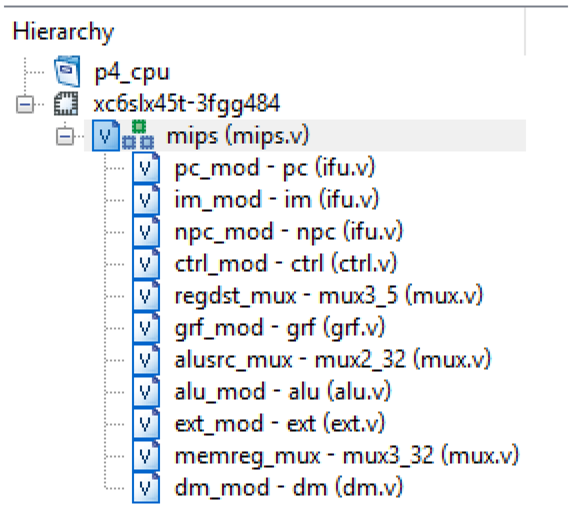


图 1 Logisim 中的顶层设计

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）。



```
p4_cpu
xc6slx45t-3fgg484
  mips (mips.v)
    pc_mod - pc (ifu.v)
    im_mod - im (ifu.v)
    npc_mod - npc (ifu.v)
    ctrl_mod - ctrl (ctrl.v)
    regdst_mux - mux3_5 (mux.v)
    grf_mod - grf (grf.v)
    alusrc_mux - mux2_32 (mux.v)
    alu_mod - alu (alu.v)
    ext_mod - ext (ext.v)
    memreg_mux - mux3_32 (mux.v)
    dm_mod - dm (dm.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：

```
module mipstb;

   reg clk;
   reg reset;

   mips uut (
      .clk(clk),
      .reset(reset)
   );

   initial begin
      clk = 0;
      reset = 0;

   end
   always #5 clk = ~clk;

endmodule
```

图 3 CPU 的 testbench 代码

2. 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。

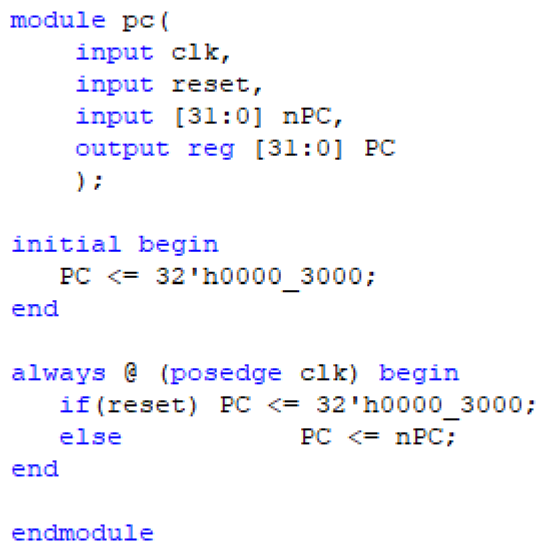
This single-cycle CPU is meant to be modularly designed and therefore it consists of several modules with different purposes.

I. 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).

A. PC (Program Counter)



```
module pc(
    input clk,
    input reset,
    input [31:0] nPC,
    output reg [31:0] PC
    );

initial begin
   PC <= 32'h0000_3000;
end

always @ (posedge clk) begin
   if(reset) PC <= 32'h0000_3000;
   else          PC <= nPC;
end

endmodule
```

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

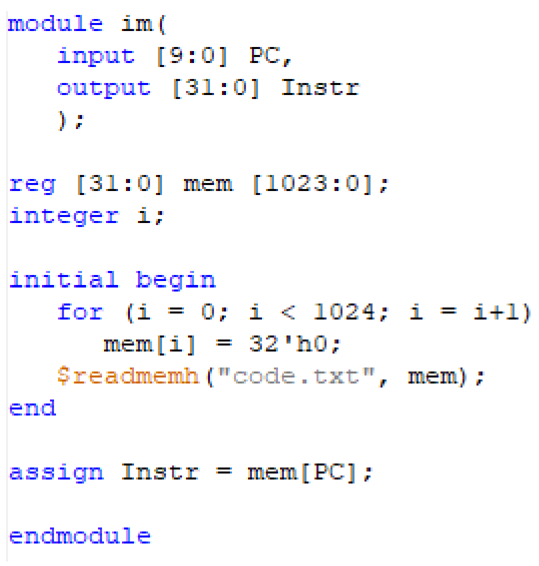
设计说明/Design Details:

- 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).

B. IM



```
module im(
   input [9:0] PC,
   output [31:0] Instr
   );

reg [31:0] mem [1023:0];
integer i;

initial begin
   for (i = 0; i < 1024; i = i+1)
      mem[i] = 32'h0;
   $readmemh("code.txt", mem);
end

assign Instr = mem[PC];

endmodule
```

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

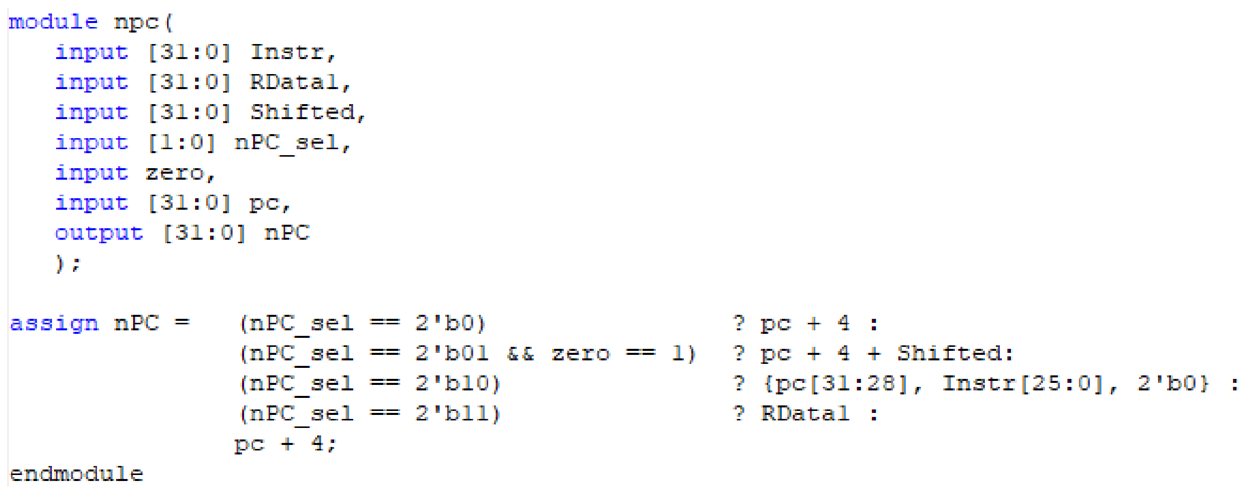
设计说明/Design Details:

- 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

C. nPC



```
module npc(
   input [31:0] Instr,
   input [31:0] RData1,
   input [31:0] Shifted,
   input [1:0] nPC_sel,
   input zero,
   input [31:0] pc,
   output [31:0] nPC
   );

assign nPC =    (nPC_sel == 2'b0)                 ? pc + 4 :
                (nPC_sel == 2'b01 && zero == 1)   ? pc + 4 + Shifted:
                (nPC_sel == 2'b10)                ? {pc[31:28], Instr[25:0], 2'b0} :
                (nPC_sel == 2'b11)                ? RData1 :
               pc + 4;
endmodule
```

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

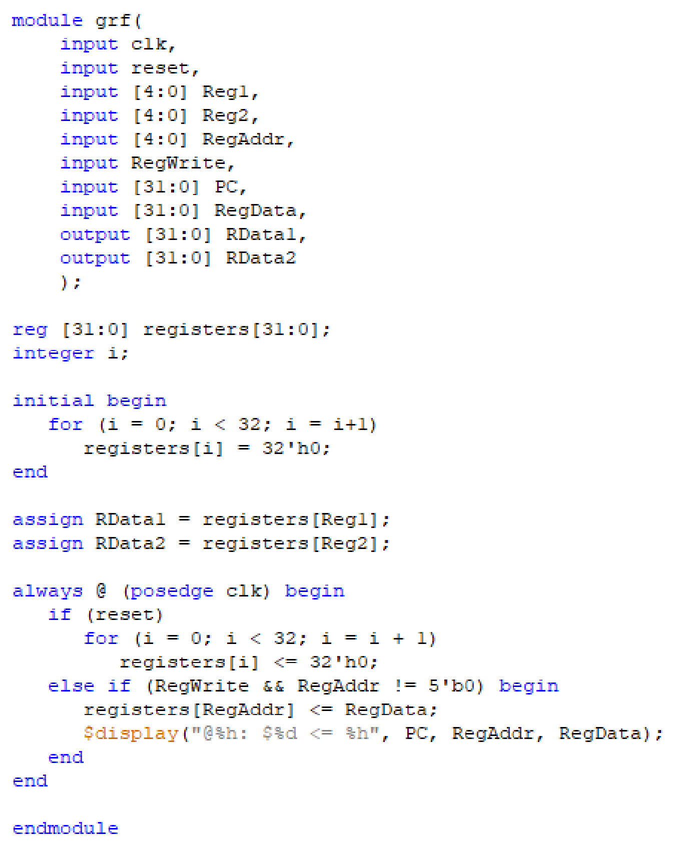
设计说明/Design Details:

- 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

II.　GRF（通用寄存器组/General Register File）



```
module grf(
    input clk,
    input reset,
    input [4:0] Reg1,
    input [4:0] Reg2,
    input [4:0] RegAddr,
    input RegWrite,
    input [31:0] PC,
    input [31:0] RegData,
    output [31:0] RData1,
    output [31:0] RData2
    );

reg [31:0] registers[31:0];
integer i;

initial begin
   for (i = 0; i < 32; i = i+1)
      registers[i] = 32'h0;
end

assign RData1 = registers[Reg1];
assign RData2 = registers[Reg2];

always @ (posedge clk) begin
   if (reset)
      for (i = 0; i < 32; i = i + 1)
         registers[i] <= 32'h0;
   else if (RegWrite && RegAddr != 5'b0) begin
      registers[RegAddr] <= RegData;
      $display("@%h: $%d <= %h", PC, RegAddr, RegData);
   end
end

endmodule
```

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

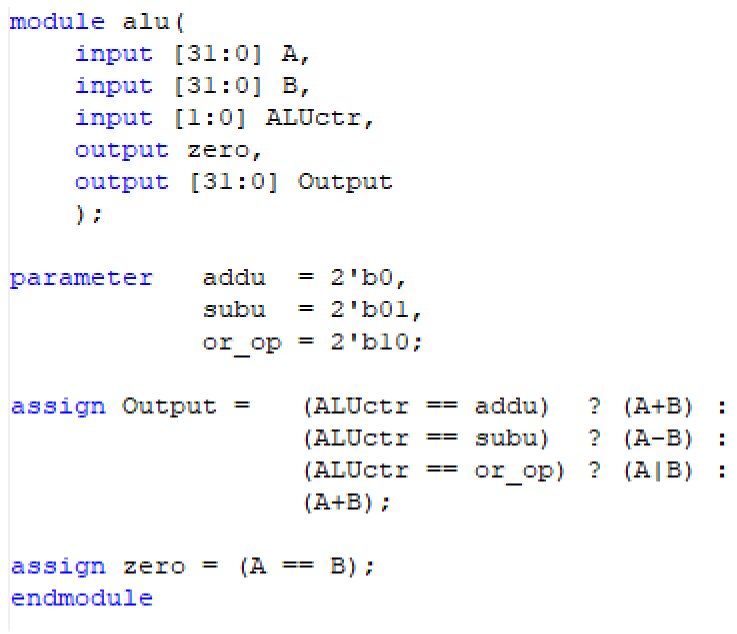
设计说明/Design Details:

- 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.

III. ALU（算术逻辑单元/Arithmetic Logic Unit）



```
module alu(
    input [31:0] A,
    input [31:0] B,
    input [1:0] ALUctr,
    output zero,
    output [31:0] Output
    );

parameter   addu  = 2'b0,
            subu  = 2'b01,
            or_op = 2'b10;

assign Output =   (ALUctr == addu)  ? (A+B) :
                  (ALUctr == subu)  ? (A-B) :
                  (ALUctr == or_op) ? (A|B) :
                  (A+B);

assign zero = (A == B);
endmodule
```

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.

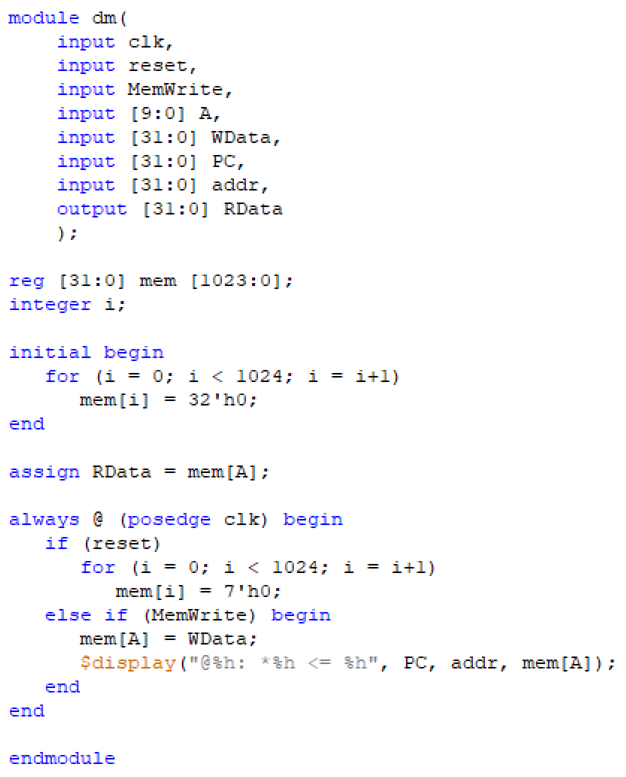
设计说明/Design Details:

- 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

IV. DM（数据存储器/Data Memory）



```
module dm(
    input clk,
    input reset,
    input MemWrite,
    input [9:0] A,
    input [31:0] WData,
    input [31:0] PC,
    input [31:0] addr,
    output [31:0] RData
    );

reg [31:0] mem [1023:0];
integer i;

initial begin
   for (i = 0; i < 1024; i = i+1)
      mem[i] = 32'h0;
end

assign RData = mem[A];

always @ (posedge clk) begin
   if (reset)
      for (i = 0; i < 1024; i = i+1)
         mem[i] = 7'h0;
   else if (MemWrite) begin
      mem[A] = WData;
      $display("@%h: *%h <= %h", PC, addr, mem[A]);
   end
end

endmodule
```

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.

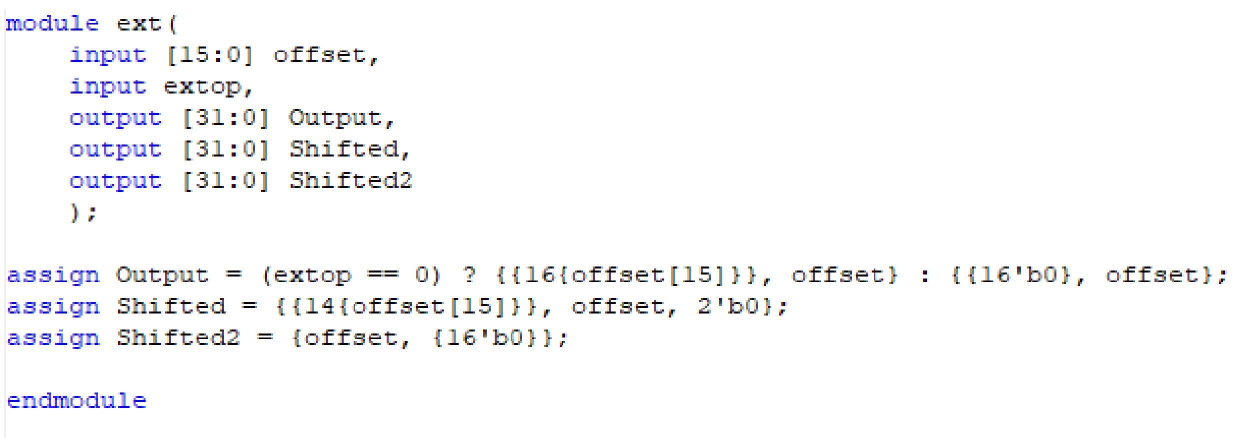
设计说明/Design Details:

- 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

V. EXT（Extender）



```
module ext(
    input [15:0] offset,
    input extop,
    output [31:0] Output,
    output [31:0] Shifted,
    output [31:0] Shifted2
    );

assign Output = (extop == 0) ? {{16{offset[15]}}, offset} : {{16'b0}, offset};
assign Shifted = {{14{offset[15]}}, offset, 2'b0};
assign Shifted2 = {offset, {16'b0}};

endmodule
```

图 10 ext.v 模块

表 7 ext.v 规格

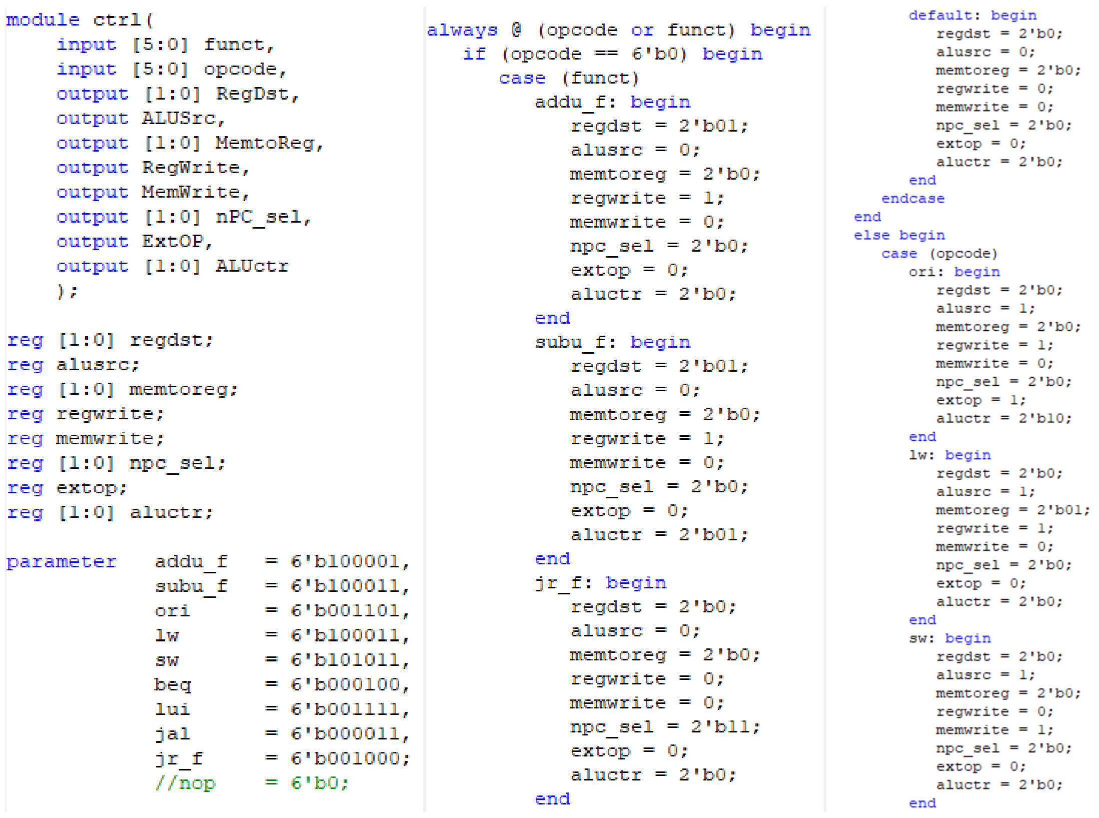
功能名称	方向	功能描述
offset [15:0]	I	将被 extended 的地址
extop	I	Extender 功能信号
Output [31:0]	O	基本 extender 输出
Shifted [31:0]	O	已被移的地址输出 1
Shifted2 [31:0]	O	已被移的地址输出 2

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||0^2))
- 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

VI.   Controller（控制器）



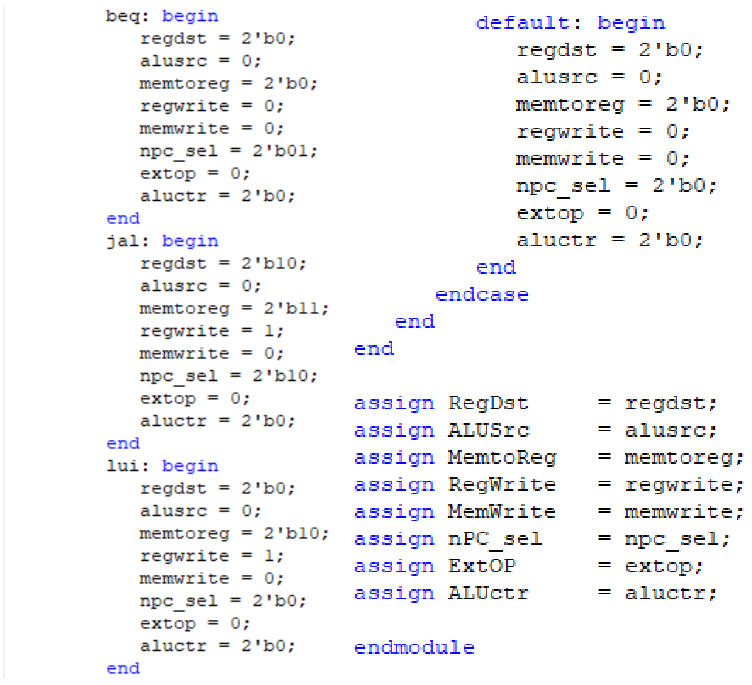
```
module ctrl(
    input [5:0] funct,
    input [5:0] opcode,
    output [1:0] RegDst,
    output ALUSrc,
    output [1:0] MemtoReg,
    output RegWrite,
    output MemWrite,
    output [1:0] nPC_sel,
    output ExtOP,
    output [1:0] ALUctr
    );

reg [1:0] regdst;
reg alusrc;
reg [1:0] memtoreg;
reg regwrite;
reg memwrite;
reg [1:0] npc_sel;
reg extop;
reg [1:0] aluctr;

parameter    addu_f    = 6'b100001,
             subu_f    = 6'b100011,
             ori       = 6'b001101,
             lw        = 6'b100011,
             sw        = 6'b101011,
             beq       = 6'b000100,
             lui       = 6'b001111,
             jal       = 6'b000011,
             jr_f      = 6'b001000,
             //nop     = 6'b0;
```

```
always @ (opcode or funct) begin
   if (opcode == 6'b0) begin
      case (funct)
         addu_f: begin
            regdst = 2'b01;
            alusrc = 0;
            memtoreg = 2'b0;
            regwrite = 1;
            memwrite = 0;
            npc_sel = 2'b0;
            extop = 0;
            aluctr = 2'b0;
         end
         subu_f: begin
            regdst = 2'b01;
            alusrc = 0;
            memtoreg = 2'b0;
            regwrite = 1;
            memwrite = 0;
            npc_sel = 2'b0;
            extop = 0;
            aluctr = 2'b01;
         end
         jr_f: begin
            regdst = 2'b0;
            alusrc = 0;
            memtoreg = 2'b0;
            regwrite = 0;
            memwrite = 0;
            npc_sel = 2'b11;
            extop = 0;
            aluctr = 2'b0;
         end
```

```
         default: begin
            regdst = 2'b0;
            alusrc = 0;
            memtoreg = 2'b0;
            regwrite = 0;
            memwrite = 0;
            npc_sel = 2'b0;
            extop = 0;
            aluctr = 2'b0;
         end
      endcase
   end
   else begin
      case (opcode)
         ori: begin
            regdst = 2'b0;
            alusrc = 1;
            memtoreg = 2'b0;
            regwrite = 1;
            memwrite = 0;
            npc_sel = 2'b0;
            extop = 1;
            aluctr = 2'b10;
         end
         lw: begin
            regdst = 2'b0;
            alusrc = 1;
            memtoreg = 2'b01;
            regwrite = 1;
            memwrite = 0;
            npc_sel = 2'b0;
            extop = 0;
            aluctr = 2'b0;
         end
         sw: begin
            regdst = 2'b0;
            alusrc = 1;
            memtoreg = 2'b0;
            regwrite = 0;
            memwrite = 1;
            npc_sel = 2'b0;
            extop = 0;
            aluctr = 2'b0;
         end
```



```
         beq: begin
            regdst = 2'b0;
            alusrc = 0;
            memtoreg = 2'b0;
            regwrite = 0;
            memwrite = 0;
            npc_sel = 2'b01;
            extop = 0;
            aluctr = 2'b0;
         end
         jal: begin
            regdst = 2'b10;
            alusrc = 0;
            memtoreg = 2'b11;
            regwrite = 1;
            memwrite = 0;
            npc_sel = 2'b10;
            extop = 0;
            aluctr = 2'b0;
         end
         lui: begin
            regdst = 2'b0;
            alusrc = 0;
            memtoreg = 2'b10;
            regwrite = 1;
            memwrite = 0;
            npc_sel = 2'b0;
            extop = 0;
            aluctr = 2'b0;
         end
```

```
         default: begin
            regdst = 2'b0;
            alusrc = 0;
            memtoreg = 2'b0;
            regwrite = 0;
            memwrite = 0;
            npc_sel = 2'b0;
            extop = 0;
            aluctr = 2'b0;
         end
      endcase
   end
end

assign RegDst     = regdst;
assign ALUSrc     = alusrc;
assign MemtoReg   = memtoreg;
assign RegWrite   = regwrite;
assign MemWrite   = memwrite;
assign nPC_sel    = npc_sel;
assign ExtOP      = extop;
assign ALUctr     = aluctr;

endmodule
```

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

3. 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
```

4. 思考题

I. (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？

文件	模块接口定义

dm.v	`dm(clk,reset,MemWrite,addr,din,dout);`
	`input  clk;  //clock`
	`input  reset;  //reset`
	`input  MemWrite;  //memory write enable`
	`input [11:2] addr;  //memory's address for write`
	`input [31:0] din;  //write data`
	`output [31:0] dout;  //read data`

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

2. 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。

II. (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。

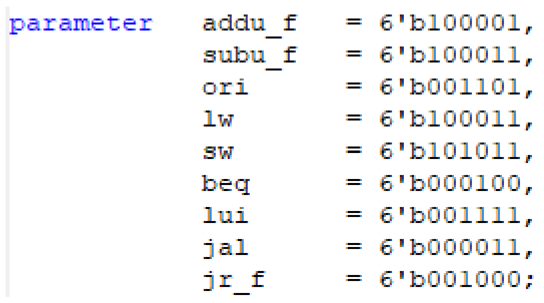
- If-else (或 case)

```
if (reset == 1)
   for (i = 0; i < 32; i = i + 1)
      registers[i] <= 32'h0;
else if (RegWrite == 1 && RegAddr != 5'b0) begin
   registers[RegAddr] <= RegData;
   $display("@%h: $%d <= %h", PC, RegAddr, RegData);
end
```

- Assign 语句

```
assign RData1 = registers[Reg1];
assign RData2 = registers[Reg2];
```

- 宏定义



```
parameter    addu_f   = 6'b100001,
             subu_f   = 6'b100011,
             ori      = 6'b001101,
             lw       = 6'b100011,
             sw       = 6'b101011,
             beq      = 6'b000100,
             lui      = 6'b001111,
             jal      = 6'b000011,
             jr_f     = 6'b001000;
```

2. 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

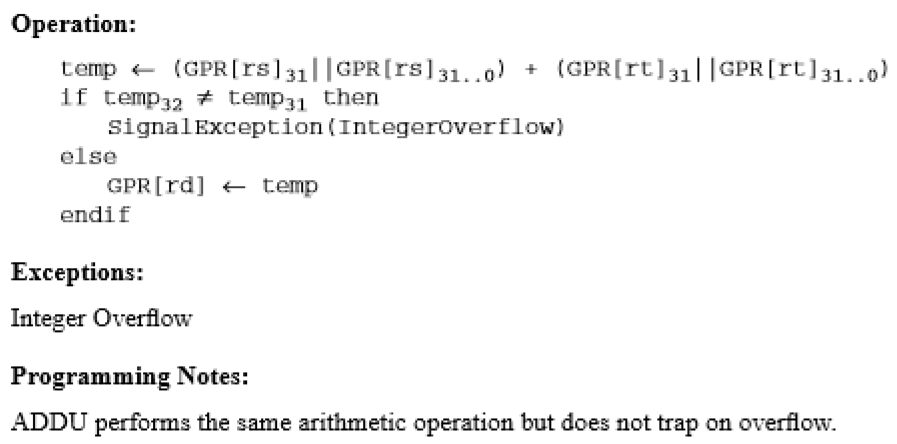
  优点：可以统一一次性定义判断值

  缺点：

III.　(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:



Operation:

```
temp ← $(GPR[rs]_{31}||GPR[rs]_{31..0})$ + $(GPR[rt]_{31}||GPR[rt]_{31..0})$
if $temp_{32} \neq temp_{31}$ then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← temp
endif
```

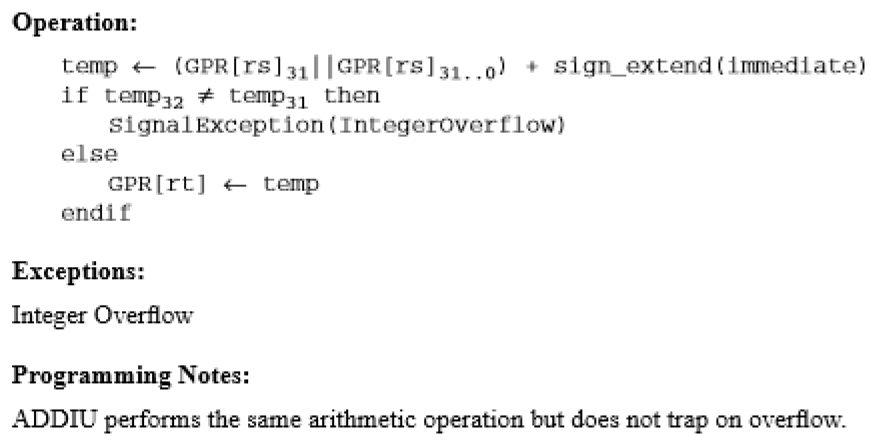
Exceptions:

Integer Overflow

Programming Notes:

ADDU performs the same arithmetic operation but does not trap on overflow.

Addi:



Operation:

```
temp ← $(GPR[rs]_{31}||GPR[rs]_{31..0})$ + sign_extend(immediate)
if $temp_{32} \neq temp_{31}$ then
    SignalException(IntegerOverflow)
else
    GPR[rt] ← temp
endif
```

Exceptions:

Integer Overflow

Programming Notes:

ADDIU performs the same arithmetic operation but does not trap on overflow.

上面的两个指令说明中已经说明了

```
if $temp_{32} \neq temp_{31}$ 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。

2. 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

3. 简要说明 jal、jr 和堆栈的关系。