计算机组成原理实验报告

16061007 王文珺

一、 CPU 设计文档

（一）整体思路

使用 Verilog 完成从 Logisim 电路到 Verilog 代码的映射，其中处理器支持{addu,subu,ori,lw,sw,beq,lui,jal,jr,nop}指令集，其中 nop 指令为空指令，不进行任何有效行为。因为 P4 是根据 P3 设计的单周期处理器进行的改进，这里首先进行整体设计如下图：

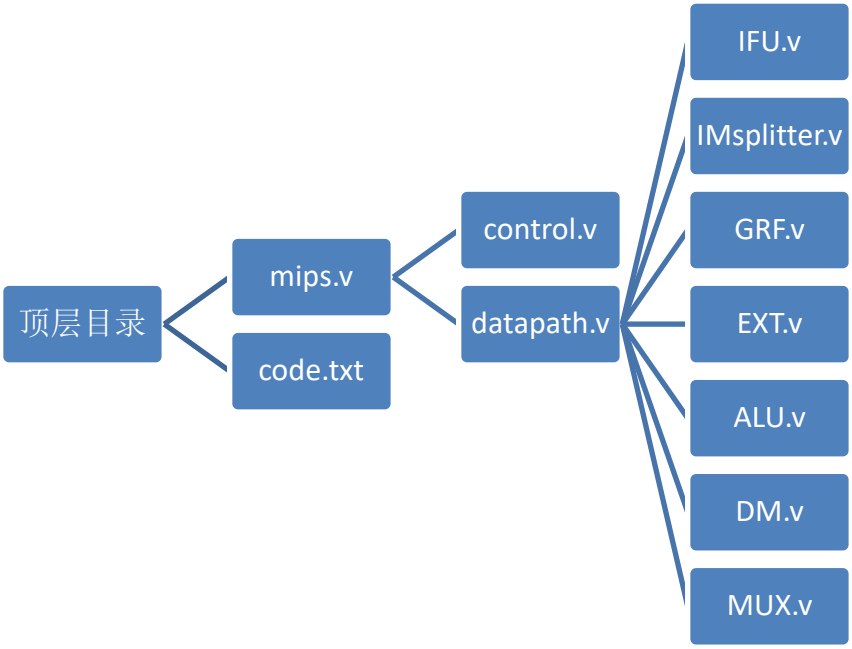

Figure 1 整体结构

（二）数据通路设计

在具体的模块化与层次之前，首先明确以下几点：、

① 处理器包括控制器和数据通路，代码储存在 code.txt 中

② Control 模块单独占一个 Verilog HDL 文件，实现单一职责

③ 模块化每一个可能使用到的单元，仅暴露出输入和输出的端口，便于对模块中的内容进行修改，使条理更加清晰

④ 在 MUX 模块中定义了不同端口数和位数要求的 MUX

1、取指令单元（IFU）

IFU 主要包括 PC 寄存器和指令存储器 IM，其中 PC 存储器起始地址为 0x00003000。IM 容量为 32bit*1024，由于增加了 jal 和 jr 指令，在 Verilog 中增加了一些信号，并且更改了复位后 PC 寄存器的值。

Table 1 IFU 端口及功能说明

序号	端口名称	I\O	功能名称	功能描述
1	RESET	I	复位	当复位信号有效时，PC 被设置为 0x00003000
2	ZERO	I	相等判断	当 ZERO 信号有效时，表示 beq 指令判断相等
3	nPC_Sel	I	地址选择	当 nPC_Sel 信号有效时，表示正在执行 beq 指令
4	IMM16	I	16 位立即数	输入 16 位立即数用于对地址进行操作
5	Instr	O	指令输出	输出当前指令编码，用于后续指令分析
6	clk	I	时钟	提供时钟信号
7	j_PC	I	跳转并储存	当 j_PC 信号有效时，储存下一条指令地址到$ra 并进行跳转到计算出来的指令地址
8	JR	I	跳转	当 JR 信号有效时，将 inpc 的值直接存储到 PC 寄存器中，进行跳转
9	inpc	I	传入地址	
10	pc	O	当前地址	输出当前指令地址，用于跳转指令的相关存储

```
initial begin
    for(i=0;i<1024;i=i+1)begin
        IM[i]=0;
    end
    $readmemh("code.txt",IM);
    PC=32'h00003000;
end
always@(posedge clk)begin
    if(reset==1)begin
        PC<=32'h00003000;
    end
    else begin
        if(zero==1&&nPC_Sel==1)begin
            PC<=PC+{{14{imm16[15]}},imm16,2'b00}+4;
        end
        else if(j_PC==1)begin
            PC<={PC[31:28],IM[PC[11:2]][25:0],2'b00};
        end
        else if(JR==1)begin
            PC<=inpc;
        end
        else begin
            PC<=PC+4;
        end
    end
end
```

2、通用寄存器组（GRF）

需要注意的是 0 号寄存器的值保持为 0，在时钟上升沿到来且 reset 信号不为 1 的时候才可以对非零号寄存器进行写入。

接下来给出端口定义与功能说明：

Table 2 GRF 端口及功能说明

序号	端口名称	I\O	功能名称	功能描述
1	Rreg1	I	寄存器 1 选择	通过多路选择器选择寄存器 1
2	Rreg2	I	寄存器 2 选择	通过多路选择器选择寄存器 2
3	Wreg	I	选择写入寄存器	当控制信号 RegWrite 有效时，在时钟上升沿将 Wdata 中的数据写入 Wreg 通过 Decoder 选择的寄存器中，
4	RegWrite	I	写入控制	
5	Wdata	I	写入的数据	
6	Rdata1	O	数据 1 输出	输出寄存器 1 的数据
7	Rdata2	O	数据 2 输出	输出寄存器 2 的数据
8	clk	I	时钟	时钟信号
9	reset	I	复位	当 reset 信号有效时，将所有寄存器值清零
10	PC	I	当前地址	用于输出当前指令地址

```
initial begin
    for(i=0;i<32;i=i+1)begin
        regs[i]=0;
    end
end

always @(posedge clk)begin
    if(reset==1)begin
        for(i=0;i<32;i=i+1)begin
            regs[i]<=0;
        end
    end
    else begin
        if(RegWrite==1&&Wreg!=5'b00000)begin
            regs[Wreg]<=Wdata;
            if(Wreg!=5'b00000)begin
                $display("@%h: $%d <= %h", PC, Wreg,Wdata);
            end
        end
    end
end
```

3、算术逻辑单元（ALU）

直接计算出所有所需的结果值，通过实例化一个 MUX 多路选择器单元进行

最终结果的选择。

接下来给出端口定义与功能说明：

Table 3 ALU 端口及功能说明

序号	端口名称	I\O	功能名称	功能描述
1	A	I	被处理数 A	将 32 位数据 A 读入 ALU
2	B	I	被处理数 B	将 32 位数据 B 读入 ALU
3	ALUOp	I	操作选择	00:加法对应 addu 指令 01:减法对应 subu 指令 10:按位或对应 ori 指令 11:按位异或对应 xori 指令
4	Result	O	结果	根据 ALUOp 通过多路选择器选择的相应输出
5	ZERO	O	相等判断	当 A=B 时，ZERO=1

```
MUX_32b_4 aluMUX(
    .in_1(Result1),
    .in_2(Result2),
    .in_3(Result3),
    .in_4(Result4),
    .select(ALUOp),
    .out(result)
);
always @* begin
    if(A==B)begin
        Zero=1;
    end
    else begin
        Zero=0;
    end
    Result1<=A+B;
    Result2<=A-B;
    Result3<=A|B;
    Result4<=A^B;
end
```

4、数据存储器（DM）

DM 容量为 32bit*1024，主要负责数据的存储写入和读取输出。这里由于增加了 lh 指令，所以首先在读取数据上实例化了两个 EXT 模块对半字进行扩展处理，在最终数据选择上通过实例化多路选择器进行了两次选择。

接下来给出端口定义与功能说明：

Table 4 DM 端口及功能说明

序号	端口名称	I\O	功能名称	功能描述
1	RESET	I	复位	当复位信号有效时，DM 中的数据被全部清零
2	Address	I	32 位地址	利用 Splitter 选择 DM 中存储的某条数据
3	WriteData	I	写入的数据	当 MemWrite 信号有效时，将 WriteData 写入
4	MemWrite	I	数据写入	Address 所选择的地址在 DM 中的位置
5	ReadData	O	读取的数据	输出 DM 被选择的位置所保存的数据
6	clk	I	时钟	时钟信号
7	pc	I	当前地址	用于输出当前指令地址
8	halfword	I	半字选择	当 halfword 信号有效时，选择半字读取并且扩展

```
EXT ext_1(
    .imm16(RAM[Address[11:2]][15:0]),
    .ExtOp(2'b01),
    .imm32(lh_1)
);

EXT ext_2(
    .imm16(RAM[Address[11:2]][15:0]),
    .ExtOp(2'b01),
    .imm32(lh_2)
);

MUX_32b_2 lh_sel(
    .in_1(lh_1),
    .in_2(lh_2),
    .select(Address[1]),
    .out(lh)
);

MUX_32b_2 fin_sel(
    .in_1(RAM[Address[11:2]]),
    .in_2(lh),
    .select(halfword),
    .out(ReadData)
);
initial begin
    for(i=0;i<1024;i=i+1)begin
        RAM[i]=0;
    end
end
always @(posedge clk)begin
    if(reset==1)begin
        for(i=0;i<=1023;i=i+1)begin
            RAM[i]<=0;
        end
    end
    else begin
        if(MemWrite==1)begin
            RAM[Address[11:2]]<=WriteData;
            $display("@%h: *%h <= %h",PC, Address,WriteData);
        end
    end
end
```

5、立即数处理单元（EXT）

EXT 本来是对立即数进行扩展（符号扩展及零扩展），但是我在这里为了统一对立即数的操作，把 lui 指令所需要的加载至高位合并了起来，并且合称该 EXT 为“立即数处理单元”，在之后的设计中，所有需要对立即数进行操作的指令 我将都合并进此立即数处理单元。

接下来给出端口定义与功能说明：

Table 5 EXT 端口及功能说明

序号	端口名称	I\O	功能名称	功能描述
1	IMM16	I	16 位立即数	输入被处理的 16 位立即数
2	ExtOp	I	操作选择	00:零扩展\01:符号扩展\10:立即数低位补 16 位 0
3	IMM32	O	32 位立即数	输出被处理后的 32 位立即数

```
always@*begin
    case(ExtOp)
        2'b00:IMM32={{16{1'b0}},imm16};
        2'b01:IMM32={{16{imm16[15]}},imm16};
        2'b10:IMM32={imm16,{16{1'b0}}};
        2'b11:IMM32={32{1'b0}};
    endcase
end
```

6、指令处理单元

这里是我自己想的用于对 IFU 取出的指令进行处理的单元，在 Logisim 中仅用多个 splitter 就可以实现，但是在电路设计上显得很复杂，所以我倾向于把他包装起来，所以在 P4 中我便设计了这样一个模块，用于直接分割指令机器码，便于后面模块的使用。

```
module IMsplitter(
    input [31:0] code,
    output [5:0] op,
    output [4:0] rs,
    output [4:0] rt,
    output [4:0] rd,
    output [5:0] func,
    output [15:0] imm16
    );
    assign imm16=code[15:0];
    assign func=code[5:0];
    assign rd=code[15:11];
    assign rt=code[20:16];
    assign rs=code[25:21];
    assign op=code[31:26];

endmodule
```

当进行完了所有的模块化设计后，就在 datapath.v 中统一管理，根据各个部件模块的输入输出端口，定义内部变量，利用模块间的逻辑关系，像接导线一样，串接各个部件模块，使之成为一个整体。最后在 mips.v 中将 datapath 和 control 连接在一起，仅留出设计要求的 clk 和 reset 端口。

Datapath 部分：

```
IFU ifu(
    .reset(reset),
    .clk(clk),
    .imm16(imm16),
    .zero(zero),
    .nPC_Sel(nPC_Sel),
    .j_PC(j_PC),
    .inpc(Rdata1),
    .JR(JR),
    .instr(instr),
    .pc(pc)
);

IMsplitter IMS(
    .code(instr),
    .op(op),
    .rs(rs),
    .rt(rt),
    .rd(rd),
    .func(func),
    .imm16(imm16)
);

MUX_5b_2 regsel_1(
    .in_1(rt),
    .in_2(rd),
    .select(RegDst),
    .out(regsel_1_out)
);

MUX_5b_2 regsel_2(
    .in_1(regsel_1_out),
    .in_2(5'b11111),
    .select(j_PC),
    .out(regsel_2_out)
);

GRF grf(
    .PC(pc),
    .reset(reset),
    .Rreg1(rs),
    .Rreg2(rt),
    .Wreg(regsel_2_out),
    .clk(clk),
    .RegWrite(RegWrite),
    .Wdata(Wdata),
    .Rdata1(Rdata1),
    .Rdata2(Rdata2)
);

EXT extend(
    .imm16(imm16),
    .ExtOp(ExtOp),
    .imm32(imm32)
);

MUX_32b_2 reg_or_imm(
    .in_1(Rdata2),
    .in_2(imm32),
    .select(ALUSrc),
    .out(ALUData)
);

ALU alu(
    .A(Rdata1),
    .B(ALUData),
    .ALUOp(ALUOp),
    .zero(zero),
    .result(ALUresult)
);

DM dm(
    .PC(pc),
    .Address(ALUresult),
    .WriteData(Rdata2),
    .MemWrite(Memwrite),
    .halfword(halfword),
    .reset(reset),
    .clk(clk),
    .ReadData(ReadData)
);

MUX_32b_2 dmout_1(
    .in_1(ALUresult),
    .in_2(ReadData),
    .select(MemtoReg),
    .out(dm_out)
);

MUX_32b_2 dmout_2(
    .in_1(dm_out),
    .in_2(pc+4),
    .select(j_PC),
    .out(Wdata)
);
```

Mips.v 部分：

```
datapath DataPath(
    .clk(clk),
    .reset(reset),
    .RegDst(RegDst),
    .ALUSrc(ALUSrc),
    .MemtoReg(MemtoReg),
    .RegWrite(RegWrite),
    .Memwrite(Memwrite),
    .nPC_Sel(nPC_Sel),
    .ExtOp(ExtOp),
    .ALUOp(ALUOp),
    .j_PC(j_PC),
    .halfword(halfword),
    .JR(JR),
    .op(op),
    .func(func)
);
```

```
control Control(
    .op(op),
    .func(func),
    .RegDst(RegDst),
    .ALUSrc(ALUSrc),
    .MemtoReg(MemtoReg),
    .RegWrite(RegWrite),
    .Memwrite(Memwrite),
    .nPC_Sel(nPC_Sel),
    .ExtOp(ExtOp),
    .ALUOp(ALUOp),
    .j_PC(j_PC),
    .halfword(halfword),
    .JR(JR)
);
```

（三）控制器设计

同样与 P3 中的设计思路一样，采用先列表的方法，只不过在 verilog 中我认为采用 case 的方法可以更加清楚明了的直接通过判断 op 和 func 的值来得到各个控制信号的值，同时由于加入了很多新的指令(xori,lh,jal,jr)表格的控制信号也根据此 CPU 要求识别的指令集，并且分析 MIPS-C 指令集，可以得到如下表格：

Table 6 控制信号真值表

func	100001	100011	N\A								001000
op	000000	000000	001101	100011	101011	000100	001111	000011	001110	100001	000000
	addu	subu	ori	lw	sw	beq	lui	jal	xori	lh	jr
RegDust	1	1	0	0	x	x	0	x	0	0	0
ALUSrc	0	0	1	1	1	0	1	x	1	1	0
MemtoReg	0	0	0	1	x	x	0	x	0	1	0
RegWrite	1	1	1	1	0	0	1	1	1	1	0
MemWrite	0	0	0	0	1	0	0	x	0	0	0
nPC_Sel	0	0	0	0	0	1	0	x	0	0	0
ExtOp	x	x	00	01	01	x	10	x	00	01	00
ALUOp	add	sub	or	add	add	sub	add	x	xori	add	add
	00	01	10	00	00	01	00		11	00	00
j_PC	0	0	0	0	0	0	0	1	0	0	0
halfword	0	0	0	0	0	0	0	0	0	1	0
jr	0	0	0	0	0	0	0	0	0	0	1

接下来给出每个控制信号的意义：

Table 7 控制信号名称及功能描述

序号	控制信号名称	功能描述
1	RegDust	当前指令为 R 型指令时有效，选择 rd 寄存器进行写入，信号无效时将选择 rt 寄存器进行写入
2	ALUSrc	当其有效时为处理过的立即数与 rs 寄存器中的数据运算，信号无效

		时为 rt 寄存中的数据与 rs 寄存器中的数据运算
3	MemtoReg	当指令为 lw 时有效，从 DM 选择相应地址段的数据传入 GRF 的数据写入端，信号无效时为将 ALU 的计算结果传入数据写入端
4	RegWrite	当需要向寄存器中写入数据时有效，将数据写入端的数据写入选择的寄存器，信号无效时不执行该操作
5	MemWrite	当指令为 sw 时有效，将 GRF 传出的 Rdata2 数据写入 DM 通过 ALU 计算出的相应地址段，信号无效时不执行该操作
6	nPC_Sel	当指令为跳转指令时有效，例如 beq，bne，j 等，判断是否生效后将选择经过计算后的跳转地址存入 PC 寄存器以便选择下一条指令，无效时即为顺序向下执行一条指令（PC+4）
7	ExtOp[1:0]	00：零扩展 01：符号扩展 10：立即数低位补 16 位 0
8	ALUOp[1:0]	00：加 01：减 10：或
9	j_PC	当指令为 jal 时有效，将选择当前指令的下一条指令地址写入$ra
10	halfword	当指令为 lh 时有效，将在 DM 中选择半字扩展和适当的数据输出
11	jr	当指令为 jr 时有效，将在 IFU 中选择读入的值写入 pc 寄存器

```
6'b100001:begin  //addu
    regDst<=1;
    aLUSrc<=0;
    memtoReg<=0;
    regWrite<=1;
    memwrite<=0;
    NPC_Sel<=0;
    extOp<=2'b00;
    aLUOp<=2'b00;
    J_PC<=0;
    Halfword<=0;
    jr<=0;
end
```

```
6'b100011:begin  //subu
    regDst<=1;
    aLUSrc<=0;
    memtoReg<=0;
    regWrite<=1;
    memwrite<=0;
    NPC_Sel<=0;
    extOp<=2'b00;
    aLUOp<=2'b01;
    J_PC<=0;
    Halfword<=0;
    jr<=0;
end
```

```
6'b001000:begin  //jr
    regDst<=0;
    aLUSrc<=0;
    memtoReg<=0;
    regWrite<=0;
    memwrite<=0;
    NPC_Sel<=0;
    extOp<=2'b00;
    aLUOp<=2'b00;
    J_PC<=0;
    Halfword<=0;
    jr<=1;
end
```

```
6'b000000:begin  //nop
    regDst<=0;
    aLUSrc<=0;
    memtoReg<=0;
    regWrite<=0;
    memwrite<=0;
    NPC_Sel<=0;
    extOp<=2'b00;
    aLUOp<=2'b00;
    J_PC<=0;
    Halfword<=0;
    jr<=0;
end
```

```
6'b001101:begin  //ori
    regDst<=0;
    aLUSrc<=1;
    memtoReg<=0;
    regWrite<=1;
    memwrite<=0;
    NPC_Sel<=0;
    extOp<=2'b00;
    aLUOp<=2'b10;
    J_PC<=0;
    Halfword<=0;
    jr<=0;
end
```

```
6'b101011:begin  //sw
    regDst<=0;
    aLUSrc<=1;
    memtoReg<=0;
    regWrite<=0;
    memwrite<=1;
    NPC_Sel<=0;
    extOp<=2'b01;
    aLUOp<=2'b00;
    J_PC<=0;
    Halfword<=0;
    jr<=0;
end
```

```
6'b001111:begin  //lui
    regDst<=0;
    aLUSrc<=1;
    memtoReg<=0;
    regWrite<=1;
    memwrite<=0;
    NPC_Sel<=0;
    extOp<=2'b10;
    aLUOp<=2'b00;
    J_PC<=0;
    Halfword<=0;
    jr<=0;
end
```

```
6'b100001:begin  //lh
    regDst<=0;
    aLUSrc<=1;
    memtoReg<=1;
    regWrite<=1;
    memwrite<=0;
    NPC_Sel<=0;
    extOp<=2'b01;
    aLUOp<=2'b00;
    J_PC<=0;
    Halfword<=1;
    jr<=0;
end
```

```
6'b100011:begin  //lw
    regDst<=0;
    aLUSrc<=1;
    memtoReg<=1;
    regWrite<=1;
    memwrite<=0;
    NPC_Sel<=0;
    extOp<=2'b01;
    aLUOp<=2'b00;
    J_PC<=0;
    Halfword<=0;
    jr<=0;
end
```

```
6'b000100:begin  //beq
    regDst<=0;
    aLUSrc<=0;
    memtoReg<=0;
    regWrite<=0;
    memwrite<=0;
    NPC_Sel<=1;
    extOp<=2'b00;
    aLUOp<=2'b01;
    J_PC<=0;
    Halfword<=0;
    jr<=0;
end
```

```
6'b000011:begin  //jal
    regDst<=0;
    aLUSrc<=0;
    memtoReg<=0;
    regWrite<=1;
    memwrite<=0;
    NPC_Sel<=0;
    extOp<=2'b00;
    aLUOp<=2'b00;
    J_PC<=1;
    Halfword<=0;
    jr<=0;
end
```

```
6'b001110:begin  //xori
    regDst<=0;
    aLUSrc<=1;
    memtoReg<=0;
    regWrite<=1;
    memwrite<=0;
    NPC_Sel<=0;
    extOp<=2'b00;
    aLUOp<=2'b11;
    J_PC<=0;
    Halfword<=0;
    jr<=0;
end
```

（四）测试程序

为了保证测试所有的指令，这里编写了如下的测试程序并给出测试期望

```
.text
ori $t0,$0,123 #测试 ori 指令
$8 被写入 0x0000007b
ori $t1,$t0,456
$9 被写入 0x000001fb
lui $t3,123 #测试 lui 指令，符号位为 0
$11 被写入 0x007b0000
lui $t4,0xffff #符号位为 1
$12 被写入 0xffff0000
ori $t4,$t4,0xffff #$t4=-1
$12 被写入 0xffffffff
jal ra #测试 jal
$31 被写入 0x00003018
addu $s0,$t0,$t3  #测试 addu，+ +
$16 被写入 0x007b007b
addu $s1,$t0,$t4  # + -
$17 被写入 0x0000007a
addu $s2,$t4,$t4  # - -
$18 被写入 0xfffffffe
subu $s3,$t0,$t3  #测试 subu，+ +
$19 被写入 0xff85007b
subu $s4,$t0,$t4  # + -
```

$20 被写入 0x0000007c

```
subu $s5,$t4,$t4  # - -
```

$21 被写入 0x00000000

```
subu $s4,$t4,$t0  # - +
```

$20 被写入 0xffffff84

```
nop  #测试 nop
```

无操作

```
jal loop_before
```

$31 被写入 0x0000303c

```
loop:
addu $a0,$a0,$a1
beq $a0,$a1,loop  #测试 beq，相等/不相等
beq $a0,$a2,loop
loop_before:
ori $a0,$0,0
```

$4 被写入 0x00000000

```
ori $a1,$0,1
```

$5 被写入 0x00000001

```
ori $a2,$0,2
```

$6 被写入 0x00000002

```
ori $s3,$0,0x000c
```

$19 被写入 0x0000000c

```
sw $t0,-8($s3)  #测试 sw,offset 为正
```

DM 中地址 0x00000004 被写入 0x0000007b

```
sw $t2,0($s3)  #offset 为 0
```

DM 中地址 0x0000000c 被写入 0x00000000

sw $t4,8($s3) #offset 为负

DM 中地址 0x00000014 被写入 0xffffffff

lw $a0,-8($s3) #测试 lw，offset 为正

$4 被写入 0x00000007b

lw $a1,0($s3) #offset 为 0

$5 被写入 0x00000000

lw $a2,8($s3) #offset 为负

$6 被写入 0x00003074

jal end

$31 被写入 0x0000000c

ra:

jr $ra #测试 jr

$31 的值被写回 pc，指令跳转回去

end:

仿真后得到该结果：

```
ISim P.20131013 (signature 0x7708f090)
This is a Full version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
@00003000: $ 8 <= 0000007b
@00003004: $ 9 <= 000001fb
@00003008: $11 <= 007b0000
@0000300c: $12 <= ffff0000
@00003010: $12 <= ffffffff
@00003014: $31 <= 00003018
@00003018: $16 <= 007b007b
@0000301c: $17 <= 0000007a
@00003020: $18 <= fffffffe
@00003024: $19 <= ff85007b
@00003028: $20 <= 0000007c
@0000302c: $21 <= 00000000
@00003030: $20 <= ffffff84
@00003038: $31 <= 0000303c
@00003048: $ 4 <= 00000000
@0000304c: $ 5 <= 00000001
@00003050: $ 6 <= 00000002
@00003054: $19 <= 0000000c
@00003058: *00000004 <= 0000007b
@0000305c: *0000000c <= 00000000
@00003060: *00000014 <= ffffffff
@00003064: $ 4 <= 0000007b
@00003068: $ 5 <= 00000000
@0000306c: $ 6 <= ffffffff
@00003070: $31 <= 00003074
ISim>
```

二、思考题

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

根据我的理解，由于 DM 的大小是 32bit*1024 字，所以对于 DM 中定义的寄存器组，他们的序号其实是字序号，而我们的 32 位处理器中，一个字为四个字节，所以每当执行一条指令时，就需要 pc+4，所以每当需要在地址中存储时，需要取[11:2]这样才能保证一次存储而没有跳跃，倘若是[9:0]则对于相邻的写入 DM 指令就会相隔三个字。addr 信号是经过数据通路中 ALU 计算得到的。

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？

清零信号是针对寄存器堆 GRF 和数据存储器 DM 还有 PC 寄存器进行清零复位操作的，由于 reset 操作是将整个 CPU 重置到初始状态，即第一条指令执行之前，所以需要多数据存储器和寄存器堆中的所有值进行清零复位。

3、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。

①使用 case 完成操作码和控制信号之间的对应：

```
case(func)
   6'b100001:begin  //addu
      regDst<=1;
      aLUSrc<=0;
      memtoReg<=0;
      regWrite<=1;
      memwrite<=0;
      NPC_Sel<=0;
      extOp<=2'b00;
      aLUOp<=2'b00;
      J_PC<=0;
      Halfword<=0;
      jr<=0;
   end
```

②assign 语句完成操作码和控制信号的值之间的对应，类似与或逻辑的操作

```
assign
RegDst=~op[5]&&~op[4]&&~op[3]&&~op[2]&&~op[1]&&~op[0]&&f[
5]&&~f[4]&&~f[3]&&~f[2]&&f[0];
assign
ALUSrc=(~op[5]&&~op[4]&&op[3]&&op[2]op[0])||(op[5]&&~op[4
]&&~op[2]&&op[1]&&op[0])
……
```

③使用宏定义

```
`define addu 6'b100001
`define subu 6'b100011
`define jr 6'b001000
……
```

然后可以使用 case 语句对 op 和 func 进行判断

4、根据你所列举的编码方式，说明他们的优缺点。

使用 case 语句的优点在于可以寻找到每个指令的代码，明确的看到在对于这条指令所有的控制信号都是什么值，并且需要添加新指令时，倘若不生成新的控制信号，即可直接复制一个 case 并更改相应的值，比较直观。缺点在于这样写代码篇幅很长，并且倘若生成了新的控制信号，则之前的所有 case 都需要进行添加信号的操作，当已有很多指令时比较麻烦。

使用 assign 语句的优点在于代码十分简洁，并且如果 assign 语句是经过简化的与或式的话，可以很直观的看出来每个控制信号分别与机器码的哪个位置有关系。缺点在于看起来不直观，倘若想寻找某一条指令的各个控制信号的话需要依次计算，并且每当添加新的控制信号都需要进行一次与或式的检索和化简，相当于利用预处理优化了代码量，减小了由于重复而容易导致的错误。

使用宏定义在我看来并不能完全完成控制器的 coding，但是优点在于它形象化了 op 和 func，把代码串形象成不同的指令，这样首先避免了在后面重复输入

0 和 1 而容易导致的错误，其次这样的定义方法更加符合我们在 logisim 中进行的与或逻辑，先定义出每条指令和他们对应的代码，然后把通过或门把不同指令结合到每个控制信号上。缺点在于宏定义一定要准确，否则后面使用都会错误。

5、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 。

addi 指令说明中 temp=(GPR[rs]31||GPR[rs]31``0)+sign_extend(immediate)
addiu 指令说明中 temp=GPR[rs]+sign_extend(immediate)

所以对于 addi 中 temp 的低 32 位和 addiu 中的 temp，都是由 rs 的值和立即数扩展后相加得到的，所以倘若不考虑溢出的情况，即 addi 指令并没有 SignalException(IntegerOverflow)分支，则两条指令均执行 GPR[rt]<-temp 操作，add 和 addu 的处理相同，低 32 位是相同的办法得到的，不考虑溢出的情况下，均执行 GPR[rt]<-temp 操作。所以说在忽略溢出的前提下，addi 与 addiu 是等价的，add 与 addu 是等价的。

6、根据自己的设计说明单周期处理器的优缺点。

单周期处理器的优点在于，每一条指令的过程清晰，直观，缺点在于每个元件功能单一不能复用，造成了资源浪费，且在处理多条指令时需要时间周期较长，且周期取决于最长数据通路的耗时，也造成了时间浪费。比如本处理器中，lw 的数据通路最长，对于 IFU 单元，lw 指令需要传出的数据已经传出，此时 IM 处于空闲状态，但是由于 lw 时间周期较长，还不能马上读入下一条指令，同样可以类比到后面的 ALU，EXT 单元等等，所有的空闲单元必须等待 GRF 存储完毕后才可以开始下一条指令的处理，时间成本很大。

7、简要说明 jal、jr 和堆栈的关系。

Jal、jr、堆栈一般在函数调用中需要配合使用，jal 可以跳转函数，同时保存下一条指令到 31 号寄存器，jr 可以跳回程序。当 jal 反复递归调用时，由于上一个函数中很多值包括 31 号寄存器的返回地址需要保存不能被更改，倘若不对这些值进行栈维护的话，下一次跳转后就会对这些值进行修改且无法复原。此时需要把需要保存的值压入栈，并且更改栈顶指针，直到 jr 跳回了这个函数时再根据先入后出的规则进行弹栈，把这些值再次于当前函数应用。总结一下就是对于需要保护的数据来说，jal 后数据进栈，jr 后数据出栈。