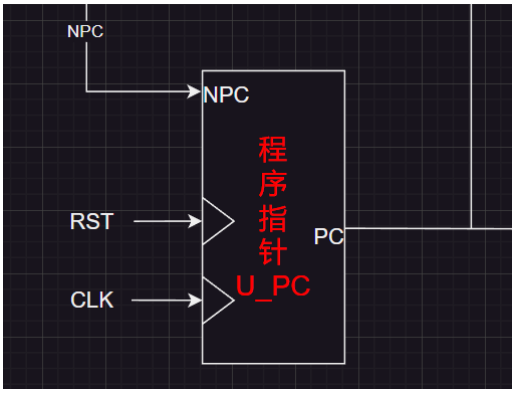
单周期CPU的底层模块

1. PC：程序计数器



1.1. 接口模块

信号名	来源/去向	描述
clk	input	时钟信号，在上升沿时执行PC的更新
rst	input	复位信号，在rst上升沿时将PC置零
NPC	input自NPC	由NPC模块输出下一周期PC的值
PC	output到NPC，指令寄存器(选指令)，加法器(+4更新)	接收NPC的值，更新后输出新的程序指针

1.2. 功能描述

1️⃣ 重置PC值：注意rst为置零信号

```
if(rst)
  PC <= 32'h0000_0000;
```

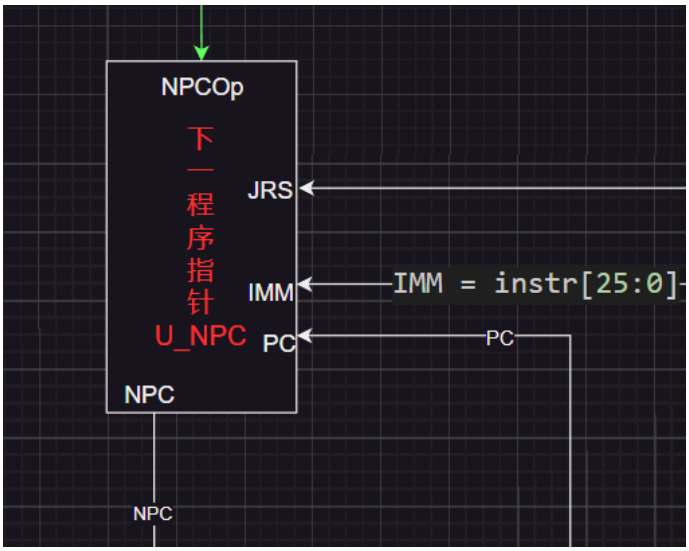
2️⃣更新PC为下条指令的地址：这是通过传入NPC值实现的，NPC的含义是下一个PC的值

```
PC <= NPC;
```

1.3. 源代码PC.v

```
module PC( clk, rst, NPC, PC );
input clk;
input rst;
input [31:0] NPC; //注意以上三者，input端口默认是wire类型
output reg [31:0] PC; //由于PC的值需要在always块中赋值，所以需要用reg类型

always @(posedge clk, posedge rst)
 if (rst)
   PC <= 32'h0000_0000;
 else
   PC <= NPC;
endmodule
```

2. NPC：指定下一个PC



2.1. 接口描述

信号名	方向	描述
PC	input自PC模块	上一周期PC的值
NPCOp	input自控制器	下一PC操作码，不同的指令对应不同逻辑
IMM	input自指令寄存器	此时是J型指令，提取出立即数，用于JUMP操作
JRS	input自寄存器组	JUMP REGISTER操作中使用的寄存器值
NPC	onput到PC	下一周期PC的值

2.2. 功能描述

1. NPC_PLUS：制字段NPCOp=00，顺序执行，让NPC指向下一位

```
`NPC_PLUS4:  NPC = PC_PLUS_4;
```

2. NPC_BRANCH：控制字段NPCOp=01，分支执行，让NPC跳到当前值+IMM代表的偏移量处

```
`NPC_BRANCH: NPC = PC_PLUS_4 + {{14{IMM[15]}}, IMM[15:0], 2'b00};
//在低位补充两个0是为了保证NPC的地址是4的倍数(指令地址必须是4的倍数)
//高位补充14个IMM[15](符号位)是为了保证NPC的地址是32位的
```

3️⃣ `NPC_JUMP`：控制字段NPCOp=10，无条件跳转，让NPC的值跳到IMM代表的字地址

```
`NPC_JUMP:   NPC = {PC_PLUS_4[31:28], IMM[25:0], 2'b00};
//PC_PLUS_4[31:28]保证跳转到的地址高4位不变，即跳转到的地址和当前地址相差不超过256M
//2'b00照样是为了对齐，让地址是4的倍数
```

4️⃣ `NPC_JUMPR`：控制字段NPCOp=11，寄存器跳转，直接使用寄存器`JRS`的值作为下一条指令的地址

```
`NPC_JUMPR:  NPC = JRS;
```

2.3. 代码实现

2.3.1. 宏定义`ctrl_encode_def.v`

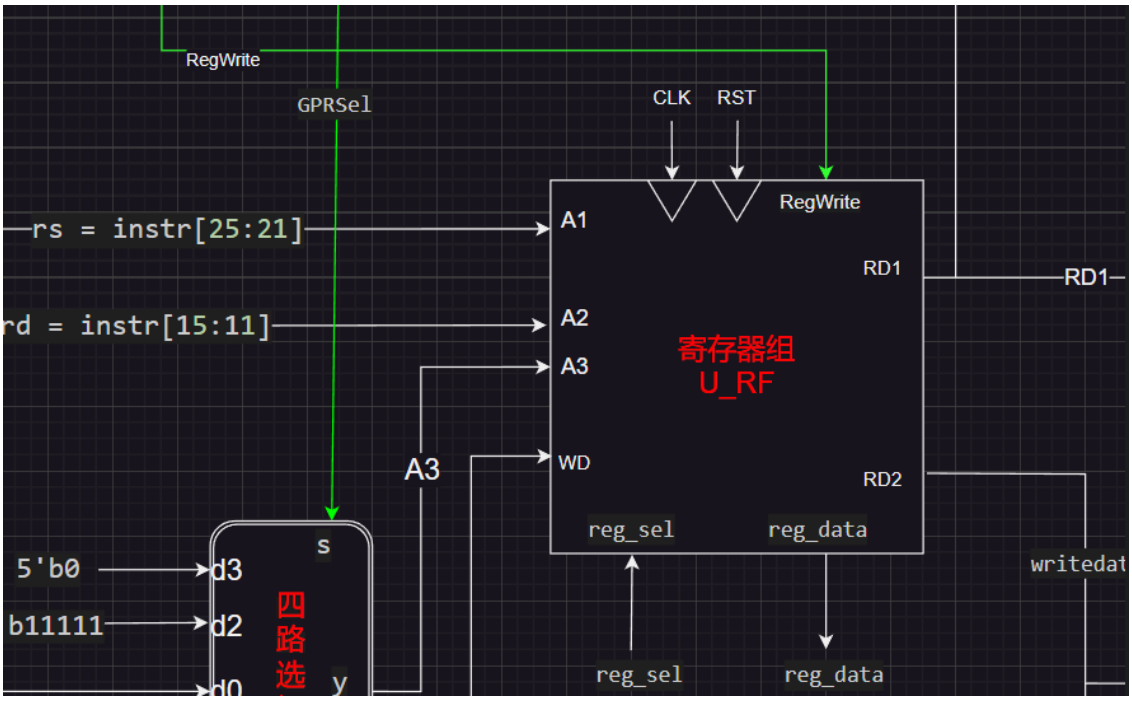
```
// NPC控制信号
`define NPC_PLUS4   2'b00
`define NPC_BRANCH  2'b01
`define NPC_JUMP    2'b10
`define NPC_JUMPR   2'b11

// ALU控制信号
`define ALU_NOP   4'b0000
`define ALU_ADD   4'b0001
`define ALU_SUB   4'b0010
`define ALU_AND   4'b0011
`define ALU_OR    4'b0100
`define ALU_SLT   4'b0101
`define ALU_SLTU  4'b0110
`define ALU_SLLV  4'b0111
`define ALU_SLL   4'b1000
`define ALU_NOR   4'b1001
`define ALU_LUI   4'b1010
`define ALU_SRL   4'b1011
`define ALU_SRLV  4'b1100
```

2.3.2. 源代码`NPC.v`

```
`include "ctrl_encode_def.v"

module NPC(PC, NPCOp, IMM, JRS ,NPC);

input  [31:0] PC;
input  [1:0]  NPCOp;
input  [25:0] IMM;
input  [31:0]  JRS;
output reg [31:0] NPC;
```

```
10
11
12  //定义PC_PLUS_4，存放顺序执行下一条指令的地址
13  //下一条指令地址 = 当前指令地址 + 指令的大小(4个字节32位)
14  wire [31:0] PC_PLUS_4;
15  assign PC_PLUS_4 = PC + 4;
16
17  //任何输入信号变化时，都会执行always块中的代码
18  //根据NPCOp的值选择操作，计算NPC的值
19  always @(*) begin
20     case (NPCOp)
21          `NPC_PLUS4:  NPC = PC_PLUS_4;
22          `NPC_BRANCH: NPC = PC_PLUS_4 + {{14{IMM[15]}}, IMM[15:0],
    2'b00};
23          `NPC_JUMP:   NPC = {PC_PLUS_4[31:28], IMM[25:0], 2'b00};
24          `NPC_JUMPR:  NPC = JRS;
25            default:      NPC = PC_PLUS_4;
26     endcase
27  end
28  endmodule
```

3. RF：寄存器



3.1. 功能描述

1️⃣读出寄存器值：将A1A2读出到对应端口，注意A1A2为寄存器号(地址)，0号寄存器不能读写

```
1  assign RD1 = (A1 != 0) ? rf[A1] : 0;//A1不为0则输出rf[A1]到RD1，否则输出0
2  assign RD2 = (A2 != 0) ? rf[A2] : 0;//A2不为0则输出rf[A2]到RD2，否则输出0
```

2️⃣写入寄存器值：在clk的上升沿时，将待写入的数据写入A3指定的寄存器

```
1  else if (RFWr) //获得写使能信号
2  begin
3  //如果要写入的目标寄存器A3为0寄存器，则不写入
```

```
if (A3 == 5'b00000)
begin
$display("******************");
rf[A3] <= 0;
end
//不为0寄存器时，写入待写入数据WD
else
begin
rf[A3] <= WD;
$display("r[%2d] = 0x%8X,", A3, WD);//打印写入的寄存器编号和写入的数据
end
end
```

3️⃣收到复位信号后将所有寄存器值清0

```
integer i;
if (rst)
begin
for (i=1; i<32; i=i+1)
rf[i] <= 0;
end
```

4️⃣Debug

```
assign reg_data = (reg_sel != 0) ? rf[reg_sel] : 0;
```

通过这一行语句，可以通过输入reg_data，改变reg_sel，来检测是否遵从0寄存器时输出0，非零时输出值(且不影响寄存器文件的其他操作)

3.2. 接口模块

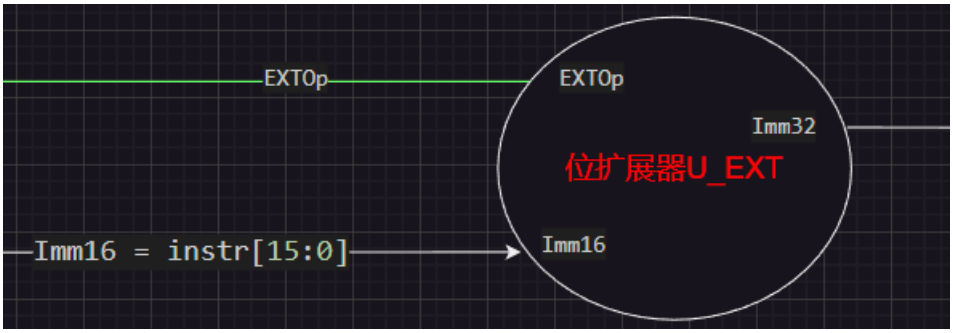
信号名	方向	描述
rst	input	Rst 上升沿时，所有寄存器复位
RFWr	input自控制器	写寄存器使能，1 时将 WD 写入寄存器
A1	input自指令寄存器	第一个源操作数所在寄存器的编号
A2	input自指令寄存器	第二个源操作数所在寄存器的编号
A3	input自一个四路选择器	先选清楚指令中哪个字段才指向目标寄存器，然后指定它
WD	input自另一个路选择器	先选中到底要把那个数据写入寄存器，然后把他给RF
RD1	output到一二路选择器	根据A1地址读出的数据，然后等待是否被选中输入ALU
RD2	output到另一二路择器	根据A2地址读出的数据，然后等待是否被选中输入ALU
Reg_sel	input	Debug 用

信号名	方向	描述
Reg_data	output	Debug 用

3.3. 源代码`RF.v`

```
module RF
(
 input clk,
 input rst,
 input RFWr,
 input [4:0]  A1, A2, A3,
 input [31:0] WD,
 output [31:0] RD1, RD2,
 input [4:0]  reg_sel,
 output [31:0] reg_data
);

reg [31:0] rf[31:0];//32个寄存器

integer i;
always @(posedge clk, posedge rst)
 if (rst) //寄存器清零
 begin
   for (i=1; i<32; i=i+1)
     rf[i] <= 0;
 end

 else if (RFWr) //写入寄存器
 begin
          if (A3 == 5'b00000)
   begin
              $display("******************");
              rf[A3] <= 0;
          end
   else
   begin
            rf[A3] <= WD;
            $display("r[%2d] = 0x%8X,", A3, WD);
          end

 end

//读取寄存器
assign RD1 = (A1 != 0) ? rf[A1] : 0;
assign RD2 = (A2 != 0) ? rf[A2] : 0;

//调试用
assign reg_data = (reg_sel != 0) ? rf[reg_sel] : 0;
endmodule
```

4. 立即数高位扩展： EXT



4.1. 功能描述

1️⃣当EXTOp=0时，进行无符号数的扩展，高位补0

2️⃣当EXTOp=1时，进行有符号数的扩展，高位补符号位

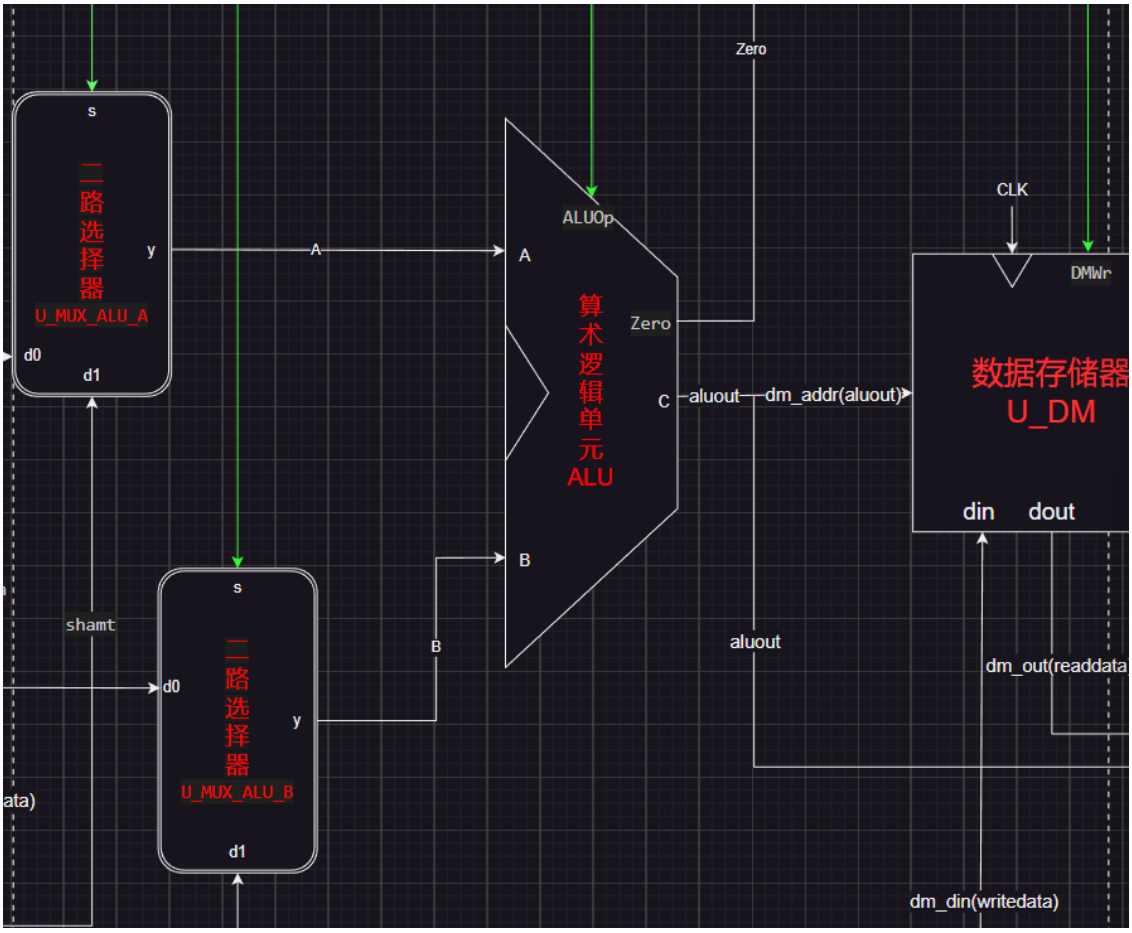
4.2. 接口模块

信号名	方向	描述
Imm16	input自指令寄存器分割所得字段	16位立即数的值
EXTOp	input自控制器	拓展方式选择信号用于选择拓展
Imm32	output到一个二路选择器	拓展后32位立即数的值，等待是否被选中输入ALU

4.3. 代码实现`EXT.v`

```
module EXT( Imm16, EXTOp, Imm32 );
input [15:0] Imm16;
input EXTOp;
output [31:0] Imm32;
assign Imm32 = (EXTOp) ? {{16{Imm16[15]}}, Imm16} : {16'b0, Imm16};

endmodule
```

5. 算术逻辑单元：ALU



5.1. 功能描述

5.1.2. 算术逻辑运算

ALUOp	功能	描述
0000	NOP	无操作，`C <- A`
0001	ADD	加法，`C <- A + B`
0010	SUB	减法，`C <- A - B`
0011	AND	逻辑与，`C <- A & B`
0100	OR	逻辑或，`C <- A`
0101	SLT	小于判定，如果A<B则C为32'd1(否则为32'd0)
0110	SLTU	无符号小于判定，逻辑同SLT，但把A和B当作无符号数比较。
0111	SLLV	变量逻辑左移，`C <- B << S`，其中`S = {27'b0,A[4:0]}`
1000	SLL	逻辑左移，`C <- B << A`
1001	NOR	逻辑非或，`C <- ~(A | B)`
1010	LUI	高位加载，将B的值放在C的高16位(低16位为0)，`C <- B << 16`

ALUOp	功能	描述
1011	SRL	逻辑右移，`C <- B >> A`
1100	SRLV	变量逻辑右移，`C <- B >> S`，其中`S = {27'b0,A[4:0]}`
default	其他	默认无操作，`C <- A`

5.1.3. 标志位

> 用zero信号接收，当运算结果为0时，标志zero=1
>
> `zero <- (C == 32'b0) ? 1:0`

5.2. 模块接口

信号名	方向	描述
A	input自一个二路选择器	源操作数1，被选择器选中后输入
B	input自一个二路选择器	源操作数2，被选择器选中后输入
ALUOp	Input自控制器	ALU控制信号，决定对 AB 进行何种计算
C	output到存储器(存储有关指令)，或一个四路选择器(待被选择是否被写回)	计算结果的输出/存储/写回
Zero	output到控制器	计算结果是否为0，用于`beq`和`bne`的判断

5.3. 代码实现

5.3.1. 宏定义：`ctrl_encode_def.v`

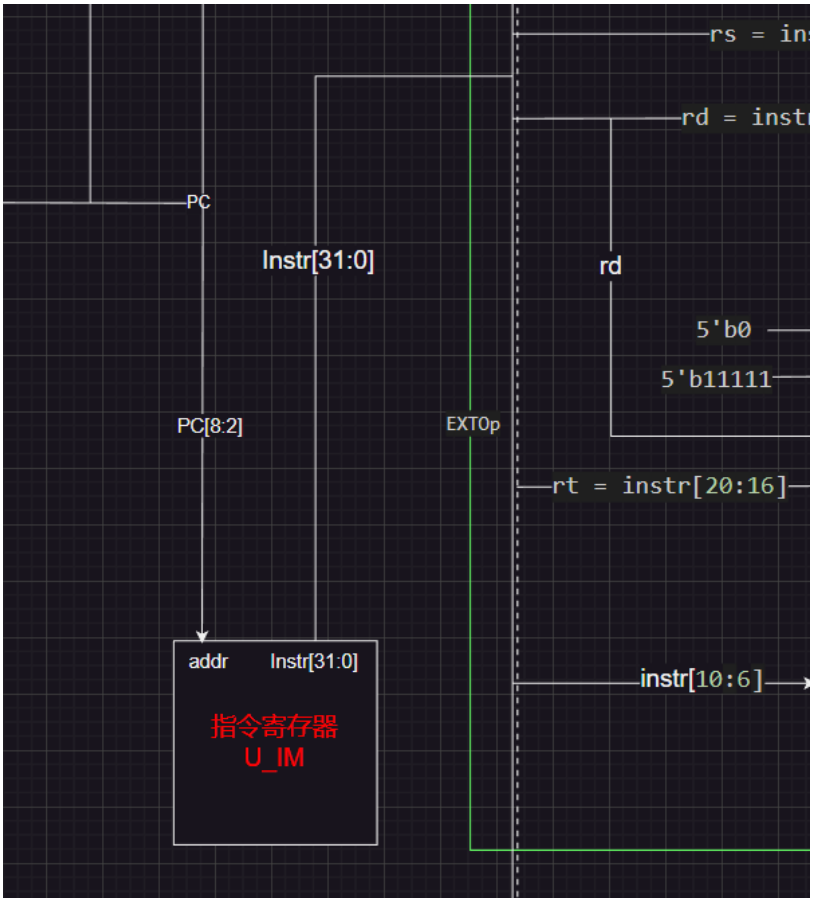
```
// NPC控制信号
`define NPC_PLUS4   2'b00
`define NPC_BRANCH  2'b01
`define NPC_JUMP    2'b10
`define NPC_JUMPR   2'b11

// ALU控制信号
`define ALU_NOP   4'b0000
`define ALU_ADD   4'b0001
`define ALU_SUB   4'b0010
`define ALU_AND   4'b0011
`define ALU_OR    4'b0100
`define ALU_SLT   4'b0101
`define ALU_SLTU  4'b0110
`define ALU_SLLV  4'b0111
`define ALU_SLL   4'b1000
`define ALU_NOR   4'b1001
```

```
`define ALU_LUI   4'b1010
`define ALU_SRL   4'b1011
`define ALU_SRLV  4'b1100
```

5.3.2. ALU的实现：`alu.v`

```
`include "ctrl_encode_def.v"
module alu(A, B, ALUOp, C, Zero);

 input signed [31:0] A, B;
 input [3:0]  ALUOp;
 output signed [31:0] C;
 output Zero;
 wire [31:0] S;
 reg [31:0] C;
 integer i;

 assign S = {27'b0,A[4:0]};
 always @( * ) begin
     case ( ALUOp )
         `ALU_NOP:  C = A;
         `ALU_ADD:  C = A + B;
         `ALU_SUB:  C = A - B;
         `ALU_AND:  C = A & B;
         `ALU_OR:   C = A | B;
         `ALU_SLT:  C = (A < B) ? 32'd1 : 32'd0;
         `ALU_SLTU: C = ({1'b0, A} < {1'b0, B}) ? 32'd1 : 32'd0;
         `ALU_SLL:  C = B << A;
         `ALU_NOR:  C = ~(A | B);
         `ALU_LUI:  C = B << 16;
         `ALU_SRL:  C = B >> A;
         `ALU_SLLV: C = B << S;
         `ALU_SRLV: C = B >> S;
        default:   C = A;
     endcase
 end
 assign Zero = (C == 32'b0);
endmodule
```

6. 指令寄存器：`im.v`



6.1. 功能&输入输出

1️⃣存储指令：在这我们在im初始化了一串指令，读取`im.v`同目录下的`studentnosorting_cut.dat`

```
initial begin
 $readmemh("studentnosorting_cut.dat", ROM);
end
```

2️⃣输入输出&寻找指令输出指令功能

```
dout <= ROM[addr]
//addr是输入的指令的地址，一般都来自于PC
//dout是根据地址找到，然后输出的指令机器码
```

6.2. 代码实现

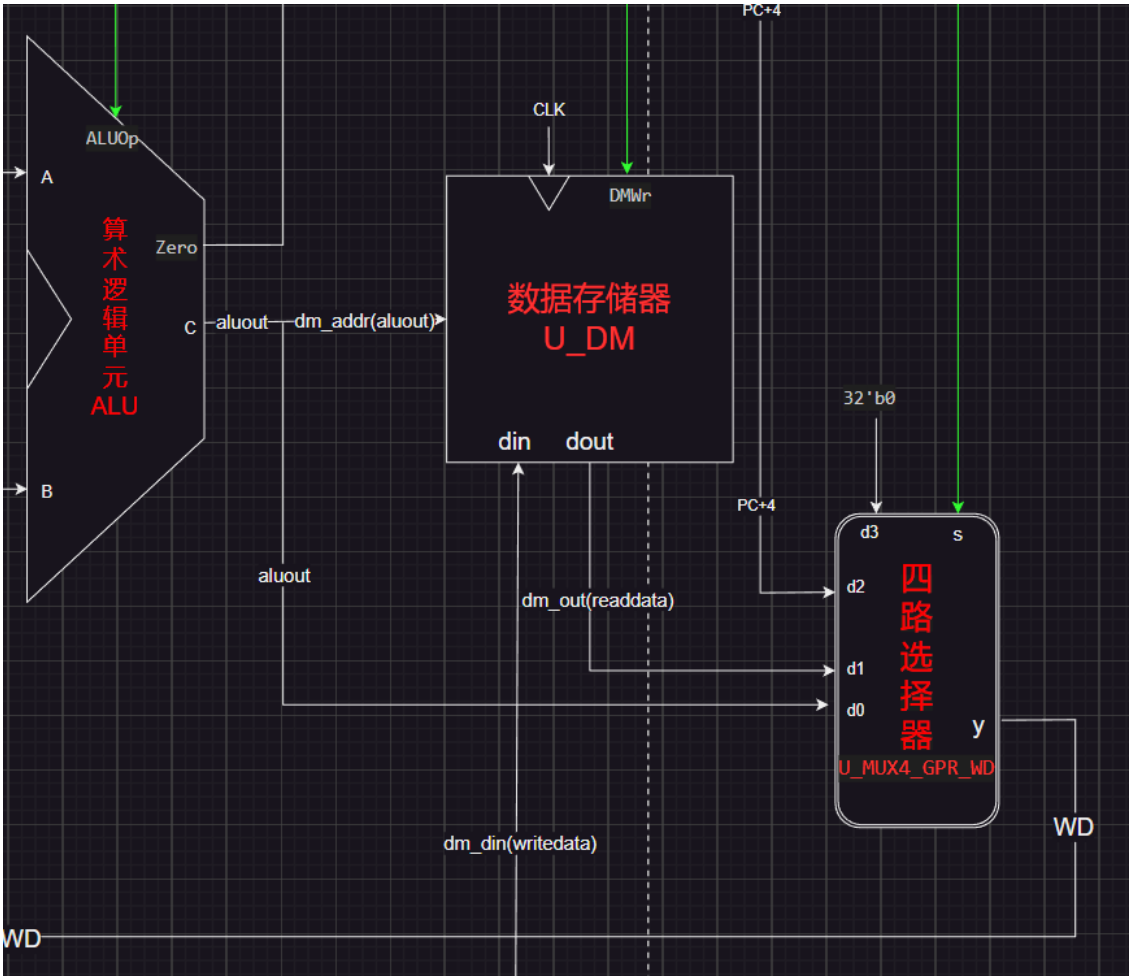
6.2.1. 预先设置好的指令序列：`studentnosorting_cut.dat`

```
3c020218
34421062
ac020100
200b0008
8c010100
00001020
2004000f
......
当然写了testbench后这个也没什么用了
```

6.2.2. `im.v`

```
//输入一个7位的addr地址，最低2位被省略意味着是以4字节为单位寻址的
//dou为输出的32位地址
module im(input  [8:2]  addr,output [31:0] dout);
//32位宽，共有128个寄存器的寄存器组ROM，128个寄存器也就需要7位寻址
reg  [31:0] ROM[127:0];
initial begin
 $readmemh("studentnosorting_cut.dat", ROM);
end
assign dout = ROM[addr];
endmodule
```

7. 内存DM：`dm.v`



7.1. 接口模块

信号名	方向	描述
clk	input	时钟信号
DMWr	input自控制器	写使能
addr	input自ALU的运算结果(内存有关指令)	待读出数据的地址，数据待写入的地址
din	input自寄存器组	写入的数据
dout	output到一个四路选择器	读出的数据，等待是否被选中写回

7.2. 功能

当addr为字地址，写使能有效时，根据地址完成读写

```
读出：dout <- dmem[addr]
写入：dmem[addr] <- din
```

7.3. 代码实现`dm.v`

```
module dm(clk, DMWr, addr, din, dout);
input          clk;
input          DMWr;
input  [8:2]   addr;
input  [31:0]  din;
output [31:0]  dout;

reg [31:0] dmem[127:0];//32位宽、128个位置的存储器组

//addeByte是addr的左移2位，将字节地址转换为字地址
wire [31:0] addrByte;
assign addrByte = addr<<2;

//数据输出操作，按照addrByte的字地址，从dmem中取出数据
assign dout = dmem[addrByte[8:2]];

//数据输入操作，按照addrByte的字地址，将数据写入dmem中
//注意这一过程需要写使能，同时打印出写入的数据
always @(posedge clk)
   if (DMWr) begin
     dmem[addrByte[8:2]] <= din;
     $display("dmem[0x%8X] = 0x%8X,", addrByte, din);
   end
endmodule
```

8. 多路选择`mux.v`

8.1. 原理：以$N=2^n$路选择器为例

8.1.1. 接口模块

信号名	方向	描述
d0,d1,...,dN-1	input	待选择的信号，共N个
s	input	s用来做选择，共n位，取值从00...00,00...01一直到11...11
y	output	选择完以后输出的信号，y=di

8.1.2. 模块功能

> 就是输入N个信号，根据s的值，确定一个然后输出

8.1.3. 代码实现

```
n'b00...01: y_r = d1;module muxN #(parameter WIDTH = 8)
                                  (d0,d1,...,dN-1,s,y);//定义待选择的数据的宽度
 input  [WIDTH-1:0] d0,d1,...,dN-1;
 input  [n-1:0] s;
 output [WIDTH-1:0] y;
```

```
 reg [WIDTH-1:0] y_r;//中间变量
 always @( * ) begin
     case(s)
         n'b00...00: y_r = d0;
         n'b00...01: y_r = d1;
         ......
         n'b11...11: y_r = dN-1;
         default:;
     endcase
 end
 assign y = y_r;
 endmodule

```

8.2. 2/4/8/16路选择器

8.2.1. 2路选择器

```
module mux2 #(parameter WIDTH = 8)
          (d0, d1,s, y);
 input [WIDTH-1:0] d0, d1;
 input s;
 output [WIDTH-1:0] y;
 assign y = ( s == 1'b1 ) ? d1:d0;
endmodule
```

8.2.2. 4路选择器

```
module mux4 #(parameter WIDTH = 8)
 (d0, d1, d2, d3,s, y);
 input  [WIDTH-1:0] d0, d1, d2, d3;
 input  [1:0] s;
 output [WIDTH-1:0] y;
 reg [WIDTH-1:0] y_r;
 always @( * ) begin
     case ( s )
         2'b00: y_r = d0;
         2'b01: y_r = d1;
         2'b10: y_r = d2;
         2'b11: y_r = d3;
         default: ;
     endcase
 end
 assign y = y_r;
endmodule
```

8.2.3. 8路选择器

```
module mux8 #(parameter WIDTH = 8)
 (d0, d1, d2, d3,d4, d5, d6, d7,s, y);
 input  [WIDTH-1:0] d0, d1, d2, d3;
 input  [WIDTH-1:0] d4, d5, d6, d7;
```

```
 input  [2:0]       s;
 output [WIDTH-1:0] y;
 reg [WIDTH-1:0] y_r;
 always @( * ) begin
     case ( s )
         3'd0: y_r = d0;
         3'd1: y_r = d1;
         3'd2: y_r = d2;
         3'd3: y_r = d3;
         3'd4: y_r = d4;
         3'd5: y_r = d5;
         3'd6: y_r = d6;
         3'd7: y_r = d7;
         default: ;
     endcase
 end
 assign y = y_r;
endmodule
```

8.2.4. 16路选择器

```
module mux16 #(parameter WIDTH = 8)
(d0, d1, d2, d3,d4, d5, d6, d7,d8, d9, d10, d11,d12, d13, d14,
d15,s, y);
 input [WIDTH-1:0] d0, d1, d2, d3;
 input [WIDTH-1:0] d4, d5, d6, d7;
 input [WIDTH-1:0] d8, d9, d10, d11;
 input [WIDTH-1:0] d12, d13, d14, d15;
 input [3:0] s;
 output [WIDTH-1:0] y;
 reg [WIDTH-1:0] y_r;
 always @( * ) begin
     case ( s )
         4'd0:  y_r = d0;
         4'd1:  y_r = d1;
         4'd2:  y_r = d2;
         4'd3:  y_r = d3;
         4'd4:  y_r = d4;
         4'd5:  y_r = d5;
         4'd6:  y_r = d6;
         4'd7:  y_r = d7;
         4'd8:  y_r = d8;
         4'd9:  y_r = d9;
         4'd10: y_r = d10;
         4'd11: y_r = d11;
         4'd12: y_r = d12;
         4'd13: y_r = d13;
         4'd14: y_r = d14;
         4'd15: y_r = d15;
         default: ;
     endcase
 end
 assign y = y_r;
endmodule
```

8.3. 本CPU中涉及到的多路选择器

> 再此就不详细展开了，详见控制器的设计，和整体设计中的CPU数据通路