Verilog 单周期处理器

一、主要模块

1、PC（PC.v）

初始值：0x30000000

端口定义：

```
module PC(
input clk,//时钟
input reset,//同步复位
input [31:0]next,//下一条地址
output reg [31:0]IAddr=32'h00003000//当前指令地址
);
```

2、IM（im.v）

容量：32bit*1024 字，地址 10 位。只读，不可写。

端口定义：

```
module Instr_Memory(
input [9:0]RAddr,//指令地址后 10 位
output [31:0]RData//指令机器码
);
```

存储部件：`reg [31:0] rom[0:1023];`

初始化：`$readmemh("code.txt",rom);`

3、GRF（GRF.v）

0 号寄存器恒为 0

端口定义：

```
module GRF(
input clk,//时钟信号
input WEnable,//写使能
input reset,//同步复位
input [4:0]RAddr1,//读地址 1
input [4:0]RAddr2,//读地址 2
input [4:0]WAddr,//写地址
input [31:0]WData,//写数据
input [31:0]IAddr,//当前指令地址，仅用于控制台输出
output [31:0]RData1,//读数据 1
output [31:0]RData2//读数据 2
);
```

4、ALU（ALU.v）

端口定义：

```
module ALU(
input [31:0]op1,//操作数 1
input [31:0]op2,//操作数 2
input [3:0]sel,//功能选择
```

```
output [31:0]result,//计算结果
output zero//计算结果为 0 标志位
);
```

选择信号：

0000---非负	0001---负数	0010---加法	0011---减法
0100---按位与	0101---按位或	0110---按位异或	0111---按位或非
1000---逻辑右移	1001---算术右移	1010---左移	1011---相等
1100---有符号小于	1101---无符号小于	1110---正数	1111---≤0

5、ext（ext.v）

容量：32bit*1024 字，地址 10 位

端口定义：

```
module ext(
input [15:0]imm,//待扩展 16 位数
input [1:0]EOp,//扩展方式
output [31:0]ext//扩展后的 32 位数
);
```

扩展方式：

00：符号扩展

01：无符号扩展

10：加载至高 16 位(lui)

11：符号扩展之后，左移两位

6、DM（DM.v）

容量：32bit*1024 字。地址 10 位，可读可写可复位（同步复位）。

端口定义：

```
module Data_Memory(
input clk,//时钟信号
input WE,
input reset,//同步复位
input [9:0]Addr,//数据地址
input [31:0]WData,//写数据
input [31:0]IAddr,//指令地址，仅用于控制台输出
output [31:0]RData//读数据
);
```

存储部件：`reg [31:0] ram[0:1023];`

7、Controller(Controller.v)

端口定义：

```
module Controller(
input [31:0]cmd,      //指令码
output Jump,          //跳转信号
output [1:0]RegSrc,  //寄存器数据来源
```

```
output MemWrite,     //写内存信号
output Branch,       //分支信号
output ALUSrc,       //ALU 操作数 2 来源
output [1:0]RegDst, //寄存器写地址选择
output RegWrite,    //写寄存器信号
output [1:0]ExtOp,  //位扩展方式
output [3:0]ALUCtrl //ALU 运算选择
);
```

详细解释：

RegSrc：00：ALU　01：DM　10：PC+4

RegDst：00：Instr[20:16]　01：Instr[15:11]　10：31（$ra）

二、数据通路

方法：定义一些 wire 型变量用于接线。之后，这些接线把模块串起来。有一些信号是由指令决定的，需要 assign，串连模块时需要用实例调用语句。对外接口只有 clk 时钟信号和 reset 同步复位。

顶层模块定义：

```
module mips(
input clk,
input reset
);
```

按照 p3 的数据通路，把模块连起来。

```
assign temp=IAddr+4;
assign next=Jump?(ALUSrc?{temp[31:28],Instr[25:0],2'b00}:RData1):((Branch&&zero)?temp+Imm:temp);
assign RegWAddr=RegDst[1]?5'd31:(RegDst[0]?Instr[15:11]:Instr[20:16]);
assign SrcB=ALUSrc?Imm:RData2;
assign WData=RegSrc[1]?temp:(RegSrc[0]?RData:ALUResult);

PC pcount(clk,reset,next,IAddr);
Instr_Memory im(IAddr[11:2],Instr);
Controller ctrl(Instr,Jump,RegSrc,MemWrite,Branch,ALUSrc,RegDst,RegWrite,ExtOp,ALUCtrl);
GRF rf(clk,RegWrite,reset,Instr[25:21],Instr[20:16],RegWAddr,WData,IAddr,RData1,RData2);
ALU a(RData1,SrcB,ALUCtrl,ALUResult,zero);
ext extender(Instr[15:0],ExtOp,Imm);
Data_Memory dm(clk,MemWrite,reset,ALUResult[11:2],RData2,IAddr,RData);
```

三、控制器设计

支持的指令集：addu　subu　ori　lui　lw　sw　beq　j　jal　jr　nop

译码器：采用 case 语句实现。定义的控制信号拼接起来，然后用一个数字赋值。数字按信号用下划线隔开，译码语句后面加上单行注释（助记符），便于区分和修改。不要指定数字位数，因为默认位宽为 32 位，对于控制信号已经足够。

定义 `reg [14:0]temp;//`这里 temp 的位宽必须和控制信号位数总和相同

```
assign {ExtOp,RegWrite,RegDst,ALUSrc,Branch,MemWrite,RegSrc,Jump,ALUCtrl}=temp;
```

然后，像 p3 一样列表格，直接写合并后的真值表：

指令	Opcode	Funct	Jump	ExtOp	RegSrc	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
addu	000000	100001	0	xx	00	0	0	0010	0	01	1
subu	000000	100011	0	xx	00	0	0	0011	0	01	1
ori	001101	xxxxxx	0	01	00	0	0	0101	1	00	1
lw	100011		0	00	01	0	0	0010	1	00	1
sw	101011		0	00	00	1	0	0010	1	xx	0
beq	000100		0	11	00	0	1	0011	0	xx	0
lui	001111		0	10	00	0	0	0010	0	00	1
j	000010		1	xx	xx	0	x	xxxx	x	xx	0
jal	000011		1	xx	10	0	x	xxxx	x	10	1
jr	000000	001000	1	xx	xx	0	x	xxxx	x	xx	0

下面就可以写 case 语句了（所有的 x 均用 0 代替）。

```
case(cmd[31:26])
    0:case(cmd[5:0])
        8:temp='b00_0_00_0_00_00_1_0000;//jr
        33:temp='b00_1_01_0_00_00_0_0010;//addu
        35:temp='b00_1_01_0_00_00_0_0011;//subu
    endcase
    2:temp='b00_0_00_1_00_00_1_0000;//j
    3:temp='b00_1_10_1_00_10_1_0000;//jal
    4:temp='b11_0_00_0_10_00_0_0011;//beq
    13:temp='b01_1_00_1_00_00_0_0101;//ori
    15:temp='b10_1_00_1_00_00_0_0101;//lui
    35:temp='b00_1_00_1_00_01_0_0010;//lw
    43:temp='b00_0_00_1_01_00_0_0010;//sw
endcase
```

四、测试

测试程序：

```
.data
arr:.space 404

.text
ori $a0,$0,100
jal memw
jal sum
j end

memw:
ori $s0,$0,1
ori $s1,$0,4
addu $s2,$a0,$s0
ori $t0,$0,1
ori $t1,$0,0
```

```
for:
beq $t0,$s2,next
sw $t0,arr($t1)
addu $t0,$t0,$s0
addu $t1,$t1,$s1
j for
next:
jr $ra

sum:
ori $s0,$0,1
ori $s1,$0,4
ori $t0,$0,0
ori $t1,$0,0
ori $v0,$0,0
for1:
beq $t0,$a0,next1
lw $s2,arr($t1)
addu $v0,$v0,$s2
addu $t1,$t1,$s1
addu $t0,$t0,$s0
j for1
next1:
jr $ra

end:
li $s2,0x87654321
li $s3,0x12345678
subu $s4,$s2,$s3
nop
```

功能：用一个函数（包括 jal 和 jr，没有返回值），把 1-100 写入从 0 开始的连续内存（按字存储），然后用另一个函数读出来逐个求和，结果写入返回值$v0（2 号）寄存器。最后加载两个 32 位立即数（可拆分为先加载高 16 位再或低 16 位）并相减。

期望结果：

寄存器：

$1:0x12340000

$2:0x000013ba（十进制 5050）

$4:0x00000064

$8:0x00000064

$9:0x00000190

$16:0x00000001

$17:0x00000004

$18:0x87654321

$19:0x12345678

$20:0x7530eca9

```
$31:0x0000300c
```

内存：0-0x18c 号字节地址（第 0-99 个字）分别对应 1-100。

导出机器码：

```
34040064 0c000c04 0c000c0f 08000c1b 34100001
34110004 00909021 34080001 34090000 11120004
ad280000 01104021 01314821 08000c09 03e00008
34100001 34110004 34080000 34090000 34020000
11040005 8d320000 00521021 01314821 01104021
08000c14 03e00008 3c018765 34324321 3c011234
34335678 0253a023 00000000
```

testbench 如下：

```
module test;
   reg clk,reset;
   initial begin
      clk=0;reset=0;
   end
   always #10 clk=~clk;
   mips cpu(clk,reset);
endmodule
```

仿真器中一直运行，直到所有寄存器稳定下来。

结果与 Mars 中完全一致。

五、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？

地址为[11:2]时，可以截取传入地址的第 11 位到第 2 位信号。由于 DM 传入地址位字节地址，而存储器是按字存储，相差 4 倍，因此后两位必然为 0，要取第 11 到 2 位作为字地址。addr 信号位寻址结果，因此来自 ALU 运算结果。

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？

针对 PC、DM、RF。PC 指示指令地址，复位时需要回到初始地址（Mars 里面为 0x3000）以便重新执行。DM、RF 为临时存储元件，重新执行之前要回到初始状态，避免上一次运行结果对后面产生影响。

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。

4、根据你所列举的编码方式，说明他们的优缺点。

1、与或逻辑：类似于 Logisim 里面，用指令的 Op 字段和 Funct 字段生成指令识别码（与逻辑），然后用指令识别码生成控制信号（或逻辑）。

例如：`J` 指令：

`Op=000010`。因此识别码：

```
j=~opcode[5]&&~opcode[4]&&~opcode[3]&&~opcode[2]&&opcode[1]&&~opcode[0]
```

Verilog 语句：`j=~Instr[31]&&~Instr[30]&&~Instr[29]&&~Instr[28]&&Instr[27]&&~Instr[26];`

然后或逻辑生成控制信号。这里，跳转指令需要使 jump 信号为真，因此 `jump=j;`（还有 `jal,jr,jalr` 也需要为真）

优缺点：不易出错，便于添加指令和修改编码，但代码冗长不易读。

2、case 语句直接编码：

控制器设计里面已经说明。注意数字要用下划线适当分隔。

优缺点：便于修改维护，但不便于扩展控制信号位数。另外，对应关系不明晰，容易出错。

3、宏定义：给不同的指令字段起不同的名称，然后再直接或逻辑生成控制信号。例如刚才的 J 指令：

Op=000010。文件开头加定义：`define J 6’b000010

控制信号 jump=j;

优缺点：和与或逻辑等效，不易出错，且易修改维护和添加指令。但是有大量宏定义需要写，对于多字段决定的指令比较困难。

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。

有符号溢出，就是计算结果符号位被进位（或借位）覆盖导致符号错误。在 add、addi 指令中，发生这种情况将会产生 IntegerOverflow 异常信号（错误算术运算）导致程序终止。但是若忽略溢出，二进制补码的加法方式决定有符号加法和无符号加法是等价的，不会发生异常就等价于无符号加法。但是如果强行使用有符号方式解读，则可能出现错误的运算结果。

6、根据自己的设计说明单周期处理器的优缺点。

优点：数据通路简单，没有延时槽，没有数据冲突和控制冲突。

缺点：和实际硬件不同，数据存储器和指令存储器分开，而在实际的体系结构中不可能。而且，寻址需要另立加法器，而算术器件是比较耗费晶体管和时间的。最致命的是，不同指令由于关键路径不同而延迟时间不同，导致时钟周期由最慢的指令决定，这严重降低执行效率。

7、简要说明 jal、jr 和堆栈的关系。

jal：跳转并链接，相当于函数入口，转移 PC 至入口地址，并把返回地址保存在 31 号寄存器中。如果其中有临时变量和参数则需要开堆栈空间（栈顶要随着移动）以保存，以便在函数结束时退栈并恢复这些值。必须保存的值是$ra。参数和临时变量视情况可保存可不保存，但在调用其它函数（并且这个函数用到了上述寄存器）之后仍想要重新使用原来值的变量必须保存（比如多层调用或递归调用）。

jr 相当于函数返回，此时从栈空间中恢复保存的变量，并退栈（栈顶回到调用前状态）转到返回地址（即上一次调用的下一条指令）。

总结：jal 时进栈，jr 时退栈。