

# 单周期 CPU

## 一、设计目的与说明

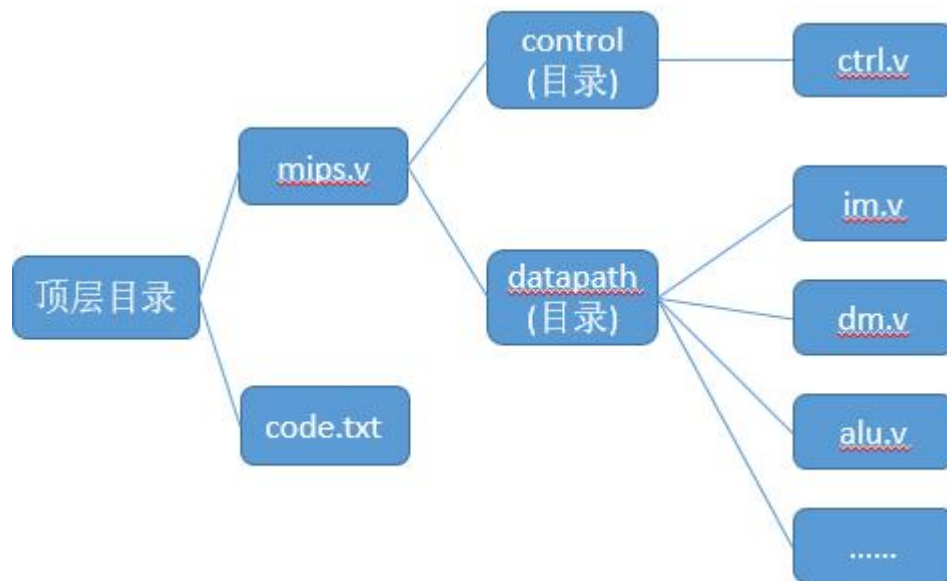
使用 verilog 搭建一个指令的单周期 CPU

处理器应支持的指令集为: {addu, subu, ori, lw, sw, beq, lui, j, jal, jr, nop};

Addu, subu 不支持溢出;

不需要考虑延迟槽。

采用模块化和层次设计, 顶层有效的驱动信号要求仅包括 clk 和 reset;



## 二、单周期数据通路设计

指令	Addr		PC	IM. A	GRF			
	A	B			RA1	RA2	WA	WD
addu	PC	4	Addr	PC	Rs	Rt	Rd	ALU
subu	PC	4	Addr	PC	Rs	Rt	Rd	ALU
ori	PC	4	Addr	PC	Rs		Rt	ALU
lw	PC	4	Addr	PC	Rs		Rt	DM. Rd
sw	PC	4	Addr	PC	Rs	Rt		
beq	PC	4	Addr   Nadd	PC	Rs	Rt		
lui	PC	4	Addr	PC	\$0		Rs	ALU
j	PC	4	Addr   Nadd	PC				
jal	PC	4	Addr   Nadd	PC			31	PC+4
jr	PC	4	Addr	PC	Rs			PC
R	PC	4	Addr	PC	Rs	Rt	Rd	ALU

```
assign rs = instr[25:21];
```

```
assign rt = instr[20:16];
```

```
assign rd = instr[15:11];
```

```
assign offset = instr[15:0];

assign op = instr[31:26];

assign func = instr [5:0];
```

### 三、 通路设计


文件	模块接口定义
mips.v	<pre>module mips(clk,reset);     input  clk; //clock     input  reset; //reset</pre>

#### 1、ifu

包含 PC（程序计数器）,IM（指令存储器）及相关逻辑。

PC 用寄存器实现，应具有复位功能。

复位后，PC 指向 **0x0000\_3000**，此处为第一条指令的地址。注意与 **MARS 中的设置**保持一致。若在复位期间对某些存储单元进行了复位操作请不要输出。

 MIPS Memory Configuration

Configuration

☐ Default
 ☒ Compact, Data at Address 0
 ☐ Compact, Text at Address 0

0x00007fff	memory map limit address
0x00007fff	kernel space high address
0x00007f00	MMIO base address
0x00007eff	kernel data segment limit address
0x00005000	.kdata base address
0x00004ffc	kernel text limit address
0x00004180	exception handler address
0x00004000	kernel space base address
0x00004000	.ktext base address
0x00003fff	user space high address
0x00003ffc	text limit address
0x00003000	.text base address
0x00002fff	data segment limit address
0x00002ffc	stack pointer \$sp
0x00002ffc	stack base address
0x00002000	stack limit address
0x00002000	heap base address
0x00001800	global pointer \$gp
0x00001000	.extern base address
0x00000000	data segment base address
0x00000000	.data base address

Apply and Close

Apply

Cancel

Reset

Ifu 代码:

```

module ifu(
    input clk,

    input reset,

    input [31:0]offset_ext,

    input [31:0]RData1,

    input [25:0]instr_index,

    input [2:0] NPCOp,

    input Zero,

    output [31:0]NPC,

    output [31:0]instr,

```

```

        output reg[31:0]jalpc
    );

    reg [31:0]PC;

    reg [31:0]im[1023:0];

    assign instr = im[PC[11:2]];

    assign NPC = PC + 4;

    initial

    begin

        $readmemh("code.txt",im);

        PC<=32'h3000;

    end

    always @(posedge clk or posedge reset)

    begin

        PC <= {PC[31:28],(instr_index<<2)};

        jalpc <= PC + 8;

        if (reset)

            PC <= 32'h3000;

        else begin

            if (NPCOp==1 && Zero==1)//beq

                PC <= PC + 4 + (offset_ext<<2);

            else if (NPCOp==2)//j

                PC <= {PC[31:28],(instr_index<<2)};

            else if (NPCOp==3)//jal

                PC <= {PC[31:28],(instr_index<<2)};

            else if (NPCOp==4)//jr

                PC <= RData1;

            else PC <= PC+4;

        end

    end

endmodule

```

包括对于 beq, j, jal, jr 的设计。

## 2、GRF

```
module GPR(  
    input clk,  
    input reset,  
    input [4:0] Rs,  
    input [4:0] Rt,  
    input [4:0] RegAddr,  
    input RegWrite,  
    input [31:0] RegData,  
    input [31:0] jalpc,  
    output [31:0] RData1,  
    output [31:0] RData2  
);  
  
reg [31:0] _reg[31:0];  
  
integer i;  
  
assign RData1=_reg[Rs];  
assign RData2=_reg[Rt];  
  
initial begin  
    for (i=0;i<32;i=i+1)  
        _reg[i]<=32'b0;  
end  
  
always @(posedge clk or posedge reset) begin  
    if (reset)  
        for (i=0;i<32;i=i+1)  
            _reg[i] <= 32'b0;  
    else if(RegWrite)begin  
        if (RegAddr==0)  
            _reg[RegAddr] <= 32'b0;  
        else if (RegAddr!=5'b11111)begin
```

```

        _reg[RegAddr] <= RegData;

        $display("%d <= %h", RegAddr, RegData);

    end

    else if (RegAddr==5'b11111)begin

        _reg[RegAddr] <= jalpc;

        $display("%d <= %h", RegAddr, jalpc);

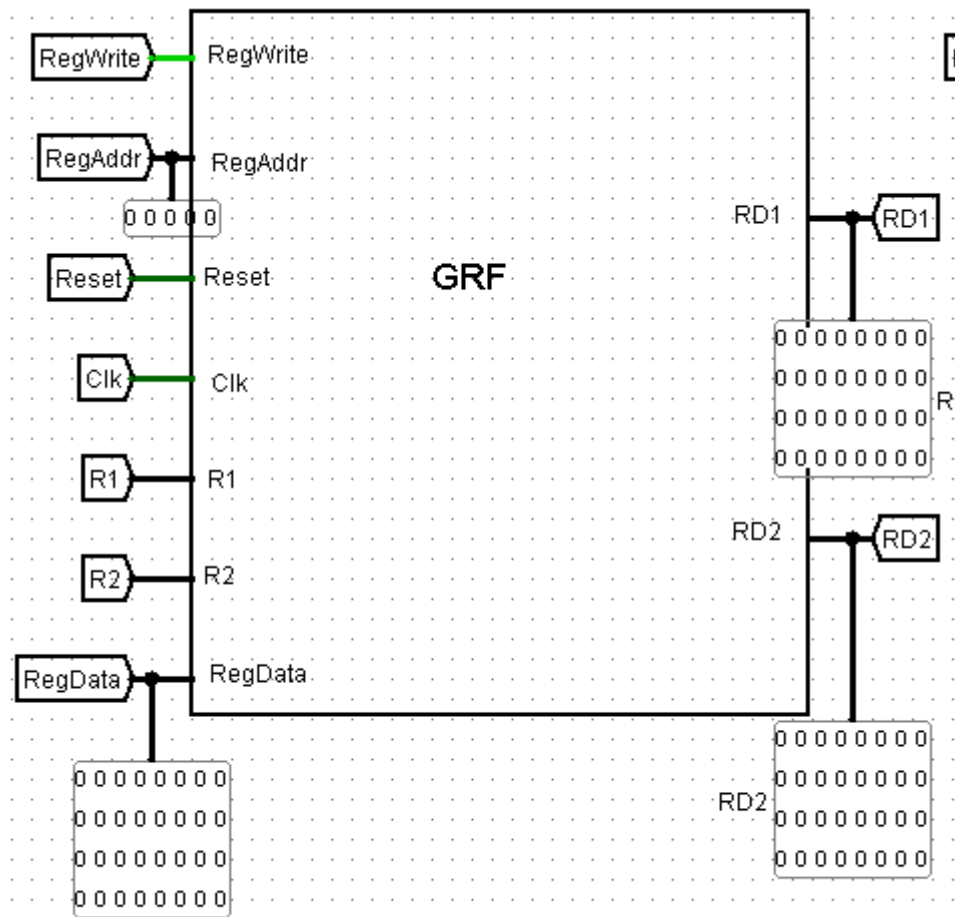
    end

end

end

endmodule

```



类比 logisim 的设计。

在 grf 模块中，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

`$display("$%d <= %h", Waddr, WData);`其中 Waddr 表示输入的 5 位写寄存器的地址，WData 表示输入的 32 位写入寄存器的值。

在 grf 中有 32 个 32 位的寄存器。0 号寄存器保持为 0。

### 3、ALU（算数逻辑单元）

```
module alu(  
  
    input [31:0] ALU_A,  
  
    input [31:0] ALU_B,  
  
    input [1:0] ALUOP,  
  
    output reg[31:0] ALU_C,  
  
    output reg ALU_ZERO  
  
);  
  
always @(*)begin  
  
    case (ALUOP)  
  
        2'b00: ALU_C =ALU_A + ALU_B;  
  
        2'b01: ALU_C =ALU_A - ALU_B;  
  
        2'b10: ALU_C =ALU_A | ALU_B;  
  
    endcase  
  
    if (ALU_C)  
  
        ALU_ZERO=0;
```

```

        else

            ALU_ZERO=1;

        end

    endmodule

```

提供 32 位加、减、或运算及大小比较功能

可以不支持溢出（不检测溢出）。

#### 4、DM（数据存储器）

```

module DM(

    input [31:0] addr,

    input [31:0] din,

    input we,

    input re,

    input clk,

    input reset,

    output [31:0] dout

);

    reg [31:0] DM[1023:0];

    integer i;

    assign dout =DM[addr[11:2]];

    initial begin

        for(i = 0; i < 1024; i = i+1)

            DM[i] = 32'b0;

        end

    always @(posedge clk or posedge reset)

        begin

```





```

always @*

begin

    case (EXTOP)

        2'b00:Dout = {16'b0,Din}; //无符号扩展

        2'b01:Dout = {Din,16'b0}; //lui

        2'b10:Dout = {16*Din[15],Din}; //有符号扩展

        default:Dout = 32'bx;

    endcase

end

endmodule

```

#### 四、Controller（控制器）

控制信号	失效时作用	有效时作用
RegDst	寄存器堆写入端地址来选择Rt字段	寄存器堆写入端地址选择 Rd字段
RegWrite	无	把数据写入寄存器堆中对应寄存器
ALUSrc	ALU输入端B选择寄存器堆输出R[rt]	ALU输入端B选择Signext输出
PCSrc	PC输入源选择 PC+4	PC输入选择beq指令的目的地址
MemRead	无	数据存储器DM读数据（输出）
MemWrite	无	数据存储器DM写数据（输入）
MemtoReg	寄存器堆写入端数据来自ALU输出	寄存器堆写入端数据来自DM输出

	addu	subu	ori	lw	sw	beq	lui	j
NPCOp	000	000	000	000	000	001	000	010
ALUOp	00	01	10	00	00	00	00	00
RegWrite	1	1	1	1	0	0	1	0
EXTOp	00	00	00	10	10	10	01	00
MemWrite	0	0	0	0	1	0	0	0
RegDst	01	01	00	00	00	00	00	00
ALUSrc	0	0	1	1	1	0	1	0
MemtoReg	00	00	00	01	00	00	00	00

```

wire R, addu, subu, jr; //R

wire ori, lw, sw, beq, lui; //I

wire j, jal; //J

//R

assign R=~op[5] && ~op[4] &&~op[3] &&~op[2] &&~op[1] &&~op[1];

```

```

    assign addu = R & func[5] & ~func[4] & ~func[3] & ~func[2] & ~func[1] &
func[0]; //100001

    assign subu = R & func[5] & ~func[4] & ~func[3] & ~func[2] & func[1] &
func[0]; //100011

    assign jr    = R & ~func[5] & ~func[4] & func[3] & ~func[2] & ~func[1] &
~func[0]; //001000

//I

    assign ori = ~op[5] & ~op[4] & op[3] & op[2] & ~op[1] & op[0];
    assign lw  = op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0];
    assign sw  = op[5] & ~op[4] & op[3] & ~op[2] & op[1] & op[0];
    assign beq = ~op[5] & ~op[4] & ~op[3] & op[2] & ~op[1] & ~op[0];
    assign lui = ~op[5] & ~op[4] & op[3] & op[2] & op[1] & op[0];

//J

    assign j = ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & ~op[0];
    assign jal = ~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0];
    assign NPCOp = (beq)?3'b001:

                (j)  ?3'b010:

                (jal)?3'b011:

                (jr) ?3'b100:

                3'b000;

    assign ALUOp[0] = subu;
    assign ALUOp[1] = ori;
    assign RegWrite = ori || R || lui || lw || jal;
    assign EXTOp[0] = lui;
    assign EXTOp[1] = beq || lw || sw;
    assign MemWrite = sw;
    assign RegDst[0] = R;
    assign RegDst[1] = jal;
    assign ALUSrc = lw || lui || sw || ori;
    assign MemtoReg[0] = lw;

```

```
assign MemtoReg[1] = jal;
```

## 五、测试程序

```
ori $8,$8,1

ori $9,$9,0

ori $10,$10,2

ori $11,$11,4

lui $12,9

lui $13,6

addu $8,$8,$10

addu $8,$8,$12

subu $13,$13,$11

subu $13,$13,$9

sw $13,($11)

ori $13,$13,100

lw $9,($11)

jal label1

subu $9,$9,$13

beq $9,$0,label2

label1:ori $9,$9,100

addu $10,$9,$9

jr $31

label2:beq $9,$0,end

end:
```

以上为测试机器码翻译的结果。

## 六、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	<pre> 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 </pre>

答：因为我们声明有 1024 个寄存器，所以需要十位，又因为每条信息查四个地址，若保持同步则前移两位。

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？

IFU, GRF, DM

因为当我们测试程序上电后，会有若干周期 reset 信号保持高电平。根据评测的设计，这数个周期内，RAM 和 GRF 不应写入任何数据。（换言之，该两元件应该同时支持 reset 操作且优先级都比写入高）。如果写入了，则会出现指令错后一条的现象（写入被 Logisim 的 logging 记录成为第一条指令，尽管此时 reset 为 1 不应执行指令）。因此，现象的实质是第一条指令被执行两次导致指令错位。因此，当 reset 为高电平时，此时应该整个电路都要被初始化。

时序逻辑有存储功能，故需要清零。

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. 直接判断：

```
Assign ori = ~op[5]&&~op[4]&&~op[3]&&~op[2]&&op[1]&&op[0];
```

2. if else

```
if (~op[5]&&~op[4]&&~op[3]&&~op[2]&&op[1]&&op[0])
ori = 1;
```

3. case

```
Case (op)
6' b000011:ori = 1;
```

4、根据你所列举的编码方式，说明他们的优缺点。

Assign 语句是组合逻辑，运行速度快；  
剩下两个都要声明一个 reg，所以是时序逻辑，运行速度慢，有时出现时钟周期现象；

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。

操作	<pre>temp ← (GPR[rs]<sub>31</sub>  GPR[rs]) + (GPR[rt]<sub>31</sub>  GPR[rt]) if temp<sub>32</sub> ≠ temp<sub>31</sub> then     SignalException(IntegerOverflow) else     GPR[rd] ← temp<sub>31..0</sub> endif</pre>
示例	<pre>add \$s1, \$s2, \$s3</pre>
其他	<pre>temp<sub>32</sub> ≠ temp<sub>31</sub> 代表计算结果溢出。 如果不考虑溢出，则 add 与 addu 等价。</pre>

当 temp<sub>32</sub> ≠ temp<sub>31</sub> 表示结果溢出。  
根据指令描述分析，只有在两个加数都是负数的情况下才会产生溢出，溢出的一位 1 相当于符号位(代表负数)，其他情况下两者的计算结果完全相同且正确。所以在不考虑溢出的情况下，溢出位即符号位被忽略，add, addi 与 addu, addiu 的结果是一致的(虽然此时结果与理论结果不一致)。

2. 根据自己的设计说明单周期处理器的优缺点。

优点：比多周期速度快，单条指令执行时间短，多周期会由指令本身决定一条指令需要几个周期。  
缺点：对于单周期的 CPU 来说，每条指令执行都需要一个周期，一条指令执行完再执行下一条指令。就是说，单周期 CPU 来说处理指令的 5 个阶段是串行执行的，耗时长。

5、简要说明 jal、jr 和堆栈的关系。

➤ jal指令: `jal label`

第一步: 将返回地址(下一条指令地址)送寄存器 `$ra`

第二部: 跳转至函数`label`

## ❖ MIPS从函数返回指令

➤ jr指令: `jr $ra`

寄存器转跳指令, 可转跳至寄存器`$ra`的值所指向位置

通过跳转语句 `jr`, 程序可以返回到造成异常的那条指令处继续执行。

栈指针的上下需要显示的通过指令来实现。因此 `mips` 通常只在子函数进入和退出的时刻才调整堆栈的指针。这通过被调用的子函数来实现。`sp` 通常被调整到这个被调用的子函数需要的堆栈的最低的地方, 从而编译器可以通过相对于 `sp` 的偏移量来存取堆栈上的堆栈变量。

子函数的返回, 使用 `jr $ra`, 如果子函数内又调用了其他的子函数, 那么`$ra` 的值应该被保存到堆栈中。 因为`$ra` 的值总是对应着当前执行的子函数的返回地址。

`Jal` 实现进栈操作, `jr` 实现出栈操作。

## 七、自己测试的代码

### 1、Addu/subu/lui

```
3c080011
```

```
3c09000a
```

```
01095021
```

```
01095823
```

```
lui $8, 17
```

```
lui $9, 10
```

```
addu $10, $8, $9
```

```
subu $11, $8, $9
```

运行结果与 `mars` 一致

```
$ 8 <= 00110000
```

```
$ 9 <= 000a0000
```

```
$10 <= 001b0000
```

```
$11 <= 00070000
```

8	0x00110000
9	0x000a0000
10	0x001b0000
11	0x00070000

2、lui \$8,17

lui \$9,10

addu \$10,\$8,\$9

subu \$11,\$8,\$9

ori \$12,\$8,100

sw \$12,(\$13)

lw \$14,(\$13)

3c080011

3c09000a

01095021

01095823

350c0064

adac0000

8dae0000

8	0x00110000
9	0x000a0000
10	0x001b0000
11	0x00070000
12	0x00110064
13	0x00000000
14	0x00110064

\$ 8 <= 00110000

\$ 9 <= 000a0000

\$10 <= 001b0000

\$11 <= 00070000

\$12 <= 00110064

\*00000000 <= 00110064

\$14 <= 00110064

3、

#addu, subu, ori, lw, sw, beq, lui, j, jal, jr

lui \$8,17

lui \$9,10



```

addu $10, $8, $9
subu $11, $8, $9
ori $12, $8, 100
sw $12, ($13)
lw $14, ($13)
lui $9, 17
beq $8, $9, label1
label1: jal label2
subu $15, $10, $8
beq $8, $9, end
label2:
    addu $11, $10, $8
    jr $31
end:
$ 8 <= 00110000
$ 9 <= 000a0000
$10 <= 001b0000
$11 <= 00070000
$12 <= 00110064
*00000000 <= 00110064
$14 <= 00110064
$ 9 <= 00110000
$31 <= 00003028
$11 <= 002c0000
$15 <= 000a0000

```

\$t0	8	0x00110000
\$t1	9	0x00110000
\$t2	10	0x001b0000
\$t3	11	0x002c0000
\$t4	12	0x00110064
\$t5	13	0x00000000
\$t6	14	0x00110064
\$t7	15	0x000a0000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x00001800
\$sp	29	0x00002ffc
\$fp	30	0x00000000
\$ra	31	0x00003028
pc		0x00003038