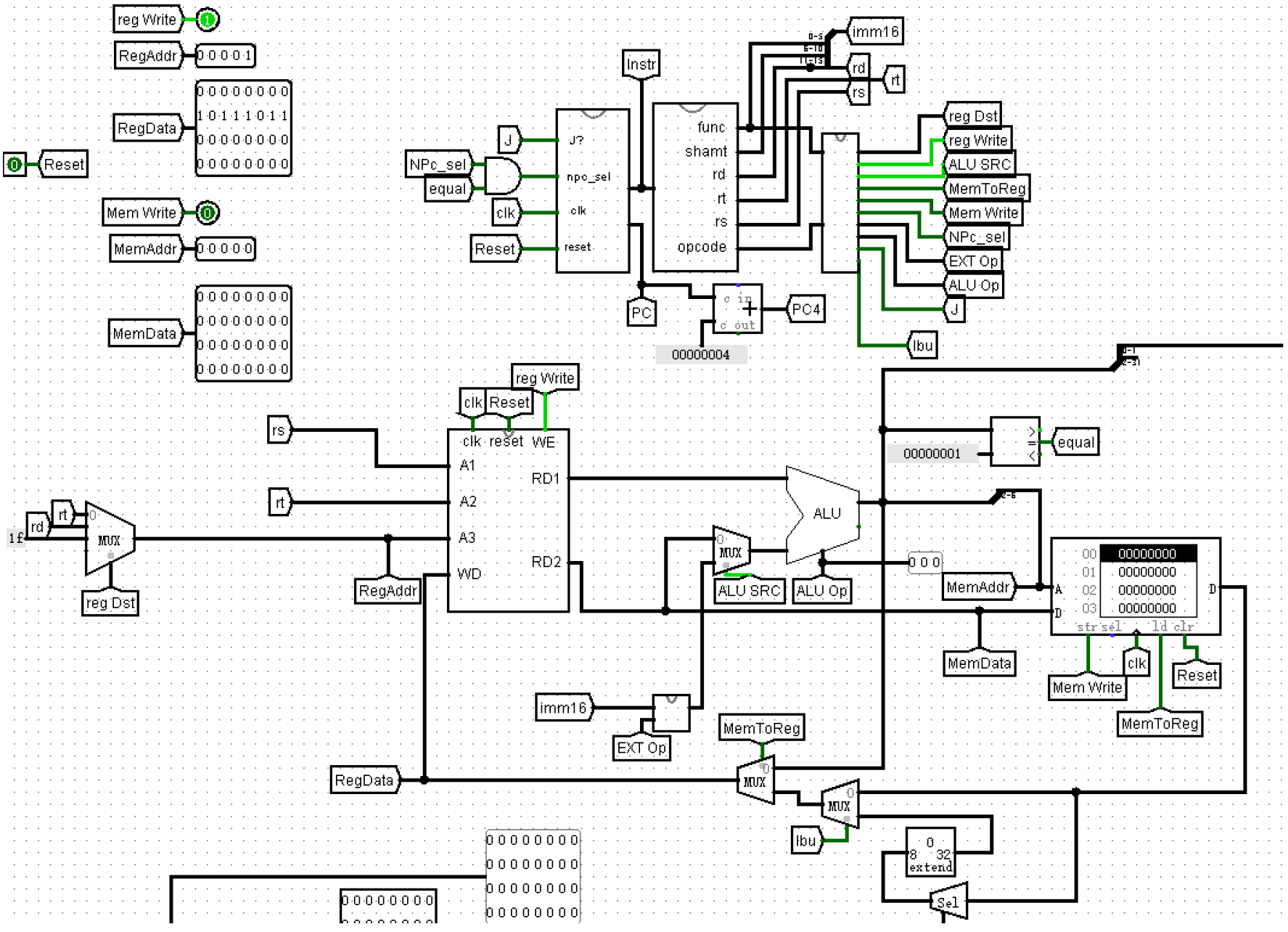
Verilog 单周期处理器

一、 设计说明

本处理器支持 MIPS 指令集中的 addu, subu, ori, lw, sw, beq, lui, j, jal, jr,nop 指令，其中 addu 和 subu 不支持进位功能。

二、 Logisim 视图



三、 主要模块

1、PC

pc 初始值：32'h0000_3000

端口定义：

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

2、npc

实现三种跳转，jarpc 输出 pc+4

端口定义：

```
module npc(
    input [31:0] pc,
    input npc_sel,
```

```
    input equal,
    input jump,
    input clk,
    input reset,
    input jr,
    input [31:0] gpr,
    input [31:0] imm32,
    input [25:0] imm26,
    output reg [31:0] npc,
    output [31:0] jarpc
    );
```

3、IM

容量：32bit*1024 字，地址 10 位。只读，不可写。输出按指令分开。

端口定义：

```
module im_4kb(
    input [31:0] Pc,
    output reg [5:0] opcode,
    output reg [4:0] rs,
    output reg [4:0] rt,
    output reg [4:0] rd,
    output reg [4:0] shamt,
    output reg [5:0] func,
    output reg [15:0] imm16,
    output reg [25:0] imm26,
    output reg [31:0] op
    );
```

存储部件：`reg [31:0] im[0:1023];`

初始化：`$readmemh("code.txt",im);`

4、GRF

0 号寄存器恒为 0

端口定义：

```
module grf(
    input [4:0] a1,
    input [4:0] a2,
    input [4:0] a3,
    input [31:0] WriteData,
    input clk,
    input reset,
    input regWrite,
    output [31:0] RD1,
    output [31:0] RD2,
    input [31:0] WPC
```

```
    );
```

5、ALU（ALU.v）

端口定义：

```
module alu(
    input [31:0] a,
    input [31:0] b,
    input [3:0] aluctr,
    output reg [31:0] result,
    output reg equal
    );
```

选择信号：

0000---加法　　0001---减法　　0010---按位与　　0011---相等

6、ext（ext.v）

端口定义：

```
module EXT(
    input [15:0] imm16,
    input [3:0] ext_op,
    output reg [31:0] imm32
    );
```

扩展方式：

0000：0 扩展　0001：符号扩展　　0010：加载至高 16 位(lui)　0011：符号扩展之后，左移两位

7、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 dm_4kb(
    input [31:0] pc,
    input [31:0] DataAddress,
    input MemWrite,
    input reset,
    input [31:0] DataIn,
    input clk,
    output [31:0] DataOut
    );
```

存储部件：`reg [31:0] ram[0:1023];`

8、Controller(Controller.v)

端口定义：

```
module ctl(
    input [5:0] opcode,
    input [5:0] func,
    output reg [1:0] Reg_Dst,
    output reg Reg_Write,
```

```
    output reg Alu_Src,
    output reg MemToReg,
    output reg Mem_Write,
    output reg NPc_Sel,
    output reg [3:0] Ext_Op,
    output reg [3:0] Alu_Op,
    output reg J,
    output reg jal,
    output reg jr,
    output reg tmp
    );
```

详细解释：

RegSrc：00：ALU　01：DM

RegDst：00：rt　01：rd　10：31

Tmp 预留加指令使用

8、MUX

内涵三种不同的复选器，32 位二选一、32 位四选一、5 位二选一。

四、 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。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
);
   EXT          Ext(imm16[15:0], Ext_Op, EXTout);
   mux5         regdst(Reg_Dst[1:0], rt[4:0], rd[4:0], 5'b11111, RegAddress[4:0]); //rt\rd\31
   grf          Grf(rs[4:0], rt[4:0], RegAddress[4:0], DataToReg[31:0], clk, reset, Reg_Write,
rd1[31:0], rd2[31:0], Pc);
       mux32in2  MuxALUSRC(Alu_Src, rd2[31:0], EXTout[31:0], DataToALU[31:0]);
       alu          ALU(rd1[31:0], DataToALU[31:0], Alu_Op[3:0], Aluresult[31:0], equal);
       dm_4kb       DM(Pc, Aluresult, Mem_Write, reset, rd2, clk, DataOut);
       mux32in2  MuxALUOrMem(MemToReg, Aluresult[31:0], DataOut, RegData);
       mux32in2     MuxRegData(jal, RegData, jarpc, DataToReg);
```

五、 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：使用与或门，搭建类似于 logisim 单周期的结构。

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	nPc_Sel	ALUCtrl	ALUSrc	RegDst	RegWrite	Jal	Jr
Nop	000000	000000	0	0000	0	0	0	0000	0	00	0	0	0

addu	000000	100001	0	Xxxx	0	0	0	0010	0	01	1	0	0
subu	000000	100011	0	Xxxx	0	0	0	0011	0	01	1	0	0
ori	001101	xxxxxx	0	0001	0	0	0	0101	1	00	1	0	0
lw	100011		0	0000	1	0	0	0010	1	00	1	0	0
sw	101011		0	0000	0	1	0	0010	1	xx	0	0	0
beq	000100		0	0011	0	0	1	0011	0	xx	0	0	0
lui	001111		0	0010	0	0	0	0010	0	00	1	0	0
j	000010		1	xxxx	X	0	x	xxxx	x	xx	0	0	0
jal	000011		1	xxxx	0	0	x	xxxx	x	10	1	1	0
jr	000000	001000	1	xxxx	X	0	x	xxxx	x	xx	0	0	1

下面就可以写 case 语句了（所有的 x 均用 0 代替）。

```
wire addu;
wire subu;
wire ori;
wire lw;
wire sw;
wire beq;
wire lui;
wire j;
wire jal;
wire jr;
assign addu = !opcode[5] && !opcode[4] && !opcode[3] && !opcode[2] && !opcode[1] && !opcode[0] && func[5]
&& !func[4] && !func[3] && !func[2] && !func[1] && func[0];
assign subu = !opcode[5] && !opcode[4] && !opcode[3] && !opcode[2] && !opcode[1] && !opcode[0] && func[5]
&& !func[4] && !func[3] && !func[2] && func[1] && func[0];
assign jr = !opcode[5] && !opcode[4] && !opcode[3] && !opcode[2] && !opcode[1] && !opcode[0] && !func[5]
&& !func[4] && func[3] && !func[2] && !func[1] && !func[0];
assign ori = !opcode[5] && !opcode[4] && opcode[3] && opcode[2] && !opcode[1] && opcode[0];
assign lw = opcode[5] && !opcode[4] && !opcode[3] && !opcode[2] && opcode[1] && opcode[0];
assign sw = opcode[5] && !opcode[4] && opcode[3] && !opcode[2] && opcode[1] && opcode[0];
assign beq = !opcode[5] && !opcode[4] && !opcode[3] && opcode[2] && !opcode[1] && !opcode[0];
assign lui = !opcode[5] && !opcode[4] && opcode[3] && opcode[2] && opcode[1] && opcode[0];
assign j = !opcode[5] && !opcode[4] && !opcode[3] && !opcode[2] && opcode[1] && !opcode[0];
assign jal = !opcode[5] && !opcode[4] && !opcode[3] && !opcode[2] && opcode[1] && opcode[0];
always@(*)begin
    RegDst[1] = jal;
    RegDst[0] = addu || subu;
    RegWrite = addu || subu || ori || lw || lui || jal;
    AluSrc = ori || lw || sw || lui;
    MemToReg = lw;
    MemWrite = sw;
    NpcSel = beq;
    ExtOp[3] = 0;
    ExtOp[2] = 0;
    ExtOp[1] = lui || beq;
    ExtOp[0] = lw || sw || beq;
    AluOp[3] = 0;
    AluOp[2] = 0;
    AluOp[1] = ori || beq;
    AluOp[0] = subu || beq;
    J = j || jal;
    Jal = jal;
    Jr = jr;
    tmp = tmp;
end
```

六、 测试

测试程序：

```
ori $28,$0,0x00000000
ori $29,$0,0x00000000
ori $1,$0,0x00003456
addu $1,$1,$1
lw $1,0x00000004($0)
sw $1,0x00000004($0)
lui $2,0x00007878
subu $3,$2,$1
lui $5,0x00001234
ori $4,$0,0x00000005
nop
sw $5,0x0000ffff($4)
lw $3,0x0000ffff($4)
beq $3,$5,0x00000003
nop
beq $0,$0,0x00000011
nop
ori $7,$3,0x00000404
beq $7,$3,0x0000000e
nop
lui $8,0x00007777
ori $8,$8,0x0000ffff
subu $0,$0,$8
ori $0,$0,0x00001100
addu $10,$7,$6
ori $8,$0,0x00000000
ori $9,$0,0x00000001
ori $10,$0,0x00000001
ddu $8,$8,$10
beq $8,$9,0xfffffffe
jal 0x00000c22
nop
addu $10,$10,$10
beq $0,$0,0xffffffff
addu $10,$10,$10
jr $31
nop
```

期望结果：

```
$28 <= 00000000
$29 <= 00000000
$ 1 <= 00003456
$ 1 <= 000068ac
$ 1 <= 00000000
*00000004 <= 00000000
$ 2 <= 78780000
$ 3 <= 78780000
$ 5 <= 12340000
$ 4 <= 00000005
*00000004 <= 12340000
$ 3 <= 12340000
$ 7 <= 12340404
$ 8 <= 77770000
$ 8 <= 7777ffff
$ 0 <= 88880001
$ 0 <= 00001100
$10 <= 12340404
$ 8 <= 00000000
$ 9 <= 00000001
$10 <= 00000001
$ 8 <= 00000001
$ 8 <= 00000002
$31 <= 0000307c
$10 <= 00000002
$10 <= 00000004
```

导出机器码：

```
341c0000  341d0000  34013456  00210821  8c010004  ac010004  3c027878
00411823  3c051234  34040005  00000000  ac85ffff  8c83ffff  10650003
00000000  10000011  00000000  34670404  10e3000e  00000000  3c087777
3508ffff  00080023  34001100  00e65021  34080000  34090001  340a0001
010a4021  1109fffe  0c000c22  00000000  014a5021  1000ffff  014a5021
03e00008  00000000
```

testbench 如下：

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

实际输出：

```
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: $28 <= 00000000
@00003004: $29 <= 00000000
@00003008: $ 1 <= 00003456
@0000300c: $ 1 <= 000068ac
@00003010: $ 1 <= 00000000
@00003014: *00000004 <= 00000000
@00003018: $ 2 <= 78780000
@0000301c: $ 3 <= 78780000
@00003020: $ 5 <= 12340000
@00003024: $ 4 <= 00000005
@0000302c: *00000004 <= 12340000
@00003030: $ 3 <= 12340000
@00003044: $ 7 <= 12340404
@00003050: $ 8 <= 77770000
@00003054: $ 8 <= 7777ffff
@00003058: $ 0 <= 88880001
@0000305c: $ 0 <= 00001100
@00003060: $10 <= 12340404
@00003064: $ 8 <= 00000000
@00003068: $ 9 <= 00000001
@0000306c: $10 <= 00000001
@00003070: $ 8 <= 00000001
@00003070: $ 8 <= 00000002
@00003078: $31 <= 0000307c
@00003088: $10 <= 00000002
@00003080: $10 <= 00000004
```

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。

七、 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？

答：

因为在 dm 中的元素是按照字形式存储，而在 mars 中是按照字节存储，所以 mars 字与字之间相差 4 而在 dm 中则是相差 1。地址为[11:2]时，可以截取传入地址的第 11 位到第 2 位信号。取 11:2 位相当于 9:0 位人为除以 4。使得我们可以取得正确的答案。

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 0x00003000）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：如本设计中 `ctl2.v`

```
assign addu = !opcode[5] && !opcode[4] && !opcode[3] && !opcode[2] && !opcode[1] && !opcode[0] && func[5]
&& !func[4] && !func[3] && !func[2] && !func[1] && func[0];
RegDst[1] = jal;
RegDst[0] = addu || subu;
```

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。

例：addu

```
if(opcode == 6'b000000) begin  //R
       case(func)
       6'b100001:
       begin
           Reg_Dst = 2'b01;
           Reg_Write = 1;
           Mem_Write = 0;
           MemToReg = 0;
           Alu_Src = 0;
           Alu_Op = 4'b0000;
           Ext_Op = 4'b0000;
           NPc_Sel = 0;
           J = 0;
           jal = 0;
           jr = 0;
           tmp = 0;
       end
```

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。每条控制信号必须每次都更改，易漏过。

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，易于构架，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。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。

jr 相当于函数返回，此时从栈空间中恢复保存的变量，退栈转到返回地址（即上一次调用的 PC+4）。