单周期CPU

1. 设计目的与说明

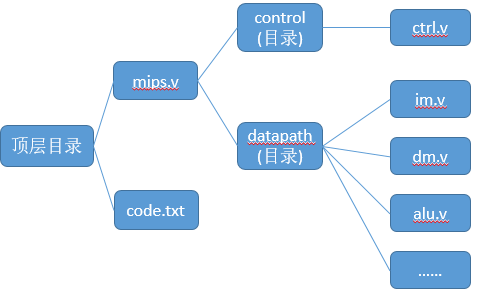
使用verilog搭建一个指令的单周期CPU

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

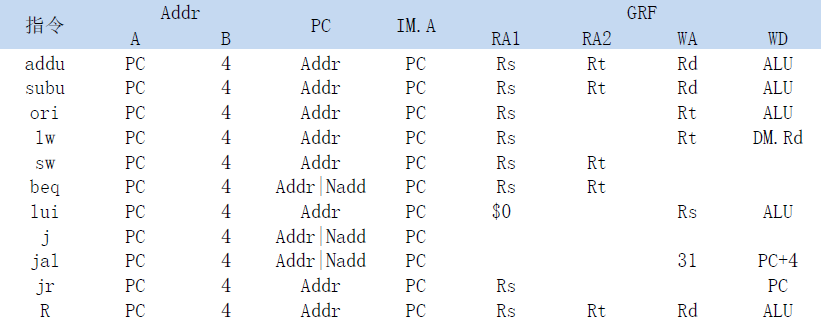
Addu,subu 不支持溢出；

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；



1. 单周期数据通路设计



assign rs = instr[25:21];

assign rt = instr[20:16];

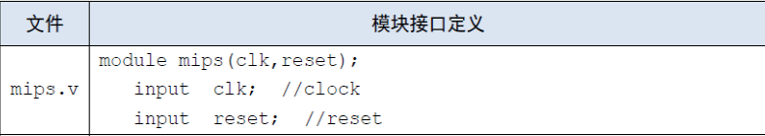
assign rd = instr[15:11];

assign offset = instr[15:0];

assign op = instr[31:26];

assign func = instr [5:0];

1. 通路设计

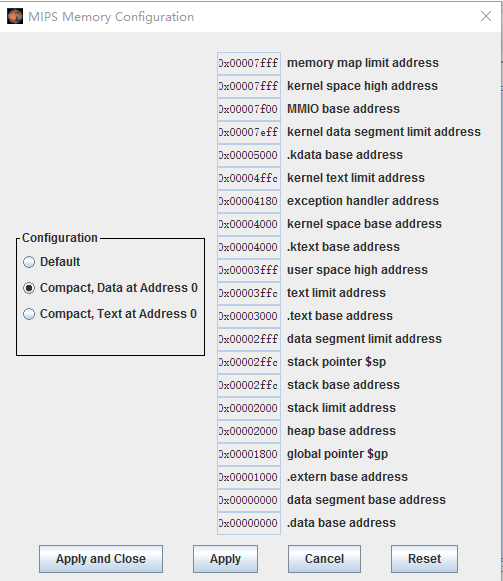


1. ifu

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。



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。

1. 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;

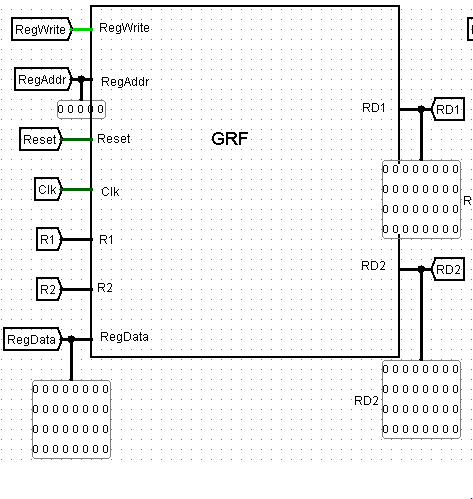
$display("$%d <= %h", RegAddr, jalpc);

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

if(reset)

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

DM[i] <= 32'b0;

else if (we)begin

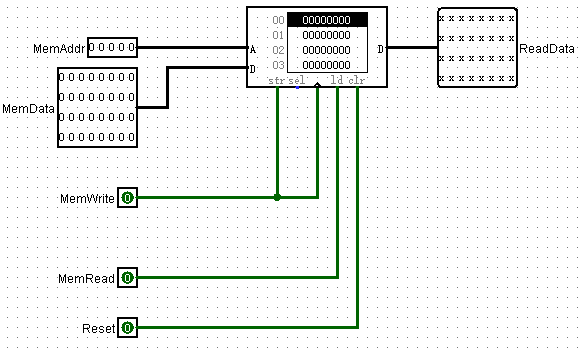
DM[addr[11:2]]<=din;

$display ("\*%h <= %h", addr, din);

end

end

endmodule



类比logisim中的设计。

在**dm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。

5、EXT

module ext(

input [15:0] Din,

input [1:0] EXTOP,

output reg [31:0] Dout

);

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;

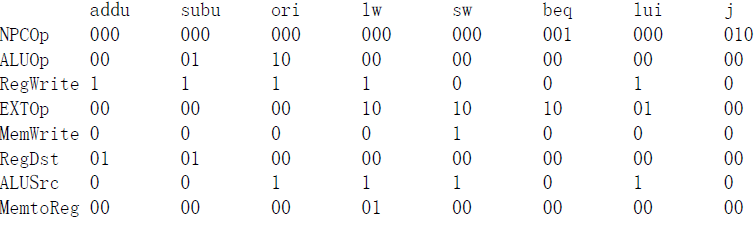
endcase

end

endmodule

四、Controller（控制器）





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

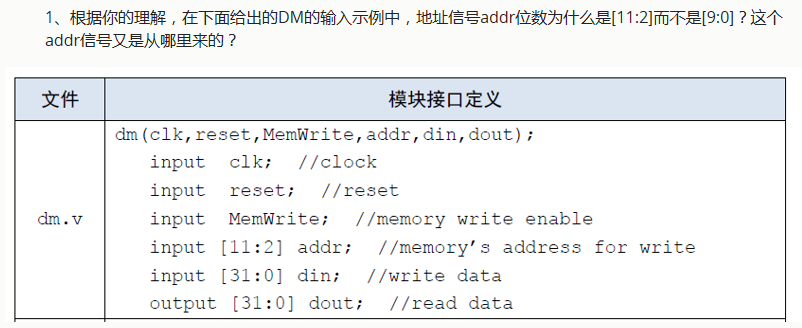
jr $31

label2:beq $9,$0,end

end:

以上为测试机器码翻译的结果。

六、思考题



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。



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。

1. 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。
2. 直接判断：

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

1. if else

if (~op[5]&&~op[4]&&~op[3]&&~op[2]&&op[1]&&op[0])

ori = 1;

1. case

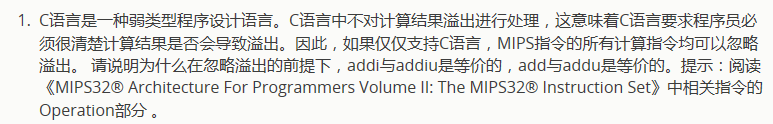
Case(op)

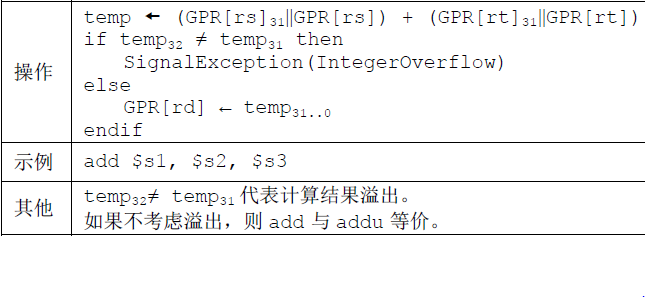
6’b000011:ori = 1;

1. 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；





当temp32！=temp31 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).



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。

1. 简要说明jal、jr和堆栈的关系。



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

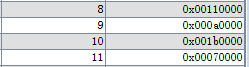
运行结果与mars一致

$ 8 <= 00110000

$ 9 <= 000a0000

$10 <= 001b0000

$11 <= 00070000



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

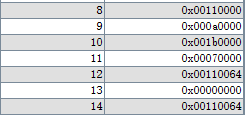
01095021

01095823

350c0064

adac0000

8dae0000



$ 8 <= 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

