Verilog开发单周期CPU

一、CPU设计方案综述

（一）总体设计概述

- 处理器应支持指令集为`{addu,subu,ori,lw,sw,beq,lui,jal,jr,nop}`
- `addu`、`subu`可以不支持溢出
- 处理器为单周期设计
- 不需要考虑延迟槽
- 支持同步复位

（二）关键模块定义

1、IFU

- 介绍：内部包括PC（程序计数器）、IM（指令存储器）及相关逻辑，PC用寄存器实现，具有同步复位功能，复位值为起始地址，IM容量为4KB
- 端口定义：

端口	方向	位宽	功能描述
clk	I	1	时钟控制信号
reset	I	1	同步复位信号
imm_1	I	32	`beq`指令的`sign_extend(offset ||0^2 )`32位立即数
imm_2	I	32	`jal`指令的拓展完成后的32位立即数
Reg_value	I	32	`rs`寄存器的值
Branch	I	2	跳转选择信号
zero	I	1	beq跳转信号，若为1，则跳转
PC	O	32	当前指令地址
PC4	O	32	下一条指令地址
Instr	O	32	当前指令

- 功能定义

序号	功能名称	功能描述
1	同步复位	在时钟上升沿到来时，如果`reset`有效，则进行复位，复位PC位0x0000_3000
2	取指令	PC从IFU取出相应的指令
3	BEQ跳转	当Branch为`2'b01`，PC=PC+4+imm_1
4	JAL跳转	当Branch为`2'b10`，PC=imm_2
5	JR跳转	当Branch为`2'b11`，PC = Reg_value

```
`define ADD4 0
`define BEQ  2'b01
`define JAL  2'b10
`define REG  2'b11
module IFU(
    input clk,
    input reset,
    input [31:0] imm_1,
    input [31:0] imm_2,
    input [31:0] Reg_value,
    input [1:0] Branch,
    input zero,

    output [31:0] PC,
    output [31:0] PC4,
    output [31:0] Instr
);
    reg [31:0] pc = 32'h0000_3000;
    reg [31:0] IM_REG[0:1023];
    wire [9:0] ADDR;


    initial begin
            $readmemh("code.txt",IM_REG);
    end

    assign ADDR = pc[11:2];
    assign Instr = IM_REG[ADDR];
    assign PC4 = PC + 4;
    assign PC = pc;

    always @(posedge clk)begin
        if(reset)
            pc <= 32'h0000_3000;
        else begin
            pc <= (Branch == `BEQ && zero == 1)  ? (pc + 4 + (imm_1 << 2)) :
                    (Branch == `JAL)  ? imm_2:
                    (Branch == `REG)  ? Reg_value:
                                        (pc + 4);
        end
    end
endmodule
```

2、GRF

- 介绍：用具有写使能的寄存器实现，寄存器总数为32个，应具有同步复位功能，其中0号寄存器的值始终为0。
- 端口定义：

端口	方向	位宽	功能描述
clk	I	1	时钟信号
reset	I	1	同步复位信号，将32个寄存器中的值全部清零 1:复位 0：无效
A1	I	5	5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD1
A2	I	5	5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD2
A3	I	5	5位地址输入信号，指定32个寄存器中的一个作为写入的目标寄存器
WE	I	1	写使能信号
Data	I	32	32位写入数据
PC	I	32	当前指令地址
RD1	O	32	A1所对应寄存器的值
RD2	O	32	A2所对应寄存器的值

- 功能定义：

序号	功能名称	功能定义
1	同步复位	在时钟上升沿到来时，如果reset有效时，则进行复位，将所有寄存器中存储的数值清零
2	读数据	读出A1、A2地址对应寄存器中所存储的数据到RD2
3	写数据	当WE有效且时钟上升沿来临时，将Data写入A3所对应的寄存器中

```
module GRF(
    //inputs
    input [4:0] A1,A2,A3,
    input clk,
    input reset,
    input WE,
    input [31:0] Data,PC,
    //outputs
    output [31:0] RD1, RD2

);
```

```
    reg [31:0] GPR [0: 31];
    integer i = 0;


    initial begin
        for(i = 0;i < 32; i = i + 1)
            GPR[i] <= 0;
    end

    assign RD1 = GPR[A1];
    assign RD2 = GPR[A2];


    always @(posedge clk)begin
        if(reset)begin
            for(i = 0;i < 32; i = i + 1)
                GPR[i] <= 0;
        end
        else begin
            if(WE)begin
                $display("@%h: $%d <= %h", PC, A3, Data);
                if(A3 != 0)
                    GPR[A3] <= Data;
            end
            else
                GPR[A3] <= GPR[A3];
        end
    end
endmodule
```

3、ALU

- 介绍：提供32位加、减、或运算以及大小比较功能，不检测溢出。
- 端口定义：

端口	方向	位宽	功能描述
SrcA	I	32	操作数1
SrcB	I	32	操作数2
ALUOP	I	3	ALU运算控制信号
Result	O	32	运算结果
Zero	O	1	判断结果是否为0

- 功能定义：

序号	功能名称	功能描述
1	或运算	当ALUOP = 3'b001，进行或运算
2	加法运算	当ALUOP = 3'b010，进行加法运算
3	减法运算	当ALUOP = 3'b011，进行减法运算
4	判断SrcA==SrcB	当ALUOP = 3'b011且Zero = 1时，代表两数相等

```
`define AND 0
`define OR  3'b001
`define ADD 3'b010
`define SUB 3'b011
module ALU(
        //inputs
        input [31:0] SrcA,SrcB,
        input [2:0] ALUOP,
        //outputs
        output [31:0] Result,
        output Zero
);
        parameter ZERO = 0;
        //calculate
        assign Result = (ALUOP == `OR)  ? SrcA | SrcB :
                        (ALUOP == `ADD) ? SrcA + SrcB :
                        (ALUOP == `SUB) ? SrcA - SrcB :
                                          0;
        assign Zero = (Result == ZERO);

endmodule
```

4、DM

- 介绍：容量为4KB，具有同步复位功能
- 端口定义：

端口	方向	位宽	功能描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
WE	I	1	写使能信号
Addr	I	32	写入地址
WD	I	32	写入的数据
PC	I	32	当前指令地址
RD	O	32	读出的数据

- 功能定义：

序号	功能名称	功能描述
1	复位	在时钟上升沿到来时，若reset有效，则进行复位
2	写入数据	当WE = 1时，将WD写入Addr地址中
3	读数据	RD始终为Addr地址的值

```
module DM(
    input clk,  //clock
    input reset,    //reset
    input WE,   //memory write enable
    input [31:0] Addr,  //memory's address for write
    input [31:0] WD,        //write data
    input [31:0] PC,

    output [31:0] RD        //read data
);



    reg [31:0] mem[0:1023];
    integer i;
    initial begin

        for(i = 0;i < 1024; i = i + 1)
            mem[i] = 0;
    end


    assign RD = mem[Addr[11:2]];

    always @ (posedge clk)begin
        if(reset)begin
            for(i = 0;i < 1024; i = i + 1)
                mem[i] <= 0;
        end
        else begin
            if(WE)begin
                $display("@%h: *%h <= %h", PC, Addr, WD);
                mem[Addr[11:2]] <=  WD;
            end
            else
                mem[Addr[11:2]] <= mem[Addr[11:2]];
        end
    end

endmodule
```

5、EXT

- 介绍：支持16bit到32bit的零拓展以及符号拓展
- 端口定义：

端口	方向	位宽	功能描述
imm_16	I	16	16位待拓展的立即数
ExtOp	I	1	拓展控制信号
Result	O	32	拓展后的结果

- 功能定义：

序号	功能名称	功能描述
1	零拓展	当ExtOp = 0时，进行零拓展
2	符号拓展	当ExtOp = 1时，进行符号拓展

```
module Ext(
    input [15:0] imm_16,
    input ExtOp,
    output [31:0] Result
);

    assign Result = (ExtOp == 1) ? {{16{imm_16[15]}}, imm_16} :
                                                {16'b0,imm_16} ;
endmodule
```

6、Controller

- 介绍：产生控制信号
- 端口定义：

端口	方向	位宽	功能描述
opcode	I	6	`Instr[31:26]`
func	I	6	`Instr[5:0]`
RegWrite	O	1	寄存器写入控制信号
RegDst	O	1	写入的寄存器选择信号
RaWrite	O	1	写入`$ra`寄存器控制信号
ExtOp	O	1	拓展控制信号
ALU_s	O	1	`SrcB`选择信号
MemWrite	O	1	内存写入信号
Branch	O	2	跳转选择信号
WD_sel	O	2	写入数据选择信号
ALUOP	O	3	运算控制信号

- 功能定义：

序号	功能名称	功能定义
1	产生控制信号	产生控制信号

```
`include "ALU.v"
`include "IFU.v"
module Controller(

    //inputs
    input [5:0] opcode, //opcode
    input [5:0] func,       //func
    //outputs
    output  RegWrite,
              RegDst,
              RaWrite,
              ExtOp,
              ALU_s,
              MemWrite,
    output [1:0] Branch,WD_sel,
    output [2:0] ALUOP

    );



    parameter ZERO = 0,
                 ADDU = 6'b100001,
                 SUBU = 6'b100011,
                 ORI  = 6'b001101,
                 LW   = 6'b100011,
                 SW   = 6'b101011,
                 BEQ  = 6'b000100,
                 LUI  = 6'b001111,
                 JAL  = 6'b000011,
                 JR   = 6'b001000;
    /*
    wire MemWrite;
    wire ExtOp;
    wire RegWrite,RegDst;
    wire RaWrite;
    wire ALU_s;
    wire [1:0] WD_sel;
    wire [1:0] Branch;
    wire [2:0] ALUOP;
    */
    // instruction
    wire zero,addu,subu,ori,lw,sw,beq,lui,jal,jr;

    assign zero = (opcode == ZERO);
    assign addu = zero & (func == ADDU);
    assign subu = zero & (func == SUBU);
    assign ori = (opcode == ORI);
    assign lw = (opcode == LW);
    assign sw = (opcode == SW);
    assign beq = (opcode == BEQ);
    assign lui = (opcode == LUI);
    assign jal = (opcode == JAL);
```

```
    assign jr = zero & (func == JR);



    assign RegDst = addu || subu;
    assign RegWrite = addu || subu || ori || lui || lw || jal;
    assign ALUOP = (addu || lw || sw) ? `ADD :
                   (subu || beq) ? `SUB :
                   (ori)  ? `OR  :
                            0 ;
    assign RaWrite = jal;
    assign ExtOp = lw || sw || beq;
    assign ALU_s = ori || lw || sw;
    assign WD_sel = (lw)                    ? 3'b1:
                    (jal)                   ? 3'b10:
                    (lui)                   ? 3'b11:
                                              0;
    assign Branch = (beq) ? `BEQ :
                    (jal) ? `JAL :
                    (jr)  ? `REG :
                            `ADD4;
    assign MemWrite = sw;

endmodule
```

（三）重要机制实现方法

```
output  RegWrite,
              RegDst,
              RaWrite,
              ExtOp,
              ALU_s,
              MemWrite,
    output [1:0] Branch,WD_sel,
    output [2:0] ALUOP
```

- 控制信号

	addu	subu	ori	beq	lw	sw	jal	jr	lui	nop
RegWrite	1	1	1		1		1		1	
RegDst	1	1								
RaWrite							1			
ExtOp				1	1	1				
ALU_s			1		1	1				
MemWrite						1				
Branch				2'b1			2'b10	2'b11		
WD_sel					3'b1		3'b10		3'b11	
ALUOP	3'b10	3'b11	3'b1	3'b11	3'b10	3'b10				

二、测试方案

(一) 典型测试样例

```
subu $28 $28 $28
subu $29 $29 $29
ori $0, $0, 0
ori $1, $1, 1
ori $2, $2, 2
ori $3, $3, 3
ori $4, $4, 4
ori $5, $5, 5
ori $6, $6, 6
ori $7, $7, 7
ori $8, $8, 8
ori $9, $9, 9
ori $10, $10, 10
ori $11, $11, 11
ori $12, $12, 12
ori $13, $13, 13
ori $14, $14, 14
ori $15, $15, 15
ori $16, $16, 16
ori $17, $17, 17
ori $18, $18, 18
ori $19, $19, 19
ori $20, $20, 20
ori $21, $21, 21
ori $22, $22, 22
ori $23, $23, 23
ori $24, $24, 24
ori $25, $25, 25
ori $26, $26, 26
ori $27, $27, 27
ori $28, $28, 28
ori $29, $29, 29
ori $30, $30, 30
ori $31, $31, 31
ori $1, $1, 1
sw $1, 4($0)
sw $1, 8($0)
sw $1, 12($0)
sw $1, 16($0)
sw $1, 20($0)
sw $1, 24($0)
sw $1, 28($0)
sw $1, 32($0)
sw $1, 36($0)
sw $1, 40($0)
sw $1, 44($0)
sw $1, 48($0)
sw $1, 52($0)
sw $1, 56($0)
sw $1, 60($0)
sw $1, 64($0)
sw $1, 68($0)
sw $1, 72($0)
sw $1, 76($0)
sw $1, 80($0)
```

```
sw $1, 84($0)
sw $1, 88($0)
sw $1, 92($0)
sw $1, 96($0)
sw $1, 100($0)
sw $1, 104($0)
sw $1, 108($0)
sw $1, 112($0)
sw $1, 116($0)
sw $1, 120($0)
sw $1, 124($0)
ori $1, $1, 1
sw $1, 0($0)
lw $2, 0($0)
lw $3, 0($0)
lw $4, 0($0)
lw $5, 0($0)
lw $6, 0($0)
lw $7, 0($0)
lw $8, 0($0)
lw $9, 0($0)
lw $10, 0($0)
lw $11, 0($0)
lw $12, 0($0)
lw $13, 0($0)
lw $14, 0($0)
lw $15, 0($0)
lw $16, 0($0)
lw $17, 0($0)
lw $18, 0($0)
lw $19, 0($0)
lw $20, 0($0)
lw $26, 0($0)
lw $27, 0($0)
lw $28, 0($0)
lw $29, 0($0)
lw $30, 0($0)
lw $31, 0($0)
ori $1 $1 907
sw $0 0($0)
lw $1 0($0)
sw $1 4($0)
lw $2 4($0)
sw $2 8($0)
lw $3 8($0)
sw $3 12($0)
lw $4 12($0)
sw $4 16($0)
lw $5 16($0)
sw $5 20($0)
lw $6 20($0)
sw $6 24($0)
lw $7 24($0)
sw $7 28($0)
lw $8 28($0)
sw $8 32($0)
lw $9 32($0)
sw $9 36($0)
```

```
sw $12 48($0)
lw $13 48($0)
sw $13 52($0)
lw $14 52($0)
sw $14 56($0)
lw $15 56($0)
sw $15 60($0)
lw $16 60($0)
sw $16 64($0)
lw $17 64($0)
sw $17 68($0)
lw $18 68($0)
sw $18 72($0)
lw $19 72($0)
sw $19 76($0)
lw $20 76($0)
sw $20 80($0)
lw $21 80($0)
sw $21 84($0)
lw $22 84($0)
sw $22 88($0)
lw $23 88($0)
sw $23 92($0)
lw $24 92($0)
sw $24 96($0)
lw $25 96($0)
sw $25 100($0)
lw $26 100($0)
sw $26 104($0)
lw $27 104($0)
sw $27 108($0)
lw $28 108($0)
sw $28 112($0)
lw $29 112($0)
sw $29 116($0)
lw $30 116($0)
sw $30 120($0)
lw $31 120($0)
sw $31 124($0)
lui $1 234
lui $2 234
lui $3 234
lui $4 234
lui $5 234
lui $10 234
lui $11 234
lui $12 234
lui $13 234
lui $14 234
lui $15 234
jal con
lui $1 222
subu $16 $16 $1
subu $17 $17 $1
subu $18 $18 $1
subu $19 $19 $1
subu $20 $20 $1
subu $24 $24 $1
```

```
subu $25 $25 $1
subu $26 $26 $1
subu $27 $27 $1
subu $28 $28 $1
subu $29 $29 $1
subu $30 $30 $1
jal end
con:
addu $16 $16 $1
addu $17 $17 $2
addu $18 $18 $3
addu $22 $22 $7
addu $23 $23 $8
addu $24 $24 $9
addu $25 $25 $10
addu $26 $26 $11
addu $27 $27 $12
addu $28 $28 $13
addu $29 $29 $14
addu $30 $30 $15
jr $31
end:
ori $1 $0 1
ori $2 $0 2
beq $1 $2 beq1
addu $1 $1 $1
beq1:
ori $12 $0 1
ori $13 $0 1
beq $12 $13 beq2
addu $2 $2 $2
beq2:
jal con2
jal end2
addu $6 $6 $6
jal end2
con2:
addu $15 $0 $31
ori $5 $0 4
addu $31 $31 $5
jr $15
addu $1 $1 $1
end2:
```

（二）自动生成`mips`代码

```
# python3.9
import os
import random
R_type = ['addu', 'subu']
I_type = ['ori', 'lui', 'lw', 'sw', 'beq']
J_type = ['jal']

R_num = 2
I_num = 5
J_num = 1
filename = "E:\CS_Project\P4\\test.asm"
```

```
label = []
cnt = 0
jal = []


class GenCode:
    def __init__(self):
        self.rs = random.randint(0, 31)
        self.rt = random.randint(0, 31)
        self.rd = random.randint(0, 31)
        self.imm16 = random.randint(0, (1 << 16)-1)
        self.mem = random.randint(0, 3071)
        self.imm26 = random.randint(0, (1 << 26)-1)
        self.list = []
        self.gen_R()
        self.gen_I()
        self.gen_J()
        self.main()

    def gen_R(self):
        random1 = random.randint(0, R_num - 1)
        type1 = R_type[random1]
        self.list.append(type1)

    def gen_I(self):
        random2 = random.randint(0, I_num - 1)
        type2 = I_type[random2]
        self.list.append(type2)

    def gen_J(self):
        random3 = random.randint(0, J_num - 1)
        type3 = J_type[random3]
        self.list.append(type3)

    def main(self):
        # sel = random.randint(0, 1)
        sel = 1
        if sel == 0:
            self.code = self.list[sel] + ' ' + '$' + \
                str(self.rd) + ' ' + ',' + '$' + \
                str(self.rs) + ' ' + ',' + '$' + str(self.rt) + '\n'
        elif sel == 1:
            if self.list[sel] == 'lw' or self.list[sel] == 'sw':
                self.code = self.list[sel] + ' ' + '$' + \
                    str(self.rt) + ',' + str(self.imm16 << 2) + \
                    '(' + '$' + '0'+')'+'\n'
            elif self.list[sel] == 'lui':
                self.code = self.list[sel] + ' ' + \
                    '$' + str(self.rt) + ',' + str(self.imm16) + '\n'
            elif self.list[sel] == 'beq':
                self.code = self.list[sel] + ' ' + '$' + \
                    str(self.rt) + ',' + '$' + \
                    str(self.rs) + ',' + 'label_' + str(cnt) + '\n'
                label.append(cnt)

            else:
                self.code = self.list[sel] + ' ' + '$' + \
                    str(self.rt) + ',' + '$' + \
```

```
                    str(self.rs) + ',' + str(self.imm16) + '\n'
        elif sel == 2:
            if self.list[sel] == 'jal':
                self.code = self.list[sel] + ' ' + 'label_' + str(cnt) + '\n'
                jal.append(cnt)
            elif self.list[sel] == 'j':
                self.code = self.list[sel] + ' ' + 'label_' + str(cnt) + '\n'


if __name__ == '__main__':
    with open(filename, 'w+') as f:
        for cnt in range(0, 200):
            a = GenCode()
            f.write(a.code)
            if random.randint(0, 5) == 1:
                f.write('nop' + '\n')
            if random.randint(0, 1) == 1 and label != []:
                ran = random.randint(0, len(label) - 1)
                f.write('label_' + str(label[ran]) + ':' + '\n')
                label.pop(ran)
            if random.randint(0, 1) == 1 and jal != []:
                ran = random.randint(0, len(jal) - 1)
                f.write('jr $ra' + '\n')
                label.pop(ran)
        while label != []:
            ran = random.randint(0, len(label) - 1)
            f.write('label_' + str(label[ran]) + ':' + '\n')
            label.pop(ran)
        while jal != []:
            ran = random.randint(0, len(jal) - 1)
            f.write('jr $ra' + '\n')
            label.pop(ran)
        f.close()
```

三、思考题

- 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？
    - DM实现过程中存储方式位`32bit * 1024`，按字存储，而Addr是以字节为单位，所以应由[9:0]改为[11:2]。
    - addr信号来自于ALU的结果输出
- 思考Verilog语言设计控制器的译码方式，给出代码示例，并尝试对比各方式的优劣。
    - 指令对应控制信号如何取值：

```
always @ * begin
    if(ori)begin
        RegWrite = 1;
        RegDst = 0;
        RaWrite = 0;
        ExtOp = 0;
        ALU_s = 1;
        MemWrite = 0;
        ALUOP = 3'b1;
        Branch = 0;
        WD_sel = 0;
    end
end
```

  - 控制信号对应每种指令如何取值：

```
assign RegWrite = addu || subu || ori || lw || lui || jal;
```

  - 第一种方式：所有的控制信号需要使用reg型且每增加一个指令就要把所有信号写一遍，每增加一个信号就需要在所有指令里写一遍；对于第二种方式，添加一个指令或信号可以省略大部分指令相同的值。因此第二种方式较优。
- 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所驱动的部件具有什么共同特点?
  - 它们都包含时序逻辑，都存在被复位的需求
- 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 。
  - addi相比于addiu多了一个如果溢出就抛出SignalException异常，如果忽略这个异常，两者自然等价；add和addu同理。
- 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，设计精简，易于维护和拓展
  - 缺点：同一时间只能执行一条指令，耗时长，时间效率低，吞吐量小