单周期CPU实验报告

1.实验目标

实现MIPS-Lite2指令集，共包含九条指令{addu,subu,ori,sw,lw,beq,lui,jal,jr}处理器为单周期设计，每一条指令都在一个周期之内执行完毕.

2.模块功能设计

2.1 PC

PC的作用是在每一次时钟周期CLK的下降沿到来时输出下一条指令的计数值，这个计数值有两种可能，如果Reset有效，则初始化为，第一条指令的地址，也就是00003000，否则就直接设置为由NPC计算好的下一条指令的地址（索引计数值）。

```
module PC(
    input [31:0] Result,
    input Reset,
    input CLK,
    output reg [31:0] Next
    );
    initial
        Next=8'h00003000;
    always @(posedge CLK or negedge Reset)
    begin
        if (!Reset)
        begin
            Next=8'h00003000;
            $display("reset: %h",Next);
        end
        else
        begin
            Next=Result;
        end
    end
endmodule
```

2.2 IM

由于我们采用的是哈佛结构，所以指令存储器和数据存储器是分开的，对于指令存储器，所需要做的只是跟据PC送入的指令地址，在“内存”中找到对应的指令内容。所谓“内存”，就是指我们在这个模块内部设置的一个寄存器，共有1024个寄存器，每一个寄存器32位。

在初始化阶段，这个寄存器需要读入指令的16进制格式。要得到指令的16进制格式，只需要将Mars中写好的MIPS汇编代码以16进制的格式导出到code.txt之中，然后用verilog之中特殊的指令$readmemh从code.txt之中读取即可。

```
module im_4k(addr,dout);
input [11:2] addr;
output[31:0] dout;
reg [31:0] im[1023:0];
initial begin
    $readmemh("D:\\asmcode\\test2.txt",im);
end
assign dout=im[addr];
endmodule
```

由于我们的im寄存器总共就最多只能存储1024条指令，而且在MIPS的架构下每一条指令都是四个字节，所以这个地址的最后两位只能是0，并且这个地址的12到31位必定是0，所以我们读取指令的时候只需要第二位到第十一位，其余的位没有必要。直接将im的第[addr]位通过assign语句连接到输出即可。

2.3 GPR

对于寄存器堆的模拟模块，主要需要考虑几个接口，读写的寄存器编号、读写的数据、以及写信号等，实现代码如下：

```
module
GPR(ReadReg1,ReadReg2,WriteReg,WriteData,RegWrite,CLK,ReadData1,ReadData2);
input [4:0] ReadReg1;
input [4:0] ReadReg2;
input [4:0] WriteReg;
input [31:0] WriteData;
output [31:0] ReadData1;
output [31:0] ReadData2;
input RegWrite;//flags
input CLK;

reg [31:0] RegFile[31:0];
integer i;
initial begin
    for (i=0;i<32;i=i+1)
        RegFile[i]=0;
end

assign ReadData1=RegFile[ReadReg1];
assign ReadData2=RegFile[ReadReg2];

always @ (negedge CLK)
begin
    if(RegWrite)
    begin
        RegFile[WriteReg]=WriteData;
        //$display("%h,%h",WriteReg,WriteData);
    end

end
endmodule
```

使用RegFile表示31个4字节的寄存器文件，在初始化的时候全部置零，直接使用assign语句将寄存器读出来的两个值连到寄存器对应下标的位置。而写数据不能使用assign语句，因为写入寄存器文件需要依靠写入信号RegWrite，只有信号有效的时候才能写入寄存器，所以需要依靠always语句检验下降沿和RegWrite条件。

2.4 dm

dm是用来模拟内存的文件，实现代码如下所示：

```
module dm_4k(addr,din,we,clk,dout);
input [11:2] addr;
input [31:0] din;
input we;
input clk;
output [31:0] dout;
reg [31:0] dm[1023:0];
integer i;
assign dout=dm[addr];

always@(posedge clk)begin
    if(we)
        dm[addr]<=din;
end;
endmodule
```

按照我的设计，从内存之中读取数据随时可以进行，不需要特殊地信号，而如果要向dm之中写入数据，就需要在clk时钟的下降沿等待MemWrite信号，如果该信号有效，才可以正常读写。

2.5 PCAdd4

对当前的PC结果加4，作为一种可能的NPC的结果如果没有branch或者J型指令，一般情况下会用PCAdd4作为下一个PC的地址，代码如下所示：

```
module PCAdd4(OldPC,Add4,Add8);
input [31:0] OldPC;
output [31:0] Add4;
output [31:0] Add8;
assign Add4=OldPC+4;
assign Add8=OldPC+8;
endmodule
```

2.6 PCBranch

PCBranch 接受ALUZero位、PCAdd4、ShiftLeft2作为输入，并依据ALUZero位确定在当前指令为branch时，是否满足跳转条件，并依据此来确定下一条指令地址。

```
module PCBranch(LeftShift2,ALUZero,Add4,Result);
input [31:0] LeftShift2;
input [31:0] Add4;
input ALUZero;
output [31:0] Result;
assign Result=ALUZero? LeftShift2+Add4:Add4;
endmodule
```

2.7 PCJump

PCJump接受PCAdd4，imm26作为输入，在当前指令为J型指令时，可以将计算出来跳转之后的地址，作为NPC的其中一个输入，代码如下所示：

```
module PCJump(Addr26,Add4,Result);
input [25:0] Addr26;
input [31:0] Add4;
output[31:0] Result;
wire [25:0] ShiftAddr26;
wire  [27:0]temp;
assign temp={Addr26,2'b00};
assign Result={Add4[31:28],temp};
endmodule
```

2.8 Extend

进行位拓展，将16位的立即数拓展为32位，有两种拓展方式，一种是零拓展，也就是前16位全部为0，另外一种是符号拓展，前16位是符号位，也就是原来的第15位。Extend使用非常频繁，所有的I型指令都需要使用这个模块将imm16进行拓展再送入到ALU之中运算。位拼接在verilog之中使用比较方便，具体代码如下所示：

```
module Extend(imm16,sign,signimm32,unsignimm32);
    input [15:0] imm16;
    input sign;
    output[31:0] signimm32;
    output[31:0] unsignimm32;
    wire [15:0] zero16;
    wire [31:0] temp1;
    wire [31:0] temp2;
    parameter z = 16'b0;
    wire [15:0] e = {16{imm16[15]}};
    assign  unsignimm32 = {z, imm16};
    assign  signimm32 = {e, imm16};
endmodule
```

2.9 ShiftLeft2

本模块比较简单，只需要将一个32位数据右移两位即可，一般是接在Extend模块之后

```
module LeftShift2(old,new);
input [31:0] old;
output [31:0] new;
assign new=old<<2;
endmodule
```

2.10 control

control unit的作用是解析im取出的指令，分割成op，rs，rt和rd（对于R型指令而言）或者op、rt、rs、imm16（对于I型指令而言）或者op imm26（对于J型指令而言）。于此同时，还要跟据这条指令，生成所有可能使用到的控制信号。为了正确表示信号与指令之间的关系，使用xlsx表示的表格如下所示：

指令	op	func	PcSrc	RegWriteSrc	RegWrite	MemWrite	RegDst	ALUSrc	ALUOperation
addu	000000	100001	00	00	1	0	01	1	00
subu	000000	100011	00	00	1	0	01	1	01
ori	001101		00	00	1	0	00	0	11
lw	100011		00	01	1	0	00	0	00
sw	101011		00		0	1		0	00
beq	000100		01		0	0		1	01
lui	001111		00	00	1	0	00	0	10
jal	000011		10	10	1	0	10		
jr	001000		11		0	0			

如图所示，

- PCSrc代表PC的result来源是NPC的哪一种，PC可能有四种来源，分别是00：PC+4；01：PCBranch；10：PCJump；11：来自寄存器GPR的ReadData1
- RegWriteSrc代表写入GPR的WriteData数据的来源，总共有三种可能的来源，分别是00：ALUResult；01：DMReadData；10：PC+4；
- RegWrite代表是否需要写入到寄存器，只有addu，subu，ori，lw，sw，lui，jar几条指令需要用到写入，其余的只需要读入即可；
- MemWrite代表是否需要将数据写入到DM之中，只有sw一条指令需要写入内存
- RegDst代表写入寄存器的来源，其中00：rt；01：rt；10：第三十一号寄存器ra
- ALUSrc：ALU第二个操作数的来源，其中0代表16位立即数imm16，1代表从GPR读到的第二个寄存器；
- ALUOperation：代表ALU需要做的操作类型，其中00是加法，01是减法，11是按位或操作，10是左移16位，专门用于load upper指令。

综上所述，control的代码如下所示：

```
module
Control(inst,rs,rt,rd,imm16,imm26,PCSrc,RegWriteSrc,RegWrite,MemWrite,ALUOperati
on,RegDst,ALUSrc);
input [31:0] inst;
output [4:0] rs;
output [4:0] rt;
output [4:0] rd;
output [15:0] imm16;
output [25:0] imm26;
output [1:0]PCSrc;
output [1:0]RegWriteSrc;
output RegWrite;
output MemWrite;
output [1:0] ALUOperation;
output [1:0] RegDst;
output ALUSrc;

wire [5:0] op;
assign op=inst[31:26];
wire [5:0] func;
assign func=inst[5:0];
assign rd=inst[15:11];
assign rt=inst[20:16];
assign rs=inst[25:21];
assign imm16=inst[15:0];
assign imm26=inst[25:00];

wire I_addu;
wire I_subu;
wire I_ori;
```

```
wire I_lw;
wire I_sw;
wire I_beq;
wire I_lui;
wire I_jal;
wire I_jr;
wire R_type;

assign R_type=!op;
assign I_addu=R_type & func[5] & ~func[4] & ~func[3] & ~func[2] & ~func[1] &
func[0];
assign I_subu=R_type & func[5] & ~func[4] & ~func[3] & ~func[2] & func[1] &
func[0];
assign I_ori=~op[5] & ~op[4] & op[3] & op[2] & ~op[1] & op[0];
assign I_lw=op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0];
assign I_sw=op[5] & ~op[4] & op[3] & ~op[2] & op[1] & op[0];
assign I_beq=~op[5] & ~op[4] & ~op[3] & op[2] & ~op[1] & ~op[0];
assign I_lui=~op[5] & ~op[4] & op[3] & op[2] & op[1] & op[0];
assign I_jal=~op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0];
assign I_jr=R_type & ~func[5] & ~func[4] & func[3] & ~func[2] & ~func[1] &
~func[0];

assign PCSrc[0]=I_beq | I_jr;
assign PCSrc[1]=I_jal | I_jr;
assign RegWriteSrc[0]=I_lw;
assign RegWriteSrc[1]=I_jal;
assign RegWrite=I_addu | I_subu | I_ori | I_lw | I_lui | I_jal;
assign MemWrite=I_sw;
assign RegDst[0]=I_addu | I_subu;
assign RegDst[1]=I_jal;
assign ALUSrc=I_addu | I_subu | I_beq;
assign ALUOperation[0]=I_subu | I_ori | I_beq;
assign ALUOperation[1]=I_ori | I_lui;
endmodule
```

2.11 MUX

对于所有输入有多个来源的情况，都需要使用多路选择器MUX来判断究竟接受哪一个来源的输入，所以必须要使用MUX来作为选择的依据，而mux跟据选择的路数和输入的位数要分成好几种：

- 5位三选一：用于判断和选择GPR的WriteReg，要写入到哪一个寄存器；
- 32位三选一：用于判断GPR的WriteData，寄存器要写的值；
- 32位四选一：用于判断NPC的来源

mux的实现有许多种选择，比如说直接使用三路选择，使用always和case语句或者使用function，实现代码如下所示：

```
module MUX3x32(x0,x1,x2,sig,y);
input [31:0] x0;
input [31:0] x1;
input [31:0] x2;
input [1:0] sig;
output [31:0] y;

function  [31:0] select;
    input [31:0] x0,x1,x2;
    input [1:0]sig;
```

```
    case(sig)
        2'b00: select=x0;
        2'b01: select=x1;
        2'b10: select=x2;
    endcase
endfunction

assign y=select(x0,x1,x2,sig);
endmodule

module MUX3x5(x0,x1,x2,sig,y);
input [4:0] x0,x1,x2;
input [1:0] sig;
output [4:0] y;

function  [4:0] select;
    input [4:0] x0,x1,x2;
    input [1:0] sig;
    case(sig)
        2'b00:select=x0;
        2'b01:select=x1;
        2'b10:select=x2;
    endcase
endfunction

assign y=select(x0,x1,x2,sig);
endmodule

module MUX2x32(x0,x1,sig,y);
input [31:0]x0,x1;
input sig;
output [31:0] y;
assign y=sig?x1:x0;
endmodule

module MUX4x32(x0,x1,x2,x3,sig,y);
input [31:0] x0,x1,x2,x3;
input [1:0]sig;
output reg[31:0] y;
always @ (x0 or x1 or x3 or x3 or sig)
begin
    case(sig)
    2'b00:y=x0;
    2'b01:y=x1;
    2'b10:y=x2;
    2'b11:y=x3;
    endcase
    //$display("select %h",y);
end
endmodule
```

2.12 顶层实体

顶层的mips实体只需要将这若干个部件使用若干个wire接线连接起来即可，必要的时候可以加入一些调试输出打印信息：

```
module mips(CLK,Reset);
```

```
    input CLK;
    input Reset;

    wire [31:0] PC,PCAdd4,PCAdd8,NextPC,PCBranch,PCJump;
    wire [31:0] Inst;
    wire [4:0] rs;
    wire [4:0] rt;
    wire [4:0] rd;
    wire [15:0] imm16;
    wire [25:0] imm26;
    wire [31:0] SignExtend;
    wire [31:0] UnsignExtend;
    wire [31:0] LeftShift2;

    wire [1:0]PCSrc;
    wire [1:0]RegWriteSrc;
    wire RegWrite;
    wire MemWrite;
    wire [1:0] ALUOperation;
    wire [1:0] RegDst;
    wire ALUSrc;

    wire [4:0] GPRWriteReg;
    wire [31:0] GPRWriteData;
    wire [31:0] GPRReadData1;
    wire [31:0] GPRReadData2;
    wire [31:0] ALUOperand2;
    wire [31:0] ALUResult;
    wire ALUZero;
    wire [31:0] DMWriteData;
    wire [31:0] DMReadData;

    PC pc(NextPC,Reset,CLK,PC);
    PCAdd4 pcadd4(PC,PCAdd4,PCAdd8);
    im_4k im(PC[11:2],Inst);
    Control
control(Inst,rs,rt,rd,imm16,imm26,PCSrc,RegWriteSrc,RegWrite,MemWrite,ALUOperati
on,RegDst,ALUSrc);
    PCJump pcjump(imm26,PCAdd4,PCJump);
    Extend extend(imm16,1,SignExtend,UnsignExtend);
    LeftShift2 leftshift2(SignExtend,LeftShift2);

    MUX3x5 mux3x5(rt,rd,5'b11111,RegDst,GPRWriteReg);
    GPR
gpr(rs,rt,GPRWriteReg,GPRWriteData,RegWrite,CLK,GPRReadData1,GPRReadData2);
    MUX2x32 mux2x32(SignExtend,GPRReadData2,ALUSrc,ALUOperand2);
    ALU alu(GPRReadData1,ALUOperand2,ALUOperation,ALUResult,ALUZero);
    dm_4k dm(ALUResult[11:2],GPRReadData2,MemWrite,CLK,DMReadData);

    MUX3x32 mux3x32(ALUResult,DMReadData,PCAdd4,RegWriteSrc,GPRWriteData);


    PCBranch pcbranch(LeftShift2,ALUZero,PCAdd4,PCBranch);

    MUX4x32 mux4x32(PCAdd4,PCBranch,PCJump,GPRReadData1,PCSrc,NextPC);

    always @(negedge CLK)
    begin
```

```
        $display("new instruction:%h %h",PC,Inst);
        $display("rs:%h,rt:%h,rd:%h,imm16:%h, imm26:%h",rs,rt,rd,imm16,imm26);
        $display("gpr readdata1: %h gpr readdata2:
%h",GPRReadData1,GPRReadData2);
        $display("signext: %h shiftleft2: %h",SignExtend,LeftShift2);
        $display("alu: %h op %h = %h",GPRReadData1,ALUOperand2,ALUResult);
        $display("RegWrite: %h, WriteReg: %h, WriteData:
%h",RegWrite,GPRWriteReg,GPRWriteData);
        $display("MemWrite: %h,DM address: %h, DM read: %h, DM write:
%h",MemWrite,ALUResult,DMReadData,GPRReadData2,);
        $display("PCSrc: %h ,PCBranch: %h, PCJump: %h
NextPC:%h",PCSrc,PCBranch,PCJump,NextPC);
    end
endmodule
```

3 测试效果

3.1 样例一

首先使用我自己写的第一个测试代码：

```
.globl main
main:
    ori $t1,$zero,4
    ori $t2,$zero,2
    addu $t3,$t1,$t2
    subu $t4,$t1,$t2
    beq $t3,$t4,main
    lui $t5, 1
    jal func1
    beq  $t2,$t4,main
    ori $t2, $zero,1
    ori $t1, $zero,2
func1:
    addu $t1,$t3,$t4
    subu $t2,$t3,$t4
    jr $ra
```

这个指令之中使用了ori、addu、subu、ori、lui等多条指令，我使用display 打印输出了每一条指令的运行情况，部分结果截图如下所示：

```
new instruction:00000000 34090004
rs:00,rt:09,rd:00,imm16:0004, imm26:0090004
gpr readdata1: 00000000 gpr readdata2: 00000000
signext: 00000004 shiftleft2: 00000010
alu: 00000000 op 00000004 = 00000004
RegWrite: 1, WriteReg: 09, WriteData: 00000004
MemWrite: 0,DM address: 00000004, DM read: 00000000, DM write: 00000000
PCSrc: 0 ,PCBranch: 00000004, PCJump: 00240010 NextPC:00000004
new instruction:00000004 340a0002
rs:00,rt:0a,rd:00,imm16:0002, imm26:00a0002
gpr readdata1: 00000000 gpr readdata2: 00000000
signext: 00000002 shiftleft2: 00000008
alu: 00000000 op 00000002 = 00000002
RegWrite: 1, WriteReg: 0a, WriteData: 00000002
MemWrite: 0,DM address: 00000002, DM read: 00000000, DM write: 00000000
PCSrc: 0 ,PCBranch: 00000008, PCJump: 00280008 NextPC:00000008
new instruction:00000008 012a5821
rs:09,rt:0a,rd:0b,imm16:5821, imm26:12a5821
gpr readdata1: 00000004 gpr readdata2: 00000002
```

对于每一条指令，都列举出了操作使用的rs，rt，rd（如果是R型指令的话），以及GPR的寄存器数据存取、ALU的运算数和运算结果，DM的存取操作，下一条指令的地址，从中可以看到，前几条指令的运行都是正确的。

3.2样例二

接下来是第二个测试样例如下所示：

```
    .data
num:    .word   0,100,1,0,0
    .text
    lw  $t0,    num         # i = 0
    lw  $t1,    num+4           # n = 100
    lw  $t2,    num+8           # step = 1
    lw  $t3,    num+12          # sum = 0
    lw  $t4,    num+16          # subsum = 0
loop:
    beq $t0,    $t1,    exit        # if i == n break
    jal func                # call func
    addu    $t0,    $t0,    $t2     # i = i + 1
    beq $t1,    $t1,    loop        # goto loop
func:
    addu    $t3,    $t3,    $t0     # sum = sum + i
    subu    $t4,    $t4,    $t0     # subsum = subsum - i
    jr  $ra             # return
exit:
    sw  $t3,    num+20
    sw  $t4,    num+24
    ori $t5,    $t3,    100     # tmp = sum | 100
    sw  $t5,    num+28
```

再生成汇编代码之后，得到的前几条指令如下所示：

Bkpt	Address	Code	Basic
	0x00003000	0x8c080000	lw $8, 0x00000000($0)
	0x00003004	0x3c010000	lui $1, 0x00000000
	0x00003008	0x8c290004	lw $9, 0x00000004($1)
	0x0000300c	0x3c010000	lui $1, 0x00000000
	0x00003010	0x8c2a0008	lw $10, 0x00000008($1)
	0x00003014	0x3c010000	lui $1, 0x00000000
	0x00003018	0x8c2b000c	lw $11, 0x0000000c($1)
	0x0000301c	0x3c010000	lui $1, 0x00000000
	0x00003020	0x8c2c0010	lw $12, 0x00000010($1)
	0x00003024	0x11090006	beq $8, $9, 0x00000006
	0x00003028	0x0c000c0d	jal 0x00003034
	0x0000302c	0x010a4021	addu $8, $8, $10
	0x00003030	0x1129fffc	beq $9, $9, 0xfffffffc
	0x00003034	0x01685821	addu $11, $11, $8
	0x00003038	0x01886023	subu $12, $12, $8
	0x0000303c	0x03e00008	jr $31
	0x00003040	0x3c010000	lui $1, 0x00000000
	0x00003044	0xac2b0014	sw $11, 0x00000014($1)
	0x00003048	0x3c010000	lui $1, 0x00000000
	0x0000304c	0xac2c0018	sw $12, 0x00000018($1)

得到的输出信息如下图所示：

```
new instruction:00000000 34090004
rs:00,rt:09,rd:00,imm16:0004, imm26:0090004
gpr readdata1: 00000000 gpr readdata2: 00000000
signext: 00000004 shiftleft2: 00000010
alu: 00000000 op 00000004 = 00000004
RegWrite: 1, WriteReg: 09, WriteData: 00000004
MemWrite: 0,DM address: 00000004, DM read: 00000064, DM write: 00000000
PCSrc: 0 ,PCBranch: 00000004, PCJump: 00240010 NextPC:00000004
new instruction:00000004 340a0002
rs:00,rt:0a,rd:00,imm16:0002, imm26:00a0002
gpr readdata1: 00000000 gpr readdata2: 00000000
signext: 00000002 shiftleft2: 00000008
alu: 00000000 op 00000002 = 00000002
RegWrite: 1, WriteReg: 0a, WriteData: 00000002
MemWrite: 0,DM address: 00000002, DM read: 00000000, DM write: 00000000
PCSrc: 0 ,PCBranch: 00000008, PCJump: 00280008 NextPC:00000008
new instruction:00000008 012a5821
rs:09,rt:0a,rd:0b,imm16:5821, imm26:12a5821
gpr readdata1: 00000004 gpr readdata2: 00000002
signext: 00005821 shiftleft2: 00016084
alu: 00000004 op 00000002 = 00000006
RegWrite: 1, WriteReg: 0b, WriteData: 00000006
MemWrite: 0,DM address: 00000006, DM read: 00000064, DM write: 00000002
```

如图所示，前几条指令的执行都是正确的，同Mars的模拟效果完全吻合，因此可以证明单周期9条指令CPU实现的正确性。

4 反思和总结

- 解决bug之一：在顶层设计连接接口的时候有一个地方使用了一个没有定义的wire，结果vivado神奇地没有报错，导致ALU的第二个运算数一直是高阻态……

- 解决bug之二：四选一的多路选择器一开始忘了声明信号的位数，结果只能选第一个选择和第二个选择……