MIPS_V2.0——多周期32位CPU

郑逸宁 15307130115

一、项目概要

本次实验在上一次实验的基础，将CPU的单周期计算模式改进为多周期计算模式。多周期的原理非常简单，就是把每条指令拆分成若干个阶段，在每个时钟周期里执行一个阶段。因为不同指令的不同特性，有些指令可以通过较少的时钟周期来完成，这样当我们减少每个时钟周期的时间之后，整个CPU的执行效率就会加快。

整个多周期CPU的由一个状态机控制，每个状态表示了当前时钟周期在处理某种指令的某一阶段。每条指令的状态划分为IF，ID，EX，MEM和WB中的若干个，用"阶段_指令类型"形式命名。由于我设计的多周期MIPS比基础要求多出了很多功能，所以在实验报告中只有文字描述的整体结构，不再使用结构图（因为真的太难画了）

二、项目亮点

1. 使用了System Verilog这种较先进的硬件设计语言
2. 多实现了beq，bne，lb，lbu，sb等指令，为设计64位指令积累经验（详见 **<四、控制单元设计>**）
3. 所有组件使用参数化构建，便于下一步升级成64位CPU（详见 **<五、数据通路设计>**）
4. 统一了数据存储器和指令存储器，实现了数据和指令的混合存储（详见 **<六、存储器设计>**）
5. 使用$display(),$stop等命令，与波形图相结合进行仿真（详见 **<七、测试>**）
6. 在Nexys4实验板上实现了：显示当前周期、显示当前阶段、查看任意内存、任意寄存器和随时暂停程序等全面的演示功能（详见 **<八、演示设计>**）

三、项目文件

根目录（/）

```
/bit/                   存放各种版本的二进制文件(.bit)
/test/                  存放各种版本的汇编文件(.s)、十六进制文件(.dat)
/images/                存放实验报告所需的图片(.png)
/source/                源代码(.sv)
.gitignore              git配置文件
memfile.dat             当前使用的十六进制文件
states.txt              Nexys4实验板演示说明
README.md               说明文档
Nexys4DDR_Master.xdc    Nexys4实验板引脚锁定文件
simulation_behav.wcfg   仿真波形图配置文件
```

源代码（/source/）

```
alu.sv                  ALU计算单元
aludec.sv               ALU控制单元，用于输出alucontrol信号
clkdiv.sv               时钟分频模块模块，用于演示
controller.sv           mips的控制单元，包含maindec和aludec两部分
datapath.sv             数据通路，mips的核心结构
flopenr.sv              时钟控制的可复位触发寄存器
flopr.sv                可复位触发寄存器
maindec.sv              主控单元
mem.sv                  指令和数据的混合存储器
mips.sv                 mips处理器的顶层模块
mux2.sv                 2:1复用器
mux3.sv                 3:1复用器
mux4.sv                 4:1复用器
mux5.sv                 5:1复用器
onboard.sv              在Nexys4实验板上测试的顶层模块
regfile.sv              寄存器文件
signext.sv              符号拓展模块
simulation.sv           仿真时使用的顶层模块
sl2.sv                  左移2位
top.sv                  包含mips和内存的顶层模块
zeroext.sv              零拓展模块
```

四、状态机设计

多周期MIPS的核心是利用状态机表示当前CPU所处理的指令及这条指令正在处理的阶段，所以状态机的设计非常重要。根据所学的知识，我将每个指令拆分成IF、ID、EX、MEM和WB五个阶段中的若干阶段，并逐一设计出这些指令的不同阶段所要产生的控制信号。状态机如下表（其中EX_LS 为Load和Store类指令共同的执行阶段，WB_L为Load类指令共有的写回阶段，WB_I为所有立即数计算指令共有的写回阶段）：

指令	IF	ID	EX	MEM	WB
RTYPE	IF	ID	EX_RTYPE		WB_RTYPE
LW	IF	ID	EX_LS	MEM_LW	WB_L
LBU	IF	ID	EX_LS	MEM_LBU	WB_L
LB	IF	ID	EX_LS	MEM_LB	WB_L
SW	IF	ID	EX_LS	MEM_SW	
SB	IF	ID	EX_LS	MEM_SB	
BEQ	IF	ID	EX_BEQ		
BNE	IF	ID	EX_BNE		
J	IF	ID	EX_J		
ADDI	IF	ID	EX_ADDI		WB_I
ANDI	IF	ID	EX_ANDI		WB_I
ORI	IF	ID	EX_ORI		WB_I
SLTI	IF	ID	EX_SLTI		WB_I

对于每个状态，我设计了21位的控制信号输出，他们分别是：

```
memwrite[1:0]       写存储器的类型，0表示不写，1表示写Word，2表示写Byte
pcwrite             用于计算pcen，pcen最后决定是否更新pc
irwrite             决定从存储器读出的数据是否当作指令进行译码
regwrite            是否写寄存器
alusrca             决定ALU的第一个运算数
branch              是否分支,用于计算pcen
iord                决定了从存储器读出的为指令还是数据
memtoreg            是否有从存储器到寄存器的数据存储
regdst              是否为R类指令
bne                 是否为bne指令,用于计算pcen
alusrcb[2:0]        决定ALU的第二个运算数
pcsrc[1:0]          决定下一个pc的计算方式
aluop[2:0]          决定了alu计算的方式，传给aludec进行具体的控制
ltype[1:0]          读存储器的类型，0表示Word，1表示写Byte，2表示写UnsignedByte
```

pcen的计算方式如下（其中zero表示ALU的计算结果是否为0）：

```
/*controller.sv*/
assign pcen = pcwrite | (branch & zero) | (bne & ~zero);
```

对于不同状态的控制信号，在下面这段核心代码中给出：

```
/*maindec.sv*/
assign {memwrite, pcwrite, irwrite, regwrite,
        alusrca, branch, iord, memtoreg, regdst,
        bne, alusrcb, pcsrc, aluop, ltype} = controls;
always_comb
    case(state)
            IF:         controls <= 21'b00_110_00000_0_001_00_000_00;
            ID:         controls <= 21'b00_000_00000_0_011_00_000_00;
            EX_LS:      controls <= 21'b00_000_10000_0_010_00_000_00;
            MEM_LW:     controls <= 21'b00_000_00100_0_000_00_000_00;
            MEM_LB:     controls <= 21'b00_000_00100_0_000_00_000_10;
            MEM_LBU:    controls <= 21'b00_000_00100_0_000_00_000_01;
            WB_L:       controls <= 21'b00_001_00010_0_000_00_000_00;
            MEM_SW:     controls <= 21'b01_000_00100_0_000_00_000_00;
            MEM_SB:     controls <= 21'b10_000_00100_0_000_00_000_00;
            EX_RTYPE:   controls <= 21'b00_000_10000_0_000_00_010_00;
            WB_RTYPE:   controls <= 21'b00_001_00001_0_000_00_000_00;
            EX_BEQ:     controls <= 21'b00_000_11000_0_000_01_001_00;
            EX_BNE:     controls <= 21'b00_000_10000_1_000_01_001_00;
            EX_J:       controls <= 21'b00_100_00000_0_000_10_000_00;
            EX_ADDI:    controls <= 21'b00_000_10000_0_010_00_000_00;
            EX_ANDI:    controls <= 21'b00_000_10000_0_100_00_011_00;
            EX_ORI:     controls <= 21'b00_000_10000_0_100_00_100_00;
            EX_SLTI:    controls <= 21'b00_000_10000_0_010_00_101_00;
            WB_I:       controls <= 21'b00_001_00000_0_000_00_000_00;
            default:    controls <= 21'b00_000_xxxxx_x_xxx_xx_xxx_xx;
        endcase
```

ALU控制单元的设计和单周期MIPS一致，更多细节见如下三个源文件

```
controller.sv
maindec.sv
aludec.sv
```

五、数据通路设计

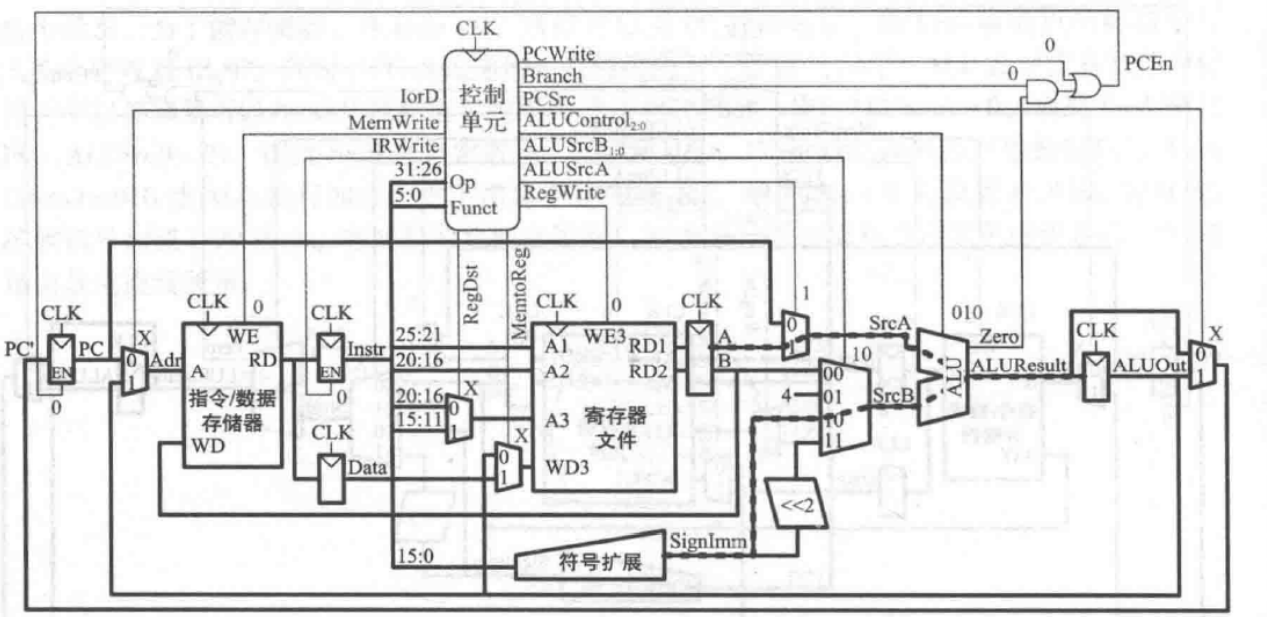


图 7-34　存储器地址计算时的数据流

上面这张图是书上有一定功能的多周期MIPS的数据通路的示意图，支持**SW**，**LW**，**BEQ**和**R**类指令。通过选择合适的ALU运算数A和运算数B，可以用ALU来计算出下一条指令的可能的地址（分支后的地址、顺序下一条的地址），通过pcsrc选择其中之一，并用pcen来决定是否进行读取指令操作。然后通过iord来决定从存储器中读出的是指令还是数据，从而决定要读取寄存器的地址。其他流程与单周期MIPS类似，这里不再赘述。

BEQ指令和**BEQ**指令非常类似，我们只要增加一个控制信号bne来修改**BEQ**指令的数据流即可。

为了支持**J**指令，我们需要将pcsrc的值增加到3个选项，也就是将原来的2:1复用器变为3:1复用器，其中跳转后指令的地址由当前指令的低26位左移2位后拼接目前pc的高4位得到。具体的3:1复用器如下:

```
/*datapath.sv*/
mux3 #(N)       pcmux(aluresult, aluout, {pc[31:28], instr[25:0], 2'b00},pcsrc, pcnext);
```

上面的数据通路实际上也支持**ADDI**和**SLTI**两个指令，但我们为了增加对**ANDI**和**ORI**指令的支持，必须增加一个零扩展模块，作为ALU的运算数B的选择之一，所以我们需要把原来的4:1复用器，变为5:1复用器。具体的5:1复用器如下:

```
/*datapath.sv*/
mux5 #(N)       srcbmux(writedata, 32'b100, signimm, signimmsh, zeroimm, alusrcb, srcb);
```

最后，为了拓展**LB**，**LBU** 这两个操作。我们对从存储区读出的数据进行选择，选择其中所需的Byte进行零拓展和符号拓展，再通过一个3:1复用器选择出，最后要使用的数据。这一块的实现如下:

```
/*datapath.sv*/
zeroext #(B,N)  lbze(mbyte, mbytezext);
signext #(B,N)  lbse(mbyte, mbytesext);
mux3 #(N)       datamux(readdata, mbytezext, mbytesext, ltype, memdata);
```

数据通路的完整代码如下，其中使用了参数化设计，便于拓展位64位版本

```
/*datapath.sv*/
module datapath #(parameter N = 32, I = 16 ,B = 8)(
    input   logic       clk, reset,
    input   logic       pcen, irwrite,
    input   logic       regwrite,
    input   logic       iord, memtoreg, regdst, alusrca,
    input   logic [2:0] alusrcb,
    input   logic [1:0] pcsrc,
    input   logic [2:0] alucontrol,
    input   logic [1:0] ltype,
    output  logic [5:0] op, funct,
    output  logic       zero,
    output  logic [N-1:0]dataadr, writedata,
    input   logic [N-1:0]readdata,
    output  logic [7:0]  pclow,
    input   logic [4:0]  checka,
    output  logic [N-1:0]check
);
    logic [4:0]     writereg;
    logic [N-1:0]   pcnext, pc;
    logic [N-1:0]   instr, data, srca, srcb;
    logic [N-1:0]   rda;
    logic [N-1:0]   aluresult, aluout;
    logic [N-1:0]   signimm;
    logic [N-1:0]   zeroimm;
    logic [N-1:0]   signimmsh;
    logic [N-1:0]   wd3, rd1, rd2;
    logic [N-1:0]   memdata, mbytezext, mbytesext;
    logic [B-1:0]   mbyte;
    assign op = instr[N-1:26];
    assign funct = instr[5:0];
    assign pclow = pc[9:2];
    flopenr #(N)    pcreg(clk, reset, pcen, pcnext, pc);
    mux2 #(N)       adrmux(pc, aluout, iord, dataadr);
    flopenr #(N)    instrreg(clk, reset, irwrite, readdata, instr);
    mux4 #(B)       lbmux(readdata[N-1:24], readdata[23:16], readdata[15:8],
                        readdata[7:0], aluout[1:0], mbyte);
    zeroext #(B,N)  lbze(mbyte, mbytezext);
    signext #(B,N)  lbse(mbyte, mbytesext);
    mux3 #(N)       datamux(readdata, mbytezext, mbytesext, ltype, memdata);
    flopr #(N)      datareg(clk, reset, memdata, data);
    mux2 #(5)       regdstmux(instr[20:16],instr[15:11], regdst, writereg);
    mux2 #(N)       wdmux(aluout, data, memtoreg, wd3);
    regfile#(N,32)  regfile(clk, regwrite, instr[25:21], instr[20:16],
                        writereg, wd3, rd1, rd2, checka, check);
    flopr #(N)      rdareg(clk, reset, rd1, rda);
    flopr #(N)      wdreg(clk, reset, rd2, writedata);
    signext#(I,N)   signext(instr[15:0], signimm);
    zeroext#(I,N)   zeroext(instr[15:0], zeroimm);
    sl2             immsh(signimm, signimmsh);
    mux2 #(N)       srcamux(pc, rda, alusrca, srca);
```

```
    mux5 #(N)       srcbmux(writedata, 32'b100, signimm, signimmsh, zeroimm, alusrcb, srcb);
    alu             alu(srca, srcb, alucontrol, aluresult, zero);
    flopr #(N)      alureg(clk, reset, aluresult, aluout);
    mux3 #(N)       pcmux(aluresult, aluout, {pc[31:28], instr[25:0], 2'b00},pcsrc, pcnext);
endmodule
```

六、存储器设计

我在对存储器的设计上选择了参数化实现，设定了单元的大小N和单元数L。目前N=32，L=256，即以字为基本单元。后续改进中会设定N=64，即以双字为基本单元。我实现了在存储器上写字（SW）、写半字（SH）和写位（SB）的操作，类似的操作可以应用到N=64的情况，可以说是为下一步的改进打下了基础。存储器的完整代码如下：

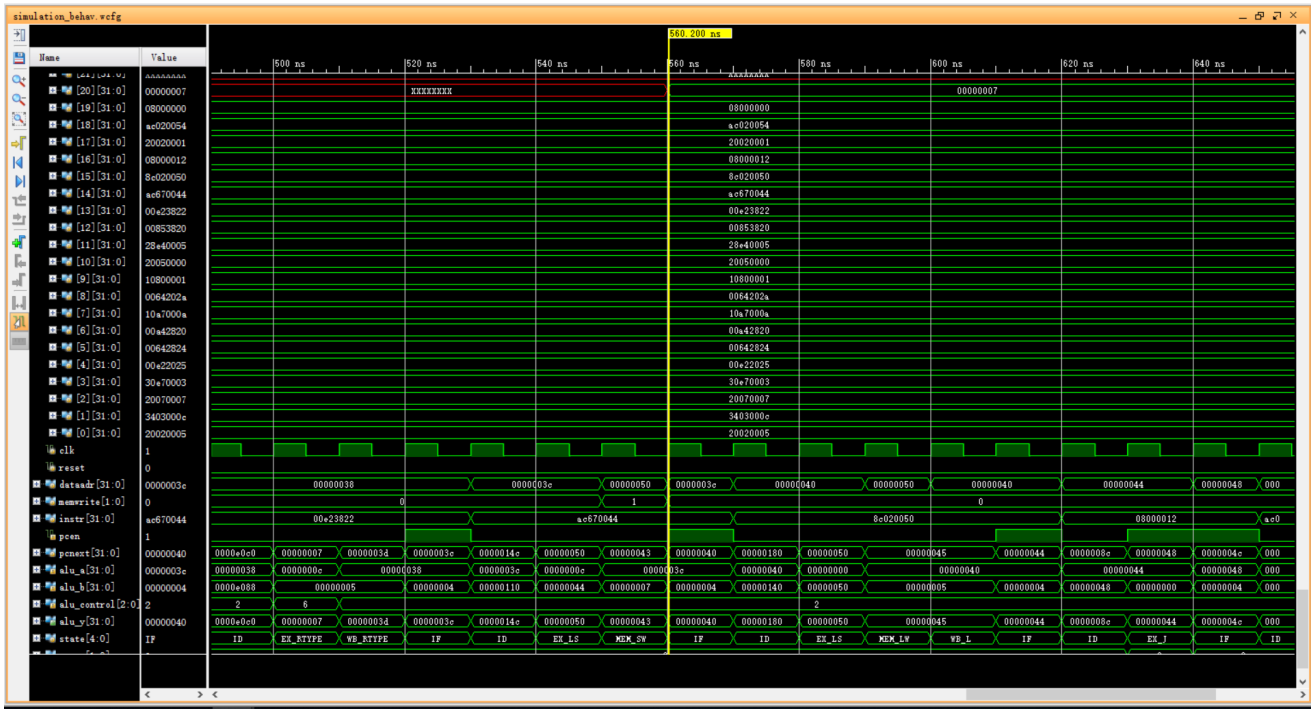
```
/*mem.sv*/
module mem#(parameter N = 32, L = 256)(
    input   logic           clk,
    input   logic [1:0]     memwrite,
    input   logic [N-1:0]   dataadr, writedata,
    output  logic [N-1:0]   readdata,
    input   logic [7:0]     checka,
    output  logic [N-1:0]   check
);
    logic [N-1:0] RAM [L-1:0];
    initial
        $readmemh("C:/Users/will131/Documents/workspace/MIPS_V2.0/memfile.dat",RAM);
    assign readdata = RAM[dataadr[N-1:2]];
    assign check = RAM[checka];
    always @(posedge clk)
        begin
        if (memwrite===1)
            RAM[dataadr[N-1:2]] <= writedata;
        else if (memwrite===2) //B
                case (dataadr[1:0])
                    2'b11:  RAM[dataadr[N-1:2]][7:0]  <= writedata[7:0];
                    2'b10:  RAM[dataadr[N-1:2]][15:8] <= writedata[7:0];
                    2'b01:  RAM[dataadr[N-1:2]][23:16]<= writedata[7:0];
                    2'b00:  RAM[dataadr[N-1:2]][31:24]<= writedata[7:0];
                endcase
        else if (memwrite===3) //H
            case (dataadr[1])
                    1:  RAM[dataadr[N-1:2]][15:0]  <= writedata[15:0];
                    0:  RAM[dataadr[N-1:2]][31:16] <= writedata[15:0];
                endcase
        end
endmodule
```

七、测试

我设计了三个程序，结合书上的程序进行测试，他们的大致内容如下，具体代码见test文件夹：

```
standard        书上的测试程序
standard2        测试除LB，LBU，SB外的15条指令的正确性
power2            生成2的次幂，并存在存储器中，主要用于演示
testls             测试LB，LBU，SB，测试数据和指令混合
```

standard2的部分波形图如下，可以查看存储器的变化、状态机中状态的变化和ALU的计算结果等。

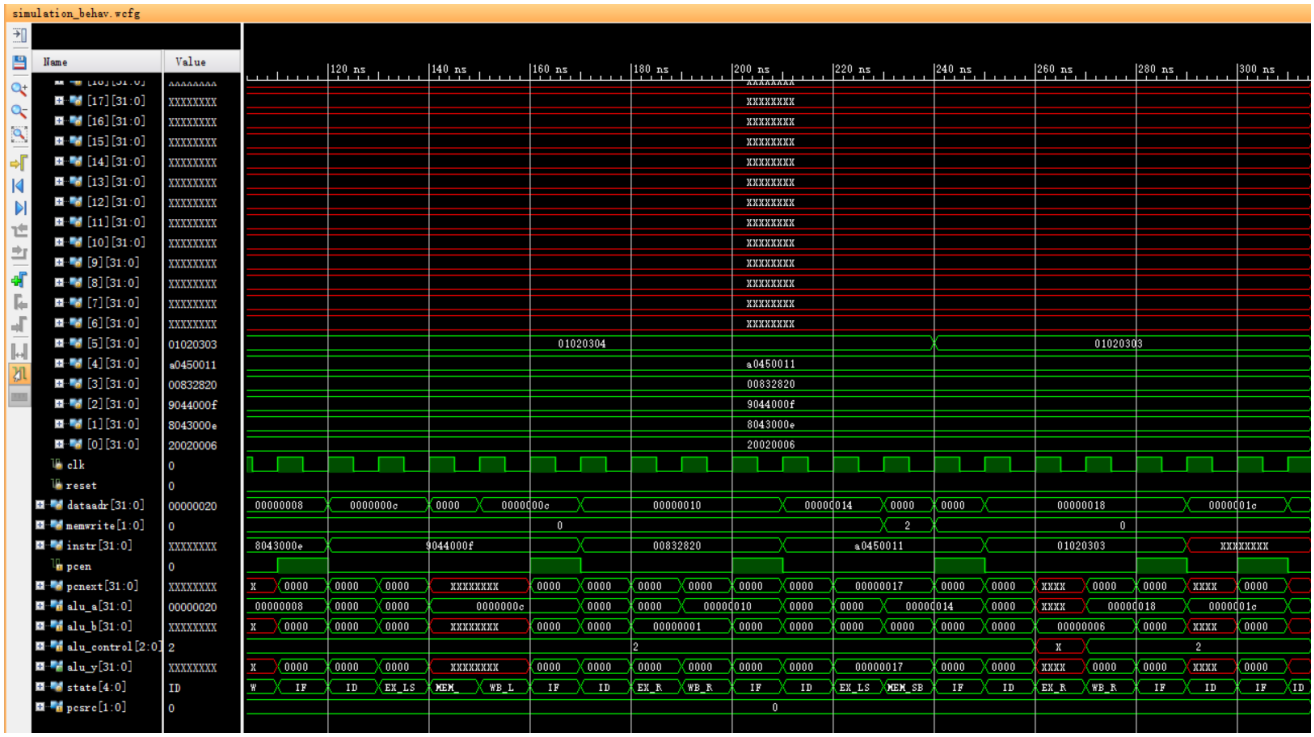


我同时通过$display()输出每个寄存器的变化来验证我程序的正确性。程序正确结束时会停在我们预设的$stop的位置。实现这些测试的核心代码在如下两处：

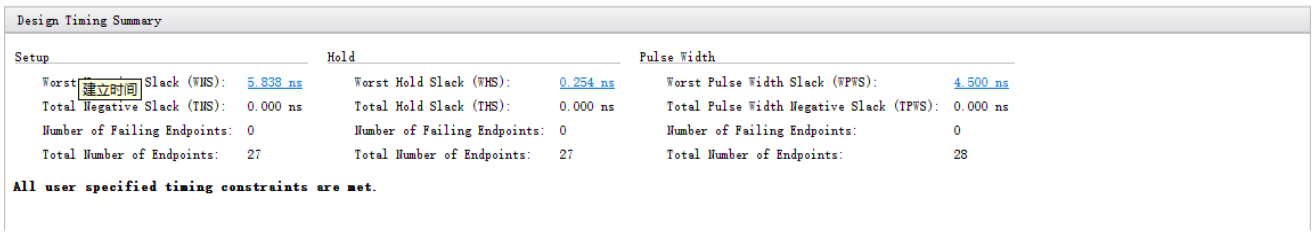
```
/*regfile.sv*/
always_ff @(posedge clk)
    if (we3) begin
        $display("REG%d=%d",wa3,wd3);
        rf[wa3] <= wd3;
    end

/*simulation.sv*/
always @(negedge clk) begin
    if (memwrite) begin
        if (dataadr === 84 & writedata === 7)begin
            $display("LOG:Simulation succeeded");
            $stop;
        end
    end
    cnt = cnt + 1;
    if(cnt === 128)
        $stop;
```

```
end
```

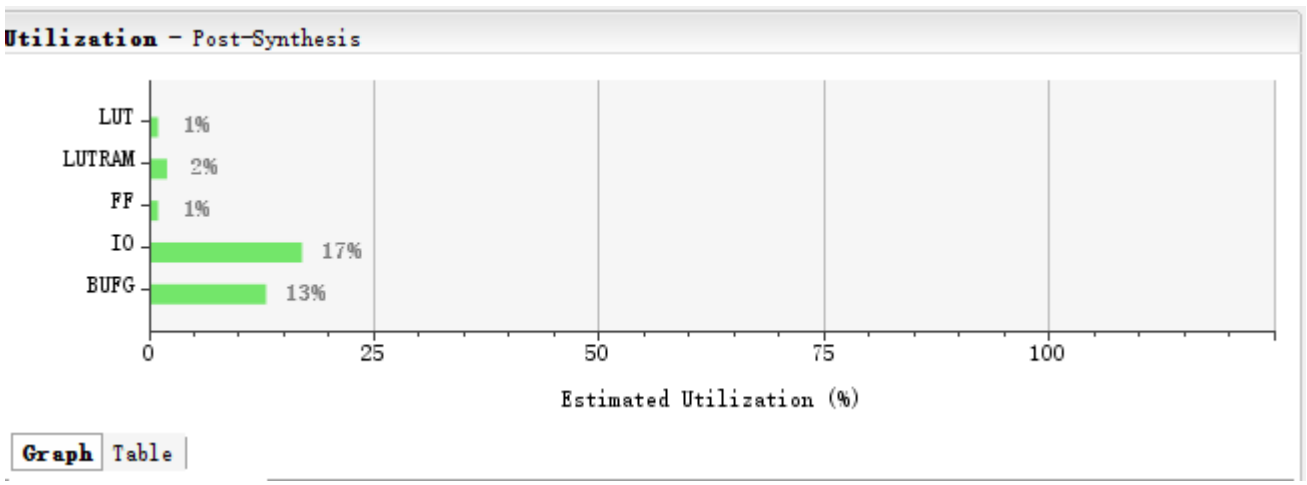
testls.s的部分波形图如下，可以看到程序读出了存储器中下标为5的字的前两位，相加后存放在了第四位。



做完仿真测试之后，我对工程进行Synthesis和Implementation，最终生成二进制文件。时钟情况如下：



工程的资源占用情况如下：



Utilization – Post-Implementation

Resource	Utilization	Available	Utilization %
LUT	819	63400	1.29
LUTRAM	312	19000	1.64
FF	280	126800	0.22
IO	36	210	17.14
BUFG	4	32	12.50

Graph **Table**

八、演示设计

由于所有的模块都有相同的clk和reset信号控制，我用两个开关来绑定这两个信号，从而可以随时选择重新执行程序或者**暂停程序**的执行。为了方便在Nexys4实验板上展示我的项目，我还实现了在8位7段显示管上显示32位数的功能（每个7段显示管表示0-f之间的一个**十六进制位**），用于显示程序的各种信息。

在一般情况下，6、7位为你查看**寄存器的序号**（0-31），4、5位显示的是那个**寄存器的值**的低两位，2、3位显示的是**PC的值**，0、1位显示的是当前**状态机的状态**（状态-序号对应表见states.txt）。其中寄存器的序号使用开关组成二进制数进行选择。当存储器被写入的时候，屏幕上会出现形如EExyEEzw的信息，表示存储器第xy个字被写入了zw（可能只写其中的某个Byte）。我们还可以通过开关（和寄存器检查的开关组一致）选择**存储器检查模式**，以字为单位检查存储器中任意地址上的值。另外，为了便于查看关键的程序执行，我还实现了用开关选择**快速运行**与**常速运行**两种模式。

具体的代码实现见onboard.sv，详细的使用说明见states.txt.