实验报告

一、 实验简述

本实验使用 modelsim 完成代码编写、编译运行和波形仿真，采用多种类型指令对编写结果进行验证，主要对 ALU 和 CPU 进行了测试。在 10 条基础指令的基础上，我尽可能多地增加了部分指令，最终共实现了 30 条指令，并更新了数据通路图和状态图，并取得了良好的测试结果。

二、 单周期 CPU 设计

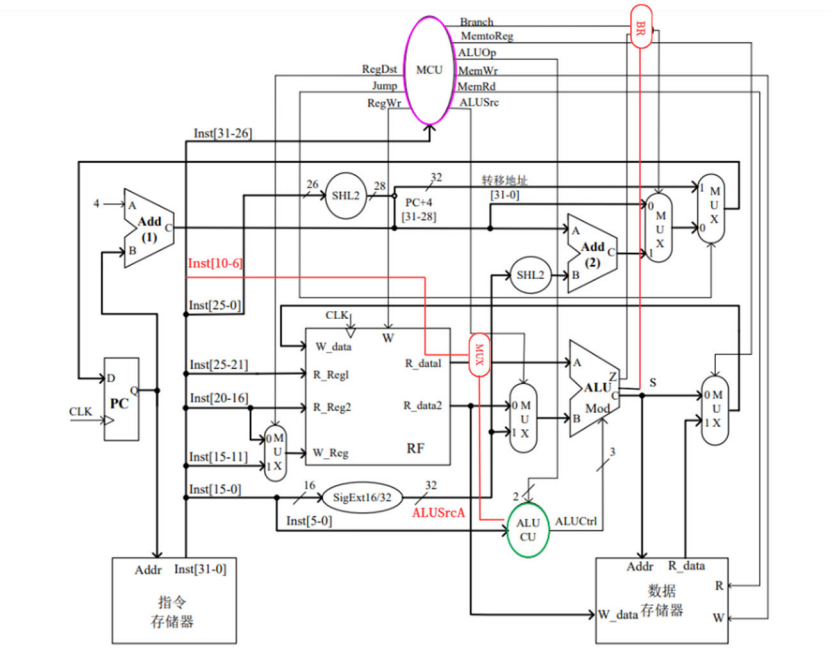
基本指令如下

指令类型	指令	指令格式类型	指令功能描述	
			取指令阶段	执行指令阶段
存储器访问指令	lw Rt, Imm16(Rs)	I-型	（1）M[PC] 或者 IR←M[PC] （2）PC←(PC)+ 4	（3）addr←(Rs)+ SigExt(Imm16) （4）Rt←M[addr]
	sw Rt, Imm16(Rs)			（3）addr←(Rs)+ SigExt(Imm16) （4）M[addr]←(Rt)
算术/逻辑运算指令	add Rd, Rs, Rt	R-型		（3）Rd←(Rs)+(Rt)，判溢出
	sub Rd, Rs, Rt			（3）Rd←(Rs)-(Rt)，判溢出
	addu Rd, Rs, Rt			（3）Rd←(Rs)+(Rt)
	and Rd, Rs, Rt			（3）Rd←(Rs)∧(Rt)
	or Rd, Rs, Rt			（3）Rd←(Rs)∨(Rt)
	nor Rd, Rs, Rt			（3）Rd←(Rs)⊕(Rt)
程序转移指令	beq Rs, Rt, Addr16	I-型		（3）if ((Rs) == (Rt) PC← (PC)+ SigExt(Addr16)×4
	j Addr26	J-型		（3）PC←PC[31~28] ‖ (Addr26)×4

单周期 CPU 实现的指令除了基本 10 条外，增加了 R-R 运算指令、R-I 运算指令以及分支指令，共 30 条。主要参考《MIPS-C 指令集》一书中的指令格式进行设计。具体指令如下

lw	sw	add	addu	and	nor	or	sllv	slt	Sltu
sra	srav	srl	srlv	sub	subu	xor	addi	addiu	andi
lui	ori	slti	sltiu	xori	beq	bgez	bgtz	blez	j

数据通路如下



修改部分用红色标出，主要包括以下部分

1. 新增 MUX，用于选择 ALU 的 A 操作数来源，由于引入了移位指令，需要判断用寄存器值还是 Shamt。
2. 新增 BR 部件，同时对 Branch 做了位扩增，以满足多种分支指令，并根据 ALU 运算结果中的 Z（零）和 S（符号）标志，进行分支判断。

主要部件设计（代码粘贴格式可能存在问题**）**

1. **CU** 设计如下，根据不同指令的操作码来设置各种控制信号，扩展的 BranchOp 用于获取分支指令的类型，其余的和书上的类似。

```
    module CU(OPCode, RegDst, RegWr, ALUSrcB, MemRd, MemWr, MemtoReg,
BranchOp, Jump, ALUOp);
    input [5:0] OPCode;
    output reg RegDst, RegWr, ALUSrcB, MemRd, MemWr, MemtoReg, Jump;
    output reg [2:0] BranchOp;
    output reg [3:0] ALUOp;

    parameter R = 6'b000000,
              lw = 6'b100011,
              sw = 6'b101011,
              j = 6'b000010,
              addi = 6'b001000,
              addiu = 6'b001001,
              andi = 6'b001100,
              lui =6'b001111,
              ori = 6'b001101,
              slti = 6'b001010,
              sltiu = 6'b001011,
              xori = 6'b001110,
              beq = 6'b000100,
              bgez = 6'b000001,
              bgtz = 6'b000111,
              blez = 6'b000110,
              bne =  6'b000101;

    always @(OPCode)
    begin
        case(OPCode)
            R:  {RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp, Jump, ALUOp} <= 14'b11000000001100;
            lw:{RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp, Jump, ALUOp} <= 14'b01110100000000;
            sw:  {RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp, Jump, ALUOp} <= 14'bx0101x00000000;
            j:{RegDst, RegWr, ALUSrcB, MemRd, MemWr, MemtoReg, BranchOp,
Jump, ALUOp} <= 14'bx0x00x0001xxxx;
```

```
          addi:{RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp, Jump, ALUOp}  <= 14'b01100000000000;
          addiu:{RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp, Jump, ALUOp} <= 14'b01100000000010;
          andi:{RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp, Jump, ALUOp} <= 14'b01100000000011;
          lui:{RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp, Jump, ALUOp} <= 14'b01100000000100;
          ori:{RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp, Jump, ALUOp} <= 14'b01100000000101;
          slti:{RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp, Jump, ALUOp} <= 14'b01100000000110;
          sltiu:{RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp, Jump, ALUOp} <= 14'b01100000000111;
          xori:{RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp, Jump, ALUOp} <= 14'b01100000001000;

          beq:{RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp,Jump, ALUOp} <= 14'bx0000x00100001;
          bgez:{RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp,Jump, ALUOp} <= 14'bx0000x01000001;
          bgtz:{RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp, Jump, ALUOp} <= 14'bx0000x01100001;
          blez:{RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp,Jump, ALUOp} <= 14'bx0000x10000001;
          bne:{RegDst,  RegWr,  ALUSrcB,  MemRd,  MemWr,  MemtoReg,
BranchOp,Jump, ALUOp} <= 14'bx0000x10100001;
       endcase
   end
   endmodule
```

2. **ALU CU** 设计如下，根据不同指令 ALUOp 操作码，设置不同的 ALUCtrl 和 ALUSrcA

```
   module ALU_CU (ALUOp,func,ALUCtrl,ALUSrcA);
   input [5:0] func;
   input [3:0] ALUOp;
   output reg [3:0] ALUCtrl;
   output reg ALUSrcA;
   always @(func or ALUOp) begin
       casex(ALUOp)
           4'b0000:{ALUCtrl, ALUSrcA} <= 5'b00000;   //lw or sw
           4'b0001:{ALUCtrl, ALUSrcA} <= 5'b00100;   //branch
           4'b0010:{ALUCtrl, ALUSrcA} <= 5'b00010;   //addiu
           4'b0011:{ALUCtrl, ALUSrcA} <= 5'b01000;   //andi
           4'b0100:{ALUCtrl, ALUSrcA} <= 5'b11010;   //lui
```

```
          4'b0101:{ALUCtrl, ALUSrcA} <= 5'b01010;   //ori
          4'b0110:{ALUCtrl, ALUSrcA} <= 5'b10000;   //slti
          4'b0111:{ALUCtrl, ALUSrcA} <= 5'b10010;   //sltiu
          4'b1000:{ALUCtrl, ALUSrcA} <= 5'b01110;   //xori
          4'b11xx: begin
            case (func)
              6'b100000: {ALUCtrl, ALUSrcA} <= 5'b00000; //add
              6'b100001: {ALUCtrl, ALUSrcA} <= 5'b00010; //addu
              6'b100010: {ALUCtrl, ALUSrcA} <= 5'b00100; //sub
              6'b100011: {ALUCtrl, ALUSrcA} <= 5'b00110; //subu
              6'b100100: {ALUCtrl, ALUSrcA} <= 5'b01000; //and
              6'b100101: {ALUCtrl, ALUSrcA} <= 5'b01010; //or
              6'b100110: {ALUCtrl, ALUSrcA} <= 5'b01110; //xor
              6'b100111: {ALUCtrl, ALUSrcA} <= 5'b01100; //nor
              6'b101010: {ALUCtrl, ALUSrcA} <= 5'b10000; //slt
              6'b101011: {ALUCtrl, ALUSrcA} <= 5'b10010; //sltu
              6'b000000: {ALUCtrl, ALUSrcA} <= 5'b10101; //sll
              6'b000010: {ALUCtrl, ALUSrcA} <= 5'b10111; //srl
              6'b000011: {ALUCtrl, ALUSrcA} <= 5'b11001; //sra
              6'b000100: {ALUCtrl, ALUSrcA} <= 5'b10100; //sllv
              6'b000110: {ALUCtrl, ALUSrcA} <= 5'b10110; //srlv
              6'b000111: {ALUCtrl, ALUSrcA} <= 5'b11000; //srav
            endcase
          end
      endcase
  end
  endmodule
```

3. ALU 设计如下，除了算术运算外，还要根据指令设置零位，符号位和溢出位，前两个用于分支判断，最后一个用于溢出判断（本实验未实现中断处理，因此虽然使用了但未实际处理）。

```
  module ALU(A,B,C,ALUCtrl,Zero,Sign,O);
  input [31:0] A, B;
  input [3:0] ALUCtrl;
  output reg [31:0] C;
  output reg Zero, Sign, O;

  parameter ADD=4'b0000,
            ADDU=4'b0001,
            SUB=4'b0010,
            SUBU=4'b0011,
            AND=4'b0100,
            OR =4'b0101,
            NOR=4'b0110,
            XOR=4'b0111,
```

```
            SLT=4'b1000,
            SLTU=4'b1001,
            SLL=4'b1010,
            SRL=4'b1011,
            SRA=4'b1100,
            LUI=4'b1101;
    always @(*)
    begin
        case(ALUCtrl)
            ADD: begin
                C <= A + B;
                O <= (A[31] == B[31] && C[31] != A[31])? 1:0;
            end
            ADDU: C <= A + B;
            SUB: begin
                C <= A - B;
                O <= (A[31] != B[31] && C[31] != A[31])? 1:0;
            end
            SUBU: C <= A - B;
            AND : C <= A & B;
            OR: C <= A | B;
            NOR:C <= ~(A | B);
            XOR:C <= A ^ B;
            SLT: begin
                if(A[31] > B[31] || A < B) C <= 1;
                else C <= 0;
            end
            SLTU:C <= (A < B)? 1:0;
            SLL:C <= (B << A);
            SRL:C <= (B >> A);
            SRA:C <= ($signed(B) >>> A);
            LUI:C <= {B[15:0],16'b0};
        endcase
    end

    always @(C) begin
        if (C == 0)
            Zero <= 1;
        else
            Zero <= 0;
        if(C[31] == 1)
            Sign <= 1;
        else
            Sign <= 0;
```

```
        if(ALUCtrl != ADD && ALUCtrl != SUB)
            O <= 0;
    end
    endmodule
```

4. **Branch5** 不同分支执行的选择部分如下

```
    module Branch5(BranchOp,Zero,Sign,Branch);
    input [2:0] BranchOp;
    input Zero,Sign;
    output reg Branch;
    always @(*) begin
        case(BranchOp)
            3'b000: Branch <= 0;
            3'b001: Branch <= Zero;
            3'b010: Branch <= ~Sign;
            3'b011: Branch <= ~(Zero | Sign);
            3'b100: Branch <= Zero | Sign;
            3'b101: Branch <= ~Zero;
        endcase
    end
    endmodule
```

5. 存储器 **RAM** 设计如下

```
    module RAM(R_data,W_data,Addr,MemRd,MemWr);
    input [31:0] W_data,Addr;
    input MemWr, MemRd;
    output [31:0] R_data;

    reg [31:0] M [255:0];

    assign R_data = MemRd? M[Addr>>2]: 32'hzzzz;

    always @(posedge MemWr)
    begin
        M[Addr] <= W_data;
    end
    endmodule
```

其余部分设计较简单，由于篇幅限制在此不做展示。

测试 CPU 的 testbench 为

```
    `timescale 10ns/1ns
    `include "CPU.v"
    `include "ALU.v"
    `include "ALU_CU.v"
    `include "Add.v"
    `include "Mux2.v"
    `include "RAM.v"
```

```
`include "ROM.v"
`include "CU.v"
`include "PC.v"
`include "Branch5.v"
`include "RF.v"
`include "SigExt.v"

module CPU_tb;
    reg clk;
    CPU cpu(.clk(clk));
    always #5 clk = ~clk;

    initial begin
        $readmemh("initial/inst.txt",cpu.instm.M);
        $readmemh("initial/RF.txt",cpu.rf.RF);
        $readmemh("initial/data.txt",cpu.datam.M);

        clk = 1 ;
        #8000 $finish;
    end
endmodule
```

单周期使用的测试指令为

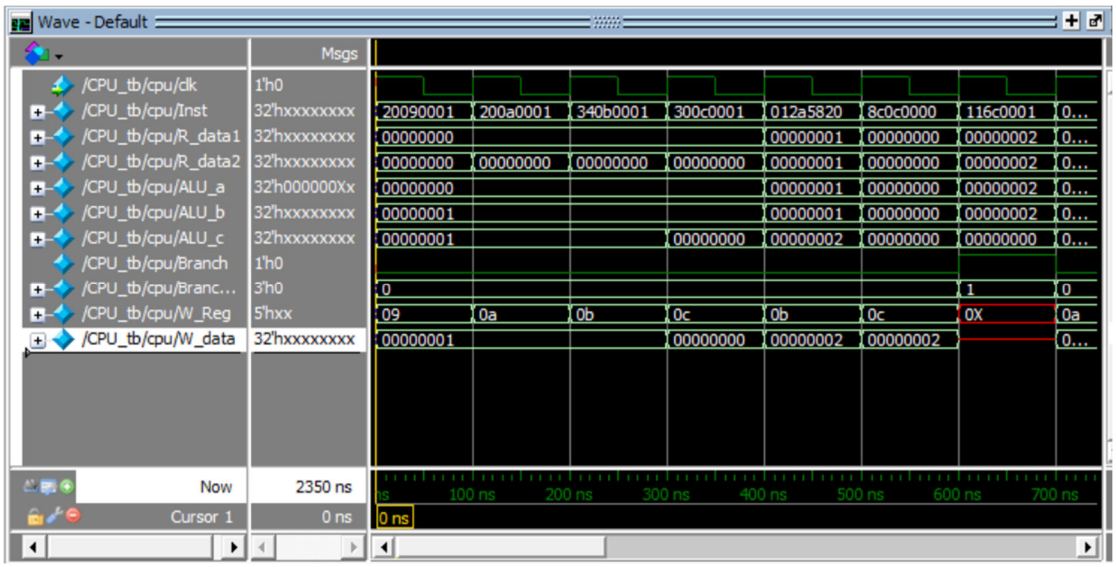
```
addi $t1, $0, 1
addi $t2, $0, 1
ori $t3, $0, 1
andi $t4, $0, 1
add $t3,$t1,$t2
lw $t4, 0($0)
beq  $t3,$t4,branch
add $t3, $t1, $t3
branch:
add $t2, $t1, $t2
sub $t3, $t1, $t2
addu $t3, $t1, $t2
and $t3, $t1, $t2
or $t3, $t1, $t2
slt $t3, $t1, $t2
sllv $t3, $t2, $t1
srlv $t3, $t2, $t1
bgez $t3, branch1
addi $t1, $0, 1
branch1:
addi $t1, $0, 2
```

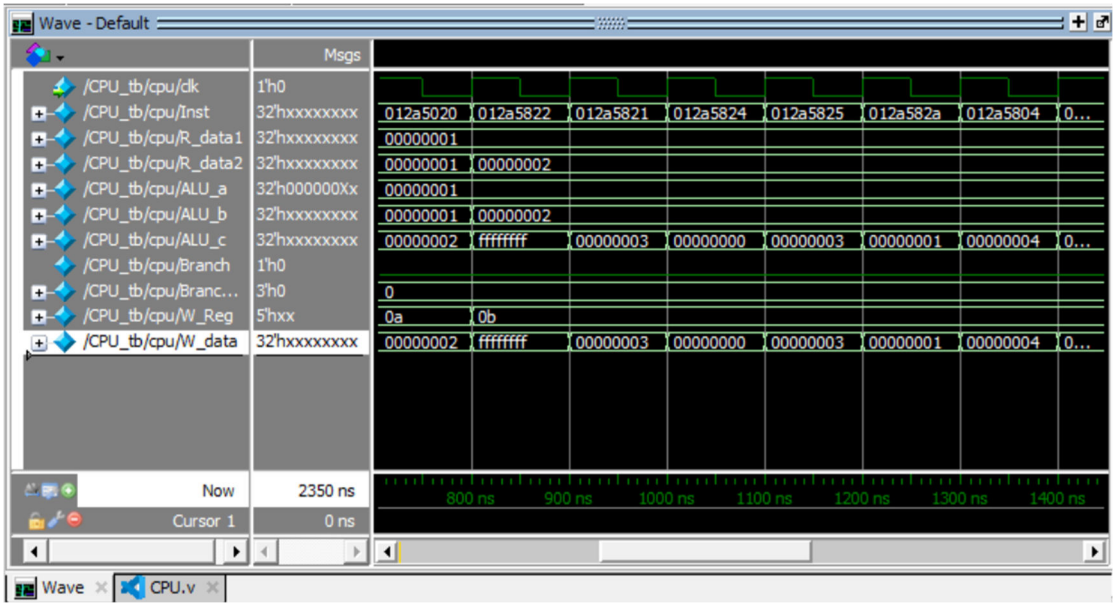
```
    sub $t2, $t3, $t1
    blez $t2, branch2
    addi $t1, $0, 1
    branch2:
    addi $t1, $0, 2
    sll $t3, $t2, 1
    sra $t3, $t2, 1
```

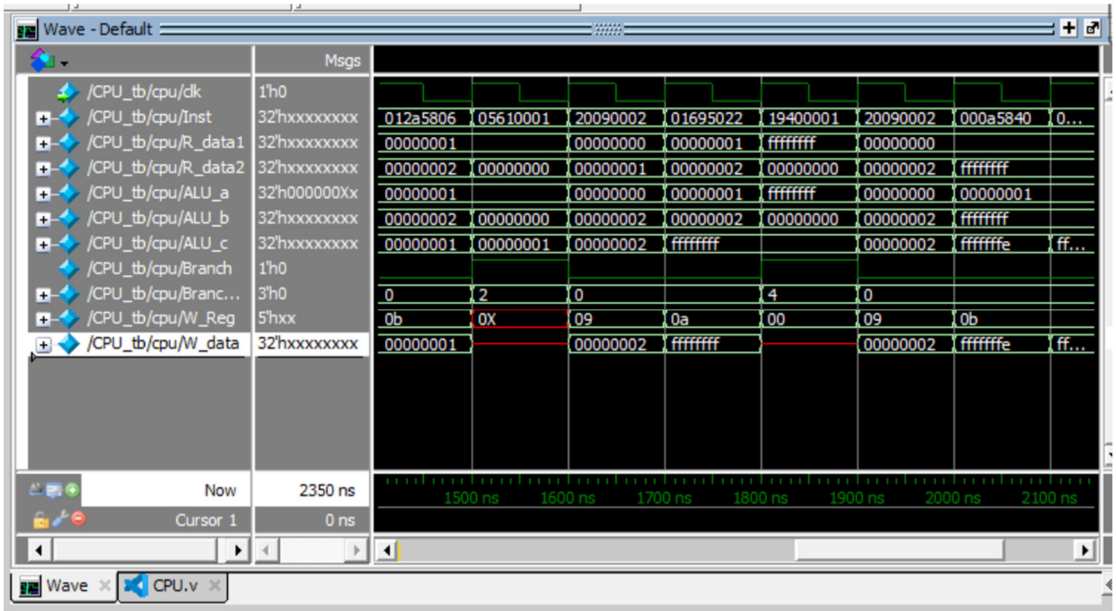
使用 Mars 将指令转为二进制码，存储在 inst.txt 文件中以备程序使用。

```
    20090001
    200a0001
    340b0001
    300c0001
    012a5820
    8c0c0000
    116c0001
    012b5820
    012a5020
    012a5822
    012a5821
    012a5824
    012a5825
    012a582a
    012a5804
    012a5806
    05610001
    20090001
    20090002
    01695022
    19400001
    20090001
    20090002
    000a5840
    000a5843
```

以上编码用于初始化指令存储器，数据存储器每 32 位全部初始化为 2，寄存器堆全部初始化为 0。测试波形如下


Wave - Default
Msgs
/CPU_tb/cpu/clk
/CPU_tb/cpu/Inst
/CPU_tb/cpu/R_data1
/CPU_tb/cpu/R_data2
/CPU_tb/cpu/ALU_a
/CPU_tb/cpu/ALU_b
/CPU_tb/cpu/ALU_c
/CPU_tb/cpu/Branch
/CPU_tb/cpu/Branc...
/CPU_tb/cpu/W_Reg
/CPU_tb/cpu/W_data
Now
2350 ns
Cursor 1
0 ns
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns

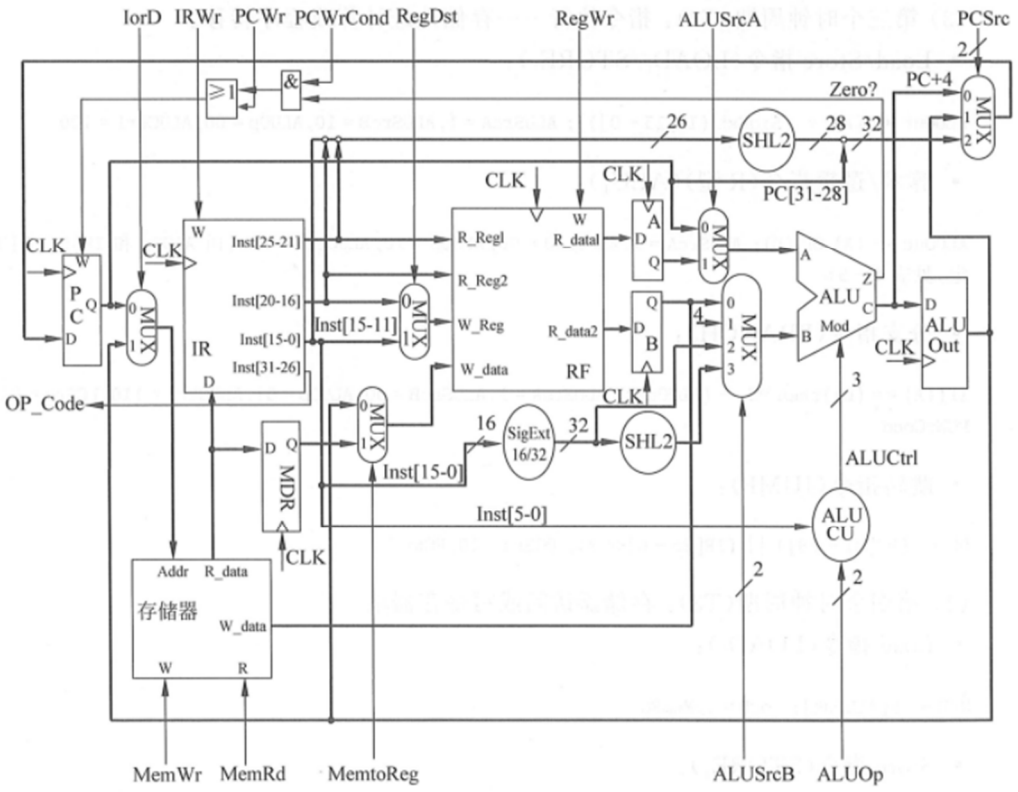

Wave - Default
Msgs
/CPU_tb/cpu/clk
/CPU_tb/cpu/Inst
/CPU_tb/cpu/R_data1
/CPU_tb/cpu/R_data2
/CPU_tb/cpu/ALU_a
/CPU_tb/cpu/ALU_b
/CPU_tb/cpu/ALU_c
/CPU_tb/cpu/Branch
/CPU_tb/cpu/Branc...
/CPU_tb/cpu/W_Reg
/CPU_tb/cpu/W_data
Now
2350 ns
Cursor 1
0 ns
800 ns
900 ns
1000 ns
1100 ns
1200 ns
1300 ns
1400 ns
Wave
CPU.v


Wave - Default
Msgs
/CPU_tb/cpu/clk
/CPU_tb/cpu/Inst
/CPU_tb/cpu/R_data1
/CPU_tb/cpu/R_data2
/CPU_tb/cpu/ALU_a
/CPU_tb/cpu/ALU_b
/CPU_tb/cpu/ALU_c
/CPU_tb/cpu/Branch
/CPU_tb/cpu/Branc...
/CPU_tb/cpu/W_Reg
/CPU_tb/cpu/W_data
Now
2350 ns
Cursor 1
0 ns
1500 ns
1600 ns
1700 ns
1800 ns
1900 ns
2000 ns
2100 ns
Wave
CPU.v

运算和转移等操作均正确，符合预期。以部分指令为例说明：前 4 条均为 R-I 型指令，使用 addi 将 t1 和 t2 置 1，ori 将 t3 置 1，andi 将 t4 置 0，然后第 5 条指令将 t1 和 t2 相加结果放入 t3，此时值为 2，第 6 条从数据存储器 lw 取出数据放入 t4，值也为 2，对 t3 和 t4 使用 beq 发现跳转到了预期地址。后续主要测试了其他运算指令和分支指令，结果均正确。

三、 多周期 CPU 设计

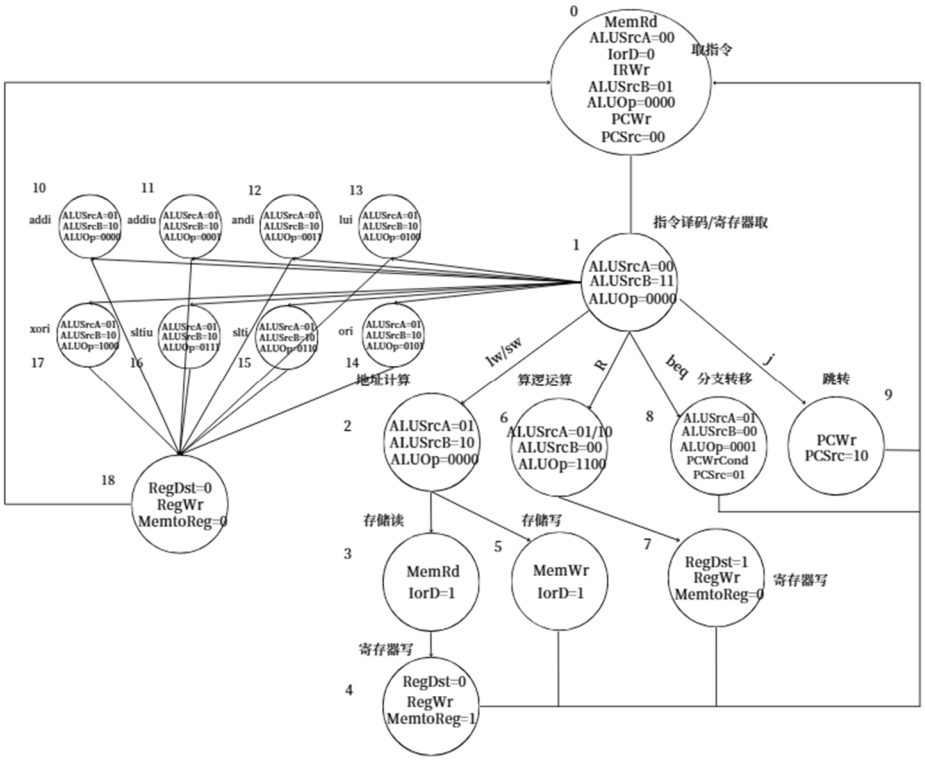
多周期 CPU 实现的指令近似，只是未新增分支指令，共 26 条。
使用课本中的数据通路



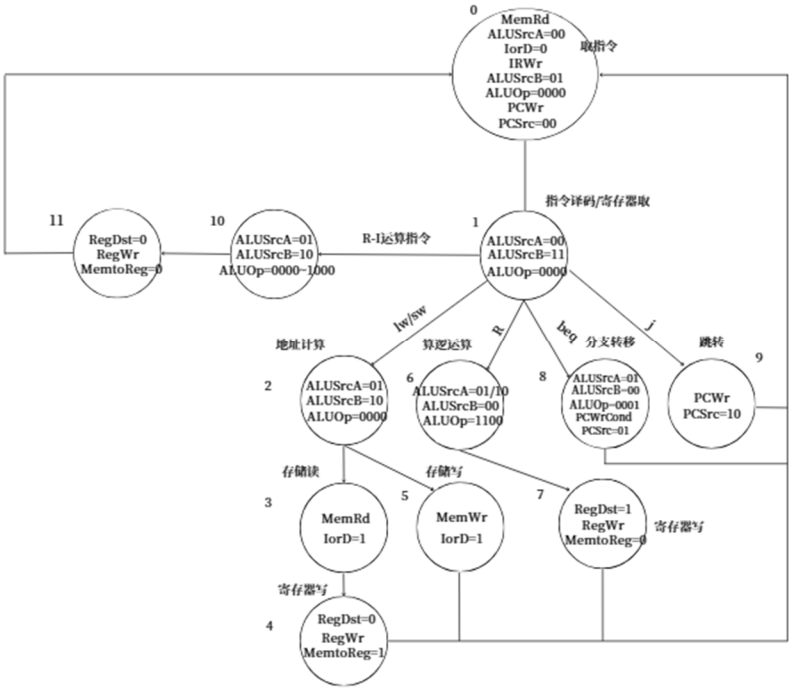
需要做出以下更改：

1. ALUSrcA 为 2 位，并使用三选一 MUX。和单周期一样，用于选择 Shamt。
2. CU 需要使用 func 字段来设置 ALUSrcA，同样是用于选择 Shamt。

由于新增了 R-I 指令，需要对状态图进行修改，新增对应指令的状态（10-17）和寄存器写状态（18），并且状态 6 对应 ALUSrcA 的取值与 func 有关。



后来认为不够精简，进行了修改，将 R-I 不同运算指令的状态合并，而在状态内部根据操作码设置 ALUOp，只增加了两个状态，得到如下结果



考虑到 PLA 实现的复杂性，未使用课本中的 PLA 控制单元设计方法。具体设计控制单元时直接显示指示状态并判断迁移的下一个状态。

由于单多周期中 ALU 和 ALU CU 变化不大，在此不再赘述，只对控制单元 CU 的设计进行展示。

```
    module
CU(PCWr,MemRd,MemWr,PCWrCond,IorD,IRWr,MemtoReg,PCSrc,ALUOp,ALUSrcA,A
```

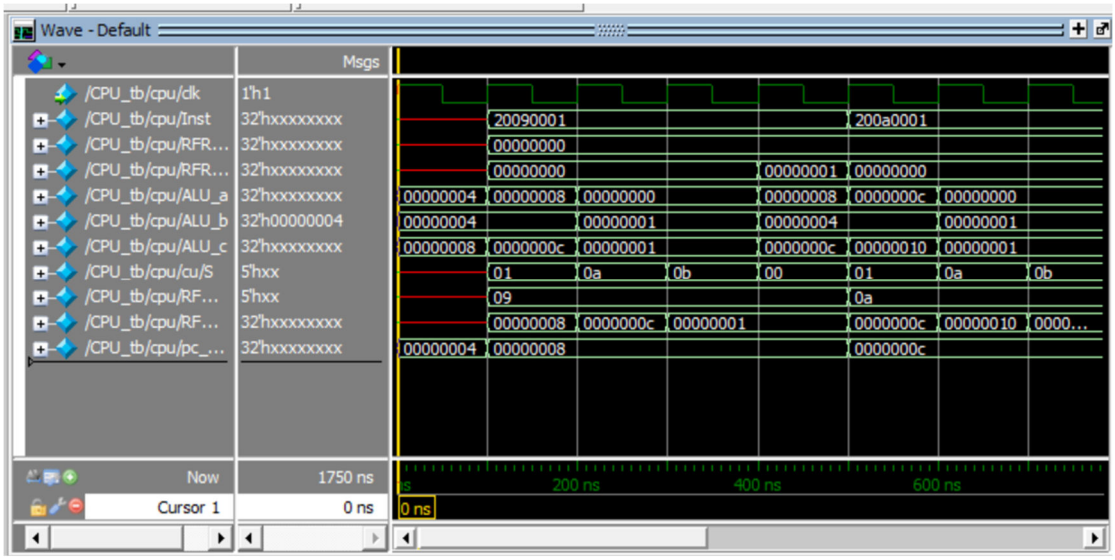
```
LUSrcB,RegWr,RegDst,Op,clk,func);
    input [5:0] func;
    input [5:0] Op;
    input clk;
    output reg [1:0] ALUSrcA, ALUSrcB, PCSrc;
    output reg [3:0] ALUOp;
    output reg PCWrCond, PCWr, IorD, MemRd, MemWr, MemtoReg, IRWr,
RegWr, RegDst;
    parameter S0=0, S1=1, S2=2, S3=3, S4=4, S5=5, S6=6, S7=7, S8=8,
S9=9, S10=10, S11=11;
    parameter R = 6'b000000,
              lw = 6'b100011,
              sw = 6'b101011,
              j = 6'b000010,
              addi = 6'b001000,
              addiu = 6'b001001,
              andi = 6'b001100,
              lui =6'b001111,
              ori = 6'b001101,
              slti = 6'b001010,
              sltiu = 6'b001011,
              xori = 6'b001110,
              beq = 6'b000100;

    reg [4:0] S, NS;
    initial S=0;
    always@(posedge clk) begin
        S <= NS;
    end
    always @(S, Op) begin
    case(S)
        S0: begin
        MemRd=1'b1;
        ALUSrcA=2'b00;
        IorD=1'b0;
        IRWr=1'b1;
        ALUSrcB=2'b01;
        ALUOp=4'b0000;
        PCWr=1'b1;
        PCSrc=2'b00;
        NS=S1;
        RegWr=1'b0;
        RegDst=1'b0;
        MemWr=1'b0;
```

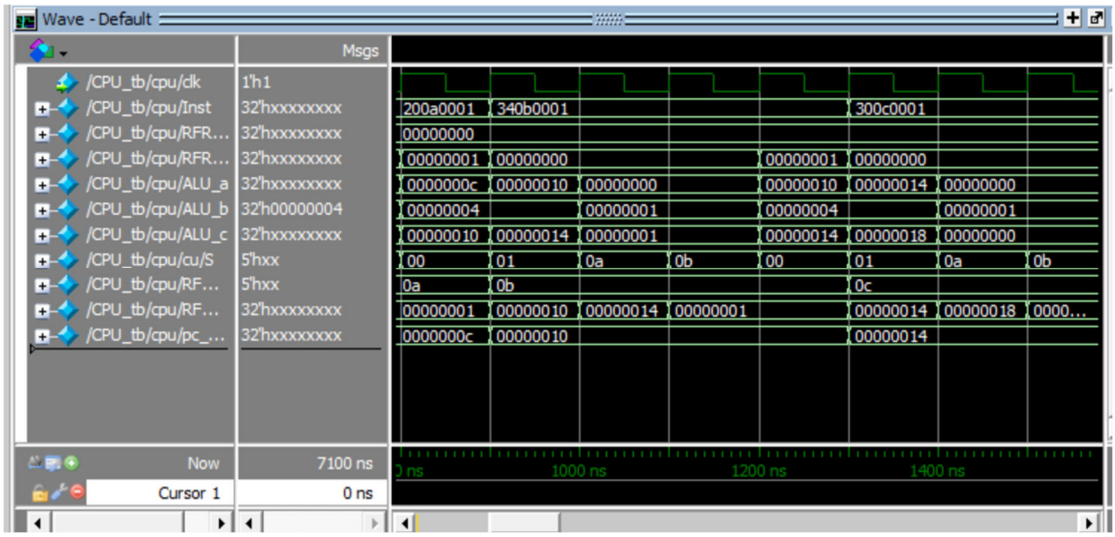
```
PCWrCond= 1'b0;
MemtoReg=1'b0;
end

S1: begin
MemRd=1'b0;
IRWr=1'b0;
ALUSrcA=2'b00;
ALUSrcB=2'b11;
PCWr =1'b0;
ALUOp= 4'b0000;
case(Op)
lw: NS=S2;
sw: NS=S2;
R: NS=S6;
beq: NS=S8;
j: NS=S9;
addi: NS=S10;
addiu: NS=S10;
andi: NS=S10;
lui: NS=S10;
ori: NS=S10;
slti: NS=S10;
sltiu: NS=S10;
xori: NS=S10;
endcase
end

S2: begin
ALUSrcA = 2'b01;
ALUSrcB= 2'b10;
ALUOp = 4'b0000;
case(Op)
lw: NS=S3;
sw: NS=S5;
endcase
end

S3: begin
MemRd=1'b1;
IorD = 1'b1;
NS=S4;
end
```

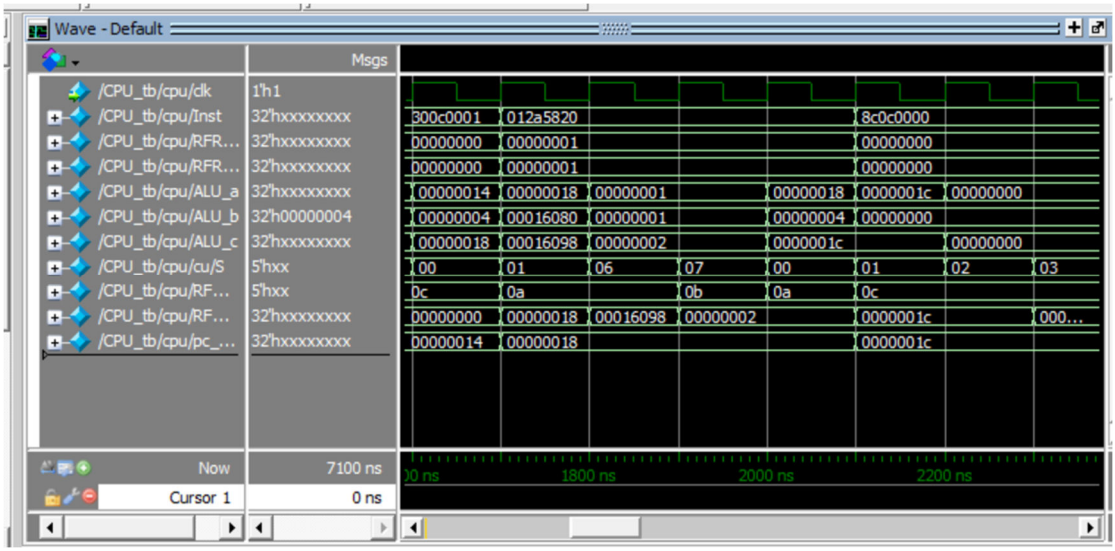
```
        S4: begin
        RegDst = 1'b0;
        RegWr = 1'b1;
        MemtoReg= 1'b1;
        MemRd=1'b0;
        NS=S0;
        end

        S5: begin
        MemWr=1'b1;
        IorD= 1'b1;
        NS=S0;
        end

        S6: begin
        if(func == 6'b000000 || func == 6'b000010 || func == 6'b000011)
ALUSrcA= 2'b10;
        else ALUSrcA= 2'b01;
        ALUSrcB= 2'b00;
        ALUOp = 4'b1100;
        NS = S7;
        end

        S7: begin
        RegDst= 1'b1;
        RegWr = 1'b1;
        MemtoReg = 1'b0;
        NS= S0;
        end

        S8: begin
        ALUSrcA= 2'b01;
        ALUSrcB= 2'b00;
        ALUOp=4'b0001;
        PCWrCond= 1'b1;
        PCSrc = 2'b01;
        NS= S0;
        end

        S9: begin
        PCWr = 1'b1;
        PCSrc= 2'b10;
        NS= S0;
        end
```

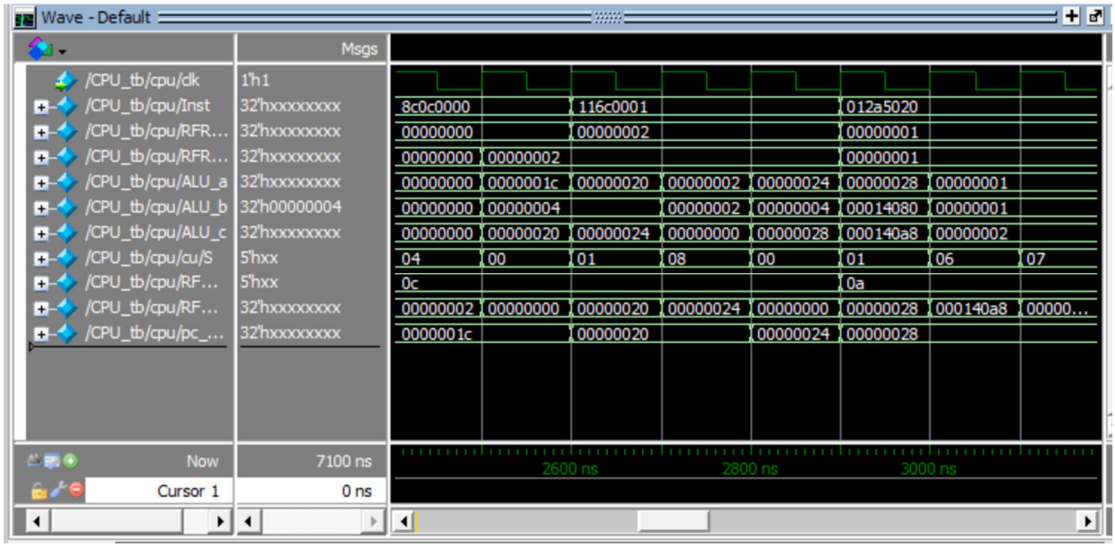
```
        // R-I
        S10: begin
        ALUSrcA = 2'b01;
        ALUSrcB= 2'b10;
        case(Op)
        addi: ALUOp = 4'b0000;
        addiu: ALUOp = 4'b0001;
        andi: ALUOp = 4'b0011;
        lui: ALUOp = 4'b0100;
        ori: ALUOp = 4'b0101;
        slti: ALUOp = 4'b0110;
        sltiu: ALUOp = 4'b0111;
        xori: ALUOp = 4'b1000;
        endcase
        NS = S11;
        end

        S11: begin
        RegDst= 1'b0;
        RegWr= 1'b1;
        MemtoReg= 1'b0;
        NS = S0;
        end

    endcase
    end
    endmodule
```

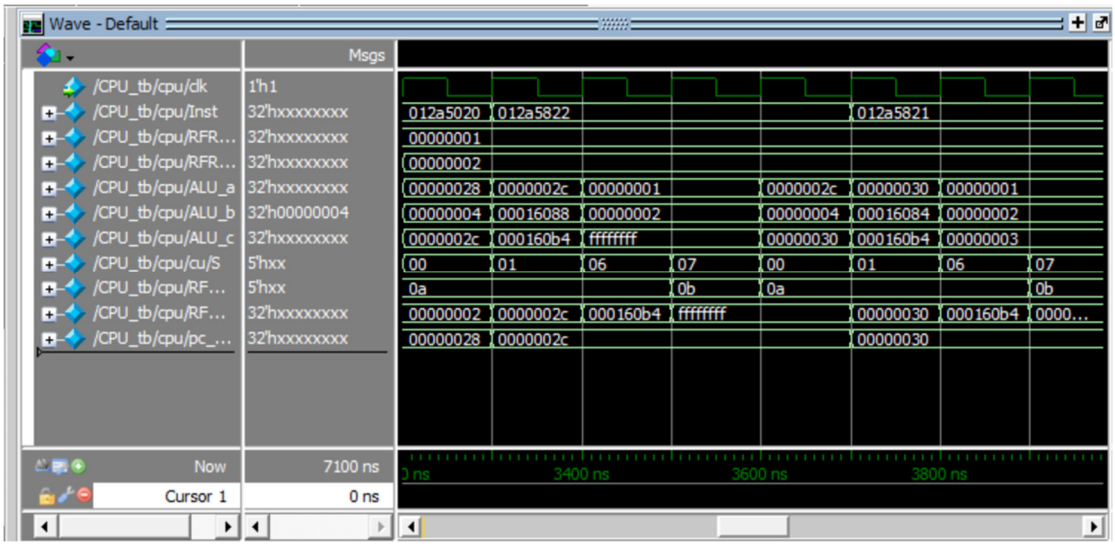
测试指令和单周期的区别是，删去了部分未实现的转移指令，以及在存储首部添加了 4 字节的数据用于 lw 测试，之后为指令的编码，演示结果如下，由于输出太多只展示部分。

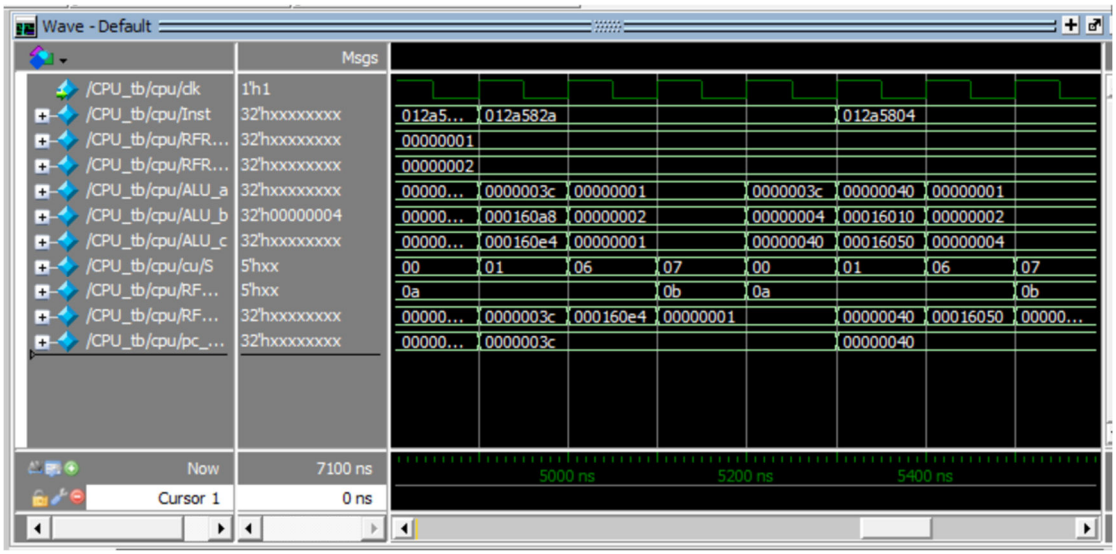


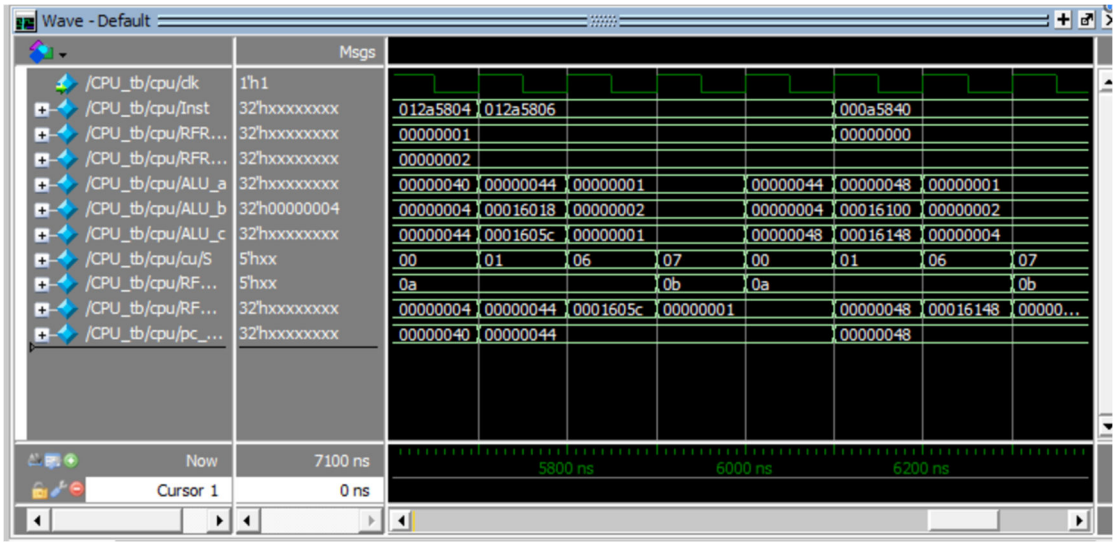

Wave - Default
Msgs
/CPU_tb/cpu/clk
/CPU_tb/cpu/Inst
/CPU_tb/cpu/RFR...
/CPU_tb/cpu/RFR...
/CPU_tb/cpu/ALU_a
/CPU_tb/cpu/ALU_b
/CPU_tb/cpu/ALU_c
/CPU_tb/cpu/cu/S
/CPU_tb/cpu/RF...
/CPU_tb/cpu/RF...
/CPU_tb/cpu/pc_...
1'h1
32'hxxxxxxxx
5'hxx
Now
7100 ns
Cursor 1
0 ns
1000 ns
1200 ns
1400 ns


Wave - Default
Msgs
/CPU_tb/cpu/clk
/CPU_tb/cpu/Inst
/CPU_tb/cpu/RFR...
/CPU_tb/cpu/RFR...
/CPU_tb/cpu/ALU_a
/CPU_tb/cpu/ALU_b
/CPU_tb/cpu/ALU_c
/CPU_tb/cpu/cu/S
/CPU_tb/cpu/RF...
/CPU_tb/cpu/RF...
/CPU_tb/cpu/pc_...
1'h1
32'hxxxxxxxx
5'hxx
Now
7100 ns
Cursor 1
0 ns
1800 ns
2000 ns
2200 ns


Wave - Default
Msgs
/CPU_tb/cpu/clk
/CPU_tb/cpu/Inst
/CPU_tb/cpu/RFR...
/CPU_tb/cpu/RFR...
/CPU_tb/cpu/ALU_a
/CPU_tb/cpu/ALU_b
/CPU_tb/cpu/ALU_c
/CPU_tb/cpu/cu/S
/CPU_tb/cpu/RF...
/CPU_tb/cpu/RF...
/CPU_tb/cpu/pc_...
1'h1
32'hxxxxxxxx
5'hxx
Now
7100 ns
Cursor 1
0 ns
2600 ns
2800 ns
3000 ns


Wave - Default
Msgs
/CPU_tb/cpu/clk
/CPU_tb/cpu/Inst
/CPU_tb/cpu/RFR...
/CPU_tb/cpu/RFR...
/CPU_tb/cpu/ALU_a
/CPU_tb/cpu/ALU_b
/CPU_tb/cpu/ALU_c
/CPU_tb/cpu/cu/S
/CPU_tb/cpu/RF...
/CPU_tb/cpu/RF...
/CPU_tb/cpu/pc_...
1'h1
32'hxxxxxxxx
5'hxx
012a5020 012a5822 012a5821
00000001
00000002
00000028 0000002c 00000001 0000002c 00000030 00000001
00000004 00016088 00000002 00000004 00016084 00000002
0000002c 000160b4 ffffffff 00000030 000160b4 00000003
00 01 06 07 00 01 06 07
0a 0b 0a 0b
00000002 0000002c 000160b4 ffffffff 00000030 000160b4 0000...
00000028 0000002c 00000030
Now
7100 ns
Cursor 1
0 ns
3400 ns
3600 ns
3800 ns


Wave - Default
Msgs
/CPU_tb/cpu/clk
/CPU_tb/cpu/Inst
/CPU_tb/cpu/RFR...
/CPU_tb/cpu/RFR...
/CPU_tb/cpu/ALU_a
/CPU_tb/cpu/ALU_b
/CPU_tb/cpu/ALU_c
/CPU_tb/cpu/cu/S
/CPU_tb/cpu/RF...
/CPU_tb/cpu/RF...
/CPU_tb/cpu/pc_...
1'h1
32'hxxxxxxxx
5'hxx
012a5... 012a582a 012a5804
00000001
00000002
00000... 0000003c 00000001 0000003c 00000040 00000001
00000... 000160a8 00000002 00000004 00016010 00000002
00000... 000160e4 00000001 00000040 00016050 00000004
00 01 06 07 00 01 06 07
0a 0b 0a 0b
00000... 0000003c 000160e4 00000001 00000040 00016050 00000...
00000... 0000003c 00000040
Now
7100 ns
Cursor 1
0 ns
5000 ns
5200 ns
5400 ns


Wave - Default
Msgs
/CPU_tb/cpu/clk
/CPU_tb/cpu/Inst
/CPU_tb/cpu/RFR...
/CPU_tb/cpu/RFR...
/CPU_tb/cpu/ALU_a
/CPU_tb/cpu/ALU_b
/CPU_tb/cpu/ALU_c
/CPU_tb/cpu/cu/S
/CPU_tb/cpu/RF...
/CPU_tb/cpu/RF...
/CPU_tb/cpu/pc_...
1'h1
32'hxxxxxxxx
5'hxx
012a5804 012a5806 000a5840
00000001 00000000
00000002
00000040 00000044 00000001 00000044 00000048 00000001
00000004 00016018 00000002 00000004 00016100 00000002
00000044 0001605c 00000001 00000048 00016148 00000004
00 01 06 07 00 01 06 07
0a 0b 0a 0b
00000004 00000044 0001605c 00000001 00000048 00016148 00000...
00000040 00000044 00000048
Now
7100 ns
Cursor 1
0 ns
5800 ns
6000 ns
6200 ns