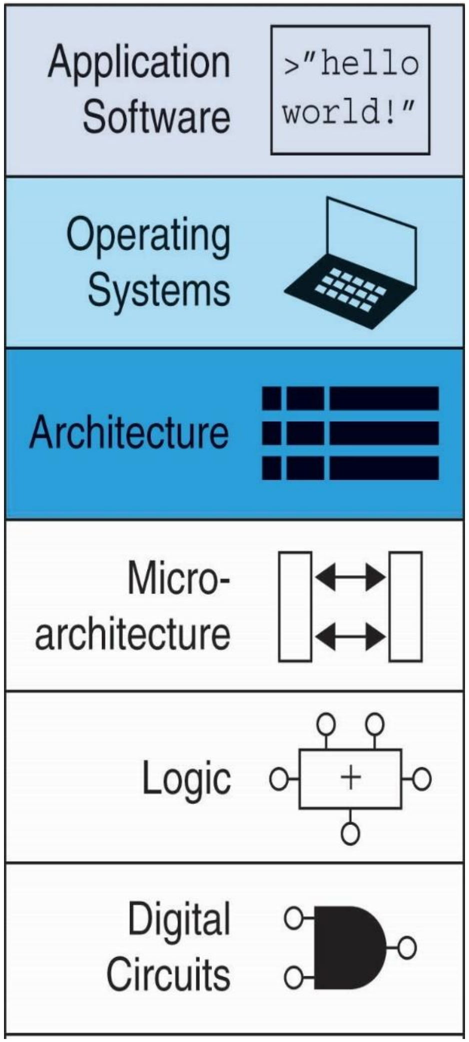
计算机体系结构实验

1. MIPS微处理器原理

- 体系结构（MIPS汇编语言）
- 微体系结构（单周期处理器）

微体系结构

- 针对同一**体系结构**有不同的**微体系结构**设计。
- 将寄存器、存储器、ALU、有限状态机、其他逻辑模块组合在一起，实现一种**体系结构**。



体系结构：程序员所见到的计算机。MIPS、X86… 未定义底层的硬件实现。

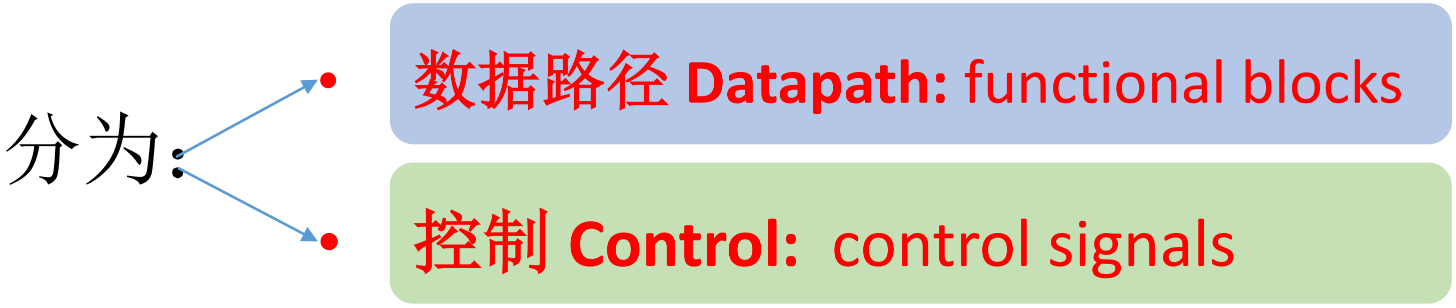
指令集（汇编语言）
寄存器 ＋ PC

微体系结构：由硬件实现一种**体系结构**。

3种微体系结构： 在性能、成本、复杂度之间折中

- **单周期:** 在一个周期中执行一条完整的指令.
- **多周期:** 利用多个较短的周期执行一条指令.
- **流水线:** Each instruction broken up into series of steps & multiple instructions execute at once.

微体系结构 (32位)



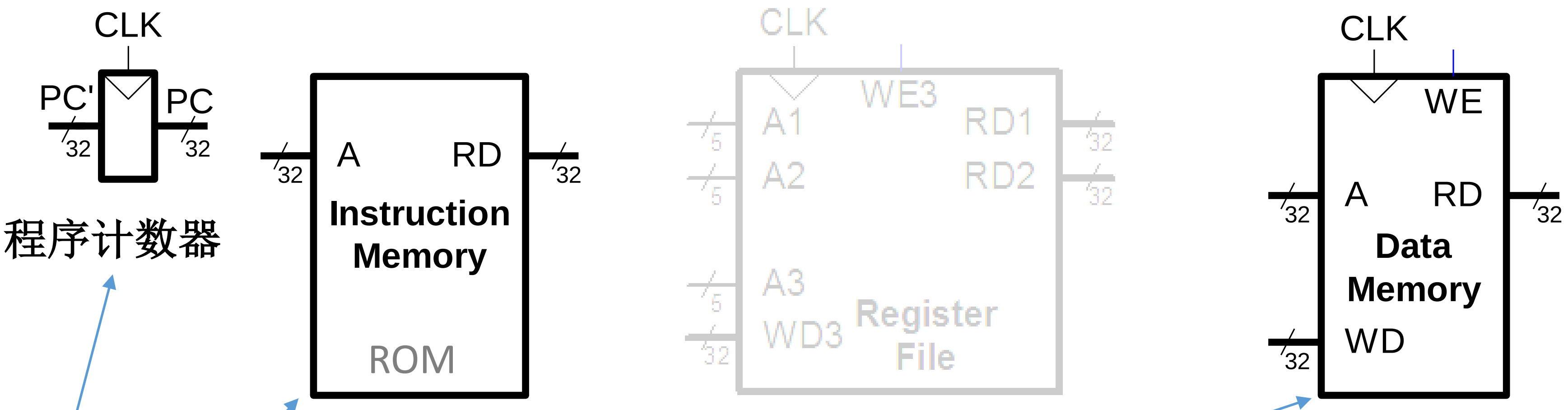
分为:

- 数据路径 **Datapath:** functional blocks
- 控制 **Control:** control signals

Consider subset of MIPS instructions:

- R-type: `and`, `or`, `add`, `sub`, `slt` (5)
- Memory: `lw`, `sw` (2)
- Branch: `beq` (1)

状态元素



- 程序计数器(**P**rogram **C**ounter)：普通32位寄存器。

  输出PC：当前指令地址。输入PC’：下一条指令地址。

- 指令存储器(**IM**)：有一个读端口的ROM。

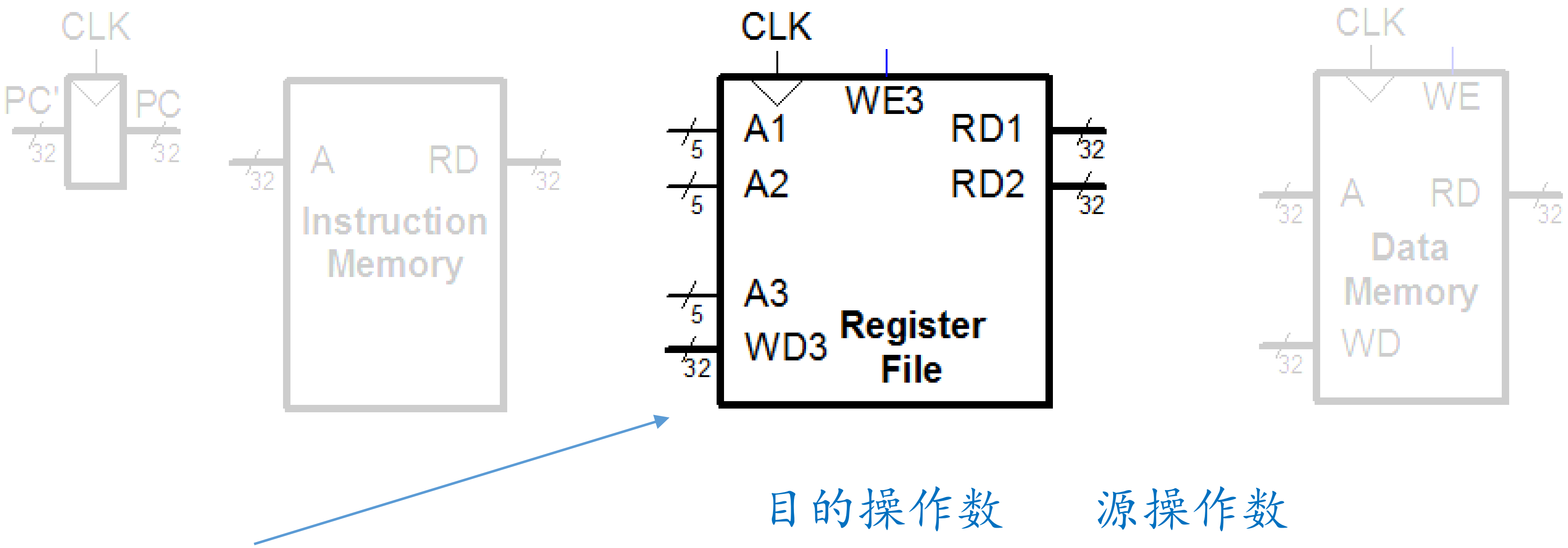
  输出RD：32位指令。　　　输入A：32位指令地址。

- 数据存储器(**DM**)：有一个读/写端口的RAM。

  如果写使能WE=0，则从地址A将数据读到输出端RD；

  如果写是能WE=1，在CLK上升沿将输入WD写入地址A。

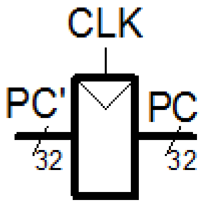
状态元素



- 寄存器文件(**R**egister **F**ile)：1个写端口，2个读端口。
  - 如果写使能WE3=1，则在CLK上升沿将数据WD3写入A3指定的寄存器中。
  - 将A1指定的寄存器数据传送到输出端口RD1;
  - 将A2指定的寄存器数据传送到输出端口RD2。

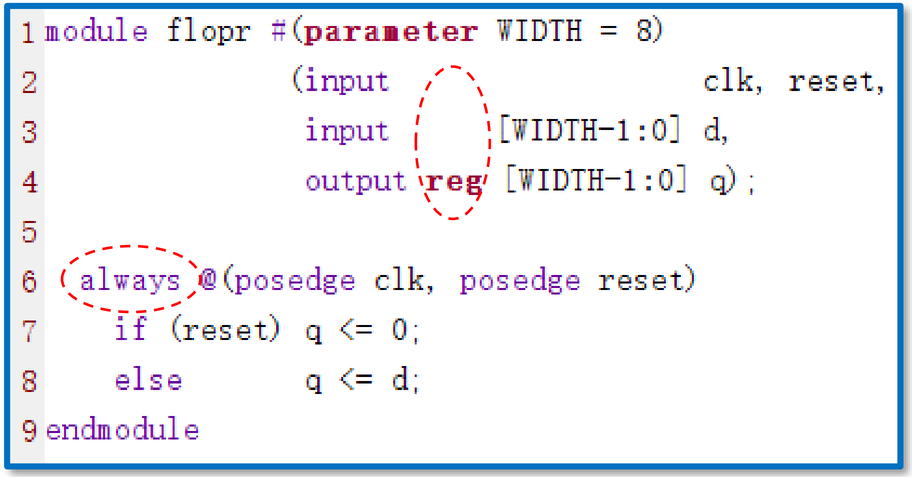
【注】5位地址可以表达32个寄存器。

程序计数器 PC



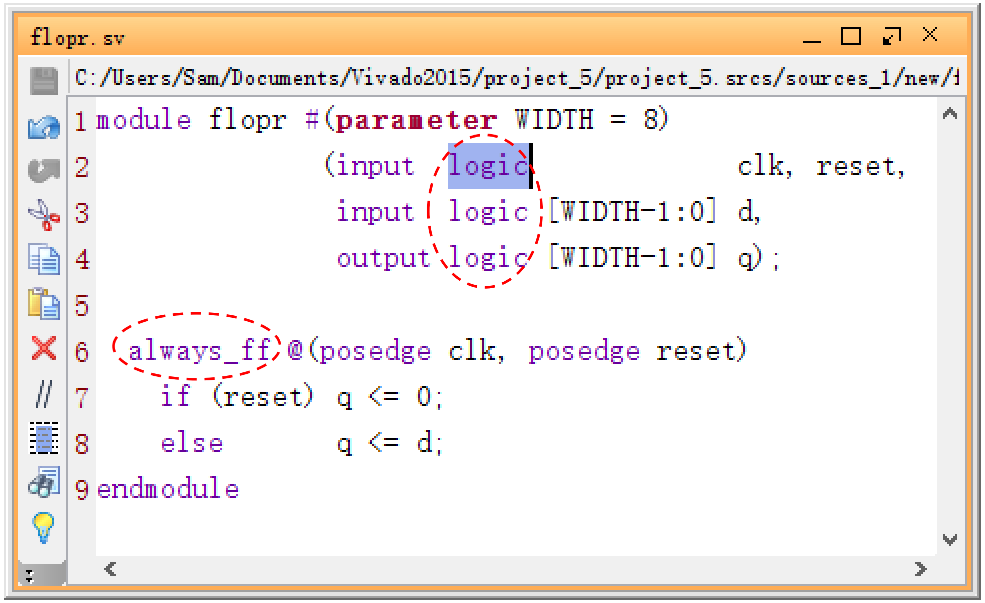
程序计数器
(32位寄存器)

Verilog
版本



```
module flopr #(parameter WIDTH = 8)
              (input              clk, reset,
               input      [WIDTH-1:0] d,
               output reg [WIDTH-1:0] q);

always @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule
```

SystemVerilog
版本


flopr.sv
C:/Users/Sam/Documents/Vivado2015/project_5/project_5.srcs/sources_1/new/f

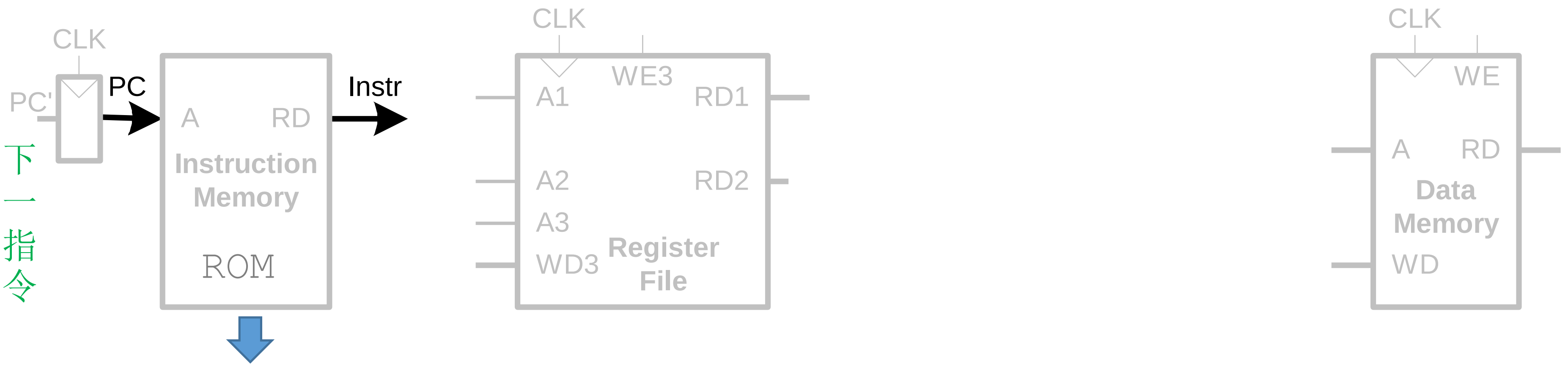
```
module flopr #(parameter WIDTH = 8)
             (input  logic            clk, reset,
              input  logic [WIDTH-1:0] d,
              output logic [WIDTH-1:0] q);

 always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule
```

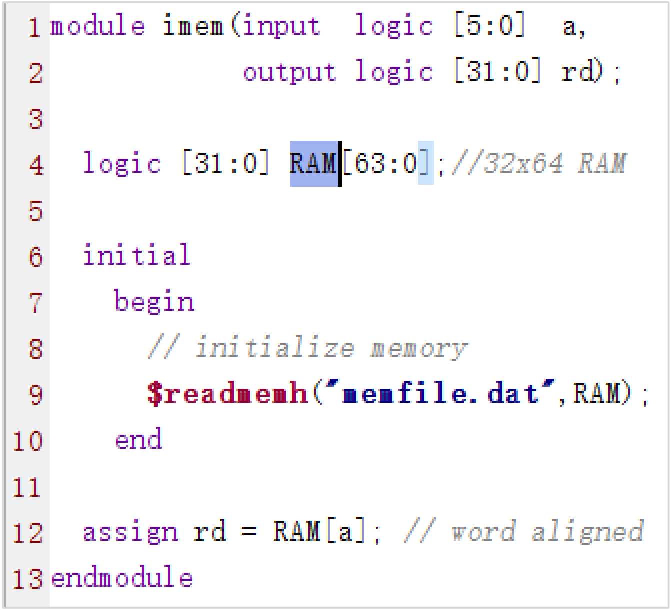
单周期 数据路径

lw rt, imm(rs)

装入字：[rt] = [Address]

STEP 1: 从指令存储器中取出指令





```
module imem(input  logic [5:0]  a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];//32x64 RAM

  initial
    begin
      // initialize memory
      $readmemh("memfile.dat",RAM);
    end

  assign rd = RAM[a]; // word aligned
endmodule
```

Verilog有2个系统任务，从文件中读取数据到存储器。

$readmemb("<数据文件名>",<存储器名>,<起始地址>,<终止地址>);

$readmemh("<数据文件名>",<存储器名>,<起始地址>,<终止地址>);

memfile.dat

```
20020005
2003000C
2067FFF7
00E22025
00642824
00A42820
10A7000A
0064202A
10800001
20050000
00E2202A
00853820
00E23822
AC670044
8C020050
08000011
20020001
AC020054
```

单周期 数据路径-2

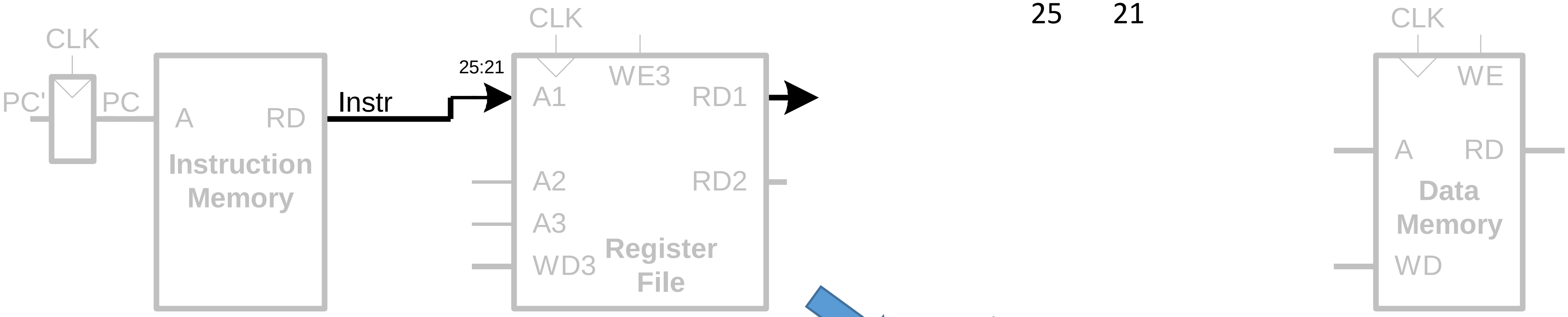
```
lw rt, imm(rs)
```

装入字：[rt] = [Address]

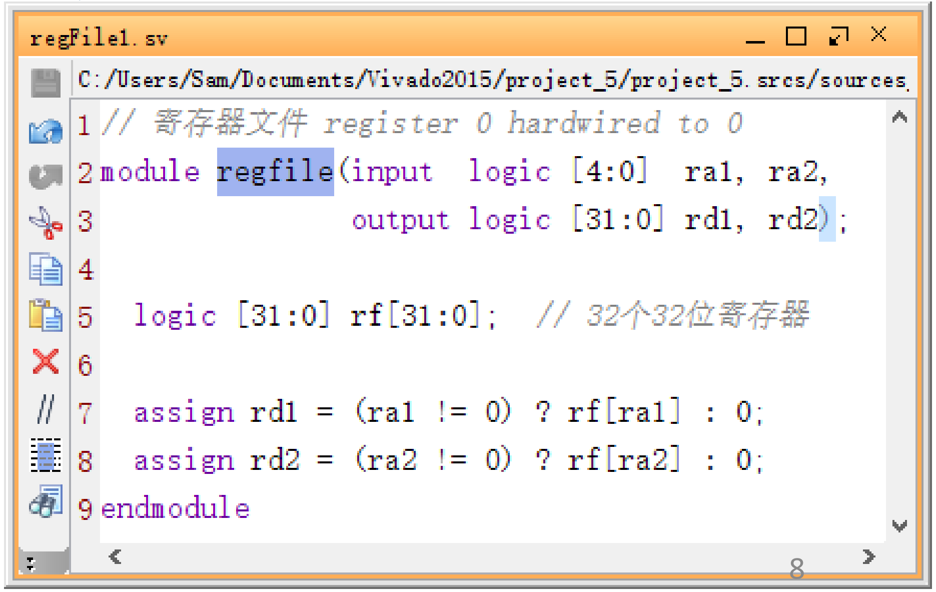
STEP 2: 从寄存器文件中读出源操作数

op (6)	rs (5)	rt (5)	Imm (16)

25 21



版本1：组合电路


regFile1.sv
C:/Users/Sam/Documents/Vivado2015/project_5/project_5.srcs/sources

```
// 寄存器文件 register 0 hardwired to 0
module regfile(input  logic [4:0]  ra1, ra2,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];   // 32个32位寄存器

  assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule
```

- 共有32个32位寄存器
- **需要区分rs、rt的地址和数据**
- 因$0一直输出0，因此当RsAddr、RtAddr为0时，RsData、RtData必须输出0.
- rd只有在写信号有效时用
- 当regWriteEn有效时，数据需要写入regWriteAddr寄存器。

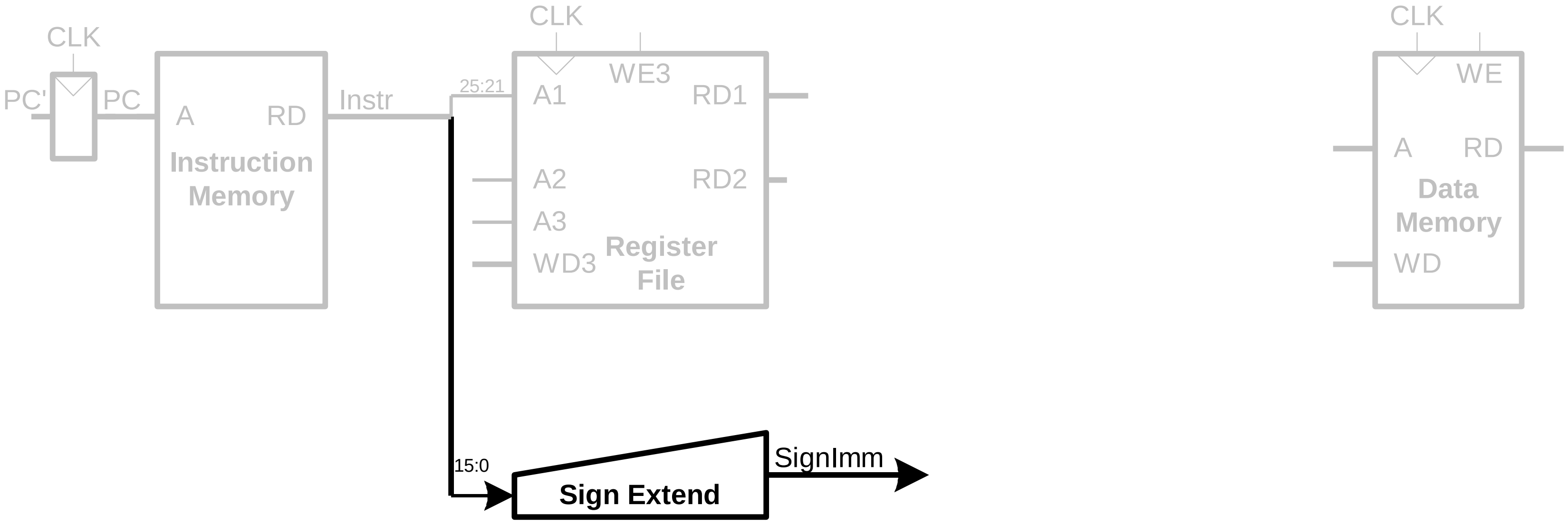
单周期数据路径 -3

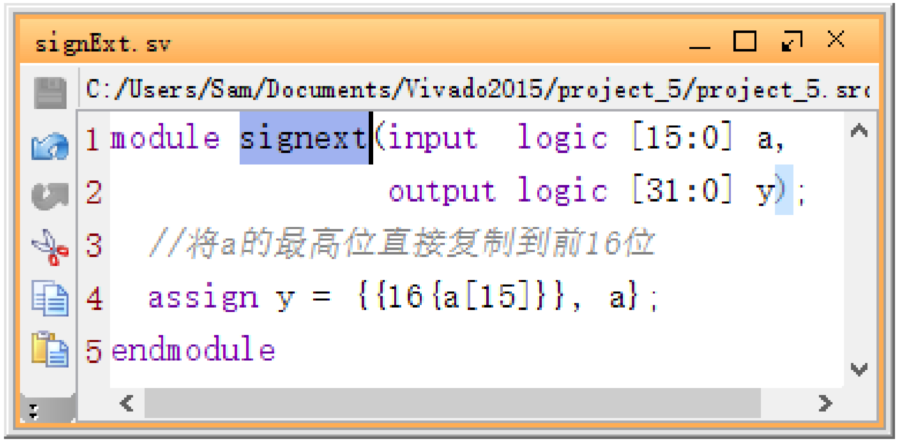
`lw rt, imm(rs)`

op (6)	rs (5)	rt (5)	imm(16)

15 0

3: 符号扩展立即数





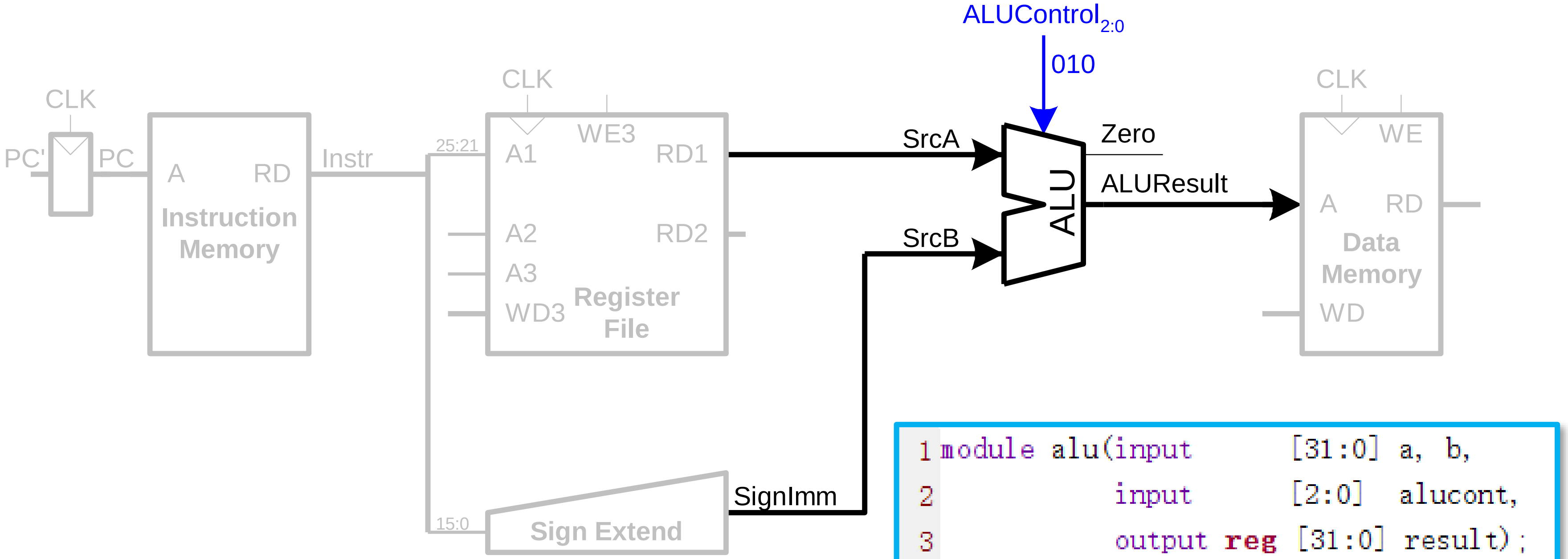
```
module signext(input  logic [15:0] a,
               output logic [31:0] y);
  //将a的最高位直接复制到前16位
  assign y = {{16{a[15]}}, a};
endmodule
```

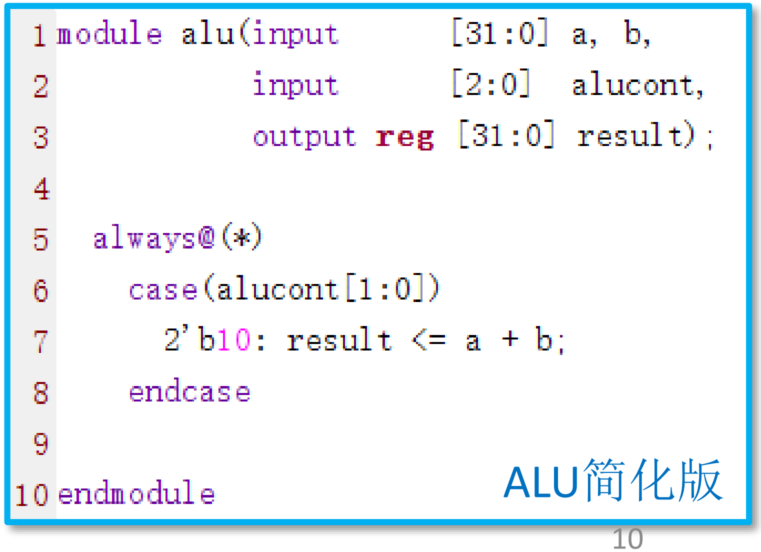
单周期数据路径 -4

lw rt, imm(rs)

op (6)	rs (5)	rt (5)	imm(16)

4: 计算存储器地址





```
module alu(input       [31:0] a, b,
           input       [2:0]  alucont,
           output reg  [31:0] result);

  always@(*)
    case(alucont[1:0])
      2'b10: result <= a + b;
    endcase

endmodule
```

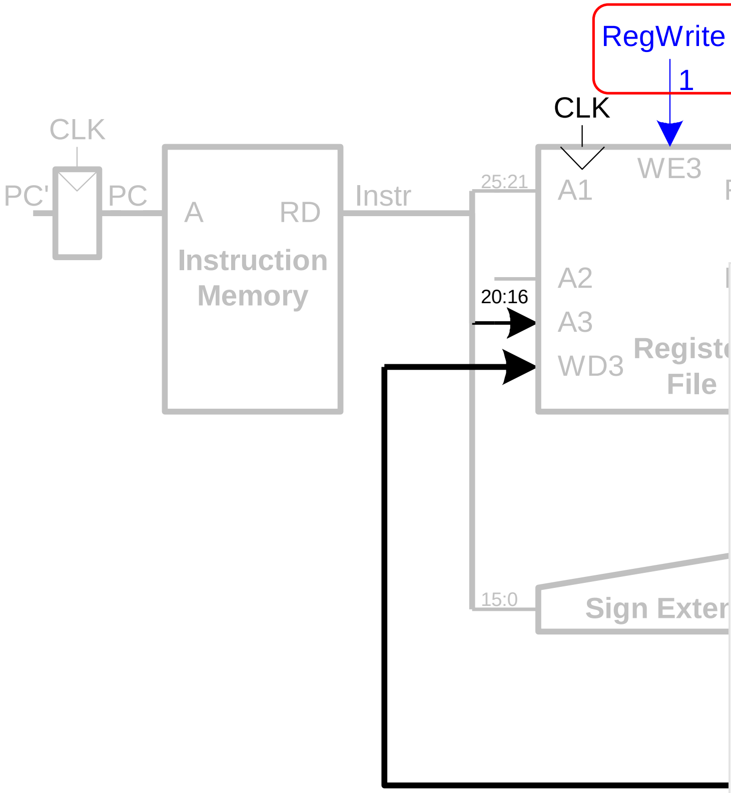
ALU简化版

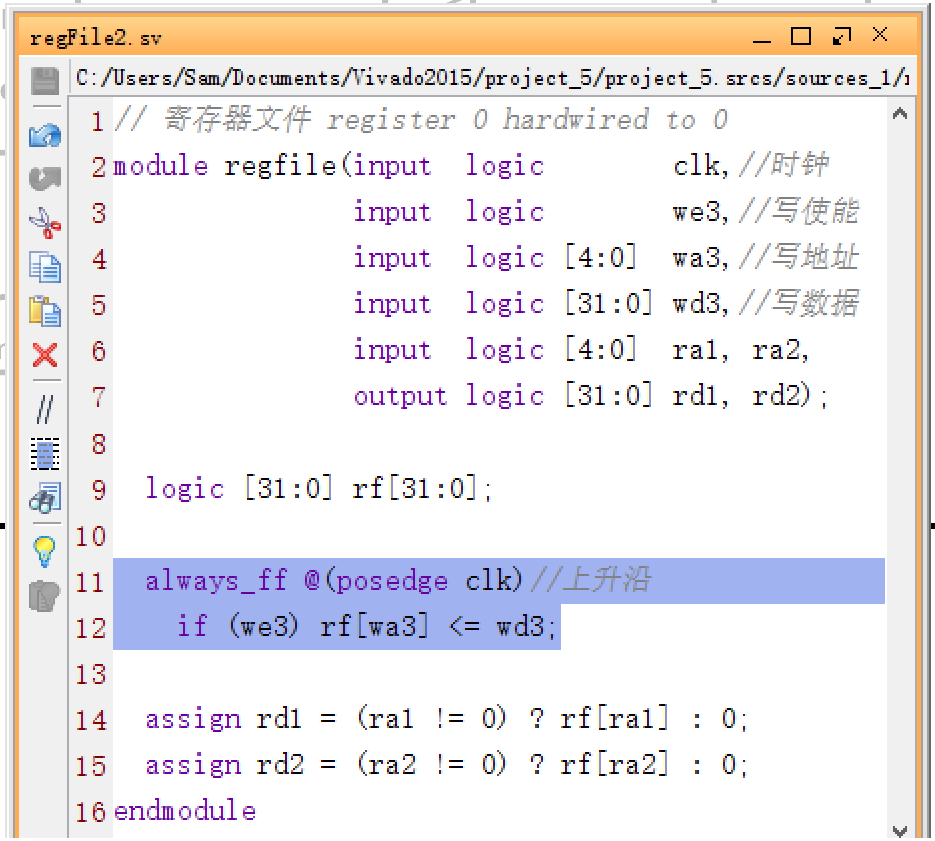
单周期数据路径 -5

lw rt, imm(rs)

op (6)	rs (5)	rt (5)	imm(16)
		20	16

5: 向寄存器文件 写入 数据




regFile2.sv

C:/Users/Sam/Documents/Vivado2015/project_5/project_5.srcs/sources_1/

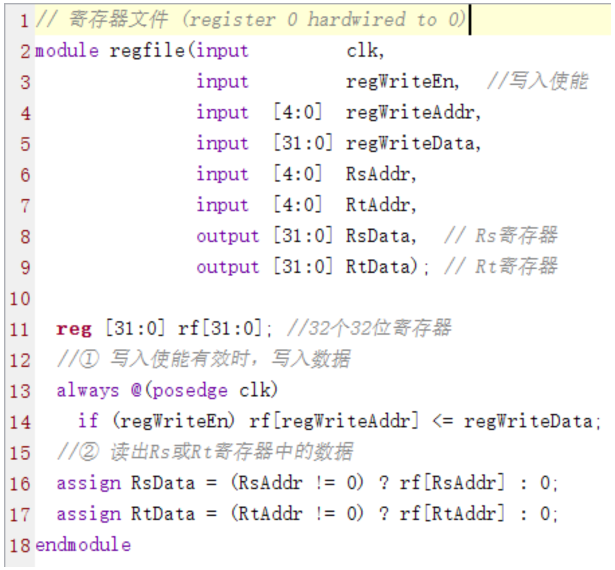
```
// 寄存器文件 register 0 hardwired to 0
module regfile(input  logic        clk, //时钟
               input  logic        we3, //写使能
               input  logic [4:0]  wa3, //写地址
               input  logic [31:0] wd3, //写数据
               input  logic [4:0]  ra1, ra2,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  always_ff @(posedge clk) //上升沿
    if (we3) rf[wa3] <= wd3;

  assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule
```

寄存器文件版本2:

写入操作为同步时序电路,

在时钟上升沿，写使能=1时

单周期 数据路径



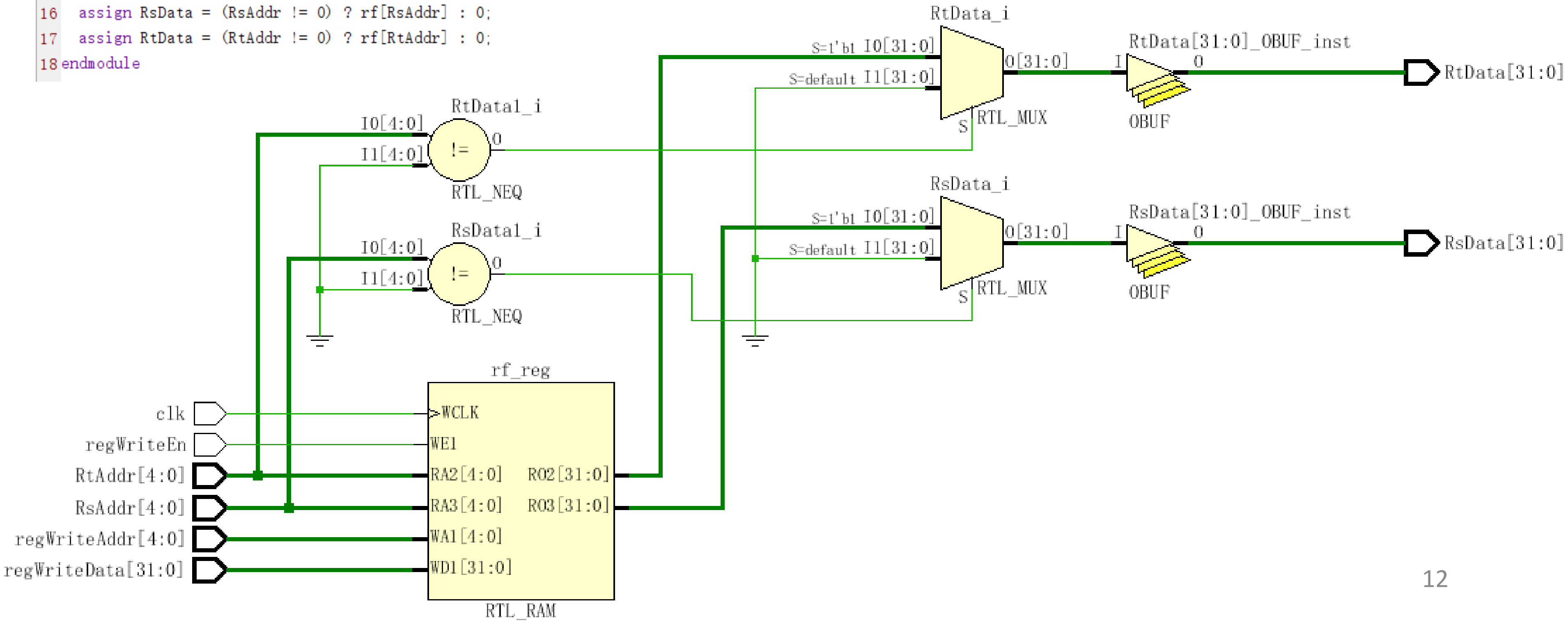
```
// 寄存器文件 (register 0 hardwired to 0)
module regfile(input         clk,
               input         regWriteEn,   //写入使能
               input  [4:0]  regWriteAddr,
               input  [31:0] regWriteData,
               input  [4:0]  RsAddr,
               input  [4:0]  RtAddr,
               output [31:0] RsData,   // Rs寄存器
               output [31:0] RtData);  // Rt寄存器

  reg [31:0] rf[31:0]; //32个32位寄存器
  //① 写入使能有效时，写入数据
  always @(posedge clk)
    if (regWriteEn) rf[regWriteAddr] <= regWriteData;
  //② 读出Rs或Rt寄存器中的数据
  assign RsData = (RsAddr != 0) ? rf[RsAddr] : 0;
  assign RtData = (RtAddr != 0) ? rf[RtAddr] : 0;
endmodule
```

lw rt, imm(rs)

装入字： [rt] = [Address]

op (6)	**rs** (5)	rt (5)	Imm (16)

25 21

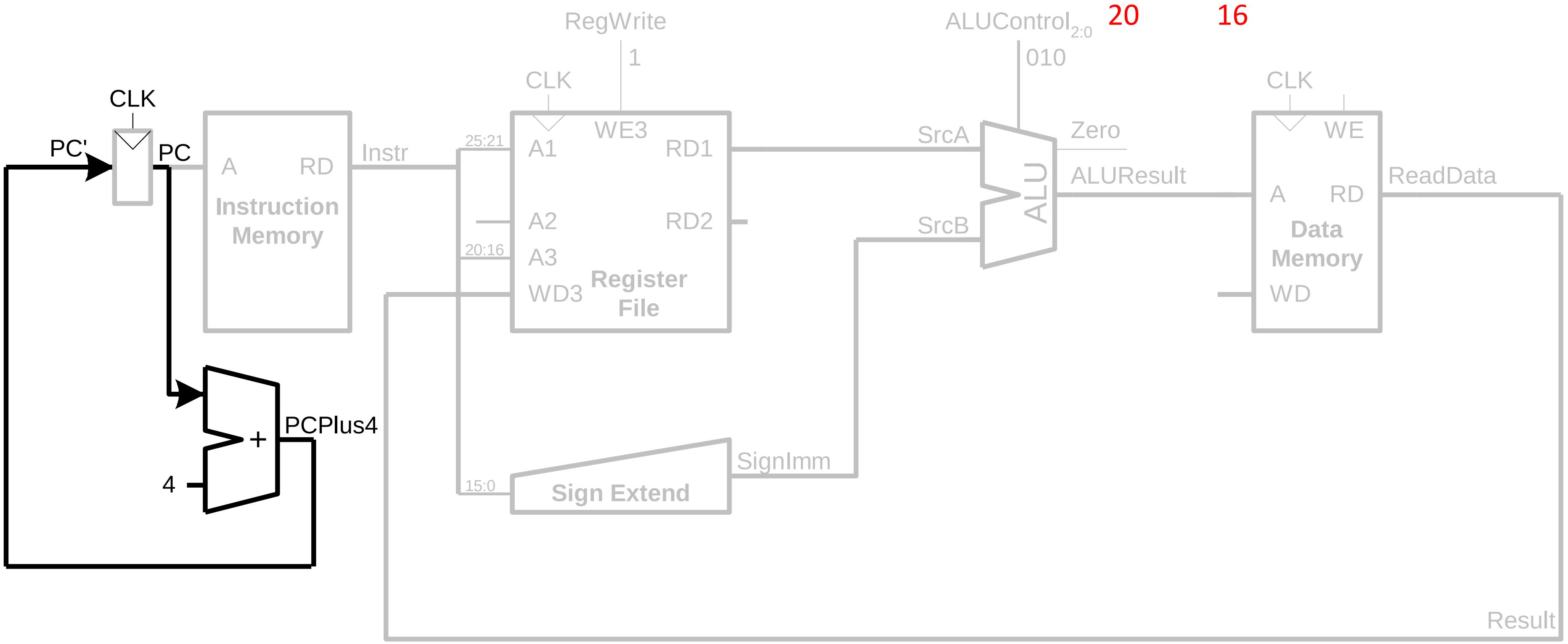


单周期数据路径 -6

lw rt, imm(rs)

6: 确定PC的下一个指令的地址

op (6)	rs (5)	rt (5)	imm(16)



```
module adder(
        input    [31:0] a, b,
        output [31:0] y );

  assign y = a + b;
endmodule
```

MIPS存储器模型是字节(8bits)寻址， 而不是字(32bits)寻址。
每一个数据**字节**都有一个**唯一的地址**，
一个32位的字包含4个8位字节，
即：每一个字地址都是4的倍数。【P185】

单周期数据路径 -7

sw rt, imm(rs)

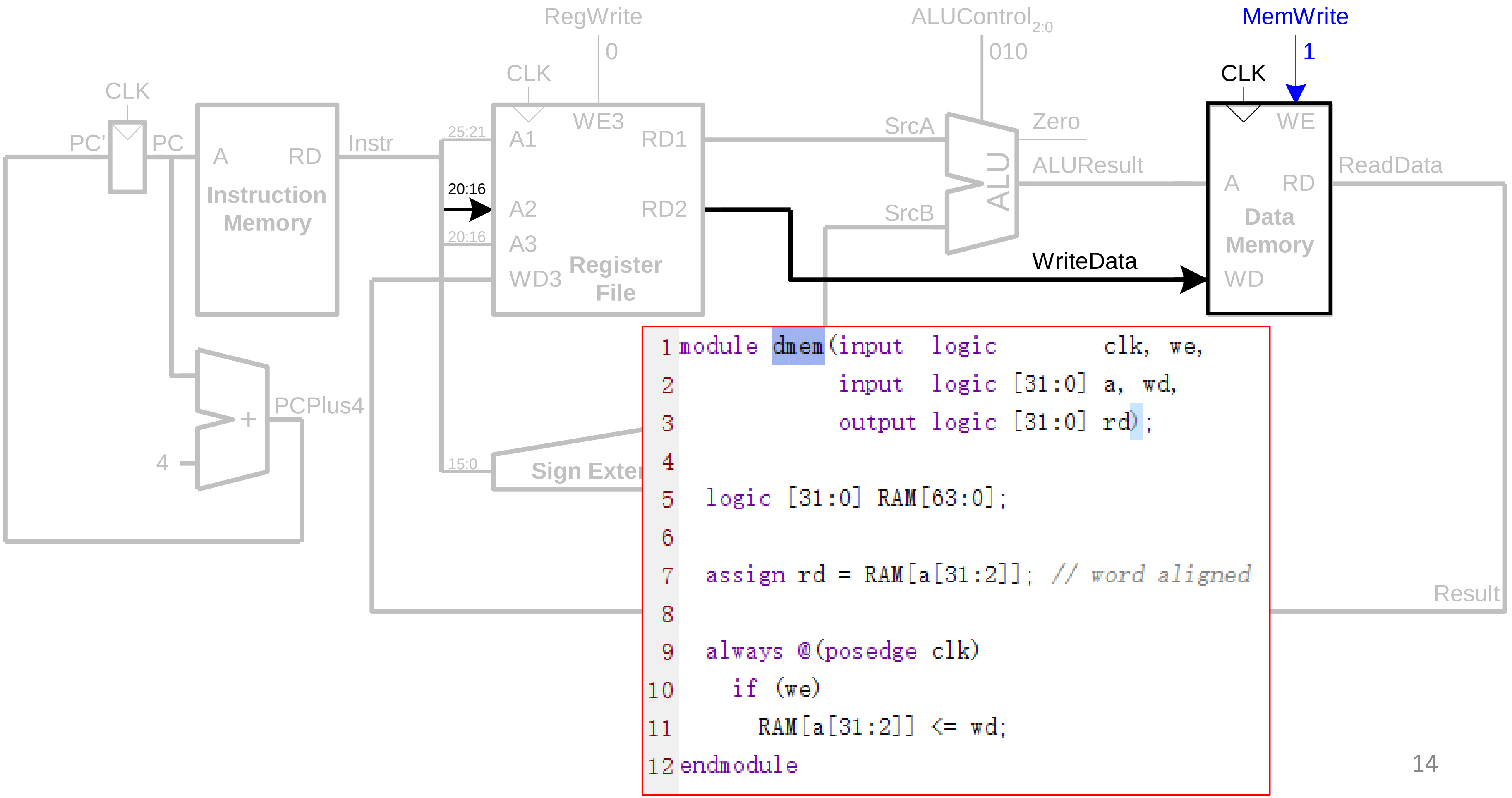
第2条汇编指令sw

存储字：[Address] = [rt]

op (6)	rs (5)	rt (5)	imm(16)

20 16

- 将数据写入数据存储器



```
module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

 logic [31:0] RAM[63:0];

 assign rd = RAM[a[31:2]]; // word aligned

 always @(posedge clk)
   if (we)
     RAM[a[31:2]] <= wd;
endmodule
```

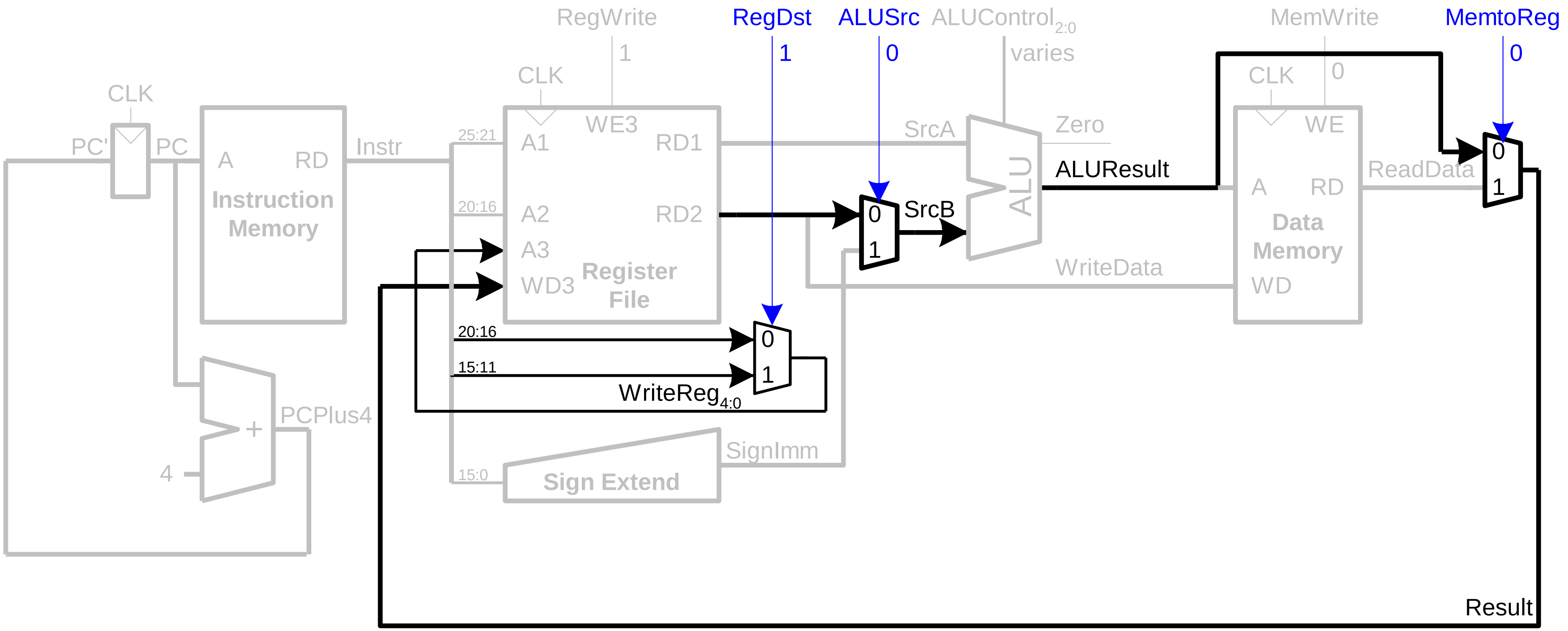
单周期数据路径 -8

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

15 11

- `add`, `sub`, `and`, `or`, `slt`
- Read from `rs` and `rt`
- Write *ALUResult* to `rd` (instead of `rt`)

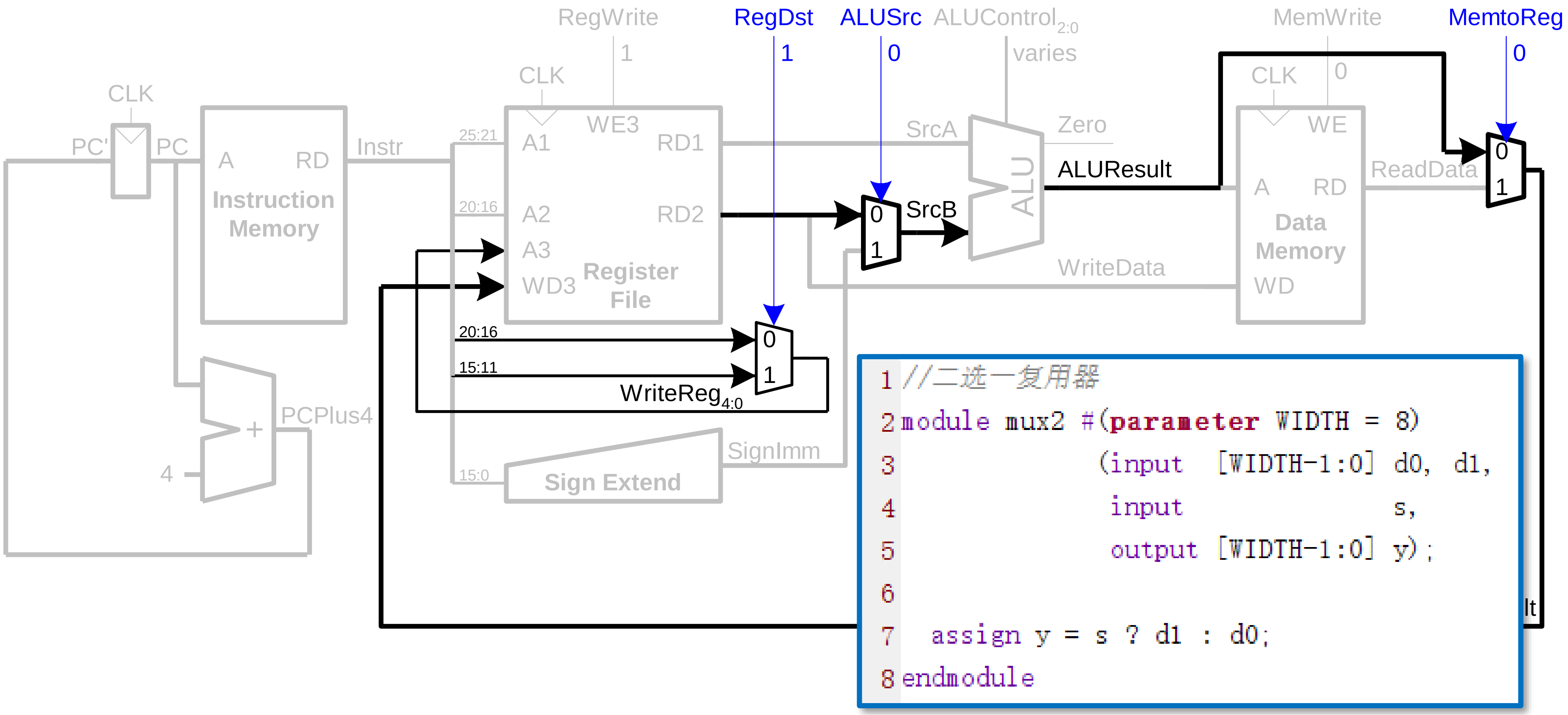


单周期数据路径 -8

- **add**, **sub**, **and**, **or**, **slt**
- Read from `rs` and `rt`
- Write *ALUResult* to `rd` (instead of `rt`)

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits



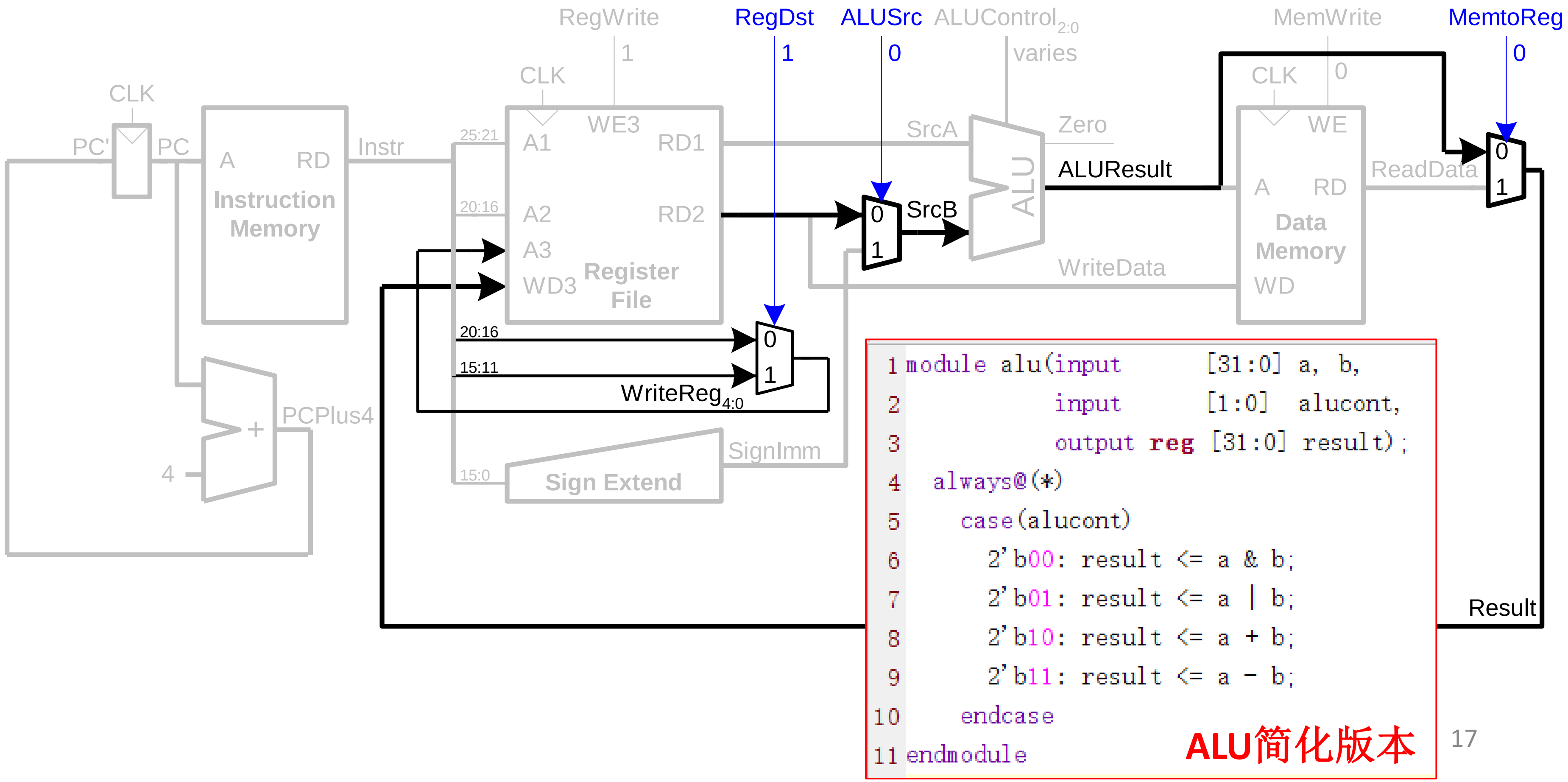
```
//二选一复用器
module mux2 #(parameter WIDTH = 8)
            (input  [WIDTH-1:0] d0, d1,
             input              s,
             output [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

单周期数据路径 -8

- **add**, **sub**, **and**, **or**, **slt**
- Read from `rs` and `rt`
- Write *ALUResult* to `rd` (instead of `rt`)

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits



```
module alu(input      [31:0] a, b,
           input      [1:0]  alucont,
           output reg [31:0] result);
  always@(*)
    case(alucont)
      2'b00: result <= a & b;
      2'b01: result <= a | b;
      2'b10: result <= a + b;
      2'b11: result <= a - b;
    endcase
endmodule
```

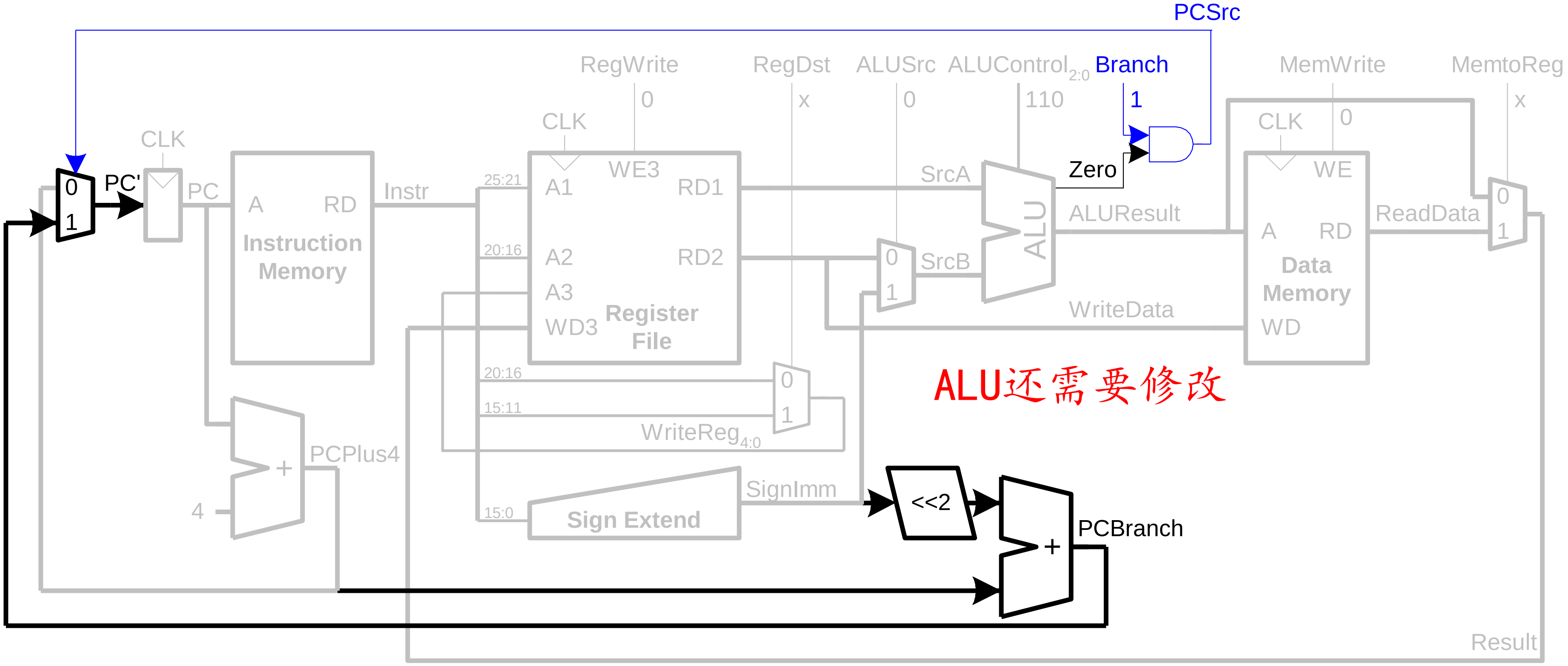
ALU简化版本

单周期数据路径 -9

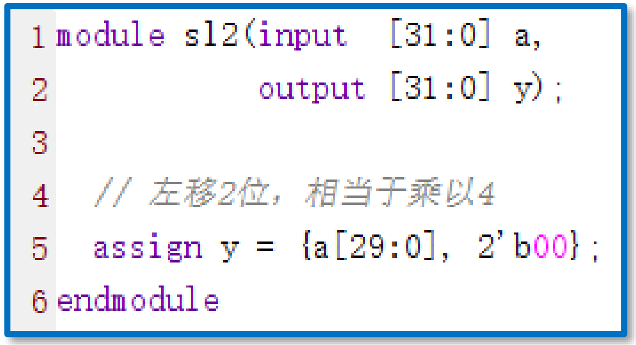
beq rs, rt, label

rs 、rt相等则转移

If ([rs]==[rt]) PC’=PC+4+(SignImm<<2)

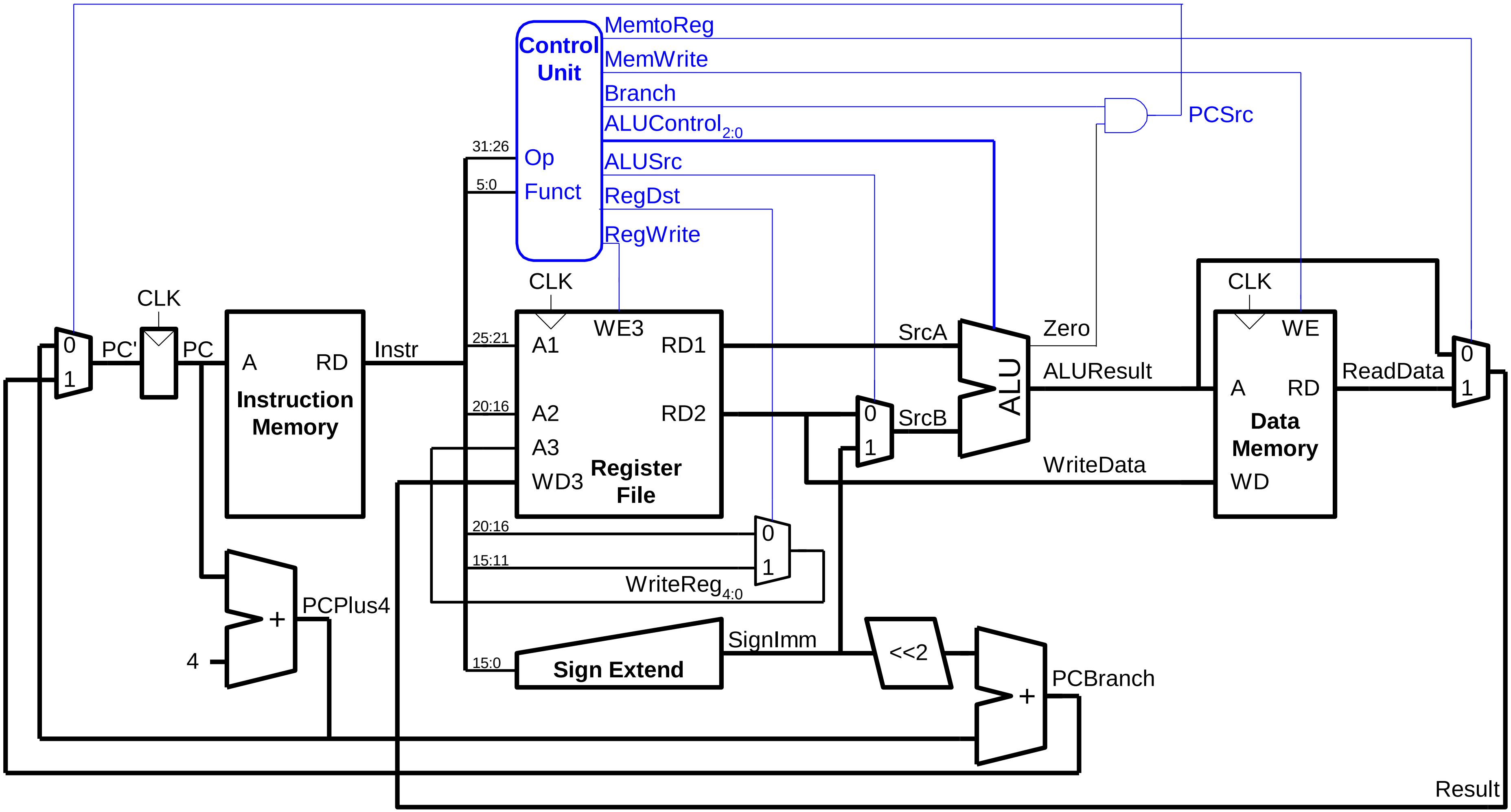


ALU还需要修改

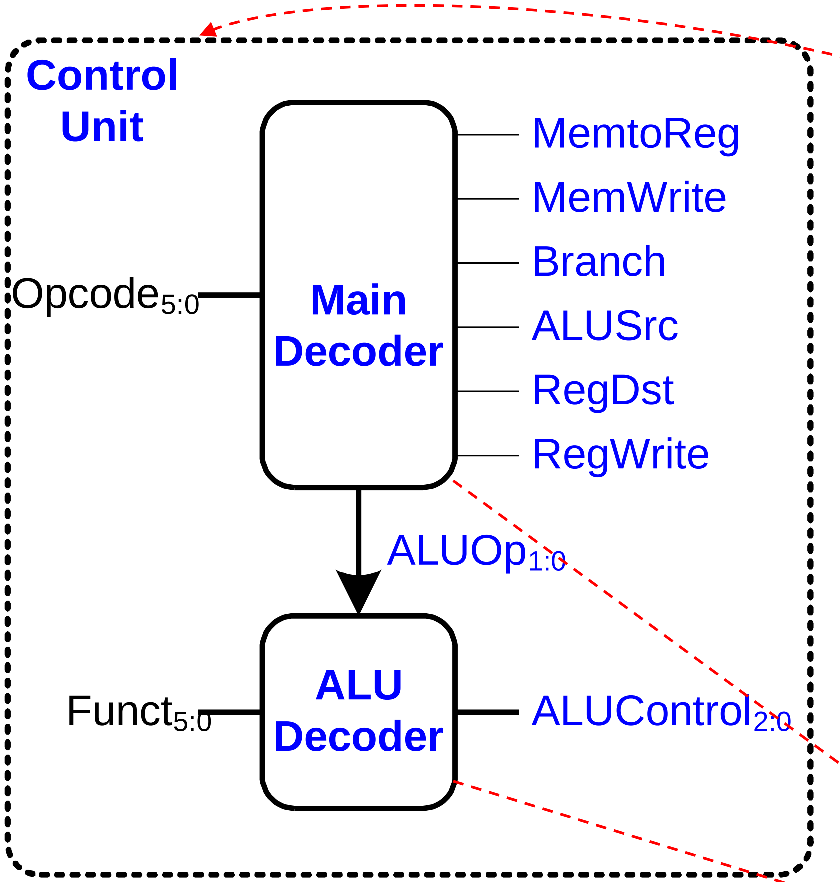


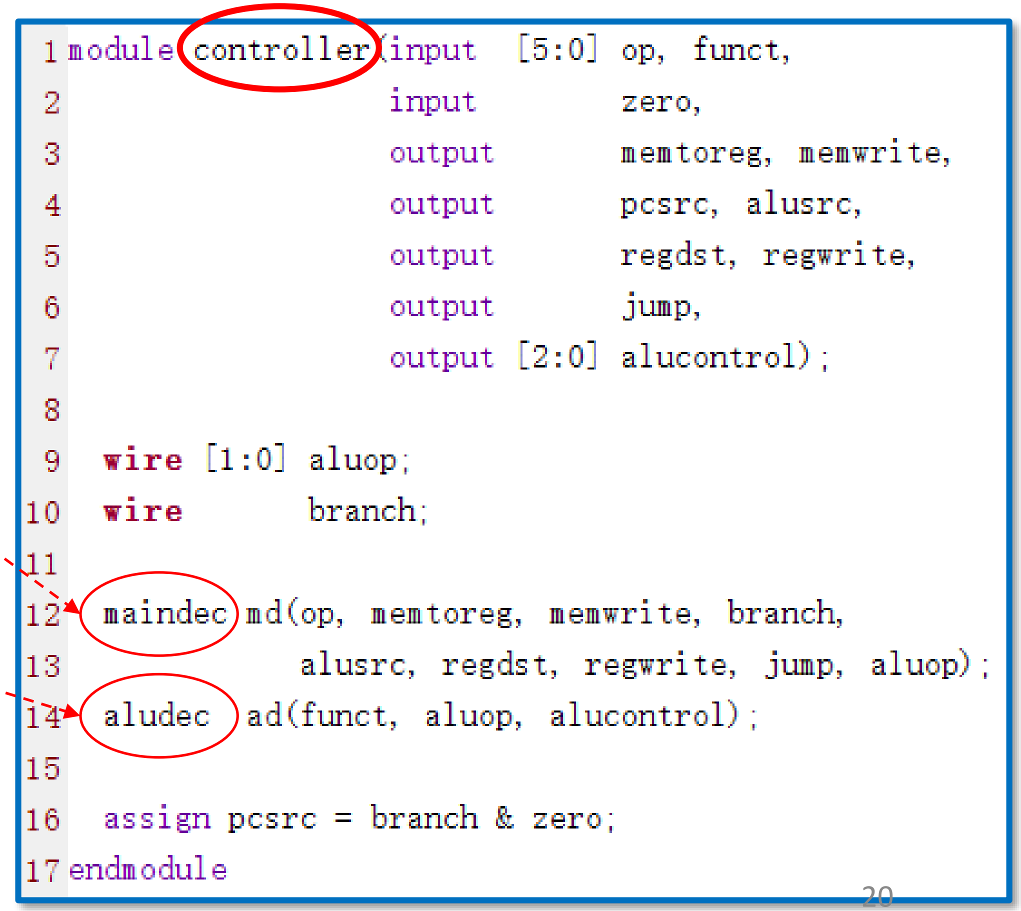
```
module sl2(input  [31:0] a,
           output [31:0] y);

  // 左移2位，相当于乘以4
  assign y = {a[29:0], 2'b00};
endmodule
```

单周期 控制


MemtoReg
Control Unit
MemWrite
Branch
PCSrc
ALUControl$_{2:0}$
31:26
Op
5:0
Funct
ALUSrc
RegDst
RegWrite
CLK
CLK
CLK
0
1
PC'
PC
A
RD
Instruction Memory
Instr
25:21
WE3
A1
RD1
SrcA
Zero
ALU
ALUResult
WE
A
RD
Data Memory
WD
ReadData
0
1
20:16
A2
RD2
0
1
SrcB
A3
WD3
Register File
WriteData
20:16
0
15:11
1
WriteReg$_{4:0}$
PCPlus4
+
4
SignImm
<<2
15:0
Sign Extend
+
PCBranch
Result

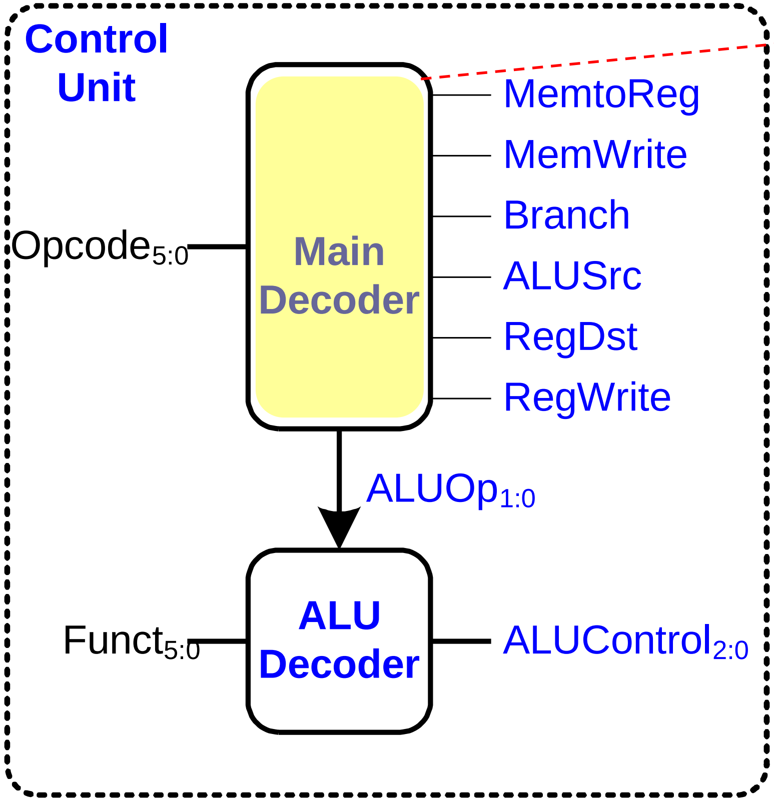
单周期 控制单元 (ALU译码器)

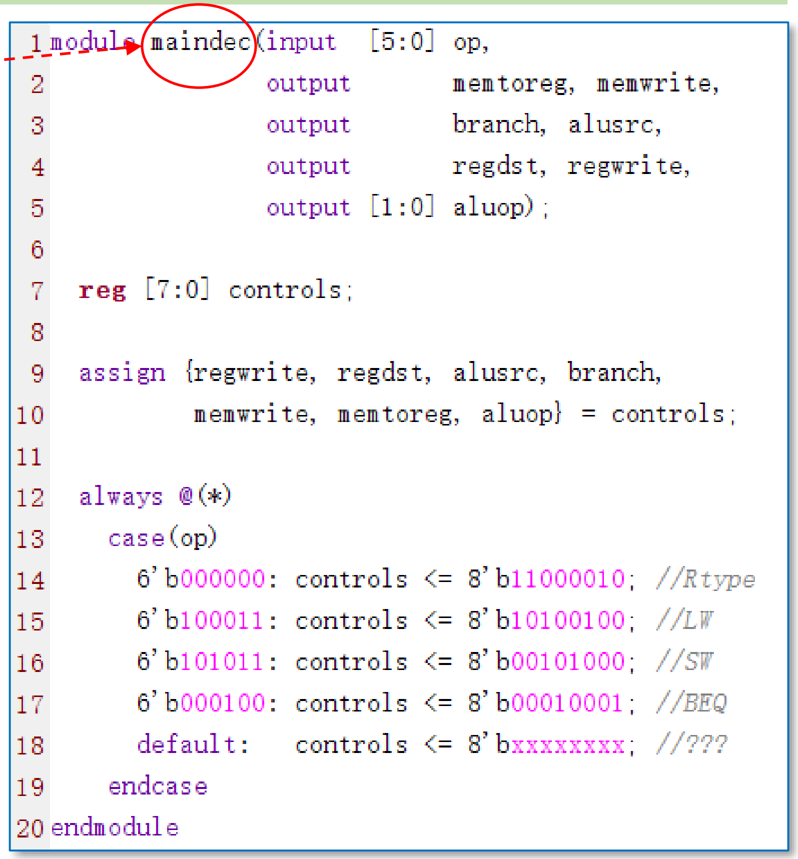




```
module controller(input  [5:0] op, funct,
                  input        zero,
                  output       memtoreg, memwrite,
                  output       pcsrc, alusrc,
                  output       regdst, regwrite,
                  output       jump,
                  output [2:0] alucontrol);

  wire [1:0] aluop;
  wire       branch;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump, aluop);
  aludec  ad(funct, aluop, alucontrol);

  assign pcsrc = branch & zero;
endmodule
```

单周期 控制单元 (主译码器)



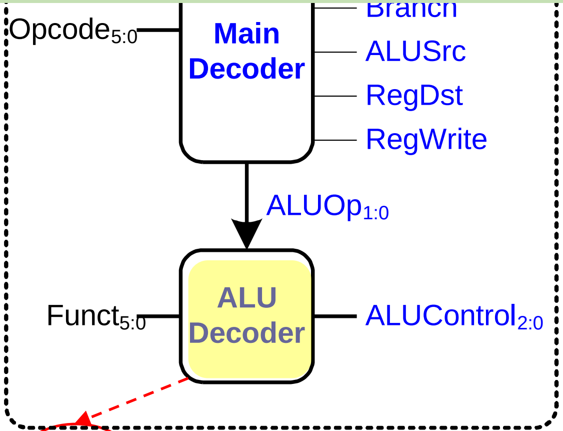


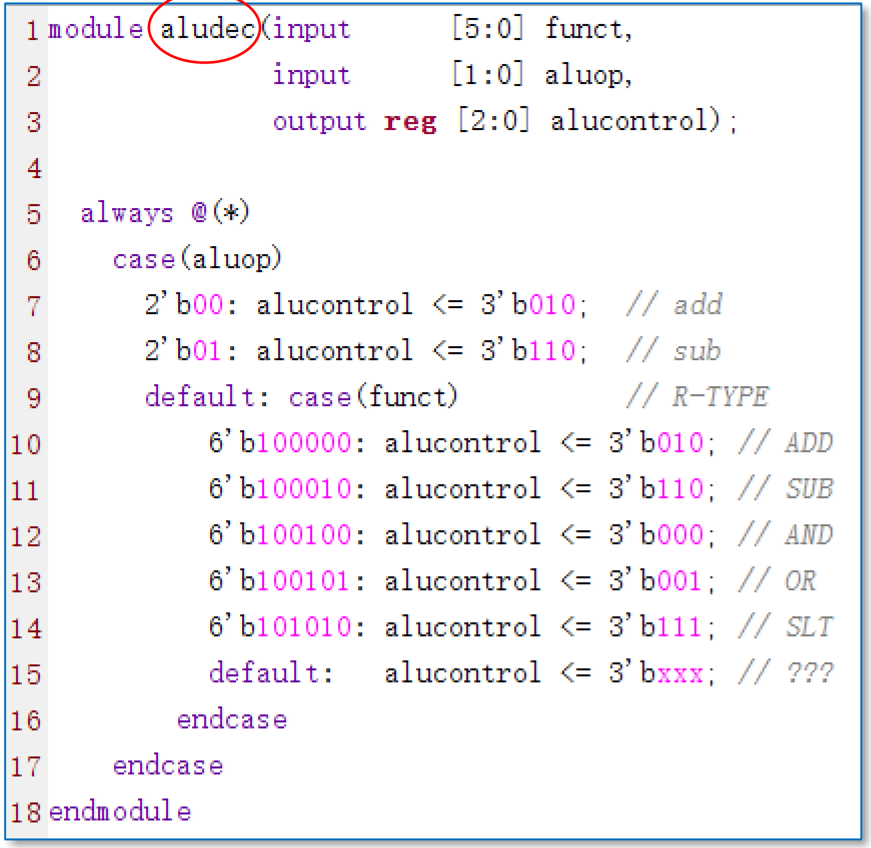
```
module maindec(input   [5:0]  op,
                   output         memtoreg, memwrite,
                   output         branch, alusrc,
                   output         regdst, regwrite,
                   output  [1:0]  aluop);

  reg [7:0] controls;

  assign {regwrite, regdst, alusrc, branch,
          memwrite, memtoreg, aluop} = controls;

  always @(*)
    case(op)
      6'b000000: controls <= 8'b11000010; //Rtype
      6'b100011: controls <= 8'b10100100; //LW
      6'b101011: controls <= 8'b00101000; //SW
      6'b000100: controls <= 8'b00010001; //BEQ
      default:   controls <= 8'bxxxxxxxx; //???
    endcase
endmodule
```

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	0	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

减法

单周期 控制单元 (ALU译码器)





```verilog
module aludec(input       [5:0] funct,
              input       [1:0] aluop,
              output reg  [2:0] alucontrol);

 always @(*)
   case(aluop)
     2'b00: alucontrol <= 3'b010;  // add
     2'b01: alucontrol <= 3'b110;  // sub
     default: case(funct)          // R-TYPE
         6'b100000: alucontrol <= 3'b010; // ADD
         6'b100010: alucontrol <= 3'b110; // SUB
         6'b100100: alucontrol <= 3'b000; // AND
         6'b100101: alucontrol <= 3'b001; // OR
         6'b101010: alucontrol <= 3'b111; // SLT
         default:   alucontrol <= 3'bxxx; // ???
       endcase
   endcase
endmodule
```

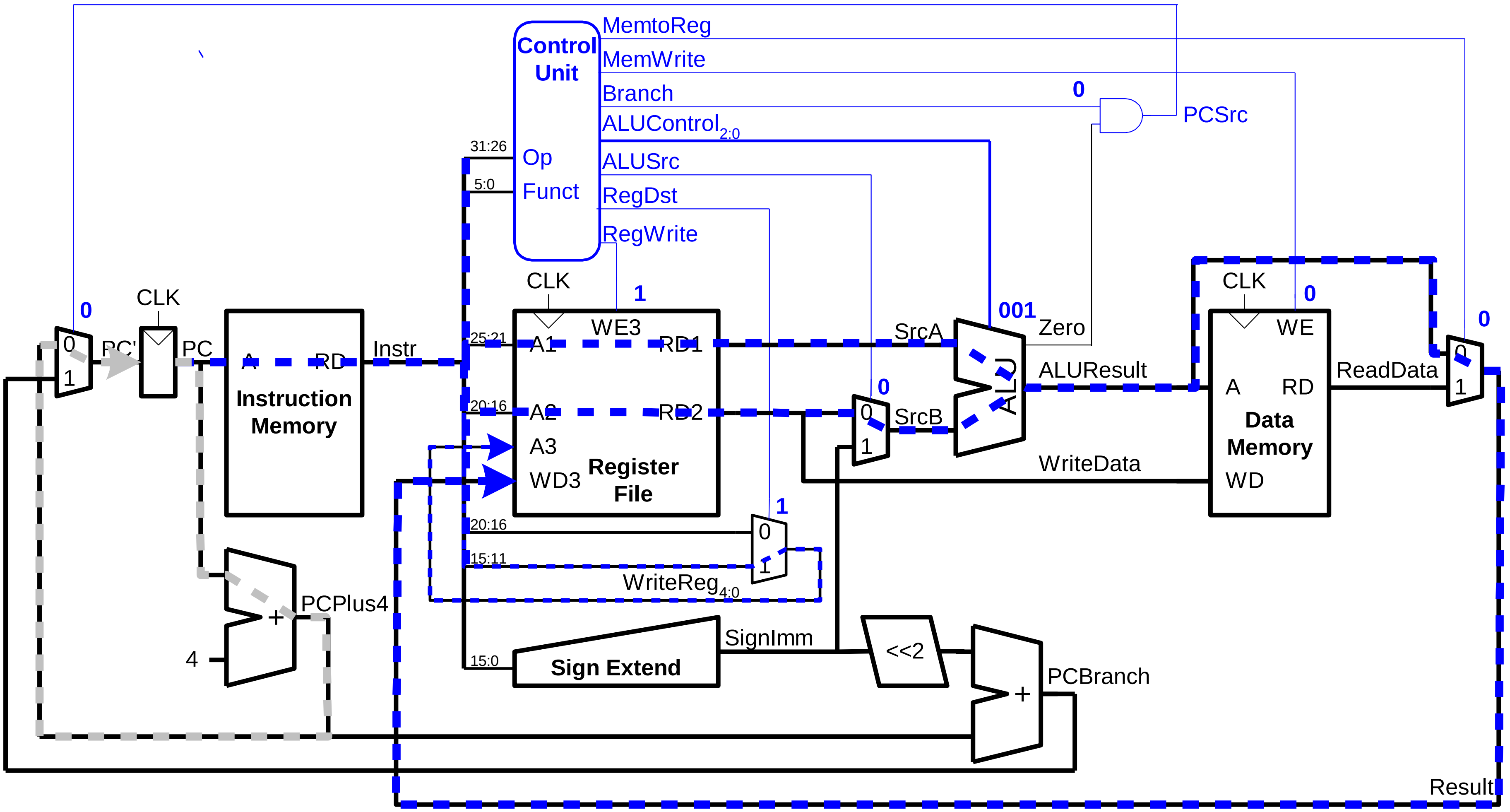
$ALUOp_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

$ALUOp_{1:0}$	$Funct_{5:0}$	$ALUControl_{2:0}$
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (add)	010 (Add)
1X	100010 (sub)	110 (Subtract)
1X	100100 (and)	000 (And)
1X	100101 (or)	001 (Or)
1X	101010 (slt)	111 (SLT)

例7.1 Single-Cycle Datapath: or

```
or rd, rs, rt
```

```
[rd]= [rs] | [rt]
```

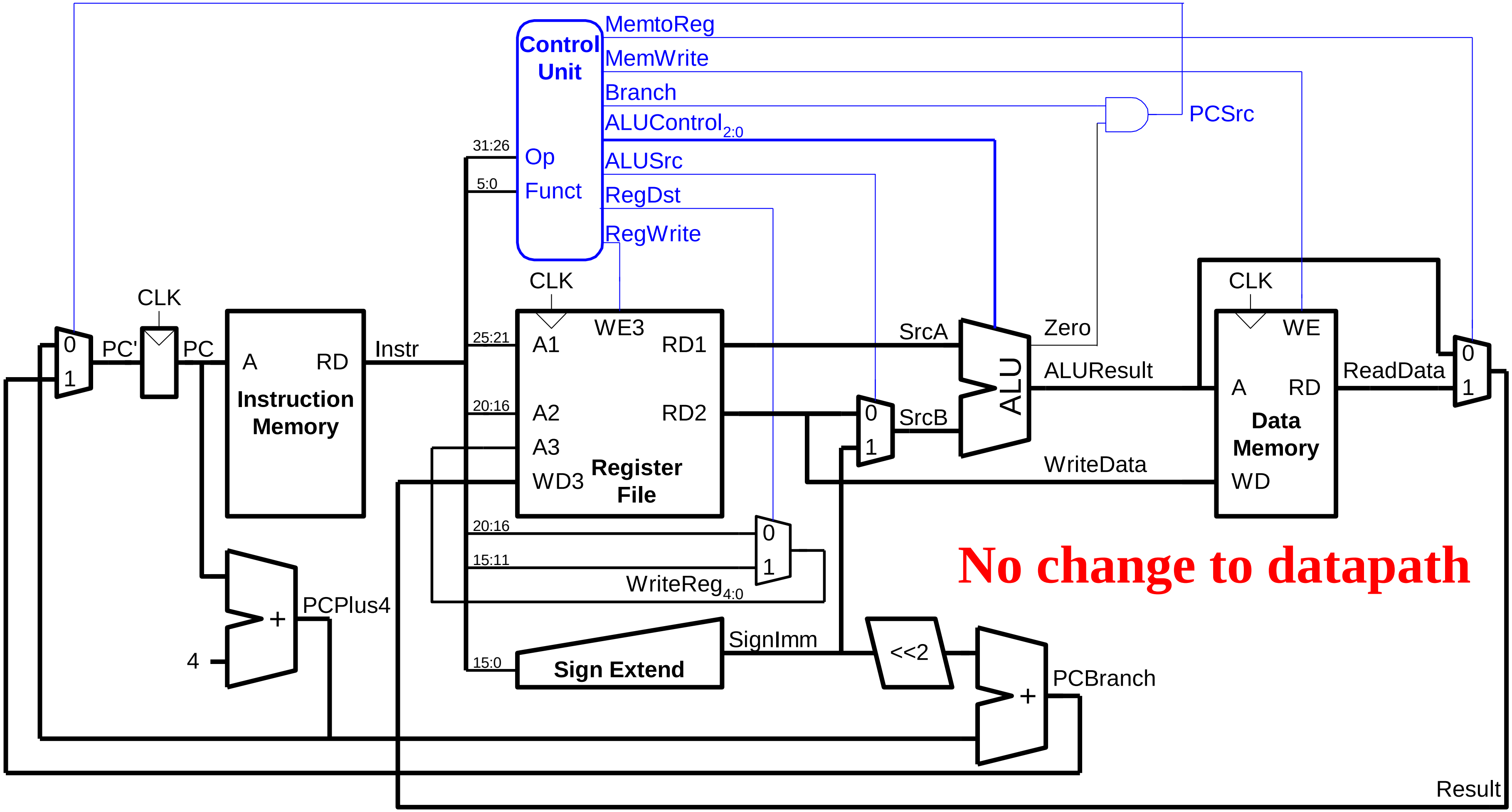


Extended Functionality: addi

立即数加法

```
addi rt, rs, imm
```

```
[rt] = [rs] + SignImm
```

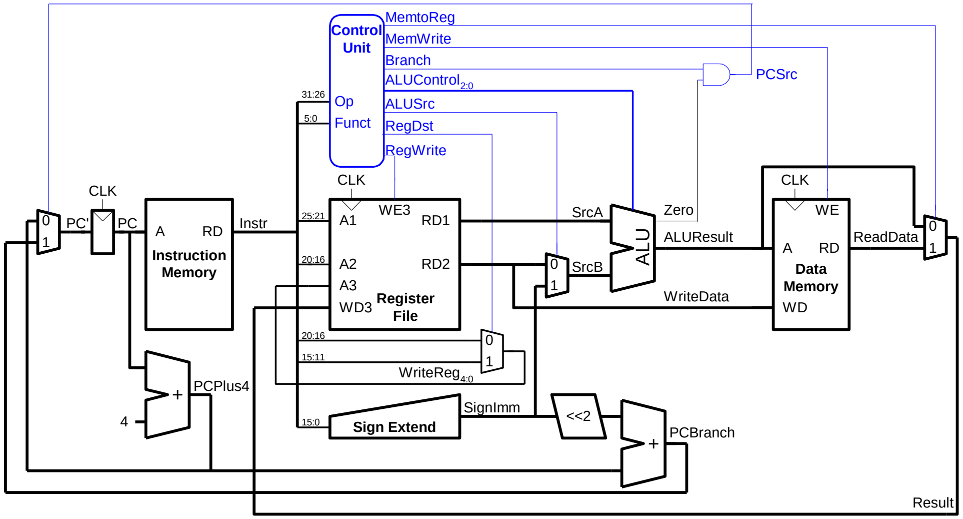


No change to datapath

Extended Functionality: addi

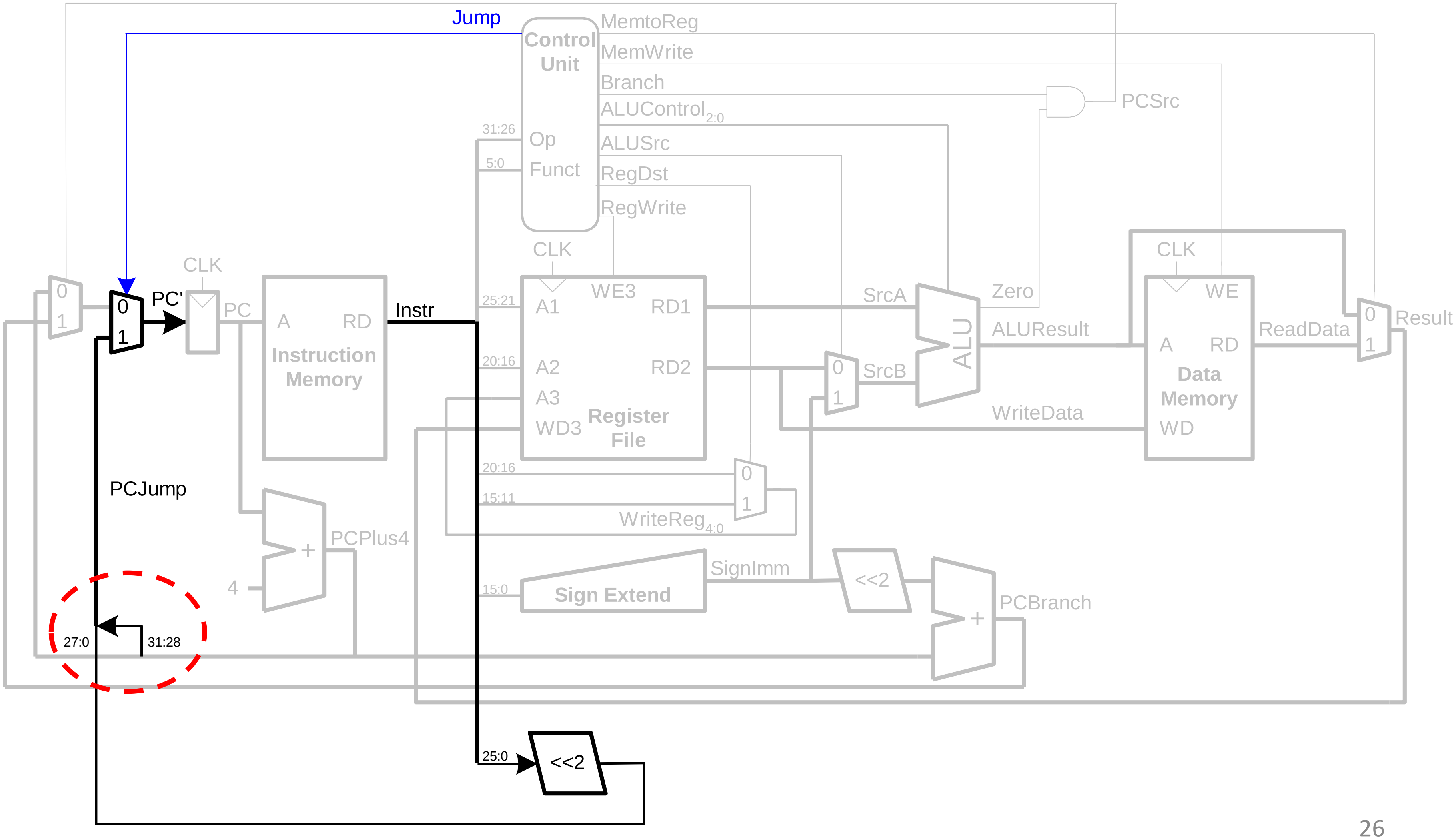
立即数加法 `addi rt, rs, imm` `[rt] = [rs] + SignImm`

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	**001000**	**1**	**0**	**1**	**0**	**0**	**0**	**00**



Extended Functionality: j

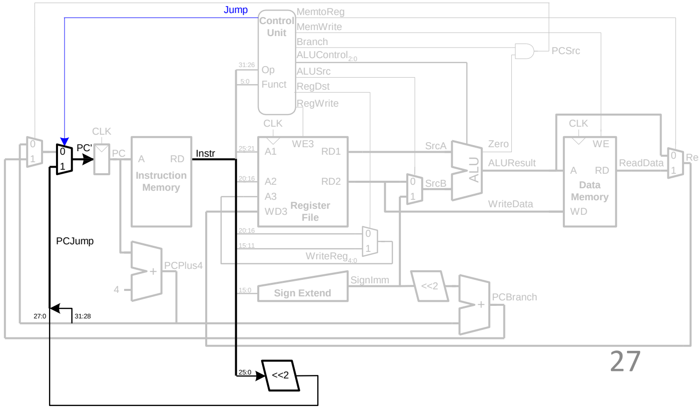
跳转 j label PC’ = {(PC+4)[31:28], addr, 2’b0}



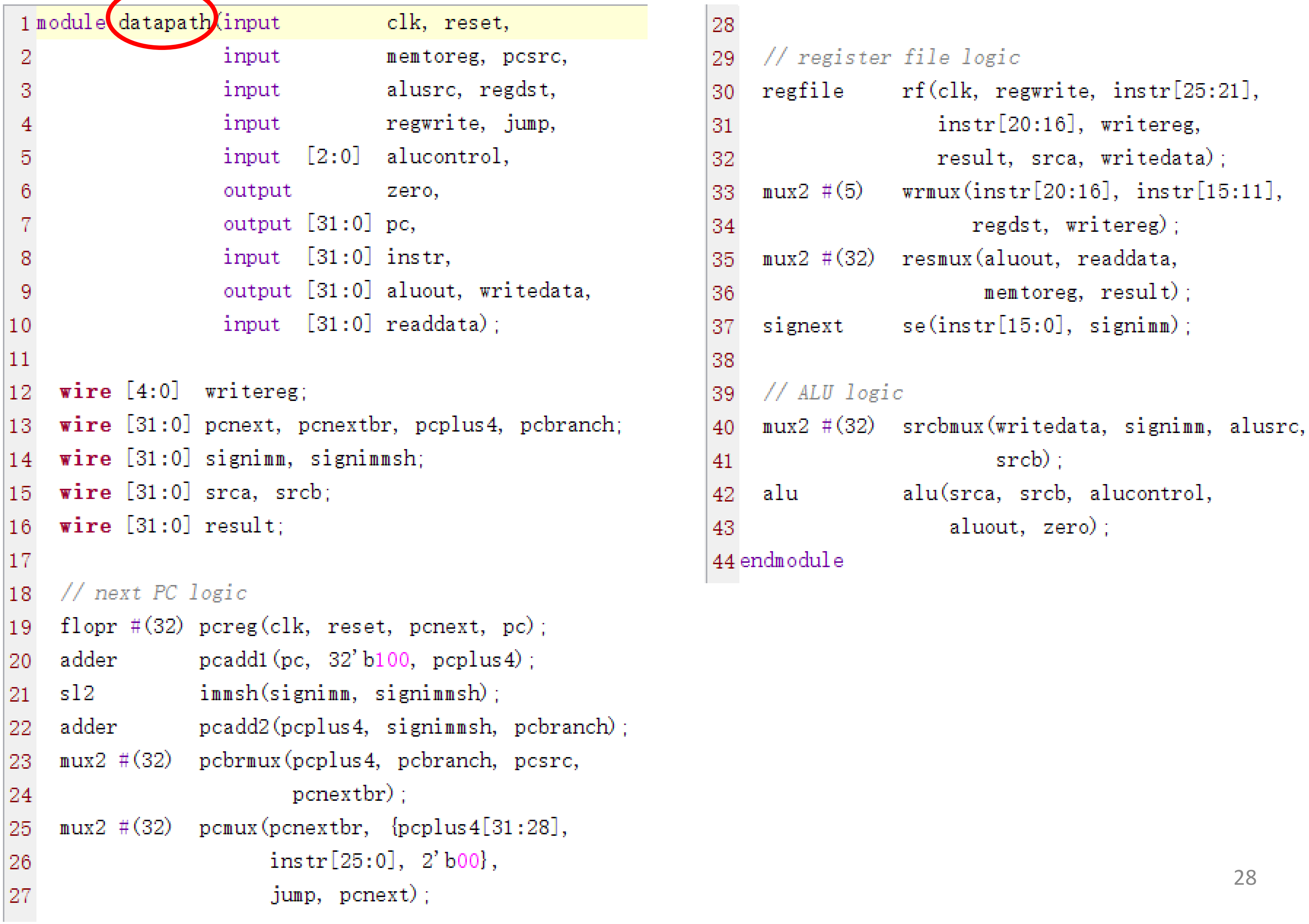
Extended Functionality: j

跳转 `j label` `PC' = {(PC+4)[31:28], addr, 2'b0}`

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
j	000010	0	X	X	X	0	X	XX	1

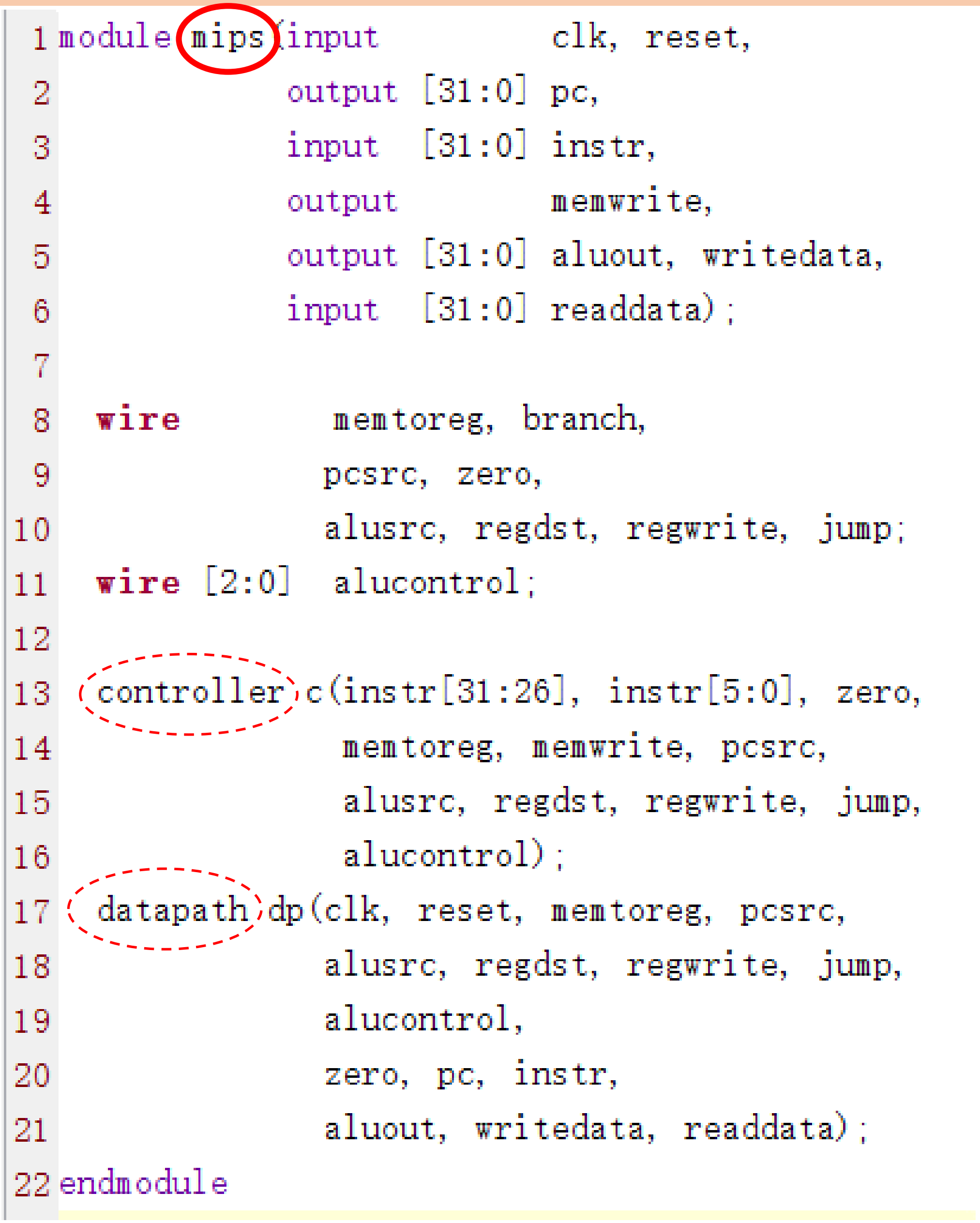


单周期 数据路径datapath 代码



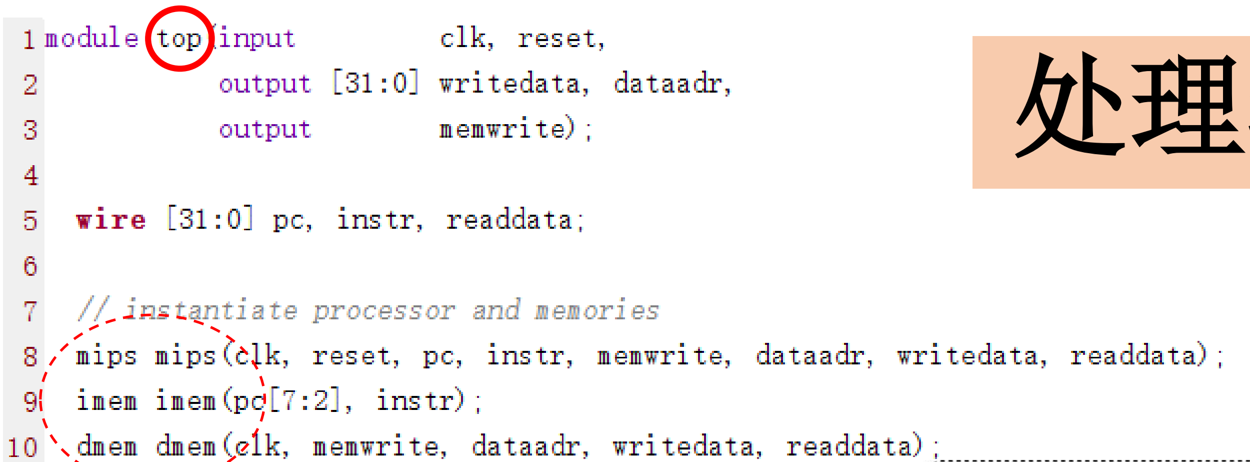
```
module datapath(input          clk, reset,
                input          memtoreg, pcsrc,
                input          alusrc, regdst,
                input          regwrite, jump,
                input   [2:0]  alucontrol,
                output         zero,
                output  [31:0] pc,
                input   [31:0] instr,
                output  [31:0] aluout, writedata,
                input   [31:0] readdata);

  wire [4:0]  writereg;
  wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  wire [31:0] signimm, signimmsh;
  wire [31:0] srca, srcb;
  wire [31:0] result;

  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext, pc);
  adder       pcadd1(pc, 32'b100, pcplus4);
  sl2         immsh(signimm, signimmsh);
  adder       pcadd2(pcplus4, signimmsh, pcbranch);
  mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc,
                      pcnextbr);
  mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28],
                    instr[25:0], 2'b00},
                    jump, pcnext);

  // register file logic
  regfile     rf(clk, regwrite, instr[25:21],
                 instr[20:16], writereg,
                 result, srca, writedata);
  mux2 #(5)   wrmux(instr[20:16], instr[15:11],
                    regdst, writereg);
  mux2 #(32)  resmux(aluout, readdata,
                     memtoreg, result);
  signext     se(instr[15:0], signimm);

  // ALU logic
  mux2 #(32)  srcbmux(writedata, signimm, alusrc,
                      srcb);
  alu         alu(srca, srcb, alucontrol,
                  aluout, zero);
endmodule
```

单周期MIPS处理器 代码

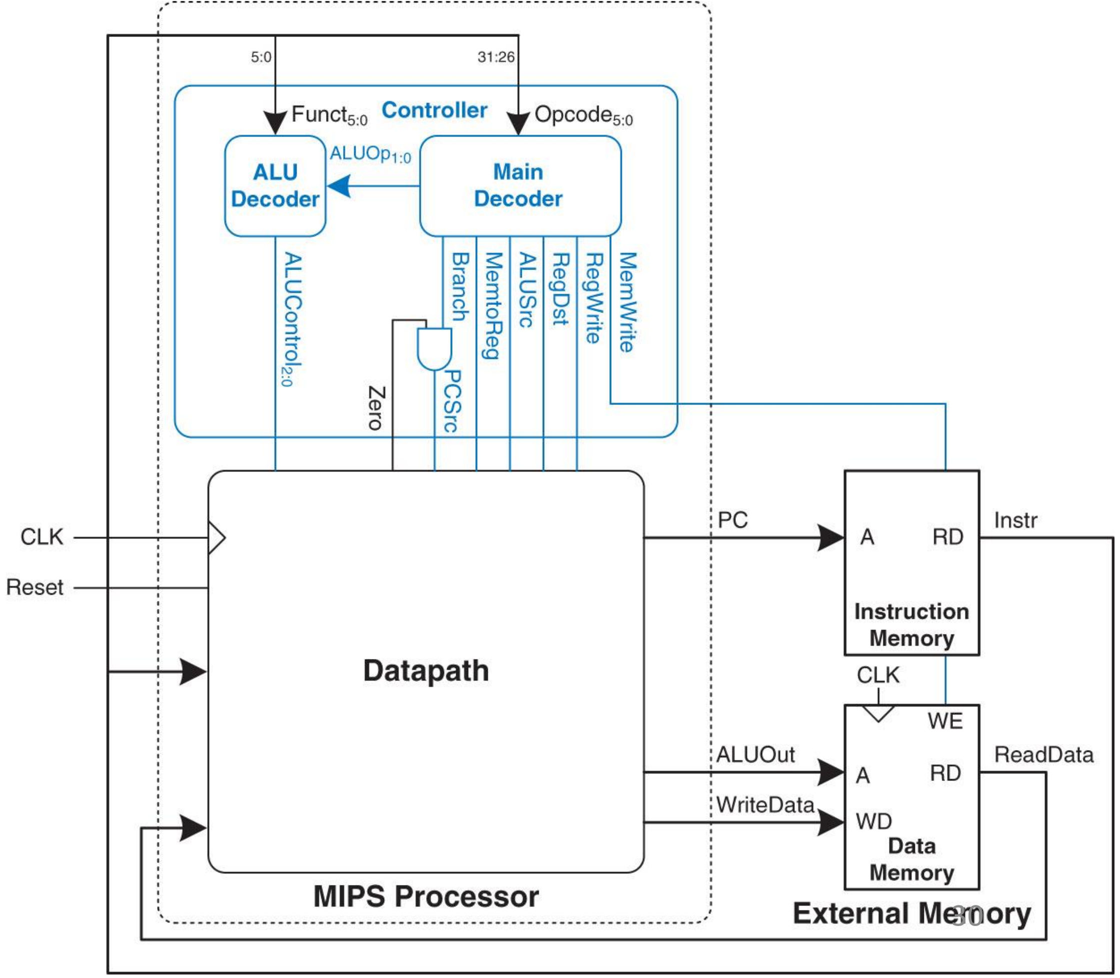


```
module mips(input          clk, reset,
            output [31:0] pc,
            input  [31:0] instr,
            output        memwrite,
            output [31:0] aluout, writedata,
            input  [31:0] readdata);

  wire         memtoreg, branch,
               pcsrc, zero,
               alusrc, regdst, regwrite, jump;
  wire [2:0]   alucontrol;

  controller c(instr[31:26], instr[5:0], zero,
               memtoreg, memwrite, pcsrc,
               alusrc, regdst, regwrite, jump,
               alucontrol);
  datapath dp(clk, reset, memtoreg, pcsrc,
              alusrc, regdst, regwrite, jump,
              alucontrol,
              zero, pc, instr,
              aluout, writedata, readdata);
endmodule
```

处理器顶层文件



```
module top(input          clk, reset,
           output [31:0] writedata, dataadr,
           output         memwrite);

  wire [31:0] pc, instr, readdata;

  // instantiate processor and memories
  mips mips(clk, reset, pc, instr, memwrite, dataadr, writedata, readdata);
  imem imem(pc[7:2], instr);
  dmem dmem(clk, memwrite, dataadr, writedata, readdata);

endmodule
```



单周期MIPS处理器 性能分析

CPI T_c

程序执行时间 = 指令数 × 每条指令的时钟周期数 × 每个周期的运行时间

CPI（每条指令的时钟周期）**C**lock cycles **p**er **i**nstruction

对于单周期每条**lw**指令运行时间：

$$T_c = \boldsymbol{t}_{\boldsymbol{pcq\_PC}} + \boldsymbol{t}_{\mathbf{mem}} + \max(\boldsymbol{t}_{\boldsymbol{RF}\mathbf{read}}, t_{sext} + t_{\mathrm{mux}}) + \boldsymbol{t}_{\mathbf{ALU}} + \boldsymbol{t}_{\mathbf{mem}} + \boldsymbol{t}_{\mathbf{mux}} + t_{RF\mathrm{setup}}$$

元件	参数	延迟 ps
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\mathrm{read}}$	150
Register file setup	$t_{RF\mathrm{setup}}$	20

单周期每条指令需要一个时钟周期，CPI=1

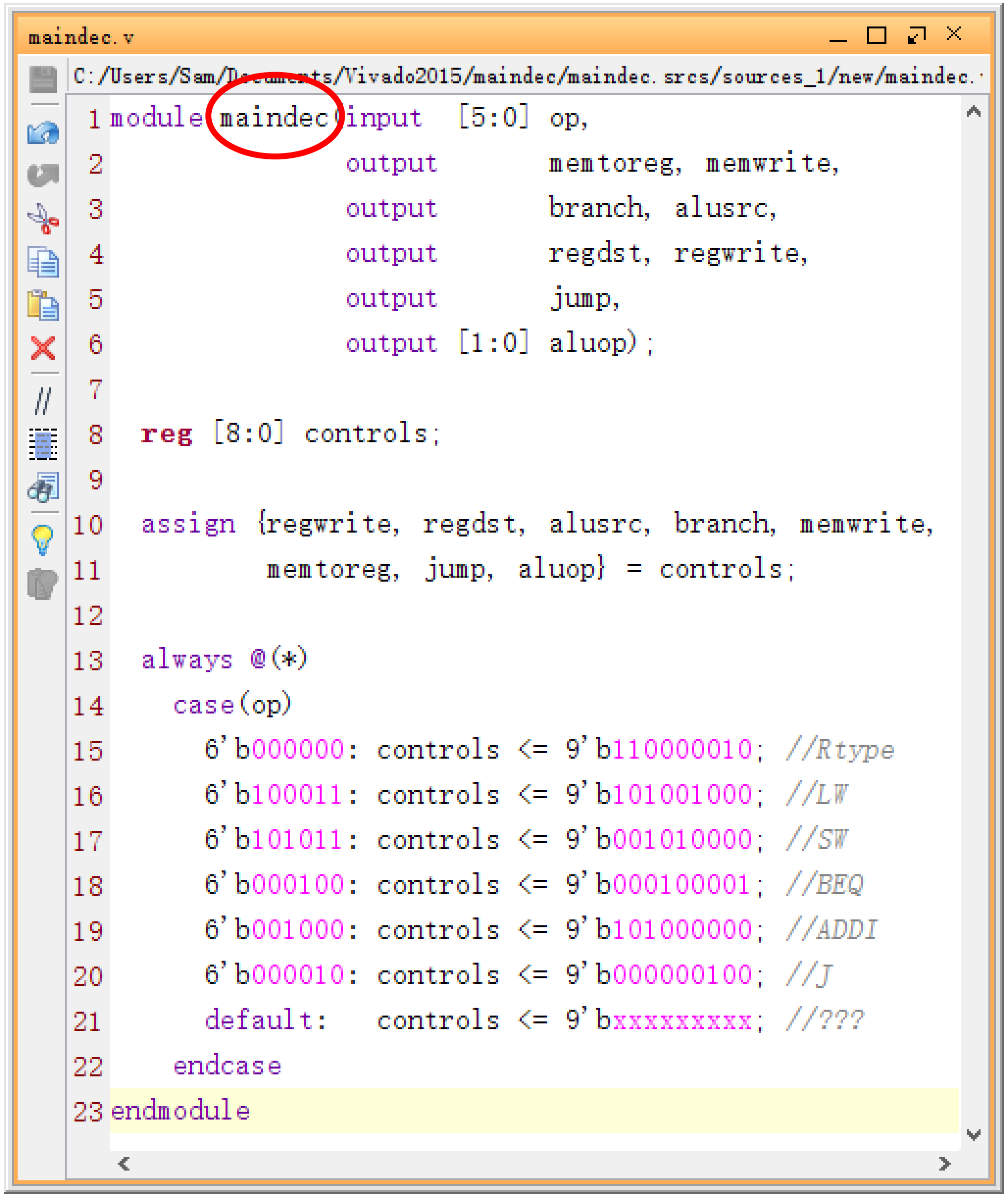
T_c =30+250+150+200+250+25+20=925 ps

因此，1000亿条指令执行时间为：

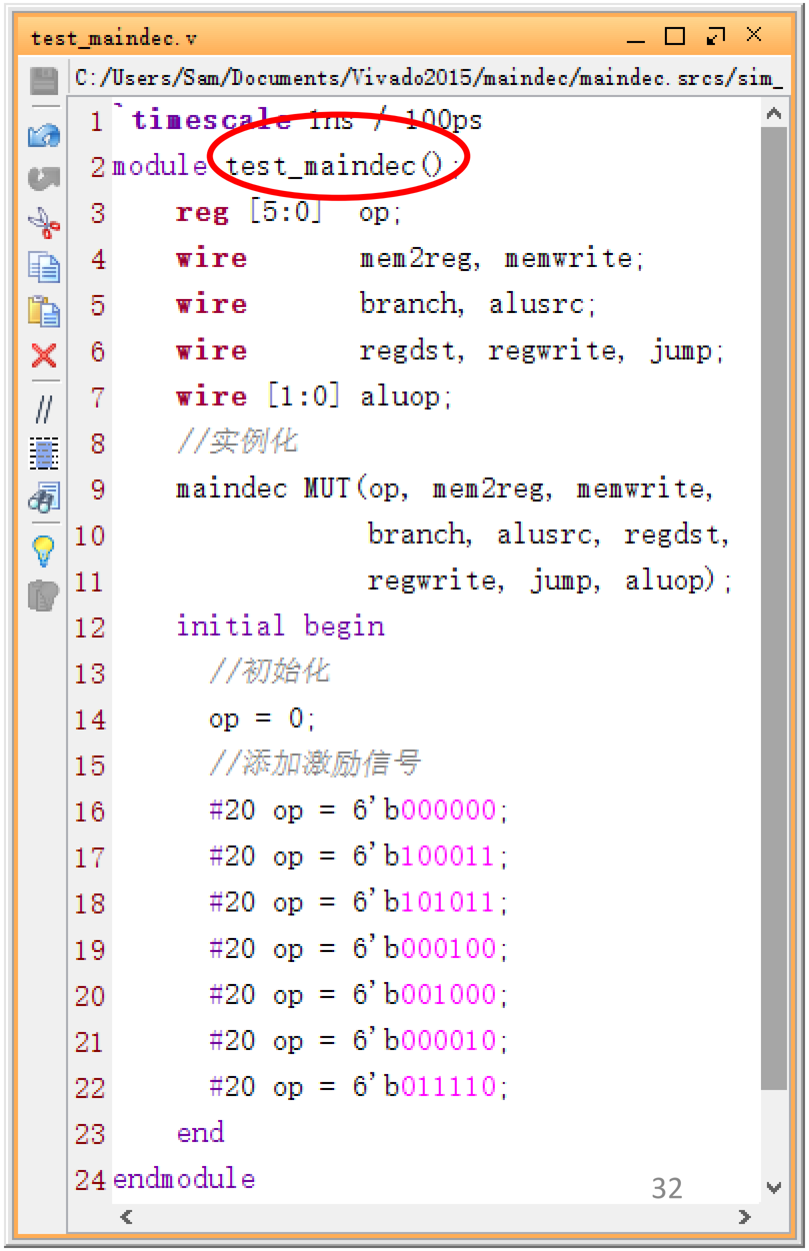
$$T = 100 \times 10^9 指令 \times \mathbf{1}\,\frac{周期}{指令} \times 925 \times 10^{-12}\,\frac{秒}{周期}$$

$$= 92.5秒$$

模块测试-maindec

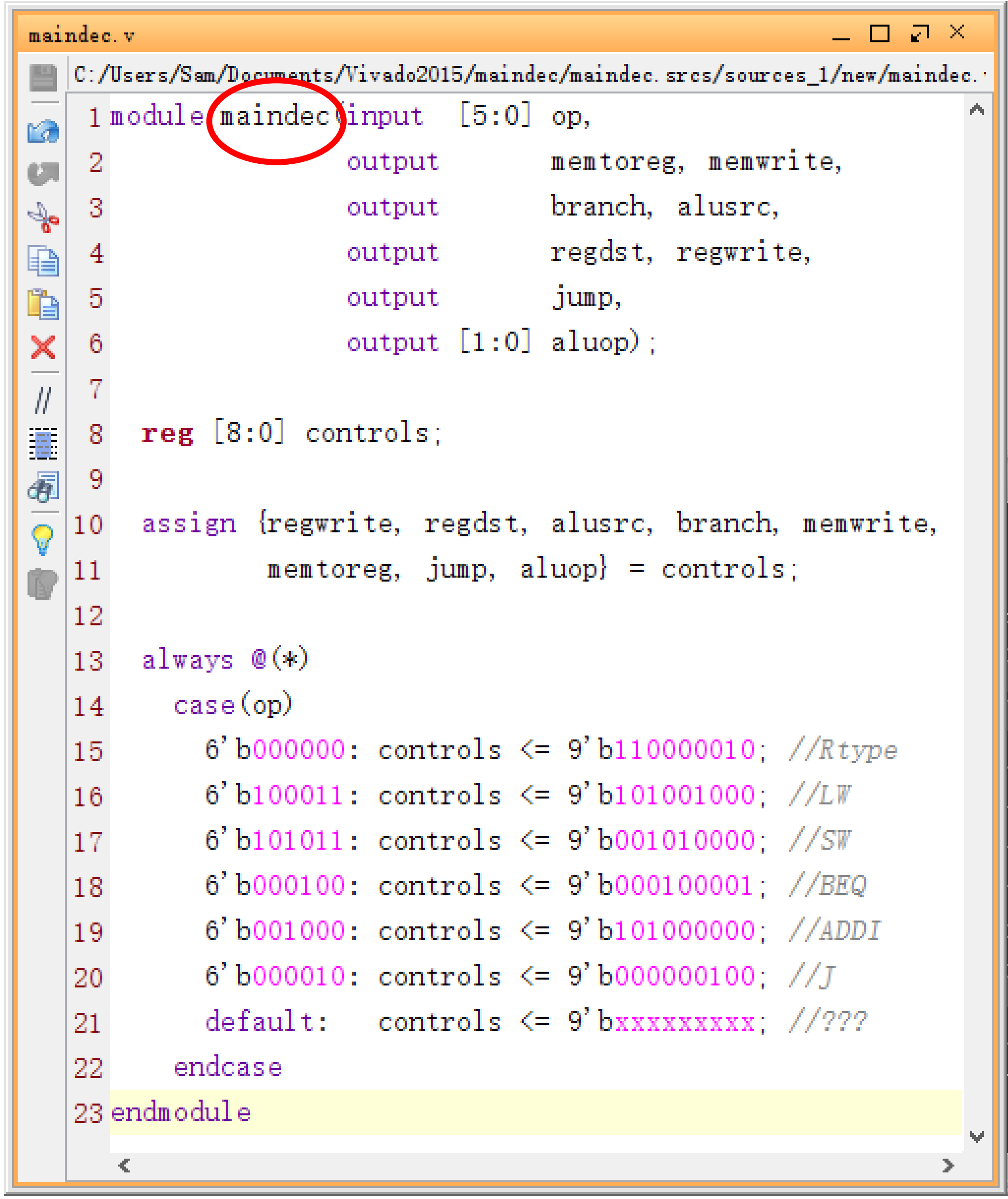

maindec.v
C:/Users/Sam/Documents/Vivado2015/maindec/maindec.srcs/sources_1/new/maindec.

```
module maindec(input   [5:0] op,
               output        memtoreg, memwrite,
               output        branch, alusrc,
               output        regdst, regwrite,
               output        jump,
               output  [1:0] aluop);

 reg [8:0] controls;

 assign {regwrite, regdst, alusrc, branch, memwrite,
             memtoreg, jump, aluop} = controls;

 always @(*)
   case(op)
     6'b000000: controls <= 9'b110000010; //Rtype
     6'b100011: controls <= 9'b101001000; //LW
     6'b101011: controls <= 9'b001010000; //SW
     6'b000100: controls <= 9'b000100001; //BEQ
     6'b001000: controls <= 9'b101000000; //ADDI
     6'b000010: controls <= 9'b000000100; //J
     default:   controls <= 9'bxxxxxxxxx; //???
   endcase
endmodule
```



```
`timescale 1ns / 100ps
module test_maindec();
    reg [5:0]  op;
    wire       mem2reg, memwrite;
    wire       branch, alusrc;
    wire       regdst, regwrite, jump;
    wire [1:0] aluop;
    //实例化
    maindec MUT(op, mem2reg, memwrite,
                branch, alusrc, regdst,
                regwrite, jump, aluop);
    initial begin
      //初始化
      op = 0;
      //添加激励信号
      #20 op = 6'b000000;
      #20 op = 6'b100011;
      #20 op = 6'b101011;
      #20 op = 6'b000100;
      #20 op = 6'b001000;
      #20 op = 6'b000010;
      #20 op = 6'b011110;
    end
endmodule
```

模块测试-maindec

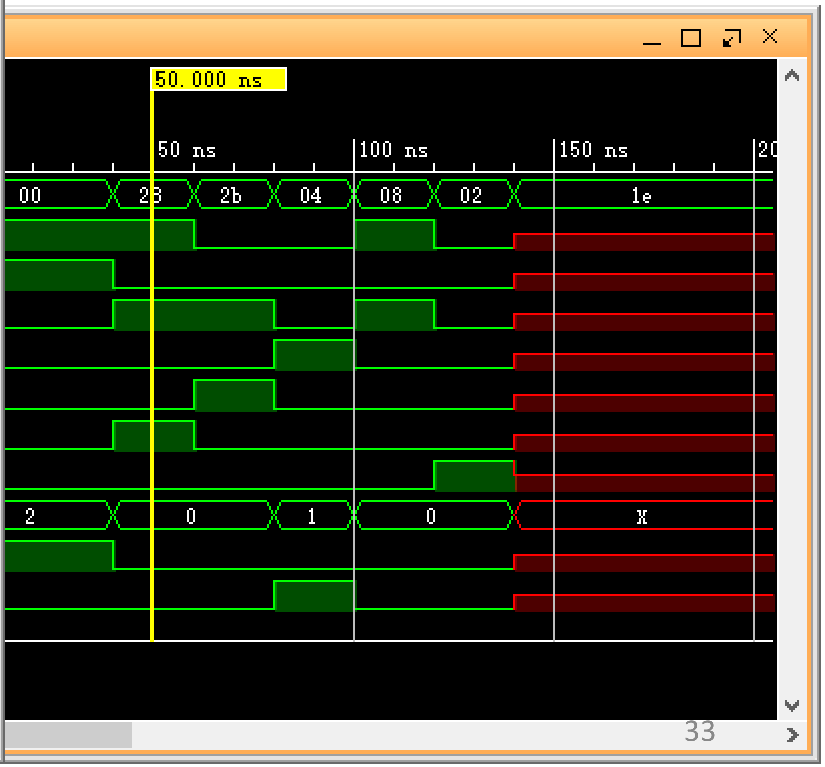


maindec.v

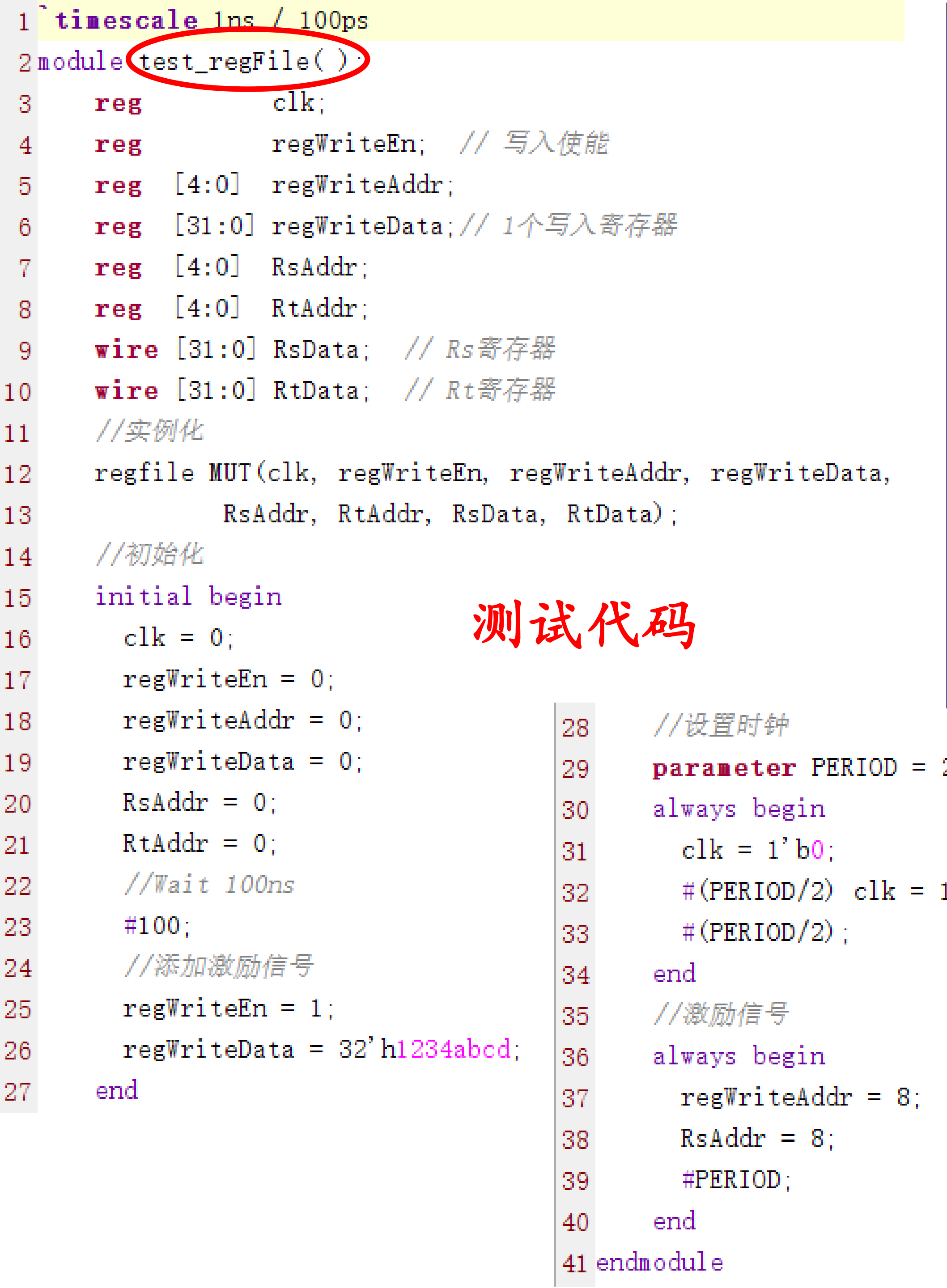
C:/Users/Sam/Documents/Vivado2015/maindec/maindec.srcs/sources_1/new/maindec.

```
module maindec(input  [5:0] op,
               output       memtoreg, memwrite,
               output       branch, alusrc,
               output       regdst, regwrite,
               output       jump,
               output [1:0] aluop);

  reg [8:0] controls;

  assign {regwrite, regdst, alusrc, branch, memwrite,
          memtoreg, jump, aluop} = controls;

  always @(*)
    case(op)
      6'b000000: controls <= 9'b110000010; //Rtype
      6'b100011: controls <= 9'b101001000; //LW
      6'b101011: controls <= 9'b001010000; //SW
      6'b000100: controls <= 9'b000100001; //BEQ
      6'b001000: controls <= 9'b101000000; //ADDI
      6'b000010: controls <= 9'b000000100; //J
      default:   controls <= 9'bxxxxxxxxx; //???
    endcase
endmodule
```

输出波形图

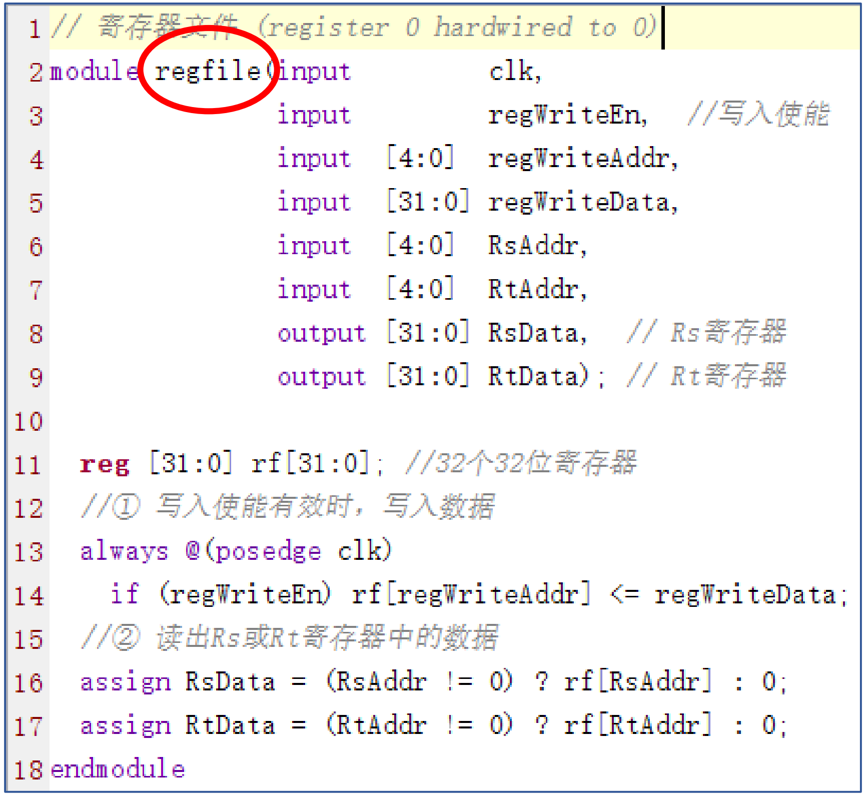


模块测试-regFile



测试代码

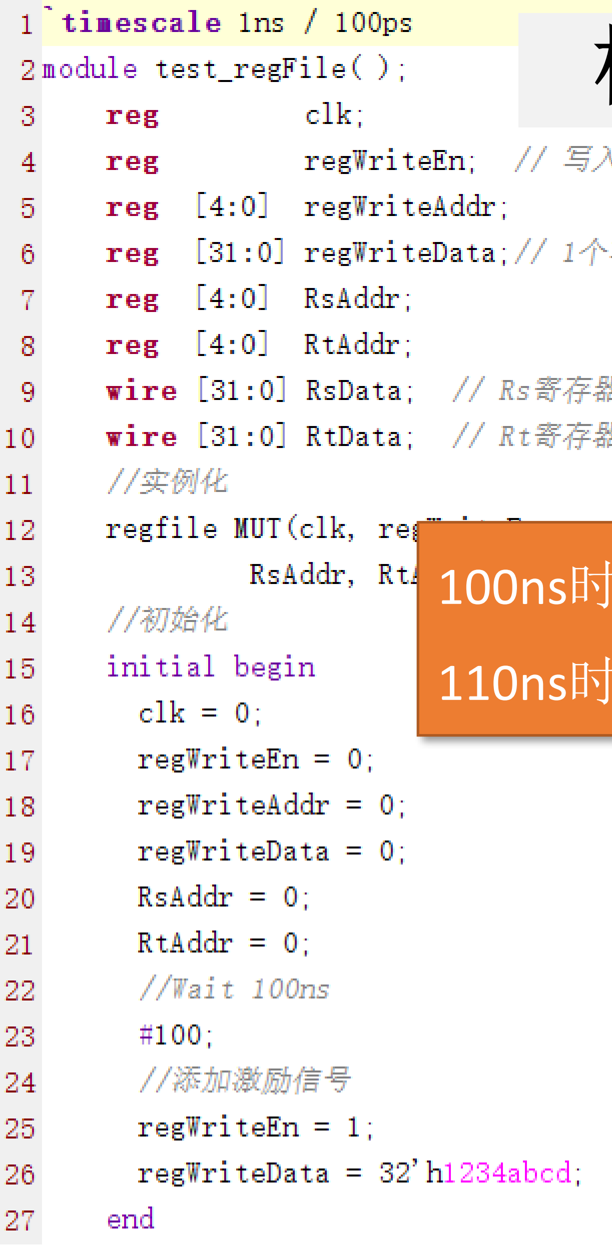
```
`timescale 1ns / 100ps
module test_regFile( );
   reg          clk;
   reg          regWriteEn;  // 写入使能
   reg   [4:0]  regWriteAddr;
   reg   [31:0] regWriteData;// 1个写入寄存器
   reg   [4:0]  RsAddr;
   reg   [4:0]  RtAddr;
   wire  [31:0] RsData;   // Rs寄存器
   wire  [31:0] RtData;   // Rt寄存器
   //实例化
   regfile MUT(clk, regWriteEn, regWriteAddr, regWriteData,
               RsAddr, RtAddr, RsData, RtData);
   //初始化
   initial begin
     clk = 0;
     regWriteEn = 0;
     regWriteAddr = 0;
     regWriteData = 0;
     RsAddr = 0;
     RtAddr = 0;
     //Wait 100ns
     #100;
     //添加激励信号
     regWriteEn = 1;
     regWriteData = 32’h1234abcd;
   end
   //设置时钟
   parameter PERIOD = 20;
   always begin
     clk = 1’b0;
     #(PERIOD/2) clk = 1’b1;
     #(PERIOD/2);
   end
   //激励信号
   always begin
     regWriteAddr = 8;
     RsAddr = 8;
     #PERIOD;
   end
endmodule
```



```
// 寄存器文件 (register 0 hardwired to 0)
module regfile(input         clk,
               input         regWriteEn,  //写入使能
               input  [4:0]  regWriteAddr,
               input  [31:0] regWriteData,
               input  [4:0]  RsAddr,
               input  [4:0]  RtAddr,
               output [31:0] RsData,  // Rs寄存器
               output [31:0] RtData); // Rt寄存器

 reg [31:0] rf[31:0]; //32个32位寄存器
 //① 写入使能有效时，写入数据
 always @(posedge clk)
   if (regWriteEn) rf[regWriteAddr] <= regWriteData;
 //② 读出Rs或Rt寄存器中的数据
 assign RsData = (RsAddr != 0) ? rf[RsAddr] : 0;
 assign RtData = (RtAddr != 0) ? rf[RtAddr] : 0;
endmodule
```

regfile源代码

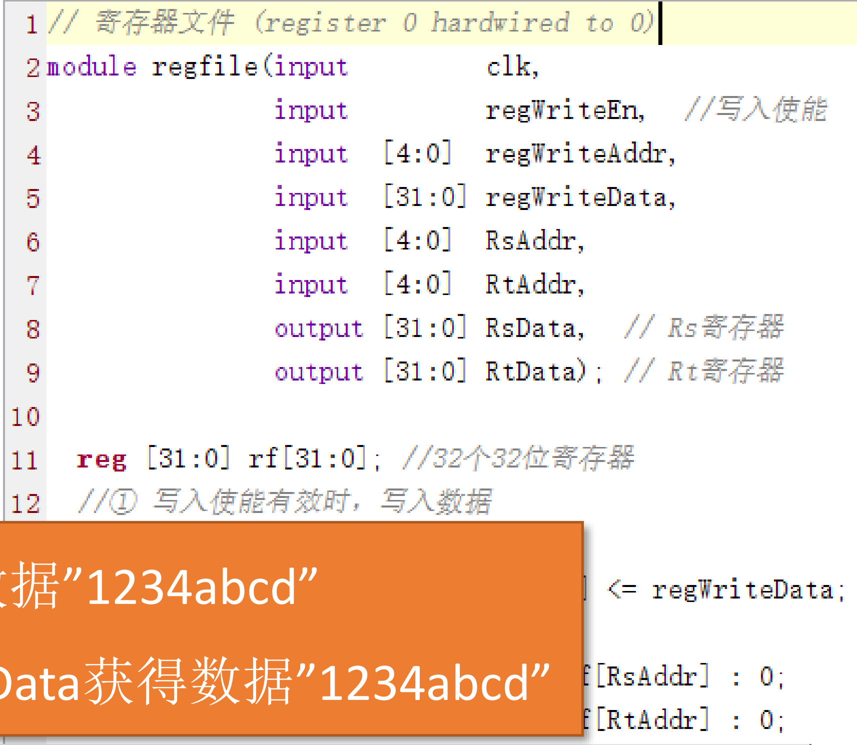
模块测试



```
`timescale 1ns / 100ps
module test_regFile( );
    reg         clk;
    reg         regWriteEn;  // 写入使能
    reg  [4:0]  regWriteAddr;
    reg  [31:0] regWriteData;// 1个写入寄存器
    reg  [4:0]  RsAddr;
    reg  [4:0]  RtAddr;
    wire [31:0] RsData;   // Rs寄存器
    wire [31:0] RtData;   // Rt寄存器
    //实例化
    regfile MUT(clk, re
            RsAddr, Rt
    //初始化
    initial begin
      clk = 0;
      regWriteEn = 0;
      regWriteAddr = 0;
      regWriteData = 0;
      RsAddr = 0;
      RtAddr = 0;
      //Wait 100ns
      #100;
      //添加激励信号
      regWriteEn = 1;
      regWriteData = 32'h1234abcd;
    end
```

```
endmodule
```

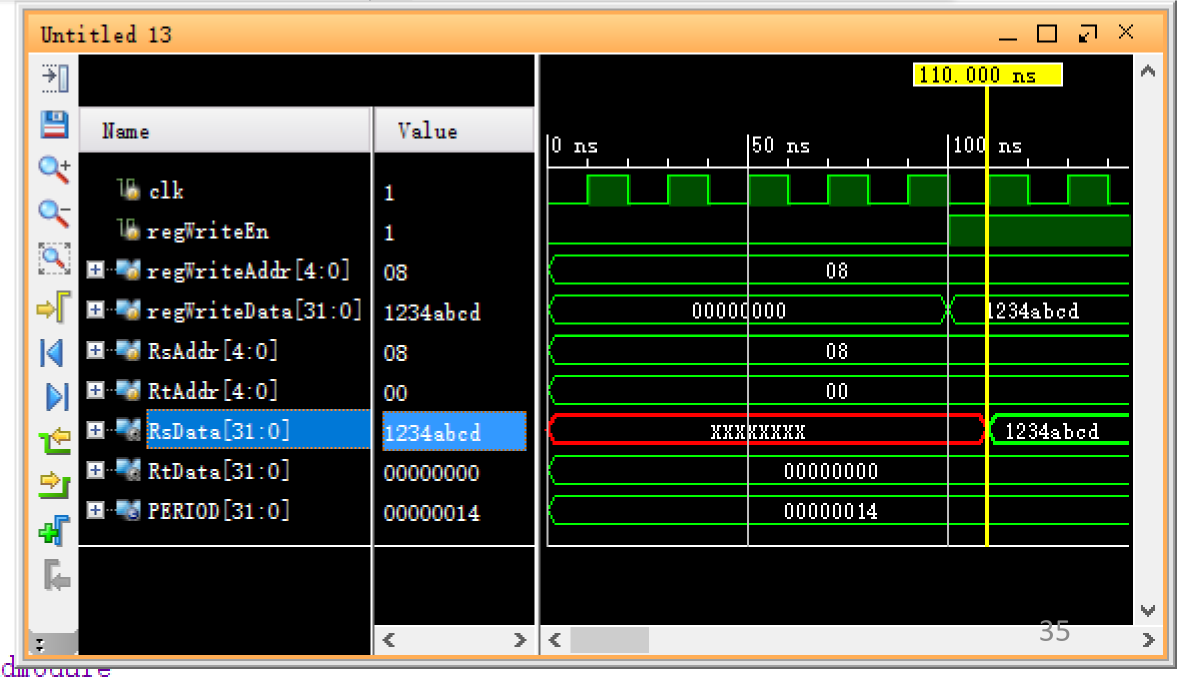
测试代码



```
// 寄存器文件 (register 0 hardwired to 0)
module regfile(input        clk,
               input        regWriteEn,   //写入使能
               input  [4:0]  regWriteAddr,
               input  [31:0] regWriteData,
               input  [4:0]  RsAddr,
               input  [4:0]  RtAddr,
               output [31:0] RsData,   // Rs寄存器
               output [31:0] RtData);  // Rt寄存器

  reg [31:0] rf[31:0]; //32个32位寄存器
  //① 写入使能有效时，写入数据
                                  ] <= regWriteData;

                                  f[RsAddr] : 0;
                                  f[RtAddr] : 0;
```

100ns时regWriteData获得数据”1234abcd”

110ns时，时钟上升沿，RsData获得数据”1234abcd”



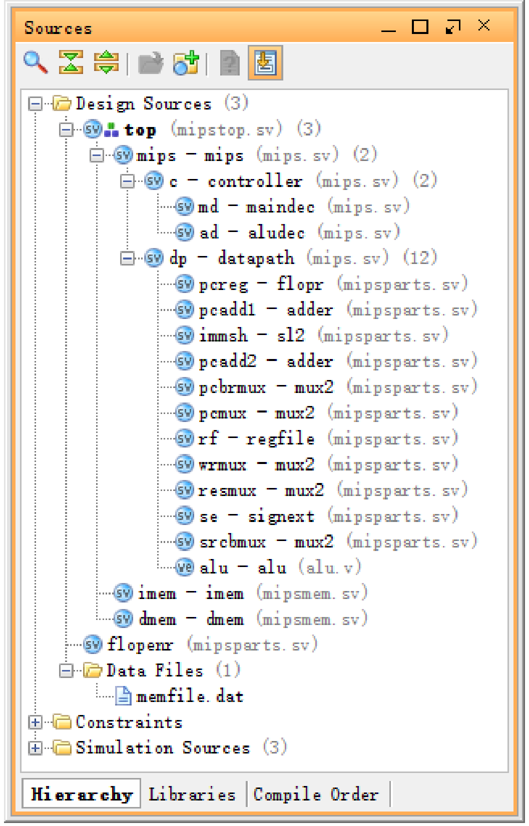
Untitled 13

Name	Value
clk	1
regWriteEn	1
regWriteAddr[4:0]	08
regWriteData[31:0]	1234abcd
RsAddr[4:0]	08
RtAddr[4:0]	00
RsData[31:0]	1234abcd
RtData[31:0]	00000000
PERIOD[31:0]	00000014

MIPS单周期处理器

sMIPS

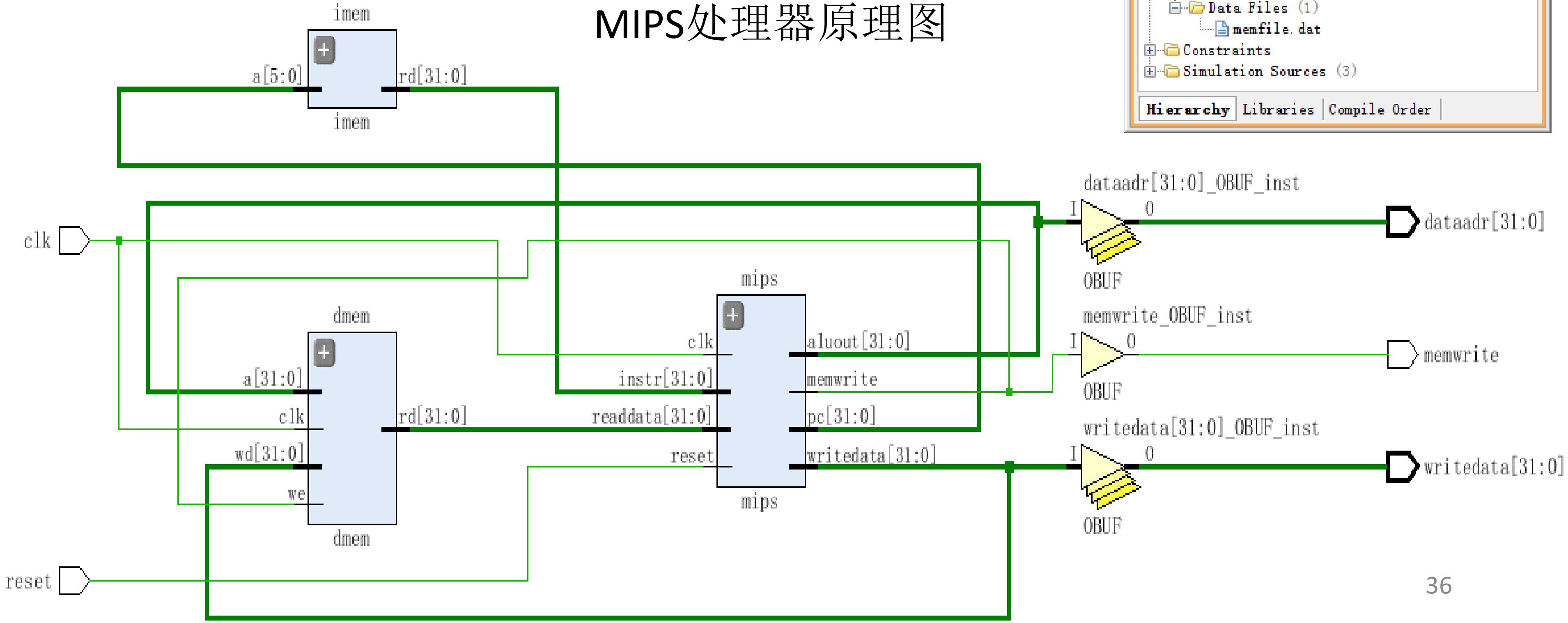
Vivado中程序组织结构



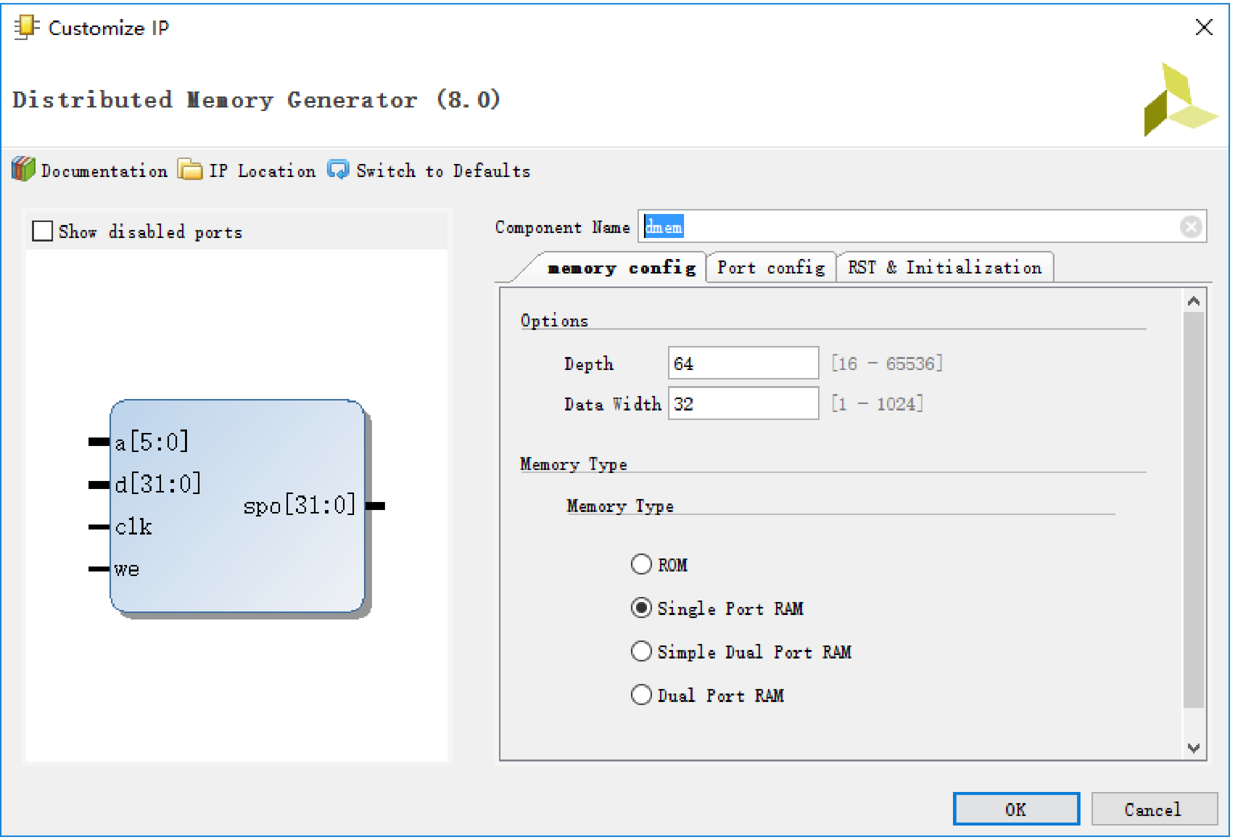
Sources

```
Design Sources (3)
  top (mipstop.sv) (3)
    mips - mips (mips.sv) (2)
      c - controller (mips.sv) (2)
        md - maindec (mips.sv)
        ad - aludec (mips.sv)
      dp - datapath (mips.sv) (12)
        pcreg - flopr (mipsparts.sv)
        pcadd1 - adder (mipsparts.sv)
        immsh - sl2 (mipsparts.sv)
        pcadd2 - adder (mipsparts.sv)
        pcbrmux - mux2 (mipsparts.sv)
        pcmux - mux2 (mipsparts.sv)
        rf - regfile (mipsparts.sv)
        wrmux - mux2 (mipsparts.sv)
        resmux - mux2 (mipsparts.sv)
        se - signext (mipsparts.sv)
        srcbmux - mux2 (mipsparts.sv)
        alu - alu (alu.v)
    imem - imem (mipsmem.sv)
    dmem - dmem (mipsmem.sv)
  flopenr (mipsparts.sv)
  Data Files (1)
    memfile.dat
Constraints
Simulation Sources (3)
```

MIPS处理器原理图



Vivado中用IP配置数据存储器dmem



Distributed Memory Generator (8.0)

□ Show disabled ports

Component Name: dmem

memory config | Port config | RST & Initialization

Options

Depth: 64 [16 – 65536]

Data Width: 32 [1 – 1024]

Memory Type

Memory Type

○ ROM

● Single Port RAM

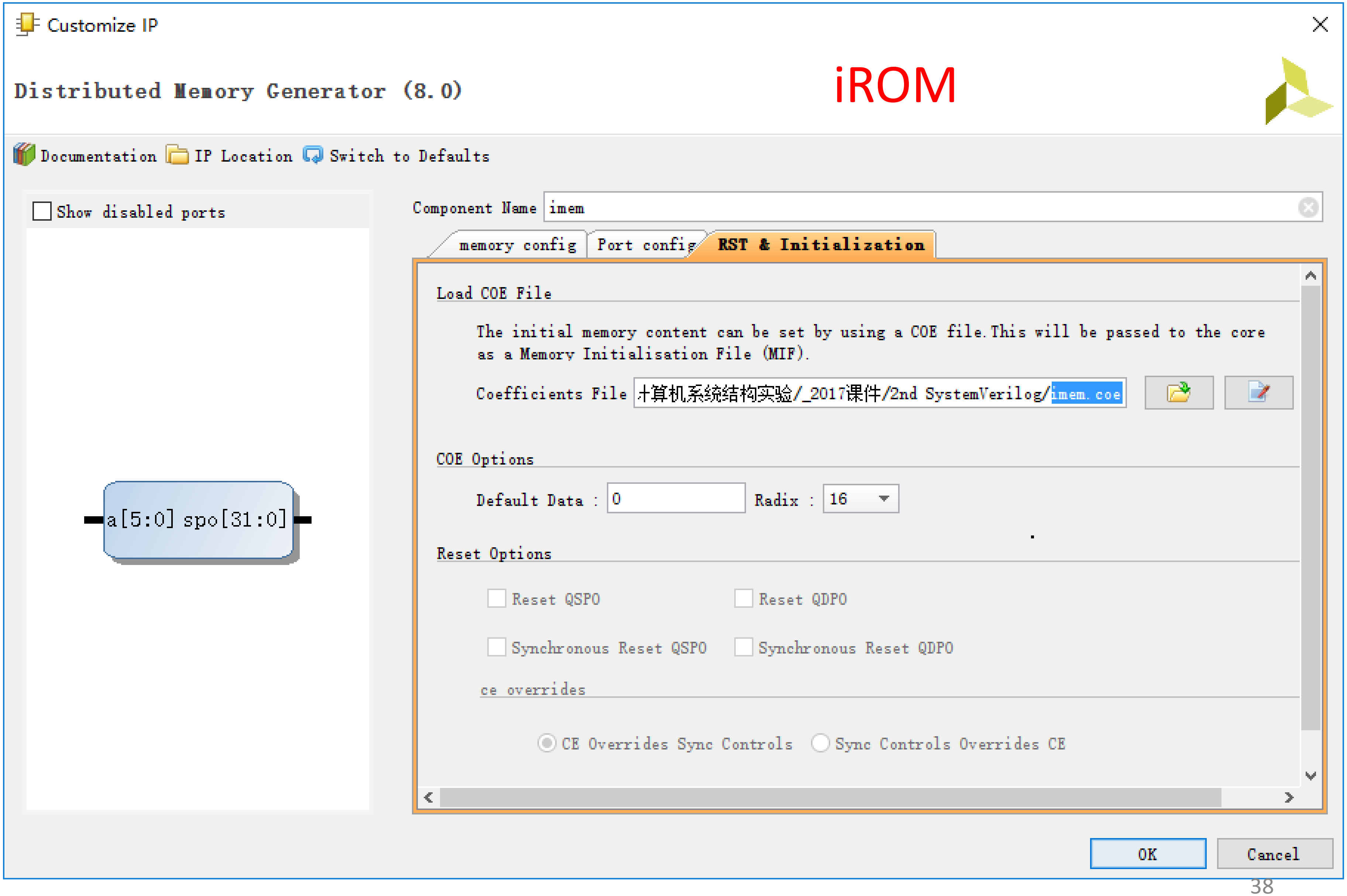
○ Simple Dual Port RAM

○ Dual Port RAM

dmem接口不配套！

```
dmem your_instance_name (
  .a(a),      // input wire [5 : 0] a
  .d(d),      // input wire [31 : 0] d
  .clk(clk),  // input wire clk
  .we(we),    // input wire we
  .spo(spo)  // output wire [31 : 0] spo
);
```

Vivado中用IP配置指令存储器imem



Customize IP

Distributed Memory Generator (8.0)

iROM

Documentation | IP Location | Switch to Defaults

□ Show disabled ports

Component Name: imem

memory config | Port config | **RST & Initialization**

Load COE File

The initial memory content can be set by using a COE file. This will be passed to the core as a Memory Initialisation File (MIF).

Coefficients File: 十算机系统结构实验/_2017课件/2nd SystemVerilog/imem.coe

COE Options

Default Data : 0 Radix : 16

Reset Options

□ Reset QSPO □ Reset QDPO

□ Synchronous Reset QSPO □ Synchronous Reset QDPO

ce overrides

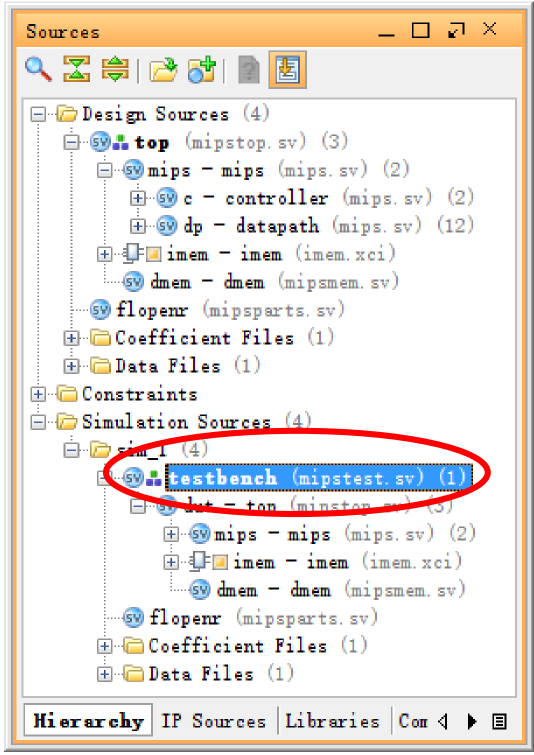
● CE Overrides Sync Controls ○ Sync Controls Overrides CE

OK Cancel

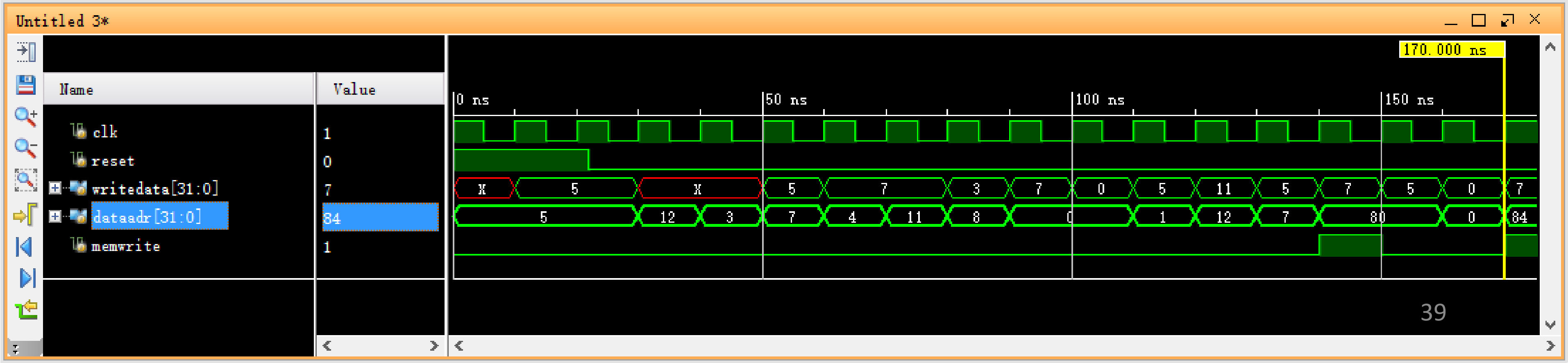
MIPS单周期处理器仿真测试

测试汇编代码+机器代码

```
#         Assembly            Description               Address   Machine
main:     addi $2, $0, 5      # initialize $2 = 5       0         20020005
          addi $3, $0, 12     # initialize $3 = 12      4         2003000c
          addi $7, $3, −9     # initialize $7 = 3       8         2067fff7
          or   $4, $7, $2     # $4 = (3 OR 5) = 7       c         00e22025
          and  $5, $3, $4     # $5 = (12 AND 7) = 4     10        00642824
          add  $5, $5, $4     # $5 = 4 + 7 = 11         14        00a42820
          beq  $5, $7, end    # shouldn't be taken      18        10a7000a
          slt  $4, $3, $4     # $4 = 12 < 7 = 0         1c        0064202a
          beq  $4, $0, around # should be taken         20        10800001
          addi $5, $0, 0      # shouldn't happen        24        20050000
around:   slt  $4, $7, $2     # $4 = 3 < 5 = 1          28        00e2202a
          add  $7, $4, $5     # $7 = 1 + 11 = 12        2c        00853820
          sub  $7, $7, $2     # $7 = 12 − 5 = 7         30        00e23822
          sw   $7, 68($3)     # [80] = 7                34        ac670044
          lw   $2, 80($0)     # $2 = [80] = 7           38        8c020050
          j    end            # should be taken         3c        08000011
          addi $2, $0, 1      # shouldn't happen        40        20020001
end:      sw   $2, 84($0)     # write mem[84] = 7       44        ac020054
```

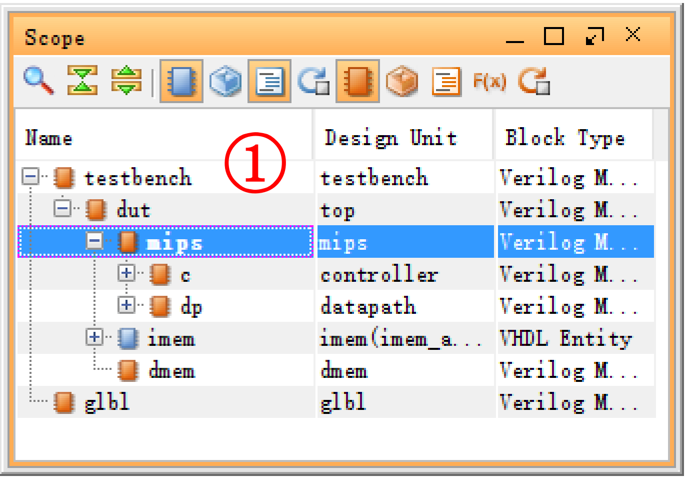


```
Sources
Design Sources (4)
  top (mipstop.sv) (3)
    mips - mips (mips.sv) (2)
      c - controller (mips.sv) (2)
      dp - datapath (mips.sv) (12)
    imem - imem (imem.xci)
    dmem - dmem (mipsmem.sv)
  flopenr (mipsparts.sv)
  Coefficient Files (1)
  Data Files (1)
Constraints
Simulation Sources (4)
  sim_1 (4)
    testbench (mipstest.sv) (1)
      dut - top (mipstop.sv) (3)
        mips - mips (mips.sv) (2)
        imem - imem (imem.xci)
        dmem - dmem (mipsmem.sv)
    flopenr (mipsparts.sv)
    Coefficient Files (1)
    Data Files (1)
Hierarchy | IP Sources | Libraries | Com
```



```
Untitled 3*
Name              Value
clk               1
reset             0
writedata[31:0]   7
dataadr[31:0]     84
memwrite          1
```

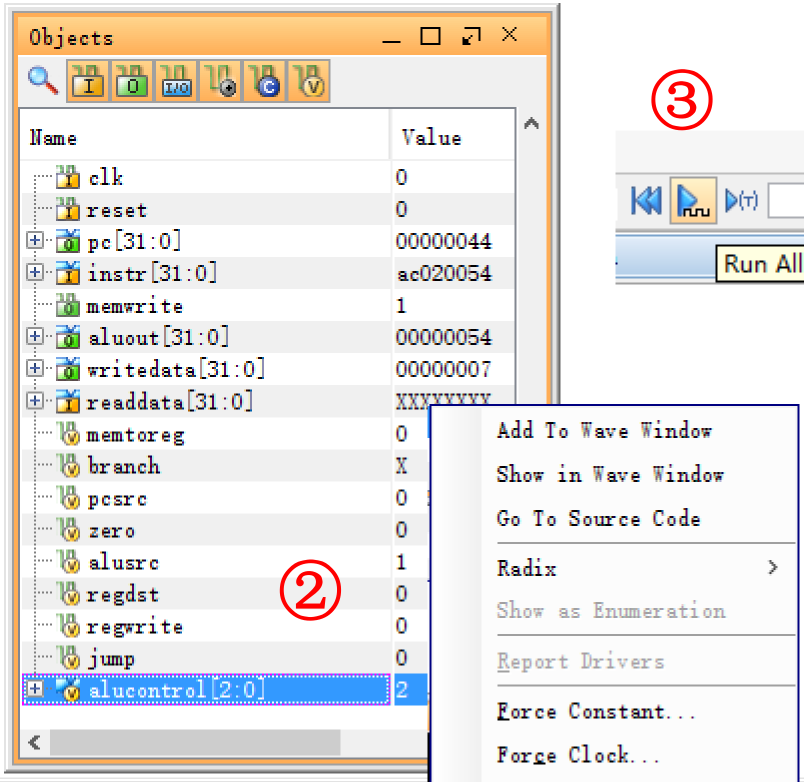
仿真时增加内部信号显示



Scope

①

Name	Design Unit	Block Type
testbench	testbench	Verilog M...
dut	top	Verilog M...
mips	mips	Verilog M...
c	controller	Verilog M...
dp	datapath	Verilog M...
imem	imem(imem_a...	VHDL Entity
dmem	dmem	Verilog M...
glbl	glbl	Verilog M...

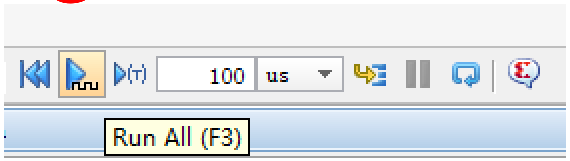


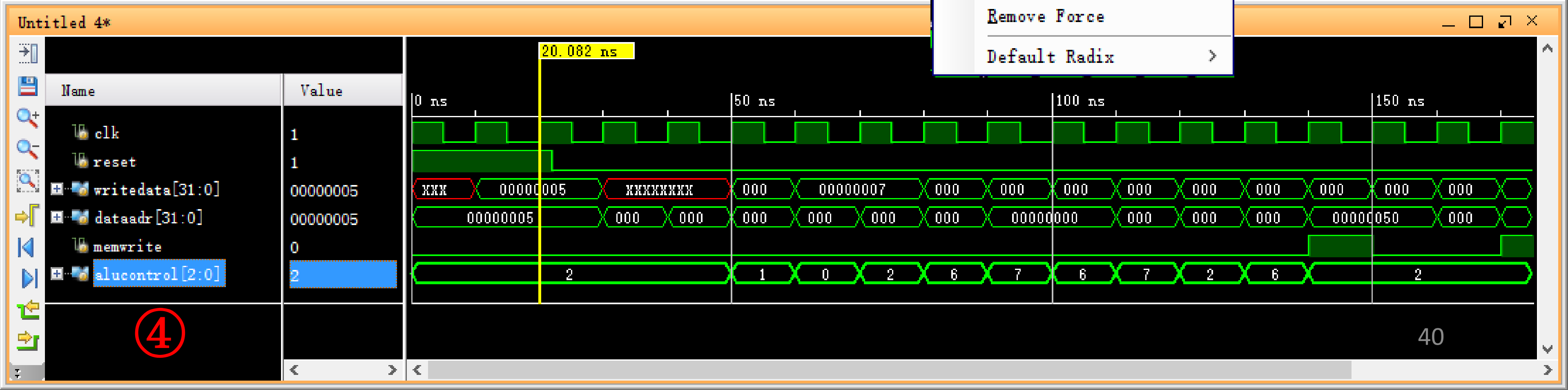
Objects

②

Name	Value
clk	0
reset	0
pc[31:0]	00000044
instr[31:0]	ac020054
memwrite	1
aluout[31:0]	00000054
writedata[31:0]	00000007
readdata[31:0]	XXXXXXXX
memtoreg	0
branch	X
pcsrc	0
zero	0
alusrc	1
regdst	0
regwrite	0
jump	0
alucontrol[2:0]	2

③





Untitled 4*

Name	Value
clk	1
reset	1
writedata[31:0]	00000005
dataadr[31:0]	00000005
memwrite	0
alucontrol[2:0]	2

④

参考资料

数字设计和计算机体系结构

Digital Design and Computer Architecture 2nd

David Money Harris，陈俊颖 译

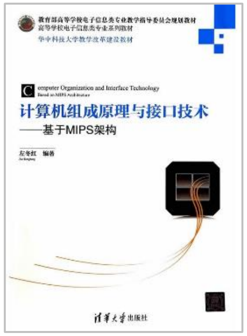
机械工业出版社，2016，**第7章 微体系结构**

计算机组成与设计：硬件/软件接口

Computer Organization and Design: The hardware / software interface

David A. Patterson，王党辉 译

机械工业出版社，2015，第五版



计算机组成原理与接口技术: 基于MIPS架构

左冬红，清华大学出版社，2014