硬件课程设计报告

题　　目	MIPS-16bit CPU
姓　　名	刘祥德
学　　号	915106840327
班　　号	9151065502(软工二班)
指 导 教 师	潘志兰老师

2018 年 8月 29 日

目录

MIPS-16bit CPU 设计报告

1. 介绍

MIPS-16bit CPU 是基于 32bit MIPS RSIV 指令集裁剪、精简后进行设计的单周期处理器，它每个指令的长度固定为 16bit。使用的开发环境为 Xilinx Vivado 2017.1。

2. 设计

2.1 主控制单元设计

控制信号名	无效时的含义	有效时的含义
ReDst	写寄存器的目标寄存器号来自 RT(位 7:4)	写寄存器的目标寄存器号来自 RD 字段(位 3:0)
Jump	Branch 结果送 PC	位 11:0 立即数右移 1 位与 PC+2 高三位拼接送 PC
Branch	PC+4 送 PC	位 3:0 符号扩展后与 PC+4 相加送 PC
MemRead	无	数据存储器读使能有效
MemtoReg	写入寄存器的数据来自 ALU	写入寄存器的数据来自数据存储器
MemWrite	无	数据存储器写使能有效
ALUSrc	第二个 ALU 操作数的来自寄存器堆 RT	第二个 ALU 操作数的来自低四位的符号扩展
RegWrite	无	寄存器堆写使能有效

表格 2.1-1 7 个控制信号的作用

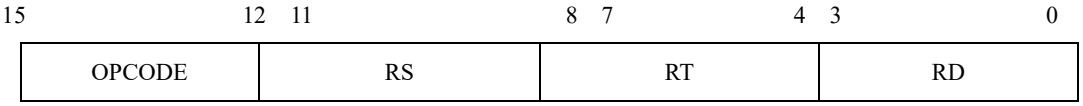
控制单元的输入为指令的高 4 位操作码。控制单元输出包括 3 个控制多选器的 1 位信号(RegWrite、MemRead、MemWrite)，一个决定是否可以转移的信号(Branch)，和一个 ALU 的 3 位控制信号(ALUOp)。分支控制信号 Branch 与 ALU

的零输入一起送入与门，其输出控制下一个 PC 的选择。

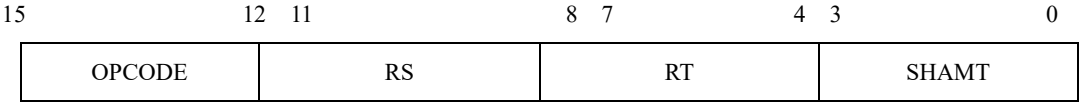
2.2 指令系统

2.2.1 指令格式

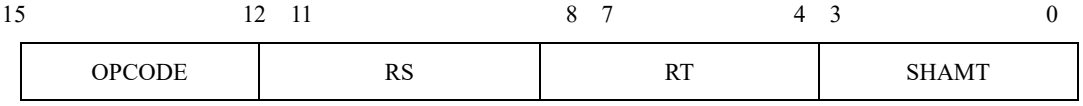
R 型指令：



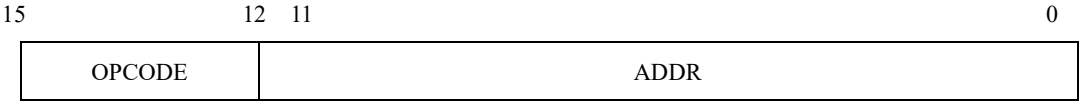
装载指令：



相等则分支指令：



跳转指令：



(1) R 型指令：寄存器操作数有 3 个 RS、RT 和 RD。RS 与 RT 都为源操作数，RD 字段为目的操作数

(2) 装载指令：RS 寄存器作为基址与 4 位的地址字段相加得到访存地址。对与装载指令而言，RT 是要存入存储器的数据所在的寄存器。对于装载指令，RT 是要存入存储器的数据所在的寄存器；

(3) 相等则分支指令：RS 与 RT 是源寄存器，用于比较是否相等。4 位地址进行符号扩展、移位后与 PC 相加以得到分支目标地址；

(4) 跳转指令：16 位跳转地址[0:0]恒为 0。[12:1]位来自跳转指令的[11:0]位。[15:13]位来自当前 PC+2 的高 4 位；

2.2.2 寄存器及其编码

命名	编号	用途
$s0-$s3	0-3	保存临时值

$t0-$t3	4-7	保存临时变量
$gp	8	全局指针寄存器
$sp	9	栈指针寄存器
$fp	10	帧指针寄存器
$ra	11	返回地址寄存器
$ar	12	地址寄存器
备用	13-14	无
$zero	15	恒 0

表格 2.2-1 MIPS 寄存器

2.2.3 指令功能说明

指令	Op_code(4)	ReDst	Jump	Branch	MemRead	MemtoReg	ALUOp(3)	MemWrite	ALUSrc	RegWrite
MOV	0011	0	0	0	0	0	010	0	1	1
ADD	0010	1	0	0	0	0	010	0	0	1
SUB	0110	1	0	0	0	0	011	0	0	1
AND	0000	1	0	0	0	0	000	0	0	1
OR	0001	1	0	0	0	0	001	0	0	1
SLT	0111	1	0	0	0	0	111	0	0	1
LW	1000	0	0	0	1	1	010	0	1	1
SW	1010	0	0	0	0	0	010	1	1	0
BNE	1110	0	0	1	0	0	011	0	0	0
JMP	1111	0	1	0	0	0	000	0	0	0

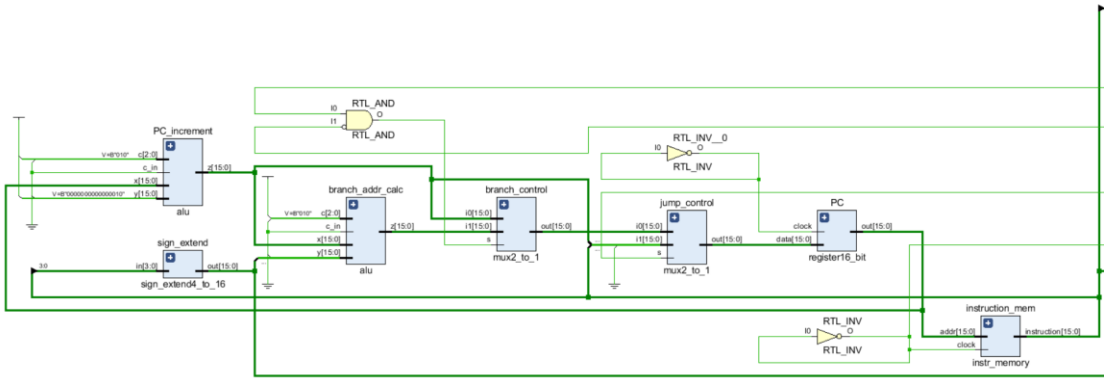
表格 2.2-2 按 OP 码译码产生的控制信号

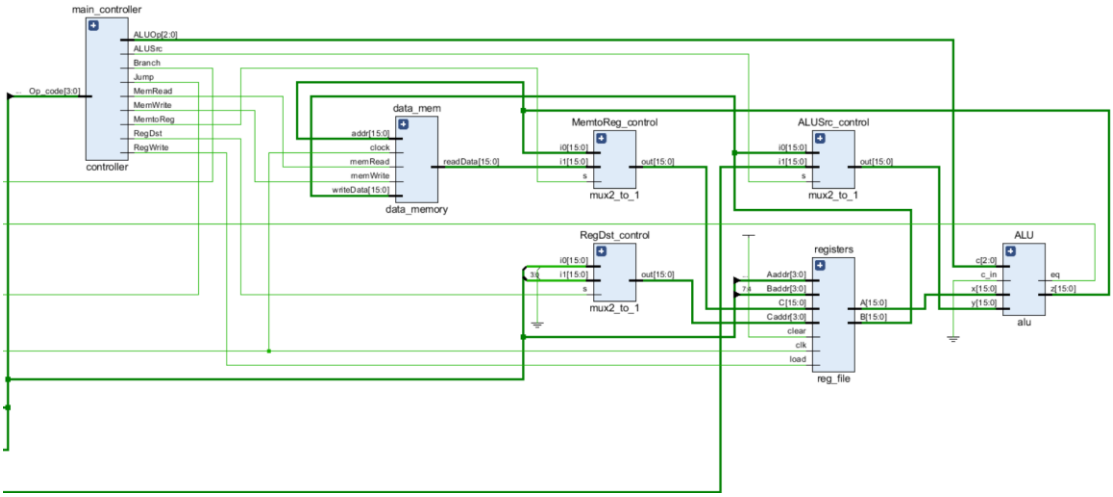
功能示例	含义	注释
MOV $s0,$s1(0002)	R[$s0] = R[$s1] +0002	移动寄存器间的值
ADD $s0,$s1,$s2	R[$s2] = R[$s1]+R[$s0]	三个寄存器操作数
SUB $s0,$s1,$s2	R[$s2] = R[$s1]-R[$s0]	三个寄存器操作数
AND $s0,$s1,$s2	R[$s2] = R[$s1]&R[$s0]	三个寄存器操作数按位与
OR $s0,$s1,$s2	R[$s2] = R[$s1] | R[$s0]	三个寄存器操作数按位或
SLT $s0,$s1,$s2	If(R[$s0]<R[$s1]) R[s2]=1	比较是否小于;用于 BNE
LW $s0,0002($s1)	R[$s0]=MEM[R[$s1]+0002]	将一个字从内存中取寄存器
SW $s0,0002($s1)	MEM [R[$s1]+0002]=R[$s0]	将一个字从寄存器中存到内存中
BNE $s0,$s1,0002	If(R[$s0]!=R[$s1]) go to PC+2+0002	不相等检测；和 PC 相关的跳转
JMP 0002	go to 0004 + [PC+2][15:11]	跳转到目的地址

表格 2.2-3 指令功能

2.3 系统架构

2.3.1 顶层原理图





2.3.2 数据通路

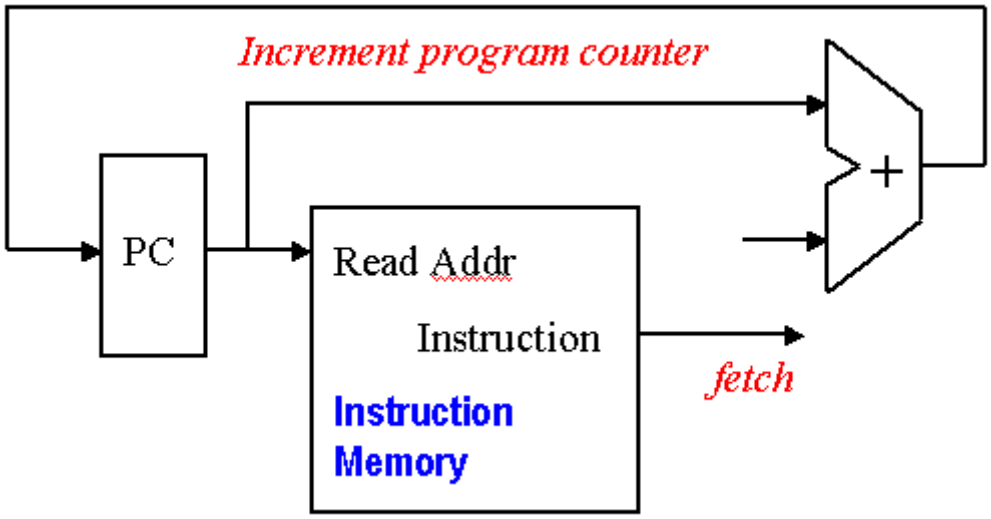


图 2.3-1 存取指令需要的两个状态单元，以及计算下一条指令地址所需要的加法器

两个状态单元分别是指令存储器(Instruction Memory)和程序计数器(PC)。因为数据通路没有写指令，所以指令存储器只提供读访问。因为指令存储器是只读的，我们讲它视为组合逻辑；任意时刻的输出都反应了输入地址处的内容，而不需要读控制信号。程序计数器是一个 16 位的寄存器，它在每个时钟周期末都被写入，所有不需要写控制信号。加法器采用具有加法功能的 ALU，它将输入两个 16 位数相加，将结果输出

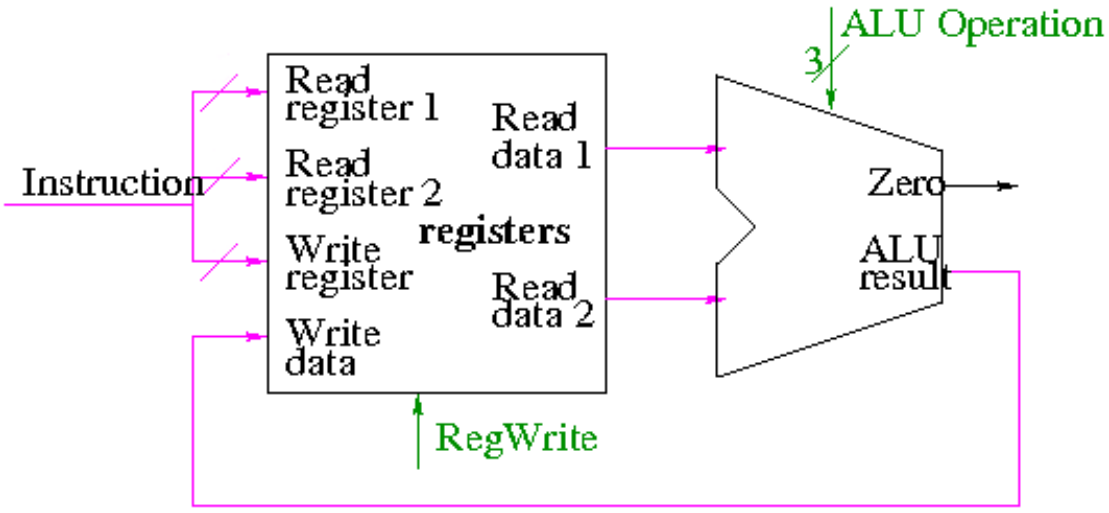


图 2.3-2 实现 R 型指令的 ALU 操作所需的两个单元——寄存器堆和 ALU

寄存器堆包括了所有的寄存器，有两个读端口和一个写端口。寄存器堆的读输出总是对应于读寄存器的号，不需要其他控制信号。但是写寄存器必须明确使能写控制信号。

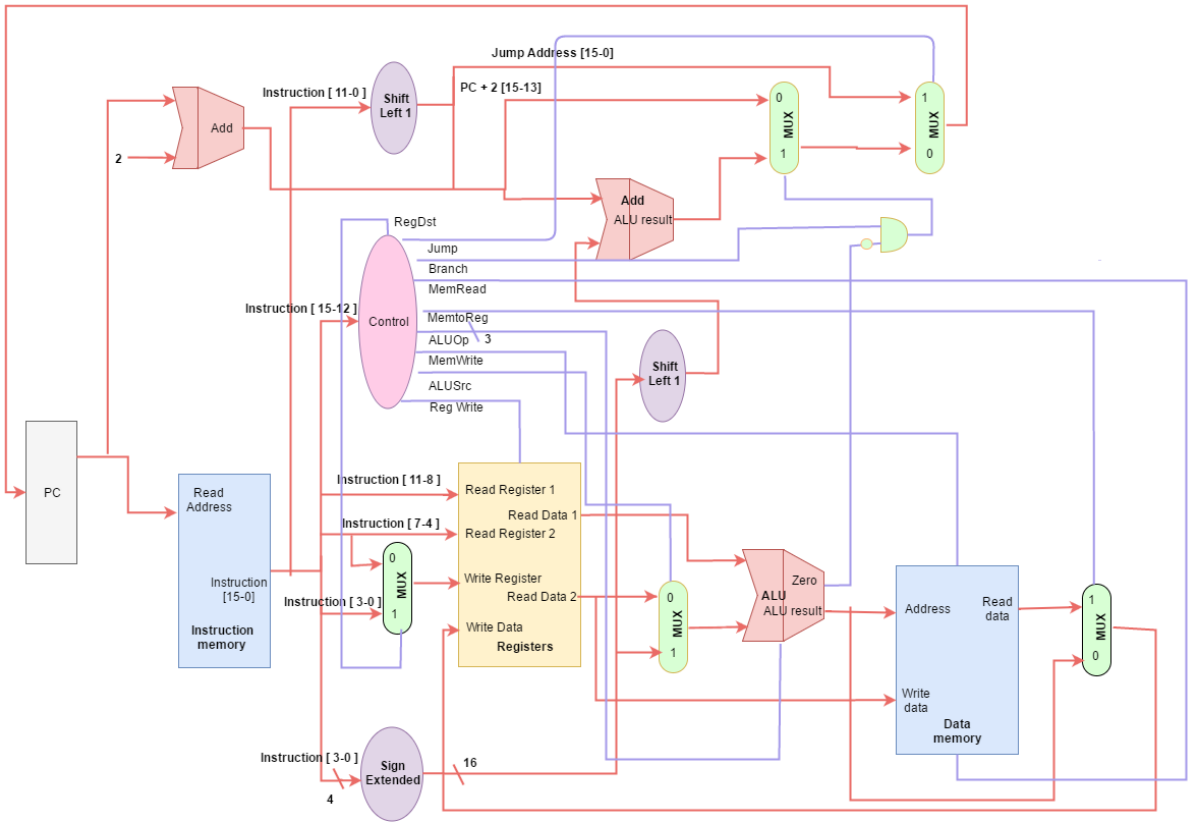


图 2.3-3 数据通路图

将 PC、指令存储器、寄存器堆、主控制器、ALU、MUX 等元件用总线连接成如上数据通路图。为了只用一个 ALU 和一个寄存器堆来创建一个数据通路，

ALU 的第二个输入和要存入寄存器堆的数据都需要两个不同的来源，在 ALU 的输入和寄存器对的输入数据处各加入一个多选器。

由于分支指令用主 ALU 对寄存器操作数进行比较，需要加法器完成分支目标的计算。此外增加了一个多选器，用于选择是顺序的指令地址 PC+2 还是分支目标地址写入 PC。

3. 仿真

3.1 CPU 周期与节拍

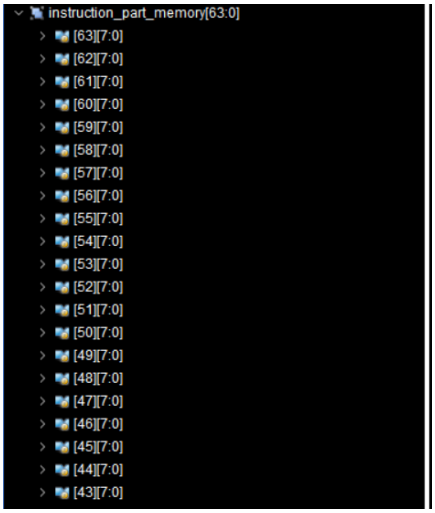
CLOCK 信号控制 CPU 周期，每 10ns 改变一次电平；clk2 控制节拍，每 1ns 改变一次电平；一个周期内含有 5 个节拍。(为了确保取指令、指令译码与读寄存器堆、执行或计算地址、存储器访问、写回五个过程有足够的时间)

图 3.1-1 周期与节拍

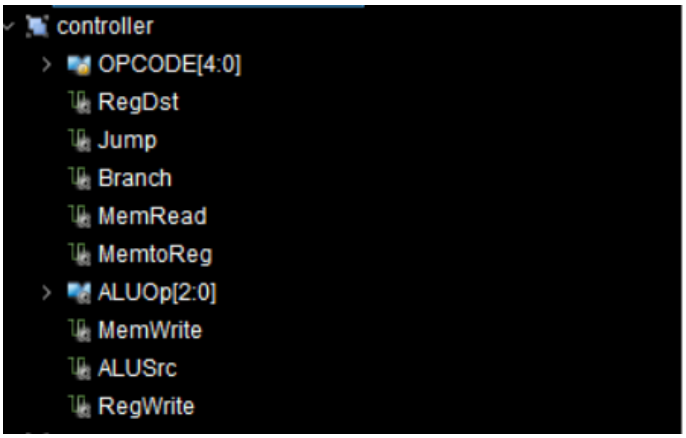
3.2 信号组织

为了方便信号与数据验证，按照数据通路图中各元件的输入与输出，将所有总线分类为 NEXT_PC、PC、instruction、controller、reg_heap、alu、memory 等几大分组，每个分组的信号意义如下：

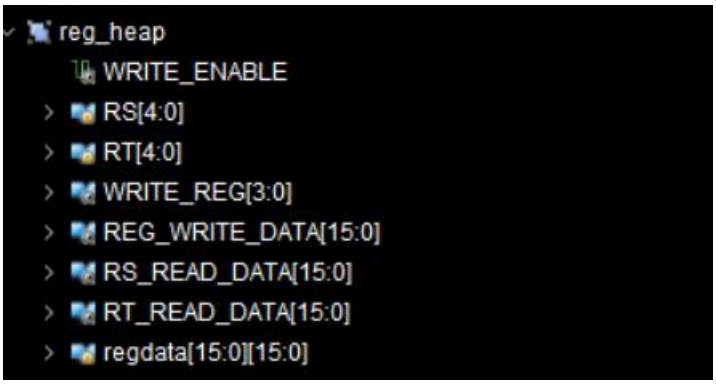
- instruction_part_memory：指令存储器中的[63:0]位的指令内容，按字节编址；(总容量为 1024*8*2)



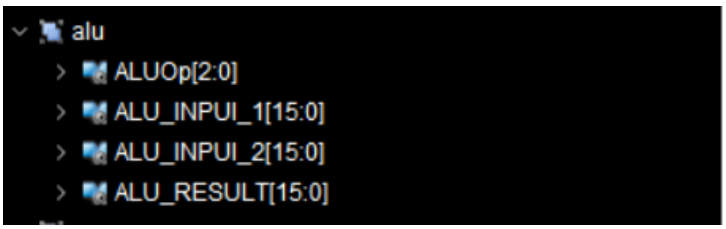
- NEXT_PC：下一条将要执行的指令地址；
- PC：当前执行的指令地址；
- Instruction：当前执行的指令内容；
- controller：控制器的输入与输出信号；



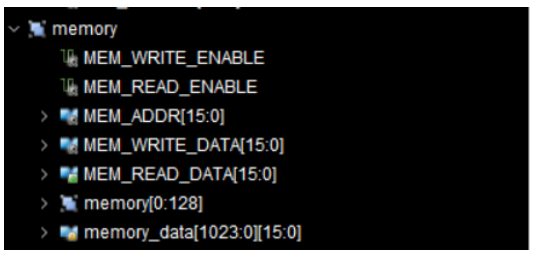
- reg_heap：寄存器堆的输入信号与输出信号，以及每个寄存器的当前值；



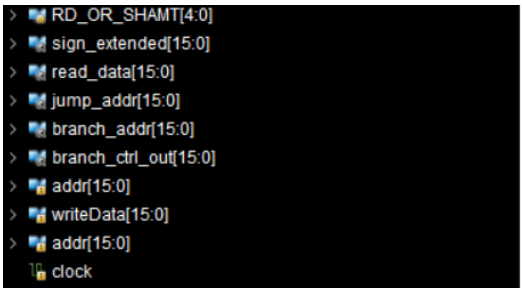
- alu：算术单元的输入信号与输出信号；



- memory：算术单元的输入信号与输出信号；



- 其他：指令低 4 位、跳转地址、分支地址等



3.3 仿真波形

3.3.1 初始化

- 指令存储器的初始化

  指令存储器的 16 进制内容如下(从上往下，大堆排序)：

3F	AC	70
C0	20	12
8C	61	E0
00	02	12
8C	00	00
12	12	…
20	10	F0
12	12	0F

- 数据存储器的初始化

  数据存储器的 16 进制内容如下(从上往下，大堆排序)：

12
34
56
78
9A
BC
DE
F1

- 寄存器的初始化

  仿真开始时，$s0,$s1,$ar 的值分别初始化为 0005、0057、000A；

  为了测试项目设计的 10 种指令，项目预加载以下指令内容到指令存储器中；

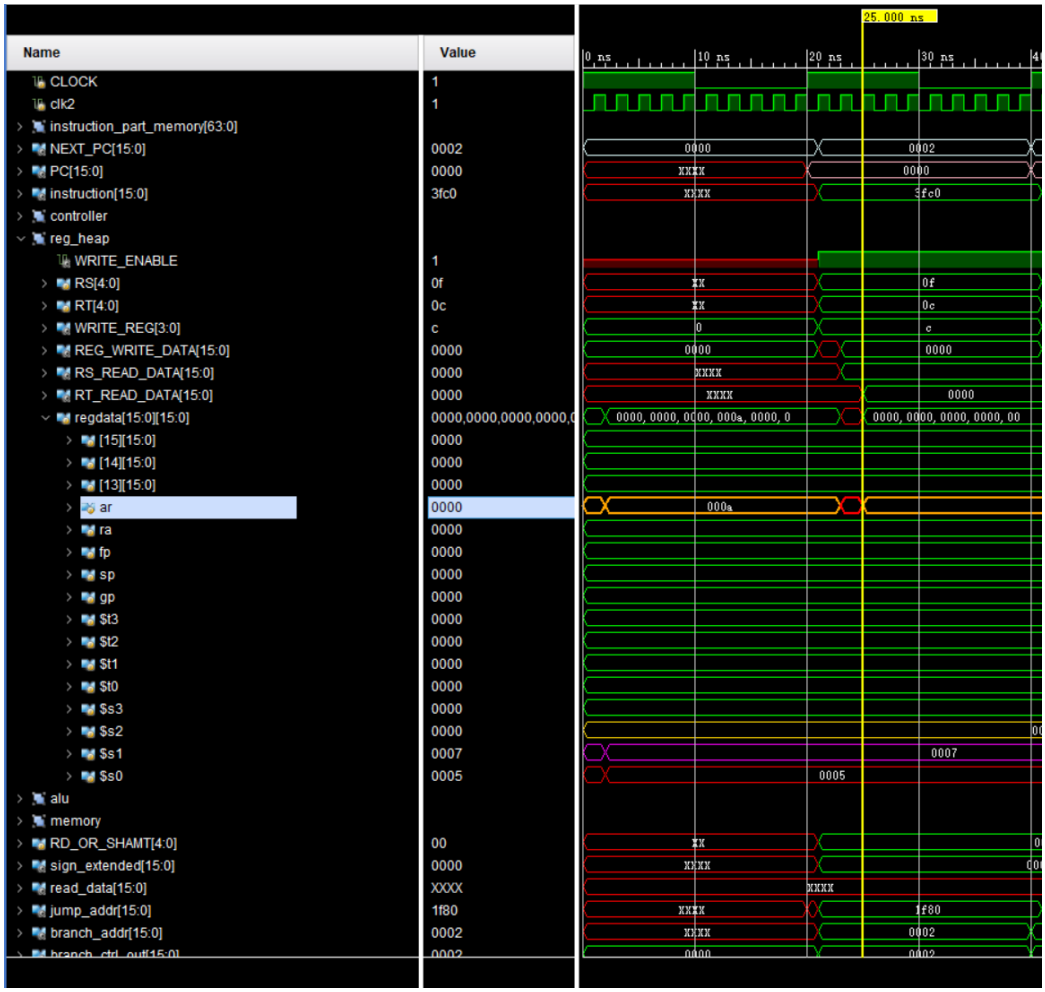
指令	功能	期待值变化
MOVE $ar,$zero	//基地址从 0 开始;	R[$ar] = 0
LW $s0,0000($ar)	//R[$s0] = MEM[R[ar] + 0000];	R[$s0] = 0012
LW $s1,0002($ar)	//R[$s1] = MEM[R[ar] + 0002];	R[$s1] = 0056
ADD $s0,$s1,$s2	//R[$s2] = R[$s0] + R[$s1];	R[$s2] = 0068
SW $s2,$ar(0)	//MEM[R[$ar] + 0]] = R[$s2];	MEM[1:0]=0068
SUB $s1,$s0,$s2	//R[$s2] = R[$s1]-R[$s0];	R[$s2] =0044
AND $s0,$s1,$s2	//R[$s2] = R[$s1] & R[$s0];	R[$s2] = 0012
OR $s0,$s1,$s2	//R[$s2] = R[$s1] | R[$s0];	R[$s2] = 0056
SLT $s0,$s1, $s2	//if(R[$s0] < R[$s1]) R[$s2] = 0001	R[$s2] = 0001
BNE $s0,$s1,2	//if(R[$s0] != R[$s1]) go to PC+2+2;	PC = 0018
JMP 0x0F	//跳转到特定地址	PC = 001E

图 3.3-1 示例指令执行(R 表示取寄存器值，MEM 表示取存储器值)

3.3.2 仿真

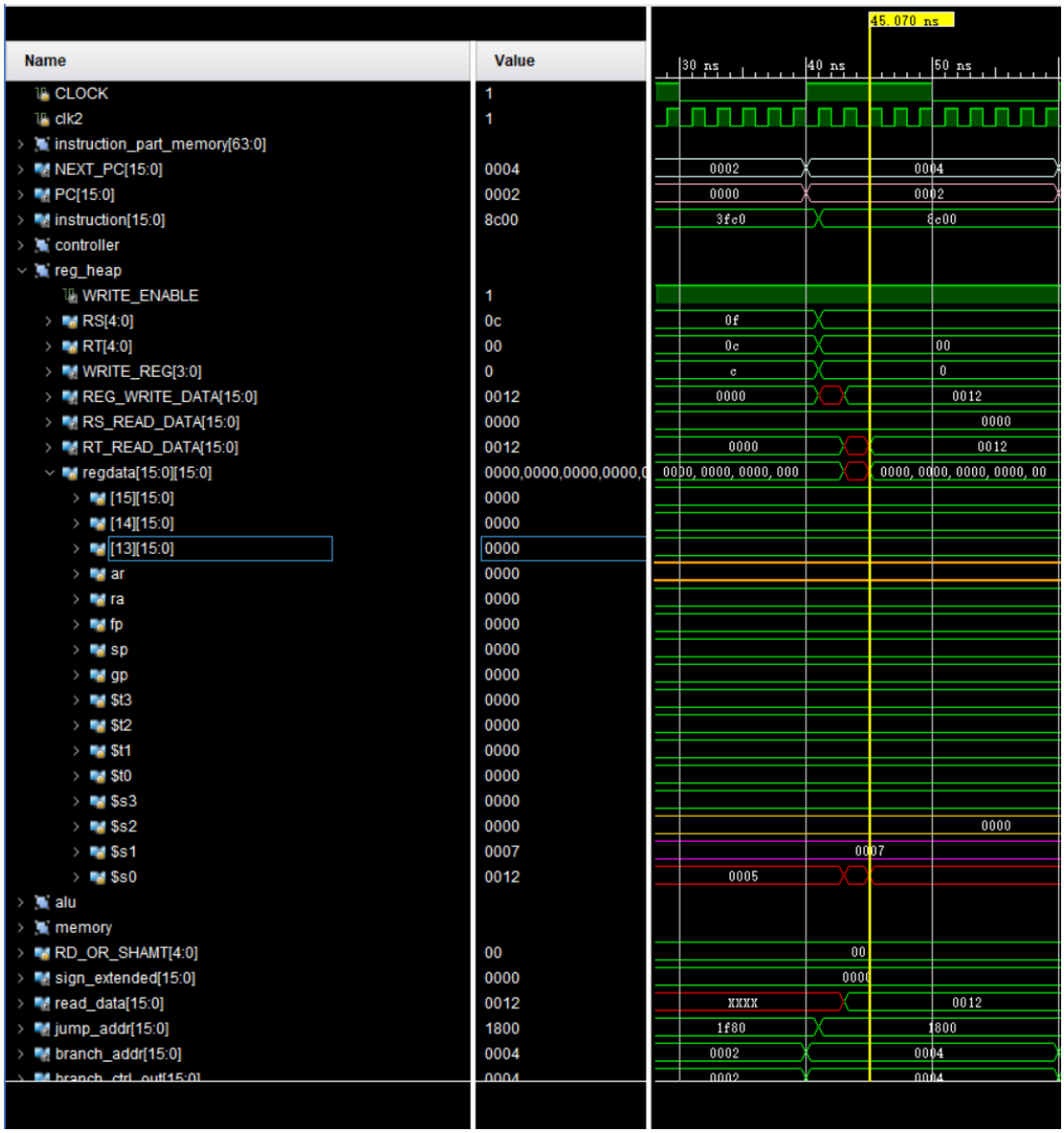
- R 型指令主要展示寄存器值的波形变化；
- 装载型指令主要展示寄存器、数据存储器值的波形变化；
- 跳转、分支型指令主要展示 PC、PC+2 的波形变化

MOVE $ar,$zero	//基地址从 0 开始;



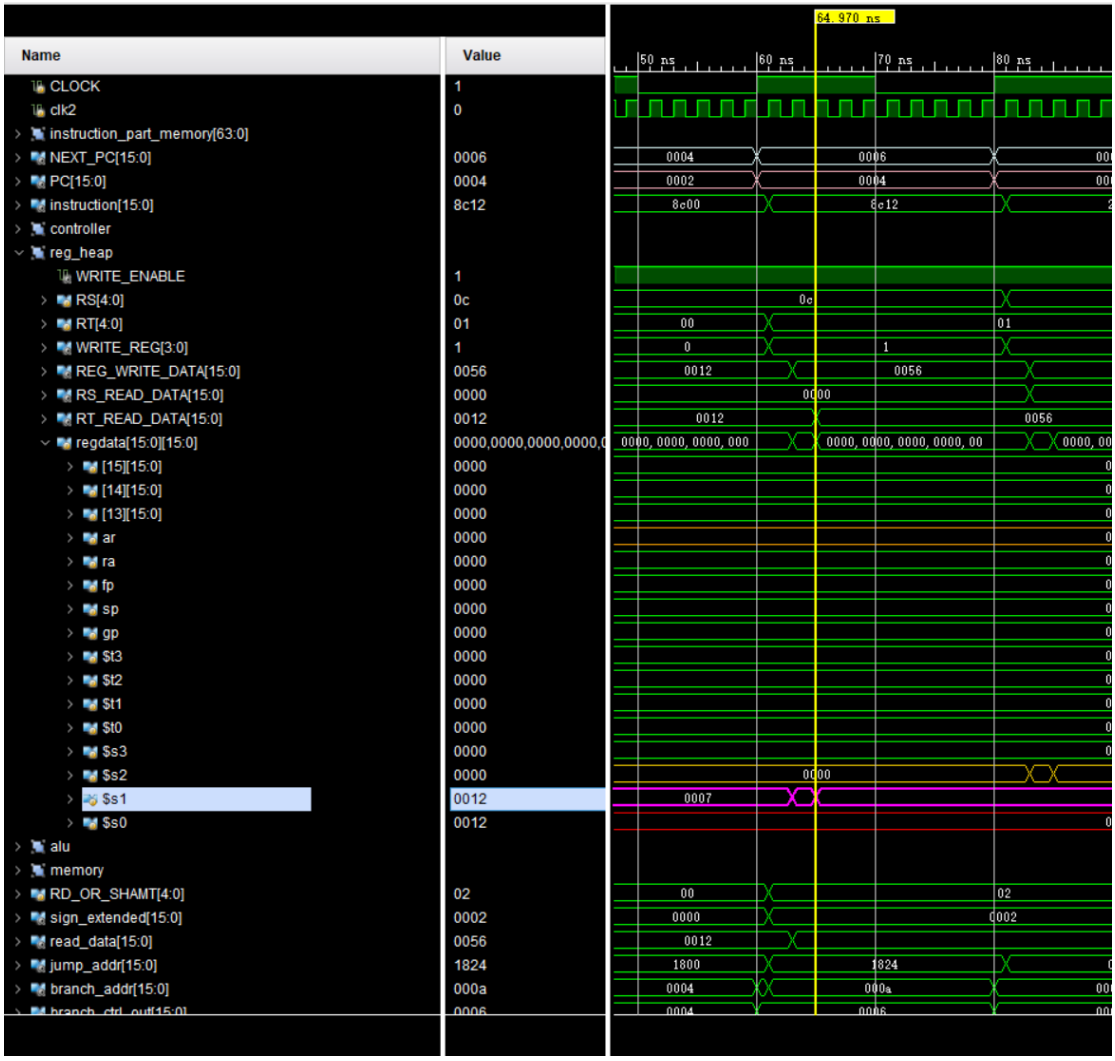
写节拍的上升沿，$ar 的值从 000A 被变为 0000；

LW $s0,0000($ar)	//R[$s0] = MEM[R[ar] + 0000];



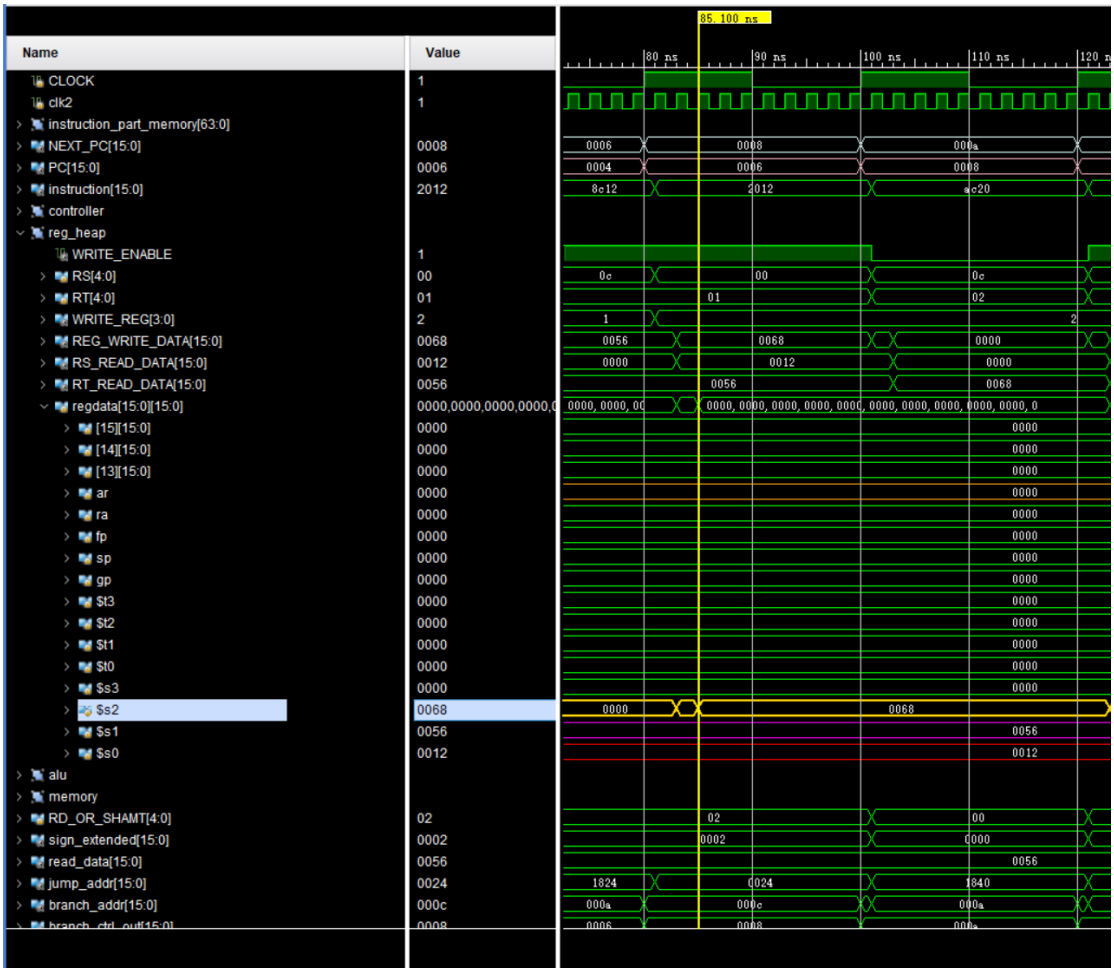
写节拍的上升沿，$s0 的值从 0005 被变为 0012；

LW $s1,0002($ar)	//R[$s1] = MEM[R[ar] + 0002];



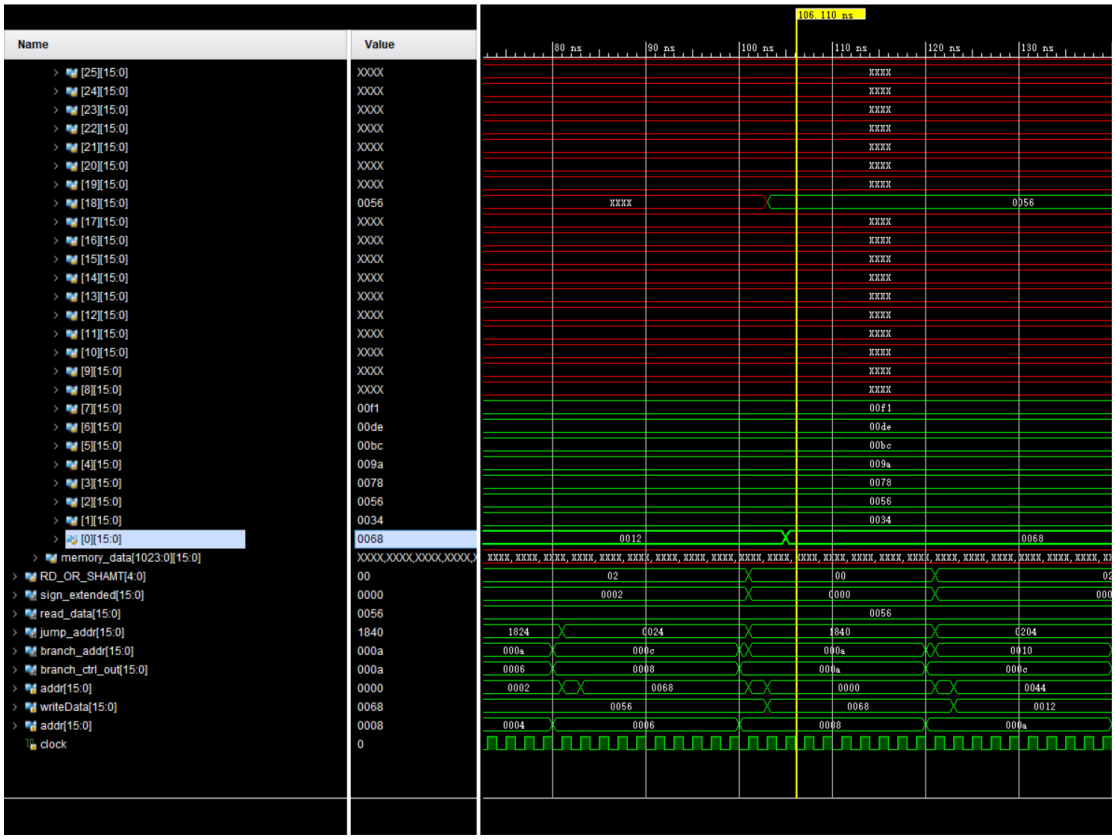
写节拍的上升沿，$s1 的值从 0005 被变为 0012；

ADD $s0,$s1,$s2	//R[$s2] = R[$s0] + R[$s1];



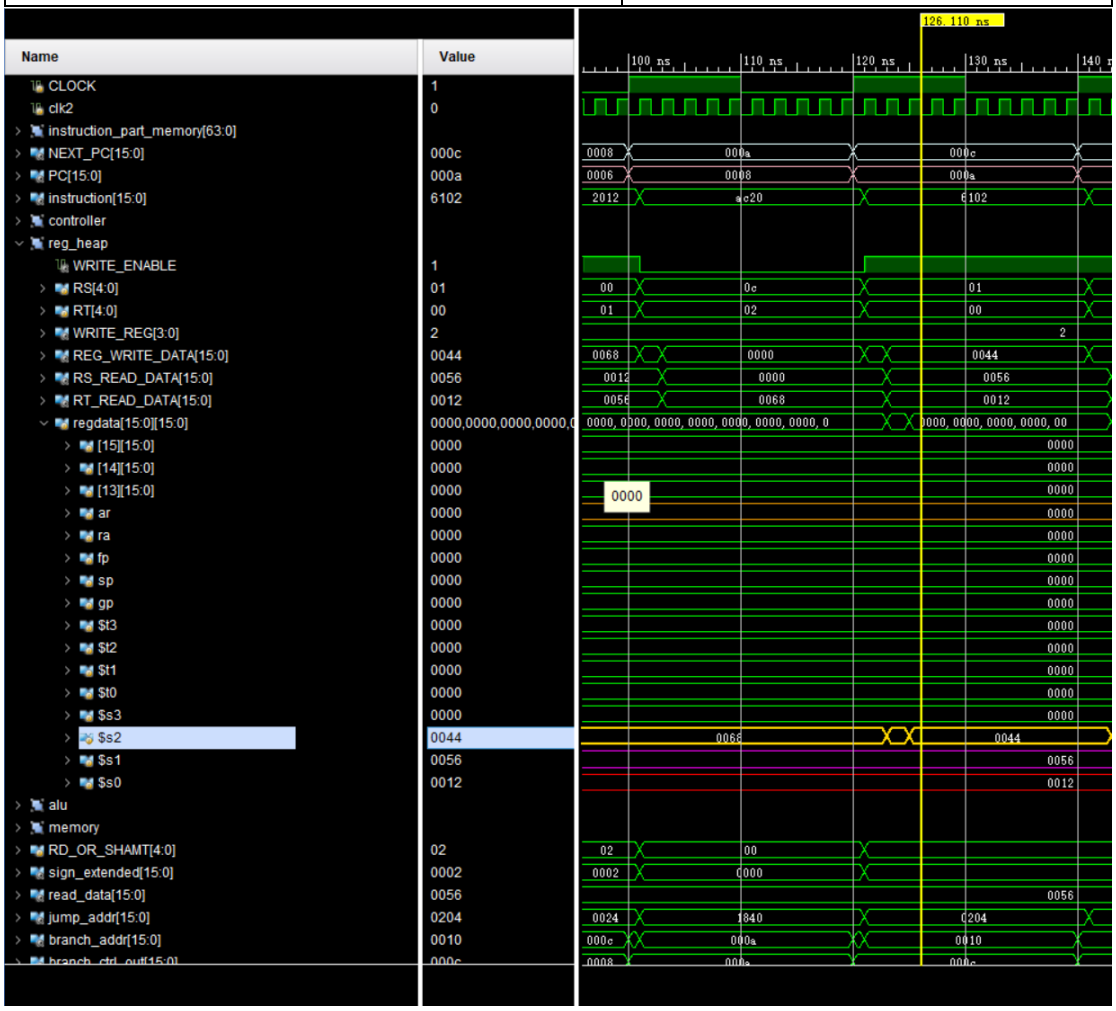
写节拍的上升沿，$s2 的值从 0000 被变为 0068；

SW $s2,$ar(0)	//MEM[R[$ar] + 0]] = R[$s2];



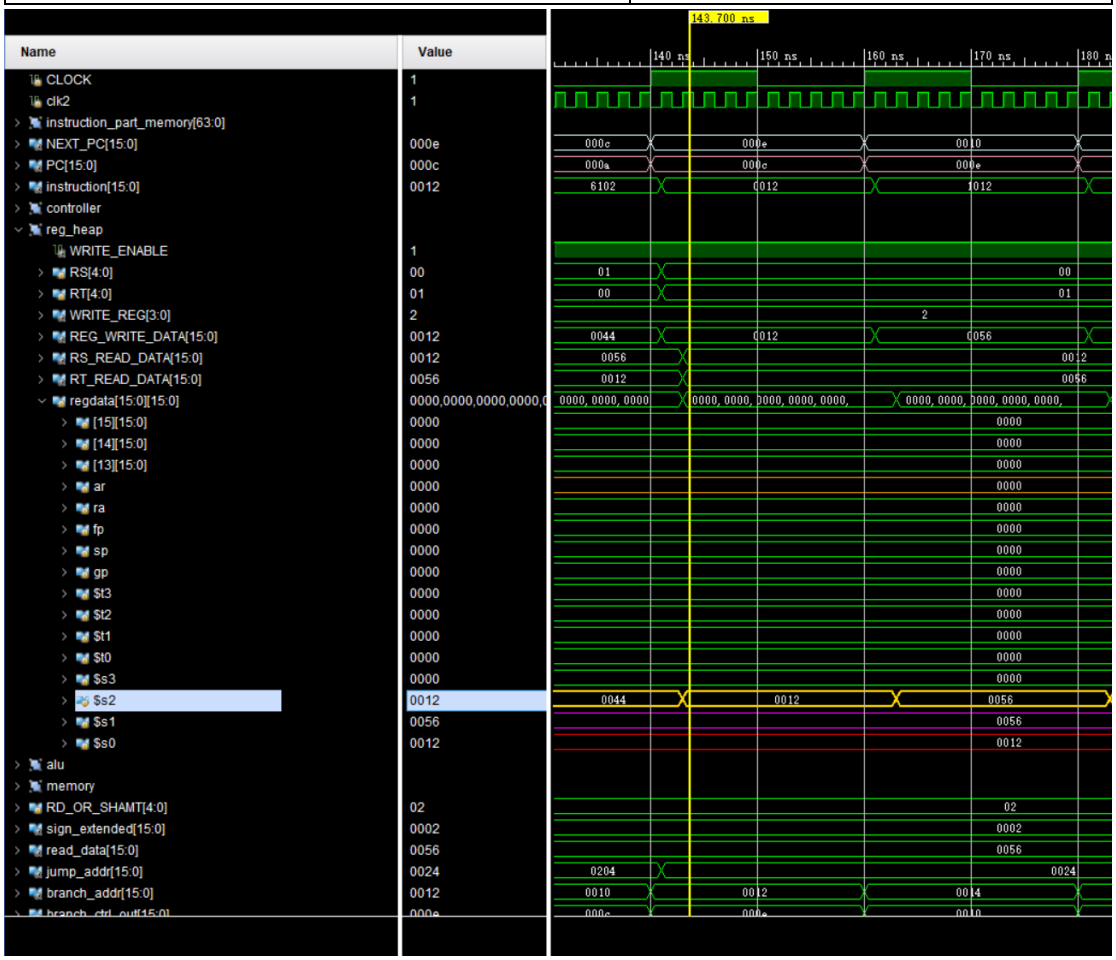
将$s2 内容保存到 MEM[0],MEM[0]值从 0012 变为 0068；

SUB $s1,$s0,$s2	//R[$s2] = R[$s1]-R[$s0];



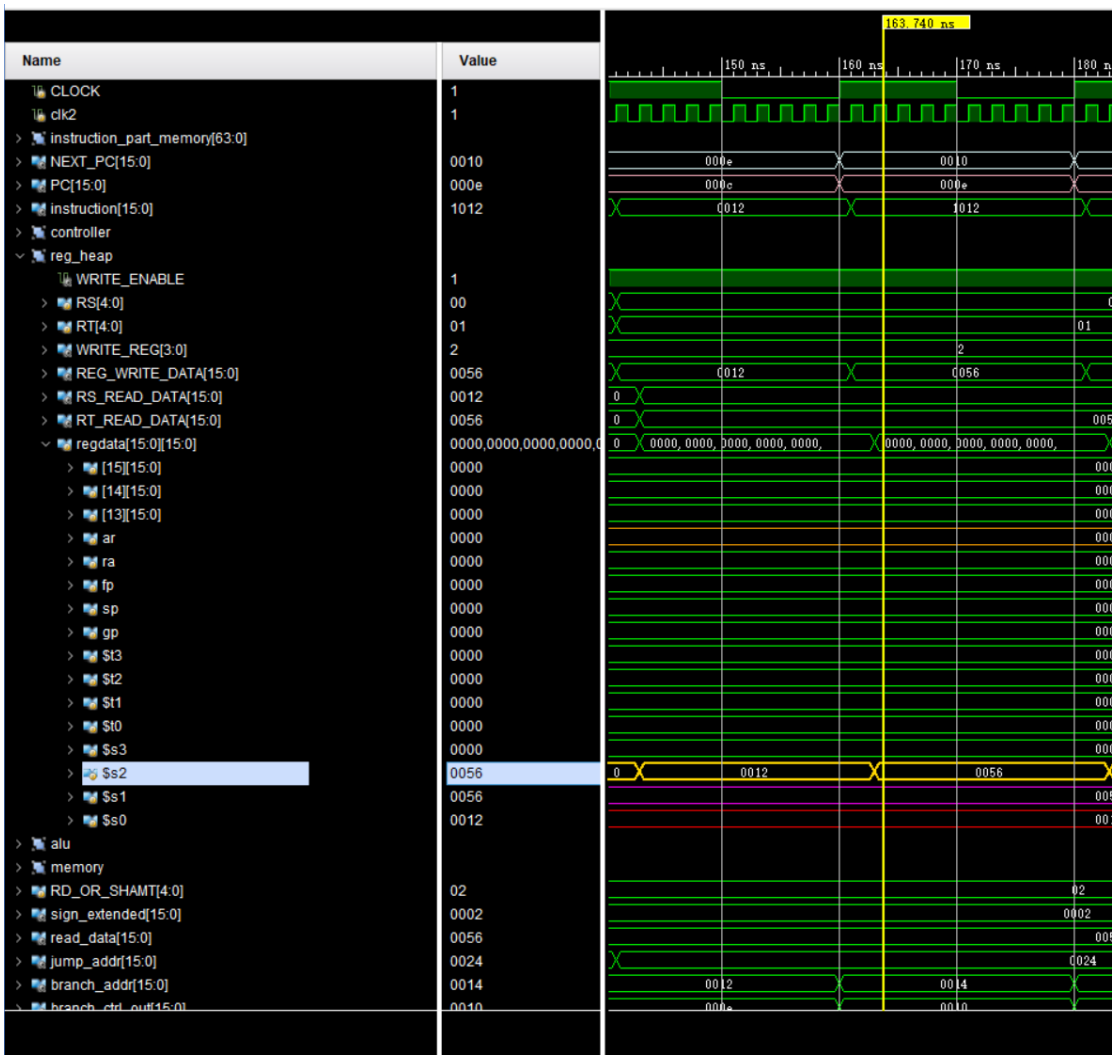
写节拍的上升沿，$s2 的值从 0068 变为 0044；

AND $s0,$s1,$s2	//R[$s2] = R[$s1] & R[$s0];



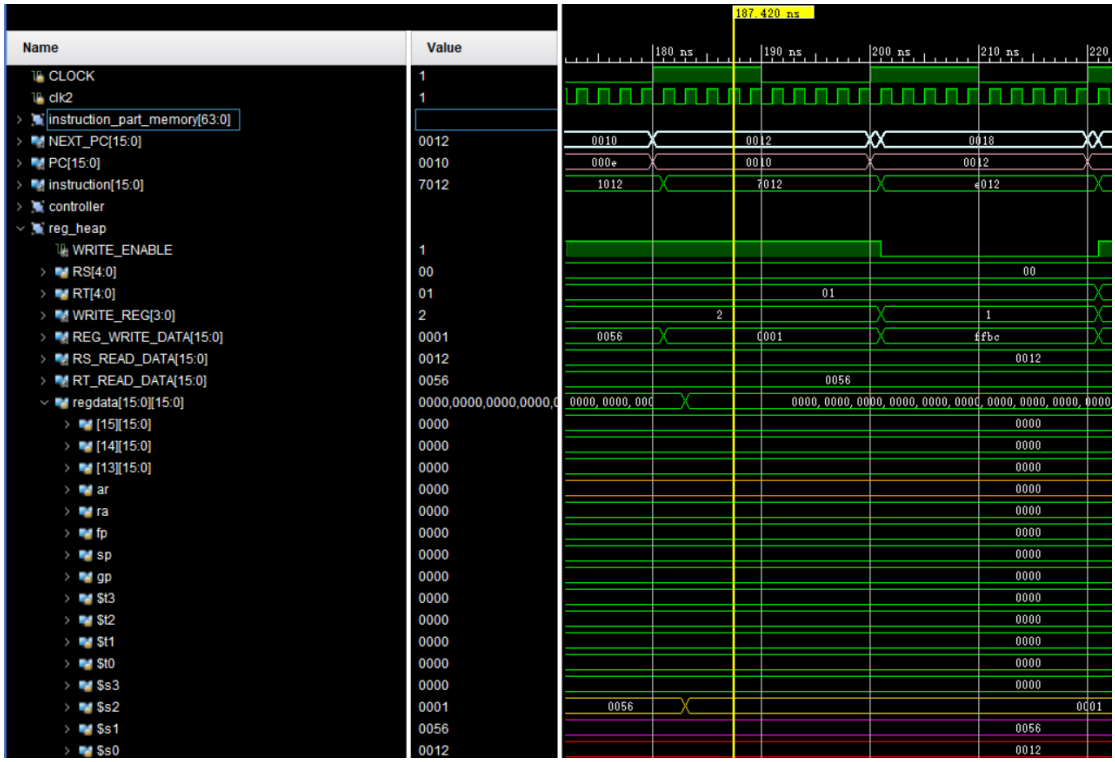
写节拍的上升沿，$s2 的值从 0044 变为 0012；

OR $s0,$s1,$s2	//R[$s2] = R[$s1] | R[$s0];



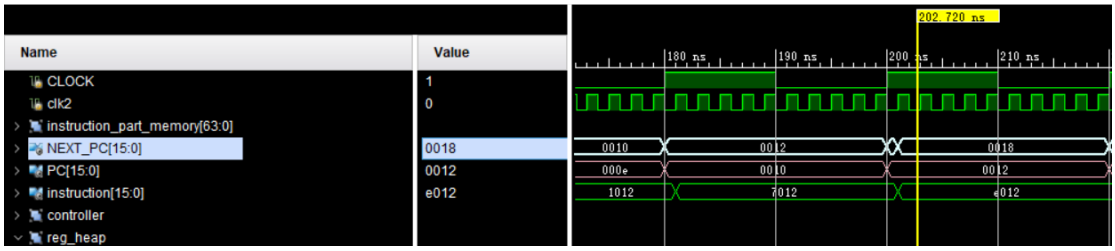
写节拍的上升沿，$s2 的值从 0012 变为 0056；

SLT $s0,$s1, $s2	//if(R[$s0] < R[$s1]) R[$s2] = 0001



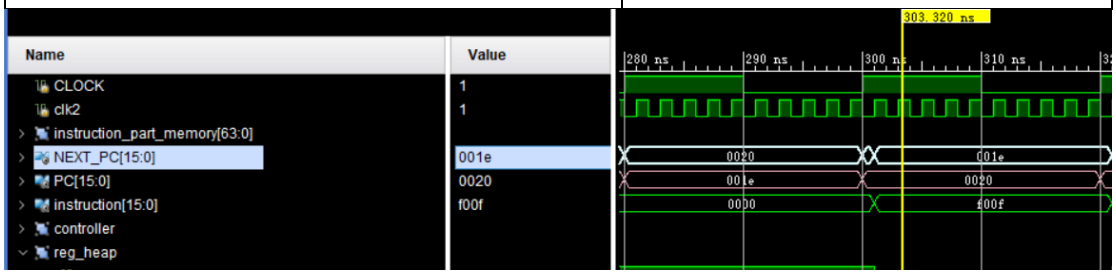
写节拍的上升沿，$s2 的值从 0056 被置为 0001；

BNE $s0,$s1,2	//if(R[$s0] != R[$s1]) go to PC+2+2;



NEXT_PC 的值从原本的 0012 变化为 0018；

JMP 0x0F	//跳转到特定地址



NEXT_PC 的值从原本的 0020 变化为 001E；

4. 总结

4.1 遇到的问题

- 问题：节拍数过少，造成寄存器堆的写回操作不及时

  解决：调整时钟周期是节拍的 10 倍关系，保证每个周期有 5 个上升沿
- 问题：always 块的敏感参数不正确，造成寄存器堆的时序逻辑错误

  解决：将敏感列表改成唯一的 postedge click;
- 问题：控制器译码信号错误，导致指令译码错误

  解决：修改映射真值表；
- 问题：MOVE 指令作用下的寄存器实际值与期待值不对，恒为 XXXX

  解决：修改 MOVE 指令的 OPCODE，译码为正确的二进制指令
- 问题：工程无法运行综合，提示 always 的 postedge 参数只能为一个

  解决：注释掉 postedge clear
- 问题：工程无法仿真，提示缺少顶层文件

  解决：将 CPU 设为顶层文件，作为仿真的入口

4.2 思考

4.2.1 节拍控制

CPU 单周期设计中，时钟周期对所有指令登场，这样时钟周期要由执行时间最长的那条指令决定。其中装载指令是执行时间最长的，它依次使用了 5 个功能单元：指令存储器、寄存器堆、ALU、数据存储器、寄存器堆，所有每个时钟周期的节拍的数目应该适应，让执行时间长的指令能够完成对所有元件的操作。

4.2.2 指令设计

本项目中，使用 16 位指令中的高 4 位作为控制器的输入，产生 7 个控制信号。显然使用 4-16 译码的方法可以产生更多的控制信号，但这样会加重控制器本来的负担。常用的方式是采用多级译码的方法，由主控制单元生成 ALUOp 作为 ALU 控制单元的输入，再由 ALU 控制单元生成真正的控制 ALU 的信号，这样可以拓展出更多的指令，同时也减少了时钟周期。

因为本项目的目的是理解 CPU 设计原理，并非完成一个具备所有 MIPS 指令

集的功能性 CPU，在本次设计中没有使用。

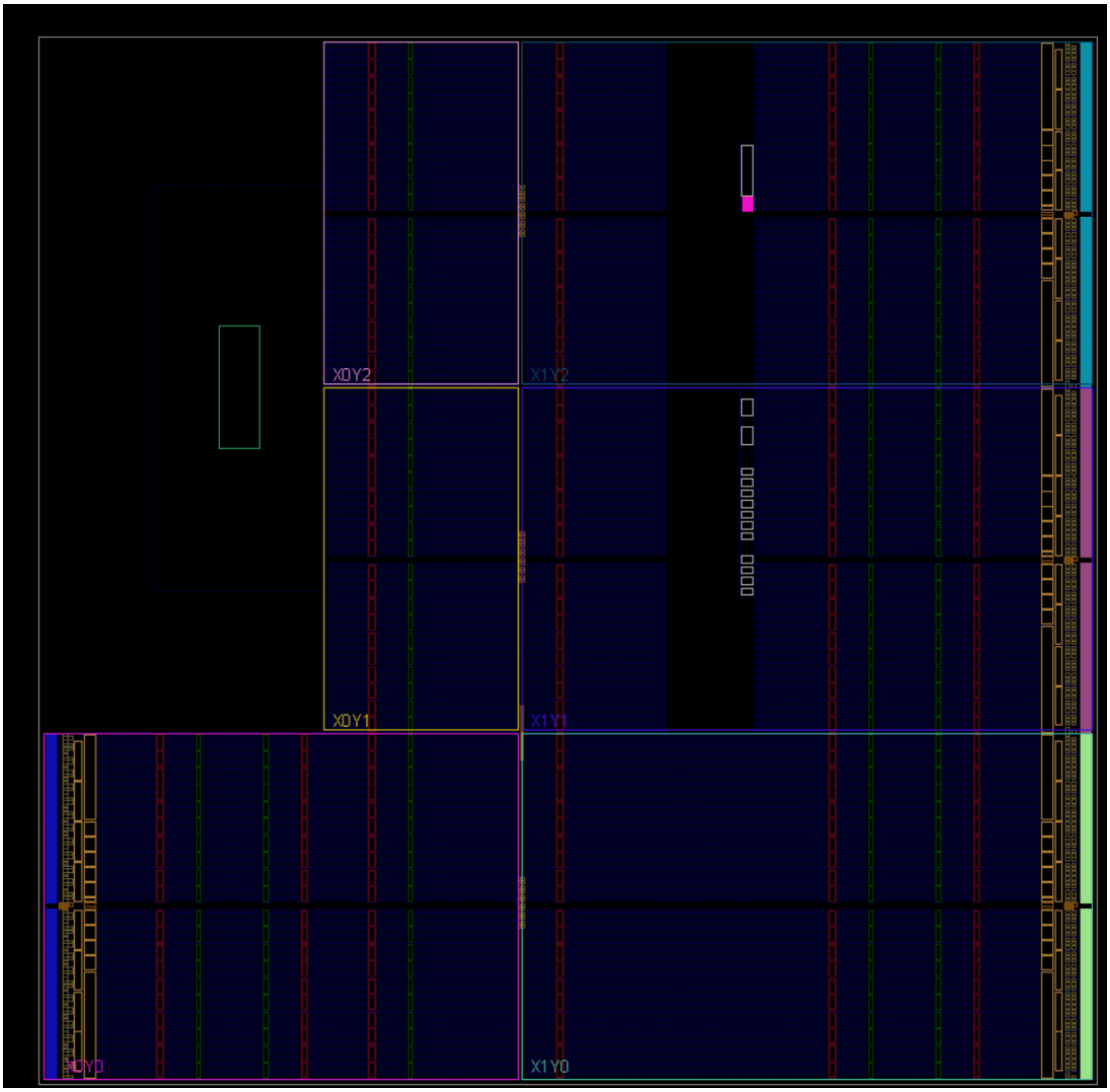


图 4.2-1 Implementation on Project, using Xilinx Vivado

5. 源代码

5.1 指令存储器

```
module instr_memory(addr, clock, instruction);
    input [15:0] addr;
    input clock;
    output reg [15:0] instruction;

    reg [7:0] memory [1023:0]; //this memory is byte addressable
```

```
    //loading some instructions
    initial
    begin
        $readmemh("instructions.mem", memory);
    end

    always @ (posedge clock)
    begin
    //two bytes are read from the memory to get the complete instruction
        instruction <= {memory[addr], memory[addr + 1]};
    end

endmodule
```

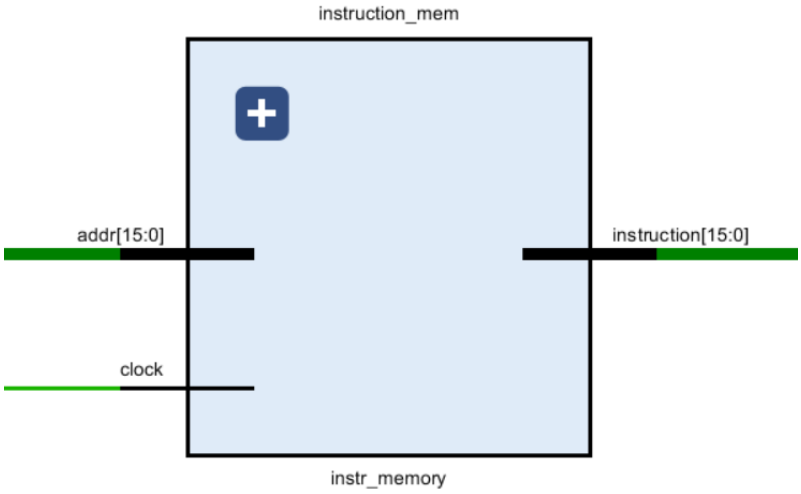


图 5.1-1 指令存储器

5.2 数据存储器

```
module data_memory(addr, memRead, memWrite, writeData, clock,
readData);
    input [15:0] addr;
    input [15:0] writeData;
    input memRead, memWrite, clock;

    output reg [15:0] readData;

    reg [15:0] memory [1023:0]; //this memory is word addresseble
(each word is 16 bits)

    initial
    begin
    $readmemh("data.mem", memory);
//  memory[0] = 16'd2;
//  memory[1] = 16'd10;
    end

    always @ (posedge clock)
        begin
            if (memWrite == 1)
                memory[addr] <= writeData;

            if (memRead == 1)
                readData <= memory[addr];
        end

endmodule
```

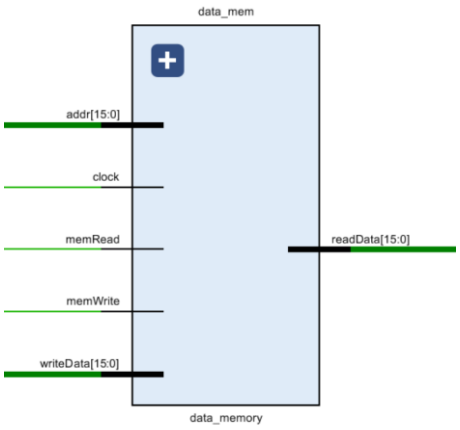


图 5.2-1 数据储存器

5.3 寄存器堆

```
module reg_file(A, B, Aaddr, Baddr, C, Caddr, load, clear, clk);
    input [3:0] Aaddr, Baddr, Caddr;
    input [15:0] C;
    input load, clear, clk;
    output reg [15:0] A, B;

    reg [15:0] regdata [15:0];

    integer i;

    initial
    begin
    i = 0;
   for (i = 0;  i < 16; i = i + 1)
       regdata[i] = 16'd0;
        #2 regdata[0] = 16'd5;
        regdata[1] = 16'd7;
        regdata[12] = 16'd10;
    end
    always
      #1  regdata[15] = 16'd0; //Con Zero Register
    // write
   always @ (posedge clk) begin
       if (load == 1)
       regdata[Caddr] = C;
       end
    always @ (posedge clk)
       begin


//          if (clear == 0)
//             begin
//                i = 0;
//                for (i = 0;  i < 16; i = i + 1)
//                   regdata[i] = 16'd0;
//             end

          A <= regdata[Aaddr];
          B <= regdata[Baddr];
       end
```

```
endmodule
```

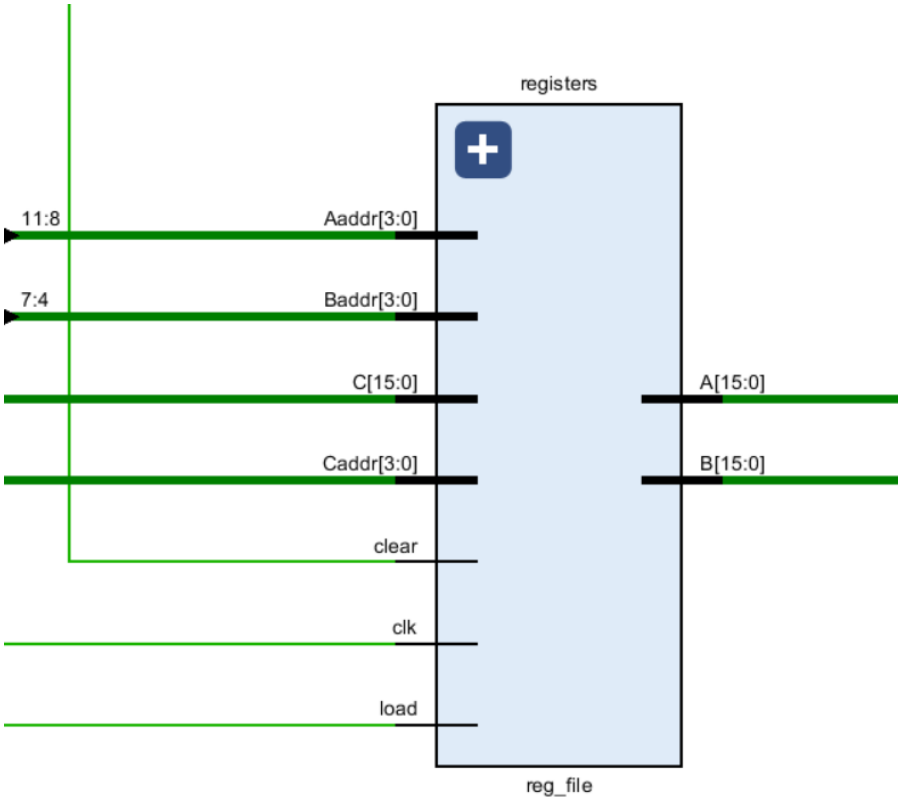


图 5.3-1 寄存器堆

5.4 ALU

```
module alu(x, y, z, c_in, c_out, lt, eq, gt, overflow, c);
    input [15:0] x, y;
    input [2:0] c;
    input c_in;

    output reg [15:0] z;
    output reg c_out, lt, eq, gt, overflow;

    always @(c, x, y, c_in)
    begin
        case (c)
            3'b000 : z = x & y;                    //And
            3'b001 : z = x | y;                    //Or
            3'b010 : begin
                        {c_out, z} = x + y + c_in;  //Add
                        overflow = (x[15] == y[15]) && (z[15] !=
x[15]); //setting overflow bit
```

```
                    end
            3'b011 : begin
                        {c_out, z} = x + ~y + 1;    //Subtract
                        overflow = (x[15] != y[15]) && (z[15] !=
x[15]); //setting overflow bit
                    end
            3'b111 : z = x < y;                     //Set-on-less-than
        endcase

        lt = x < y;     //less than (is x < y?)
        eq = x == y;    //equal (is x = y?)
        gt = x > y;     //greater than (is x > y?)
    end
endmodule
```

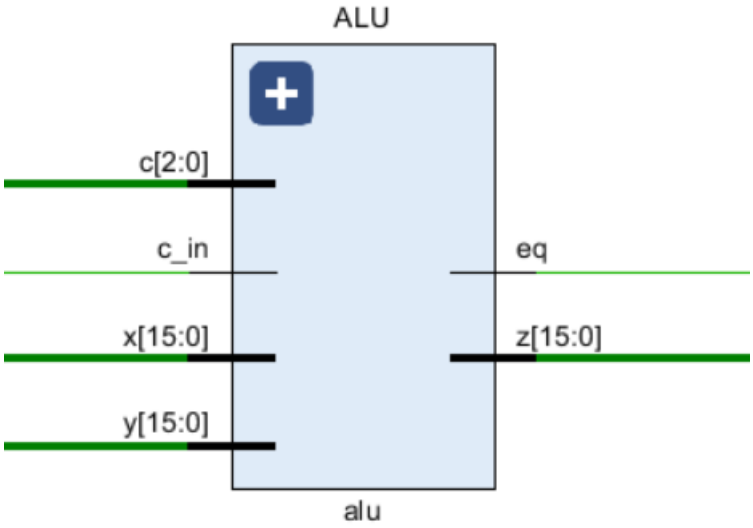


图 5.4-1 ALU

5.5 控制器

```
module controller(Op_code, RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite);
    input [3:0] Op_code;
    output reg RegDst, Jump, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;
    output reg [2:0] ALUOp; //this is to be directly connected to the ALU control input
                            //000 - AND,
                            //001 - OR,
```

```
                            //010 - ADD,
                            //011 - SUB,
                            //111 - SET_ON_LESS_THAN


    always @(Op_code)
    begin
        case (Op_code)
            //MOV instruction
           4'd3 : {RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite} = 11'b00000010011;
            //ADD instruction
            4'd2 : {RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite} = 11'b10000010001;
            //SUB instruction
            4'd6 : {RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite} = 11'b10000011001;
            //AND instruction
            4'd0 : {RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite} = 11'b10000000001;
            //OR instruction
            4'd1 : {RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite} = 11'b10000001001;
            //SLT instruction
            4'd7 : {RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite} = 11'b10000111001;
            //LW instruction
            4'd8 : {RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite} = 11'b00011010011;
            //SW instruction
            4'd10 : {RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite} =11'b00000010110;
            //BNE instruction
            4'd14 : {RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite} =11'b00100011000;
            //JMP instruction
            4'd15 : {RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite} =11'b01000000000;
            default : $display("Invalid signal");
        endcase
    end
endmodule
```

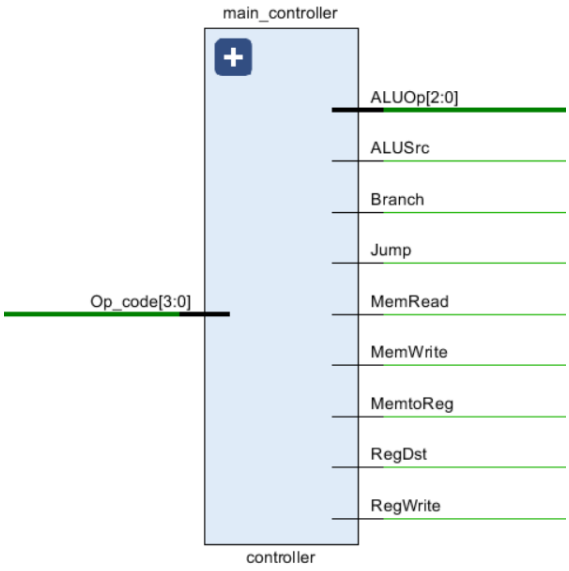


图 5.5-1 主控制器

5.6 2 选 1 选择器

```
module mux2_to_1 (out, i0, i1, s);

    input [15:0] i0, i1;
    input s;

    output reg [15:0] out;

    initial
        out = 16'd0;

    always @(s, i0, i1)
    begin

        case (s)
            1'b0 : out = i0;
            1'b1 : out = i1;
            default : $display("Invalid signal");
        endcase
    end
endmodule
```

5.7 符号扩展

```
module sign_extend4_to_16(in, out);
    input [3:0] in;
    output reg [15:0] out;
    always @(in)
    begin
        out[3:0] = in[3:0];
        out[15:4] = {in[3], in[3], in[3], in[3], in[3], in[3], in[3], in[3], in[3], in[3], in[3], in[3]};
    end
endmodule
```

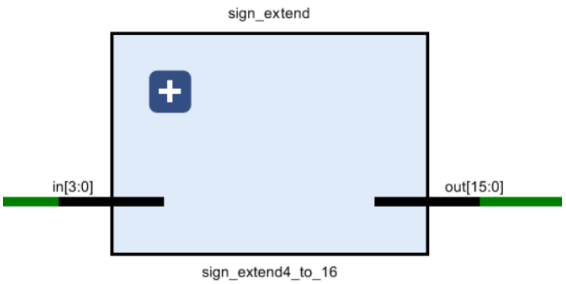


图 5.7-1 符号扩展

5.8 PC

```
module register16_bit(data, clock, out);
    input [15:0] data;
    input clock;
    output reg [15:0] out;

    initial
    begin
        out = 16'd0;
    end

    always @(posedge clock)
    begin
        out = data;
    end
endmodule
```

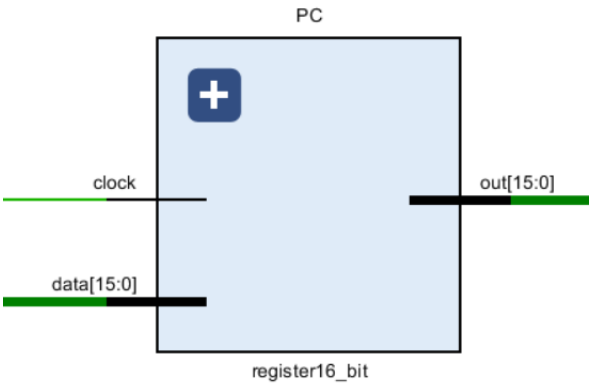


图 5.8-1 PC