华中农业大学课程实验报告

课程名称：计算机组成原理

专业班级：	计科 2102
学　　号：	2021307220712
姓　　名：	高星杰
同组成员：	夏鑫
指导老师：	别丽华
报告日期：	2023/6/2

信息学院计算机科学系

华中农业大学计算机组成原理实验报告原创性声明

本人郑重声明：所呈交的计算机组成原理实验报告时本人独立思考并完成。除了文中特别加以标注的内容外，本报告不包含任何其他个人或集体已经发表或撰写的成果作品。本人完全意识到本声明的法律后果由本人承担。

作者签名：　高星杰　夏鑫

2023 年　6 月　2 日

目录

一、 实验目的

1. 参考提供的实验指导和资料，基于 MIPS 指令集，设计一个 20+的单周期 CPU。
2. 采用 VerilogHDL 语言进行设计，并下载到 FPGA 开发板上。
3. 能跑测试程序，并利用开发板上的显示单元进行指令的跟踪执行。

二、 实验环境

1. 操作系统：Windows7
2. 编译器：Quartus II
3. 芯片：Cydone V: 5CSEMA5F31C6

三、 实验设计与步骤

1、 设计基本器件

● 算术逻辑单元（alu）

```
module alu(a_,b_,aluc_,r,z,v);
         output wire v;
         input [31:0] a_,b_;
         input [3:0] aluc_;
         output [31:0] r;
         output z;
         wire [31:0] d_and=a_&b_;
         wire [31:0] d_or=a_|b_;
         wire [31:0] d_xor=a_^b_;
         wire [31:0] d_lui={b_[15:0],16'h0};
         wire [31:0] d_and_or=aluc_[2]?d_or:d_and;
         wire [31:0] d_xor_lui=aluc_[2]?d_lui:d_xor;
         wire [31:0] d_as/*synthesis keep*/;
         wire [31:0] d_sh/*synthesis keep*/;
         assign v = ~aluc_[2] & ~a_[31] & ~b_[31] & r[31] & ~aluc_[1] & ~aluc_[0] |
          ~aluc_[2] &  a_[31] &  b_[31] &~r[31] & ~aluc_[1] & ~aluc_[0] |
                                  aluc_[2] & ~a_[31] &  b_[31] & r[31] & ~aluc_[1] & ~aluc_[0]
|aluc_[2] &  a_[31] & ~b_[31] &~r[31] & ~aluc_[1] & ~aluc_[0] ;
         addsub32 as32(a_,b_,aluc_[2],d_as);
         shift shifter(b_,a_[4:0],aluc_[2],aluc_[3],d_sh);
         mux4x32 select(d_as,d_and_or,d_xor_lui,d_sh,aluc_[1:0],r);
         assign z=~|r;
endmodule
```

● 寄存器组

```
/*
该模块为寄存器组
输入有 rf_clk(时钟), rw ra rb(5 位 选择寄存器地址); busW(32 位 写入寄存器内容); RegWr(1 位 写使能)
输出有 busA busB(32 位 输出寄存器内容)
*/
module regfile(rf_clk, rw, ra, rb, busW, RegWr, busA, busB,reset);
input [4:0] rw, ra, rb;
input [31:0] busW;
input RegWr, rf_clk,reset;
output [31:0] busA, busB;
reg [31:0] registers [0:31];
always @(posedge rf_clk)
begin
          if(RegWr) //写入寄存器
                    begin
                              registers[rw] = busW;
                    end
end
assign busA = registers[ra];
assign busB = registers[rb];
endmodule
```

- 拓位器

```
//符号拓展器 16-30
//根据 flag 1 是符号拓展
//根据 flag 0 是零拓展
//其他为异常情况
module extendedThirty(in,out,flag);
input [15:0]in;
output reg [29:0]out;
input flag;
always@(*)
begin
          case(flag)
          1'b1:if(in[15]==1)out={{14{1'b1}},in};
                              else out={{14{1'b0}},in};
          1'b0:out = {{14{1'b0}},in};
          endcase
end
endmodule
//符号拓展器 16-32
//根据 flag 1 是符号拓展
//根据 flag 0 是零拓展
//其他为异常情况
module extendedThirtyTwo(in,out,flag);
input [15:0]in;
output reg [31:0]out;
input flag;
always@(*)
begin
          case(flag)
          1'b1:if(in[15]==1)out={{16{1'b1}},in};
                              else out={{16{1'b0}},in};
          1'b0:out = {{16{1'b0}},in};
          endcase
end
endmodule
```

- 选择器

```
/*
该模块为 5 位 2 选 1 选择器
输入有 in0, in1(5 位 待选数据); ctr(1 位 选择信号)
输出有 out(5 位 选择的数据)
*/
module mux2x5(in0, in1, ctr, out);
input [4:0] in1, in0;
input ctr;
```

```
output [4:0] out;
assign out = ctr?in1:in0;
Endmodule
/*
该模块为 32 位 2 选 1 选择器
输入有 in0, in1(32 位 待选数据); ctr(1 位 选择信号)
输出有 out(32 位 选择的数据)
*/
module mux2x32(in0, in1, ctr, out);
input [31:0] in1, in0;
input ctr;
output [31:0] out;
assign out = ctr?in1:in0;
Endmodule

module mux4x32 (a0,a1,a2,a3,s,y);
          input  [31:0] a0,a1,a2,a3;
          input  [1:0]  s;
          output [31:0] y;

          function  [31:0] select;
                    input [31:0] a0,a1,a2,a3;
                    input [1:0] s;
                    case  (s)
                              2'b00: select = a0;
                              2'b01: select = a1;
                              2'b10: select = a2;
                              2'b11: select = a3;
                    endcase
          endfunction
          assign y = select(a0,a1,a2,a3,s);
endmodule
```

- 移位器

```
module shift (d,sa,right,arith,sh);
     input [31:0] d;
     input [4:0] sa;
     input       right,arith;
     output [31:0] sh;
     reg [31:0]    sh;
     always @* begin
          if (!right) begin
              sh = d << sa;
          end else if (!arith) begin
              sh = d >> sa;
          end else begin
              sh = $signed(d) >>> sa;
          end
     end
endmodule
```

- 指令存储器 ROM

```
/*
该模块为读取指令模块
输入为 addr(5 位 指令地址)
输出为 inst(32 位 指令)
*/
module scinstmem (a,inst);
          input [31:0] a;
          output [31:0] inst;
          wire [31:0] rom [0:48];
/* 不带异常的程序    assign rom[5'h00] = 32'h3c010000;        // (00)main: lui r1,0
          assign rom[5'h01] = 32'h34240050;        // (04)      ori r4,r1,80
          assign rom[5'h02] = 32'h20050004;        // (08)      addi r5,r0, 4
          assign rom[5'h03] = 32'h0c000018;        // (0c)call: jal sum     .0110.0000
          assign rom[5'h04] = 32'hac820000;        // (10)      sw r2,0(r4)
          assign rom[5'h05] = 32'h8c890000;        //        (14)                lw
          r9,       0(r4)
          assign rom[5'h06] = 32'h01244022;        //        (18)                sub
          r8,       r9.       r4
          assign rom[5'h07] = 32'h20050003;        //        (1c)                addi
          r5,       r0.       3
          assign rom[5'h08] = 32'h20a5ffff;        //        (20)loop2:addi      r5,
          r5,       -1
          assign rom[5'h09] = 32'h34a8ffff;        //        (24)                ori
          r8,       r5,       0xffff
          assign rom[5'h0A] = 32'h39085555;        //        (28)                xori
          r8.       r8,       0x5555
          assign rom[5'h0B] = 32'h2009ffff;        //        (2c)                addi
          r9,       r0,       -1
          assign rom[5'h0C] = 32'h312affff;        //        (30)                andi
          r10,      r9,       0xffff
          assign rom[5'h0D] = 32'h01493025;        //        (34)                or
          r6.       r10,      r9
          assign rom[5'h0E] = 32'h01494026;        //        (38)                xor
          r8,       r10,      r9
          assign rom[5'h0F] = 32'h01463824;        //        (3c)                and
          r7,       r10,      r6
          assign rom[5'h10] = 32'h10a00001;        //        (40)                beq
          r5,       r0,       shift
          assign rom[5'h11] = 32'h08000008;        //        (44)                j
          loop2
          assign rom[5'h12] = 32'h2005ffff;        //        (48)shift:addi      r5.
          r0,       -1
          assign rom[5'h13] = 32'h000543c0;        //        (4c)                sll
          r8.       r5.       15
          assign rom[5'h14] = 32'h00084400;        //        (50)                sll
          r8,       r8,       16
          assign rom[5'h15] = 32'h00084403;        //        (54)                sra
          r8,       r8,       16
          assign rom[5'h16] = 32'h000843c2;        //        (58)                srl
          r8.       r8.       15
          assign rom[5'h17] = 32'h08000017;        //        (5c)finish:         j
          finish
          assign rom[5'h18] = 32'h00004020;        //        (60)sum:  add       r8,
          r0,       r0
          assign rom[5'h19] = 32'h8c890000;        //        (64)loop: lw        r9,
          (r4)
          assign rom[5'h1A] = 32'h20840004;        //        (68)                addi
          r4,       r4,       4
          assign rom[5'h1B] = 32'h01094020;        //        (6c)                add
          r8,       r8,       r9
          assign rom[5'h1C] = 32'h20a5ffff;        //        (70)                addi
          r5,       r5,       -1
          assign rom[5'h1D] = 32'h14a0fffb;        //        (74)                bne
          r5,       r0,       loop
          assign rom[5'h1E] = 32'h00081000;        //        (78)                sll
          r2f       r8f       0
          assign rom[5'h1F] = 32'h03e00008;        //        (7c)                jr
          r31                 0000 0011 1110 0000*/

          assign  rom[6'h00] = 32'h0800_001d; //带异常的程序
          assign  rom[6'h01] = 32'h0000_0000; //
          assign  rom[6'h02] = 32'h401a_6800; //
          assign  rom[6'h03] = 32'h335b_000c; //
          assign  rom[6'h04] = 32'h8f7b_0020;
          assign    rom[6'h05] = 32'h0000_0000;
          assign    rom[6'h06] = 32'h0360_0008;
          assign    rom[6'h07] = 32'h0000_0000;
          assign    rom[6'h0C] = 32'h0000_0000;
          assign    rom[6'h0D] = 32'h4200_0018;
          assign    rom[6'h0E] = 32'h0000_0000;
          assign    rom[6'h0F] = 32'h0000_0000;
          assign    rom[6'h10] = 32'h401a_7000;
          assign    rom[6'h11] = 32'h235a_0004;
          assign    rom[6'h12] = 32'h409a_7000;
          assign    rom[6'h13] = 32'h4200_0018;
          assign    rom[6'h14] = 32'h0000_0000;
          assign    rom[6'h15] = 32'h0000_0000;
          assign    rom[6'h16] = 32'h0800_0010;
          assign    rom[6'h17] = 32'h0000_0000;
          assign    rom[6'h1A] = 32'h0000_0000;
          assign    rom[6'h1B] = 32'h0800_0010;
          assign    rom[6'h1C] = 32'h0000_0000;
          assign    rom[6'h1D] = 32'h2008_000f;
          assign    rom[6'h1E] = 32'h4088_6000;
          assign    rom[6'h1F] = 32'h8c08_0048;

          assign    rom[6'h20] = 32'h8c09_004c;
          assign    rom[6'h21] = 32'h0109_4020;
          assign    rom[6'h22] = 32'h0000_0000;
          assign    rom[6'h23] = 32'h0000_000c;
          assign    rom[6'h24] = 32'h0000_0000;
          assign    rom[6'h25] = 32'h0128_001a;
          assign    rom[6'h26] = 32'h0000_0000;
          assign    rom[6'h27] = 32'h3404_0050;
          assign    rom[6'h28] = 32'h2005_0004;
          assign    rom[6'h29] = 32'h0000_4020;
          assign    rom[6'h2a] = 32'h8c89_0000;
          assign    rom[6'h2b] = 32'h2084_0004;
          assign    rom[6'h2c] = 32'h0109_4020;
          assign    rom[6'h2d] = 32'h20a5_ffff;
          assign    rom[6'h2e] = 32'h14a0_fffb;
          assign    rom[6'h2f] = 32'h0000_0000;
          assign    rom[6'h30] = 32'h0800_0030;
          assign inst = rom[a[7:2]];
endmodule
```

- 数据存储器 RAM

```
module scdatamem(WrEn,Adr,Clk,DataIn,DataOut);
input Clk;
input WrEn;
input [31:0] Adr, DataIn;//地址是 32 位
output [31:0] DataOut;
reg [31:0] data [31:0];
assign DataOut=data[Adr[6:2]];
always@(posedge Clk)
```

```
begin
          if(WrEn)
          data[Adr[6:2]]=DataIn;

end
integer i;
initial begin
          for (i = 0;i < 32;i = i + 1)
                    data[i] = 0;
          data[5'h08] = 32'h0000_0030;//异常处理程序的地址
          data[5'h09] = 32'h0000_003c;
          data[5'h0a] = 32'h0000_0054;
          data[5'h0b] = 32'h0000_0068;
          data[5'h12] = 32'h0000_0002;
          data[5'h13] = 32'h7fff_ffff;
          data[5'h14] = 32'h0000_00A3;
          data[5'h15] = 32'h0000_0027;
          data[5'h16] = 32'h0000_0079;
          data[5'h17] = 32'h0000_0115;
end
endmodule
```

2、 分析指令（确定要实现的功能）

```
 sll/srl/sra    rd, rt, sa  # rd <--rt   shift sa
```

这是 3 条移位指令(Shift Left/ Right Logical/Arithmetic)，5 位的 sa (Shift Amount)指定移位的位数。

```
lui rt,  imm  # rt <--  imm << 16
```

lui (Load Upper Immediate)指令把 16 位立即数 imm 左移 16 位，
存入 rt 寄存器。它与 ori 指令合作，可以为一个 32 位的寄存器赋任意值：lui 赋高 16 位，ori 赋低 16 位。

```
addi  rt, rs, imm # rt <-- rs+ imm(符号扩展)
```

addi (Add Immediate)是立即数的加法指令。注意目的寄存器号是 t，立即数要符号扩展到 32 位。因为是符号扩展，因此 MIPS 指令系统中没有类似于 subi 这样的指令。

```
andi/ori/xori  rt，rs, imm # rt<-- rs op imm(零扩展)
```

这 3 条是逻辑操作指令(And/Or/Xor Immediate)，
因此立即数要零扩展。

```
lW  rt, offset (rs) # rt <-- memory[rs + offset ]
```

lw (Load Word)是- -条取存储器字的指令。寄存器 rs 的内容与符号扩展的 offset 相加，得到存储器地址。从存储器取来的数据存人 rt 寄存器。注意，offset 就是前面讲的立即数。

```
SW rt, offset (rs) # memory[rs + offset ]  <-- rt
```

SW(Store Word)是一条存字的指令，与 lw 方向相反，把 rt 寄存器的内容放入存储器。存储器地址的计算与 lw 相同。

```
beq rs, rt, label # if (rs==rt)  PC <-- label
```

beq (Branch on Equal)是一-条条件转移指令。当寄存器 rs 的内容与寄存器 rt 的内容相等时，转移到 label。如果程序计数器 PC 是 beq 指令的地址，则 label= PC+4 +offset << 2。offset 左移两位导致 PC 的最低两位永远是 0，这是因为 PC 是字节地址而一条指令要占 4 个字节。offset 是要符号扩展的，因此 beq 能实现向前和向后两种转移。

```
bne rs,rt,label # if(rs != rt)PC<-- label
```

与 beq 类似，但 bne (Branch on Not Equal)是在两个寄存器的内容不相等时转移。

```
 j target # PC <-- target
```

j(Jump)是一条跳转指令。target 是跳转的目标地址，32 位，由 3 部分组成:最高 4 位来自于 PC+4 的高 4 位，中间 26 位是指令中的 address，最低两位为 0。这条指令在生成目标地址时不需要任何电路进行计算，只需把 3 部分地址拼接起来就行。以下的两条指令也不需要计算。

```
jal  target #  r31<-,一  PC + 8; PC <-- target
```

jal (Jump and Link)指令与 j 类似， 但要把返回地址保存在 r31 中。即 jal 是子程序调用指令。jal 下一条指令的地址是 PC+4, 为什么返回地址是 PC+ 8?这是因为 MIPS 指令系统实现流水线的延迟转移功能，详见第 8 章。注意，寄存器号 31 是约.定好的，该号码并不出现在指令中。因此在设计电路时，应当由硬件为 jal 指令产生这个号码。

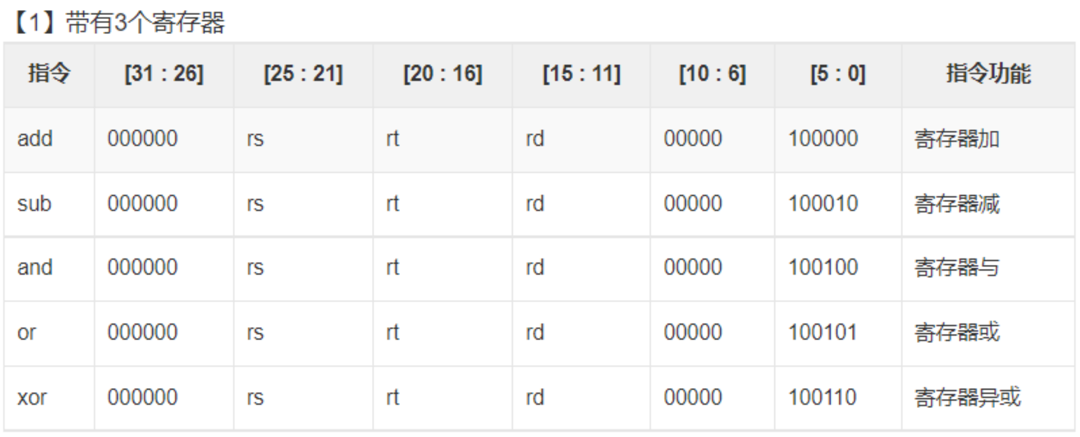
```
jr rs #PC<-- rs
```

jr (Jump Register)也是一-条跳转指令，它把 rs 寄存器的内容写入 PC。如果指定 rs 为 31，则 jr 是从子程序返回的指令。

到此，基本指令格式与功能分析完毕

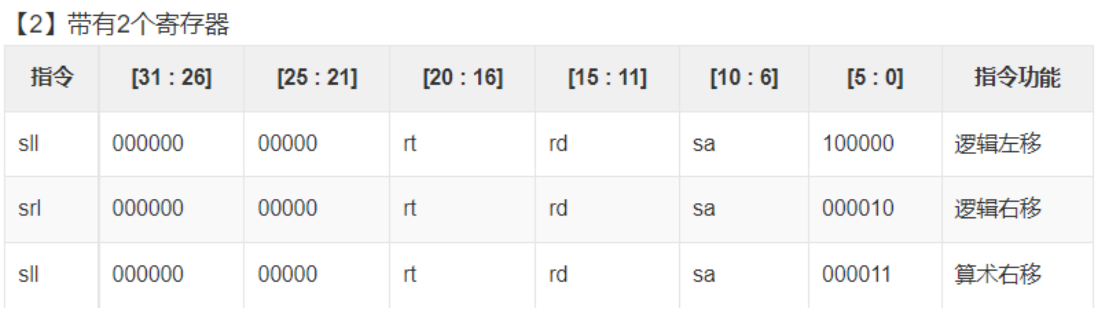
3、 根据指令设计基本器件之间的连接方式（数据通路）

1. 首先看 add 指令可以当实现了这条指令后带有 3 个寄存器的 r 型指令就基本实现了


【1】带有3个寄存器

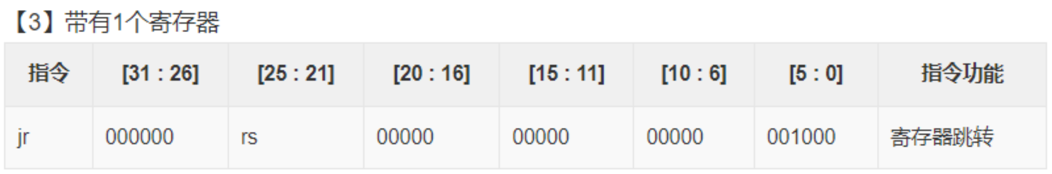
指令	[31 : 26]	[25 : 21]	[20 : 16]	[15 : 11]	[10 : 6]	[5 : 0]	指令功能
add	000000	rs	rt	rd	00000	100000	寄存器加
sub	000000	rs	rt	rd	00000	100010	寄存器减
and	000000	rs	rt	rd	00000	100100	寄存器与
or	000000	rs	rt	rd	00000	100101	寄存器或
xor	000000	rs	rt	rd	00000	100110	寄存器异或

2. 然后再实现带有两个寄存器的 r 型指令


【2】带有2个寄存器

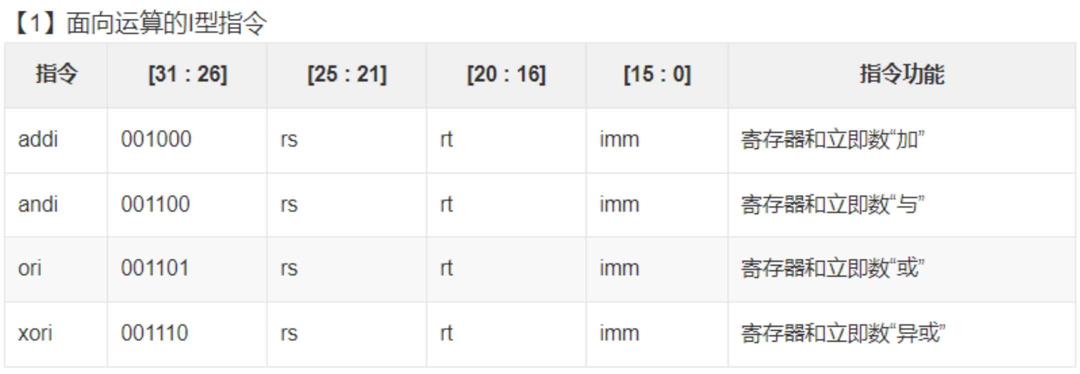
指令	[31 : 26]	[25 : 21]	[20 : 16]	[15 : 11]	[10 : 6]	[5 : 0]	指令功能
sll	000000	00000	rt	rd	sa	100000	逻辑左移
srl	000000	00000	rt	rd	sa	000010	逻辑右移
sll	000000	00000	rt	rd	sa	000011	算术右移

3.之后再实现带有一个寄存器的 r 型指令


【3】带有1个寄存器

指令	[31 : 26]	[25 : 21]	[20 : 16]	[15 : 11]	[10 : 6]	[5 : 0]	指令功能
jr	000000	rs	00000	00000	00000	001000	寄存器跳转

4.最后再实现 I 型指令和 J 型指令


【1】面向运算的I型指令

指令	[31 : 26]	[25 : 21]	[20 : 16]	[15 : 0]	指令功能
addi	001000	rs	rt	imm	寄存器和立即数"加"
andi	001100	rs	rt	imm	寄存器和立即数"与"
ori	001101	rs	rt	imm	寄存器和立即数"或"
xori	001110	rs	rt	imm	寄存器和立即数"异或"

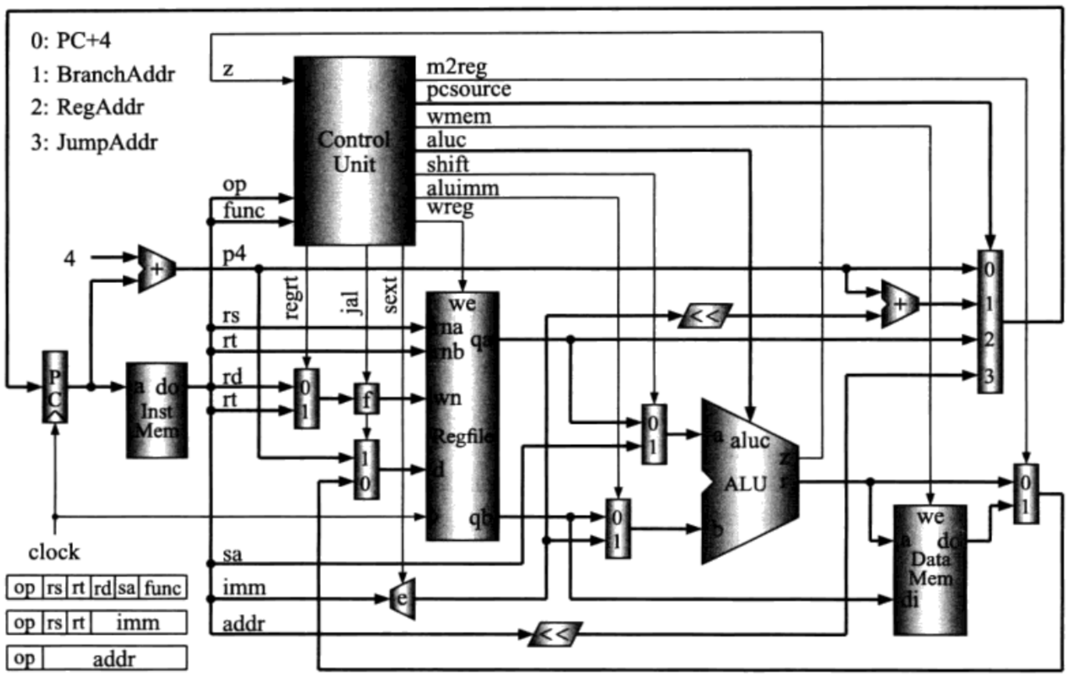
【2】面向访存的I型指令

指令	[31 : 26]	[25 : 21]	[20 : 16]	[15 : 0]	指令功能
lw	100011	rs	rt	imm	从存储器种读取数据
sw	101011	rs	rt	imm	把数据保存到存储器

（2）具体J型指令

指令	[31 : 26]	[25 : 0]	指令功能
j	000010	address	无条件跳转
jal	001100	address	调用与联接

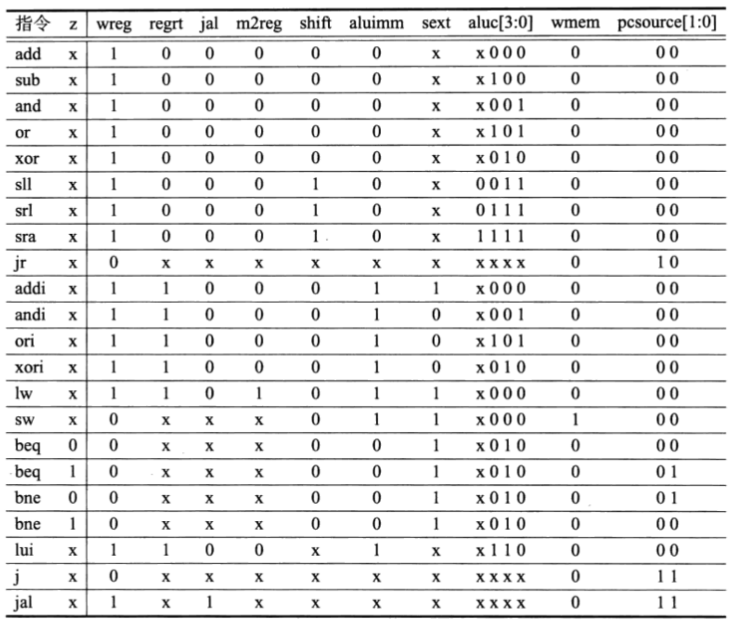
根据他们具体的功能就可以逐渐实现数据通路的图



具体每条指令数据通路图[1]的设计就不再重复了，就是跟踪一条指令从开始到结束的具体过程。

其中就包括设计控制信号产生器。

表 5.3　控制信号的真值表

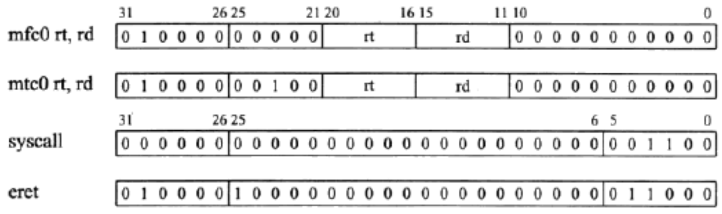


指令	z	wreg	regrt	jal	m2reg	shift	aluimm	sext	aluc[3:0]	wmem	pcsource[1:0]
add	x	1	0	0	0	0	0	x	x 0 0 0	0	0 0
sub	x	1	0	0	0	0	0	x	x 1 0 0	0	0 0
and	x	1	0	0	0	0	0	x	x 0 0 1	0	0 0
or	x	1	0	0	0	0	0	x	x 1 0 1	0	0 0
xor	x	1	0	0	0	0	0	x	x 0 1 0	0	0 0
sll	x	1	0	0	0	1	0	x	0 0 1 1	0	0 0
srl	x	1	0	0	0	1	0	x	0 1 1 1	0	0 0
sra	x	1	0	0	0	1	0	x	1 1 1 1	0	0 0
jr	x	0	x	x	x	x	x	x	x x x x	0	1 0
addi	x	1	1	0	0	0	1	1	x 0 0 0	0	0 0
andi	x	1	1	0	0	0	1	0	x 0 0 1	0	0 0
ori	x	1	1	0	0	0	1	0	x 1 0 1	0	0 0
xori	x	1	1	0	0	0	1	0	x 0 1 0	0	0 0
lw	x	1	1	0	1	0	1	1	x 0 0 0	0	0 0
sw	x	0	x	x	x	0	1	1	x 0 0 0	1	0 0
beq	0	0	x	x	x	0	0	1	x 0 1 0	0	0 0
beq	1	0	x	x	x	0	0	1	x 0 1 0	0	0 1
bne	0	0	x	x	x	0	0	1	x 0 1 0	0	0 1
bne	1	0	x	x	x	0	0	1	x 0 1 0	0	0 0
lui	x	1	1	0	0	x	1	x	x 1 1 0	0	0 0
j	x	0	x	x	x	x	x	x	x x x x	0	1 1
jal	x	1	x	1	x	x	x	x	x x x x	0	1 1

我们在设计控制信号产生器使用的是类似设计微程序一样的，面向指令写控制信号，每条指令对应控制信号的值不同[1]（详情请看代码文件）。

4、　设计中断与异常的总体思路（设计要实现的功能）

1.首先搞清楚中断和异常（溢出异常、外部中断、系统中断、未知指令异常）的大致过程，具体过程包括一下步骤：
(1) 检测异常（确认异常、关中断、保存地址、跳转）
(2) 处理异常
(3) 返回主程序
2.根据要实现的异常设计，检测异常的控制信号产生器
3.设计处理异常程序（写在指令储存器（rom）中）
4.设计返回主程序的功能
其中第一步是直接硬件实现的，后两步是通过实现四条指令来实现的。



5、　分析四条异常指令（分析要实现的功能）

```
mfc0  rt, rd # (rt) <--c0
```

mfc0 是将三个异常与中断寄存器中的内容移到指定的内存中。

```
mtc0 rt,rd  # ct<--(rt)
```

mtc0 是将内存中的内容移到指定异常与中断寄存器。

```
syscall
```

syscall 实现系统调用

```
eret
```

eret 实现返回和开中断

6、 根据异常与中断功能和四个异常指令添加相关器件的连接方式

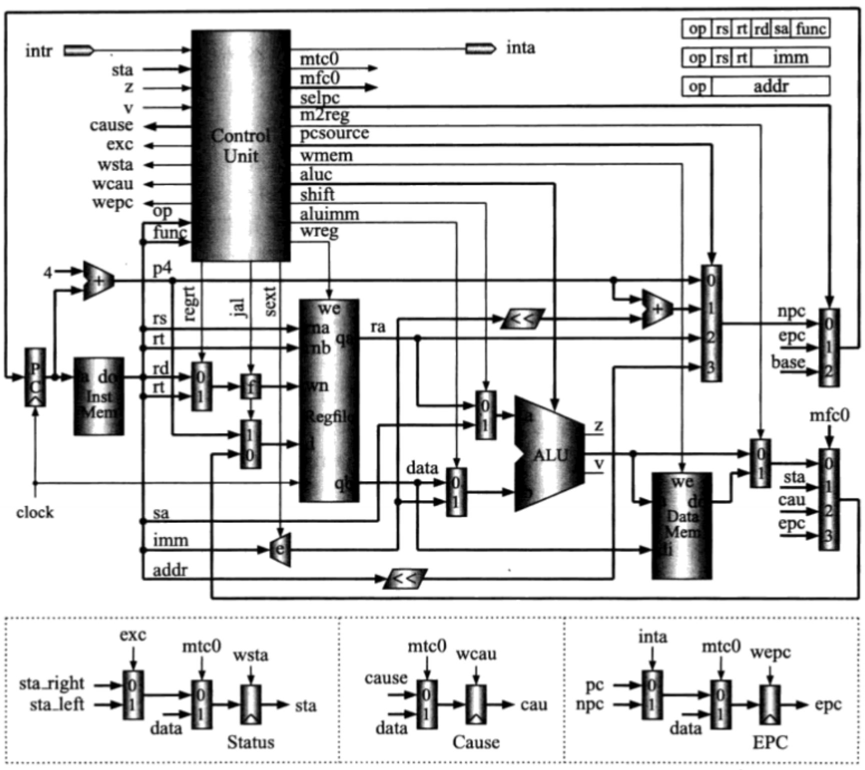


图 6.11 带有处理异常或中断功能的单周期 CPU

可以看见 ram 的数据来源增加了异常寄存器[1]，下址 pc 生成也增加了 base（异常处理程序的总入口），epc（保存的返回地址）。

四、 过程与调试

1、Signaltap 仿真（调试和 debug）

仿真调试的步骤：

经过非常多次的调试（总共花的时间要比编写程序花的时间要多几倍）可以总结出来一般步骤：

- 首先是分析错误出错的大概位置（例如执行到某个指令后出现死循环）
- 在 signaltap 中添加关心的信号，可以删除不关心的信号，因为可能导致编译速度过慢。

- 然后分析结果，把所有的可能原因考虑一下，逐渐缩小范围，最终找到错误的地方。

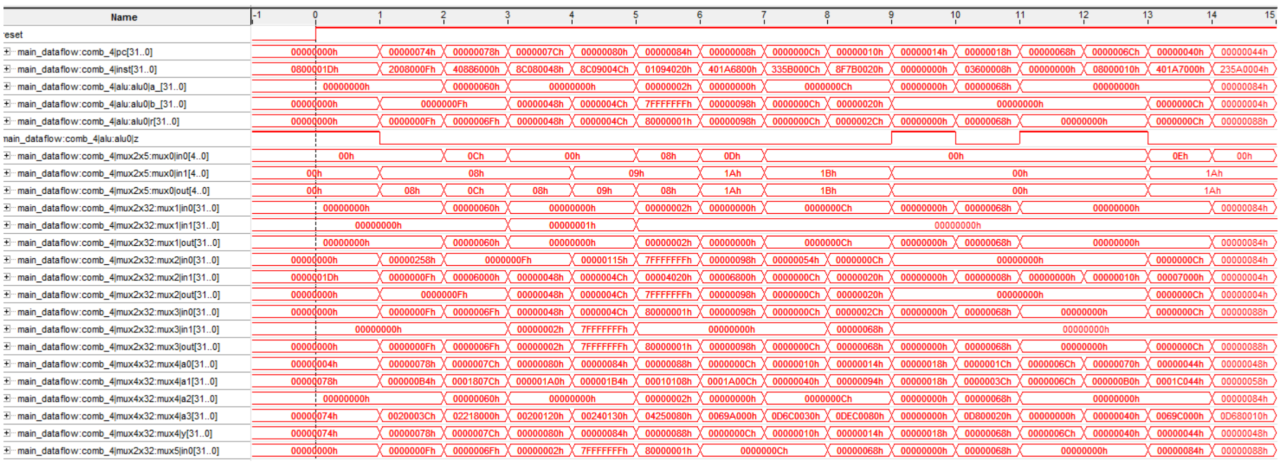
因为调试过程十分繁琐复杂，并且看似 signaltap 出现的错误相同但是导致他们的原因却可能大相径庭，所以没有保存这个过程中的截图，仅说明几点需要注意的地方：

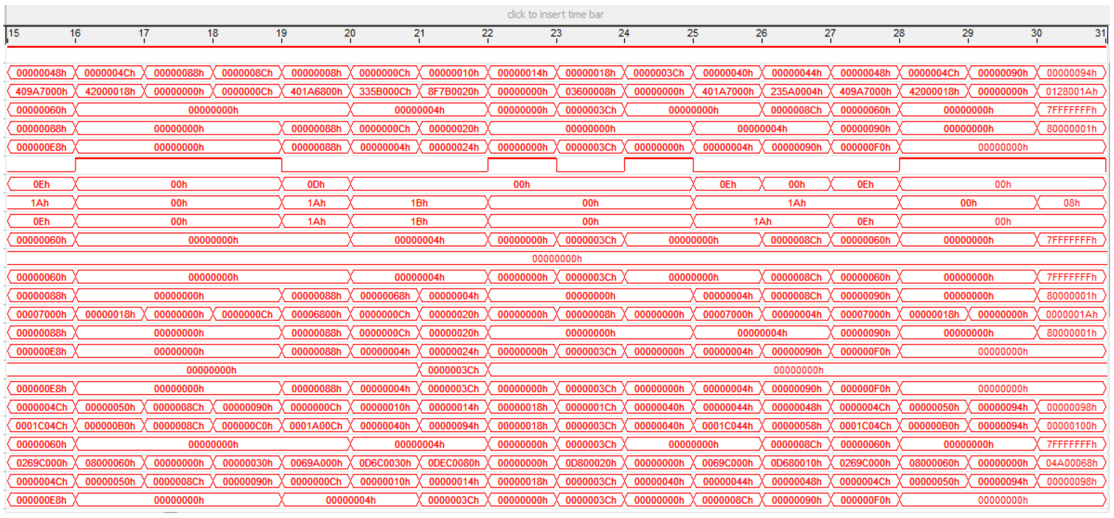
- 注意变量名，如果使用变量较多十分容易产生这个问题，大小写不同导致使用的变量不是一个，然后也没有报错，排查起来十分费神费力。
- 注意参数对应，使用对应的模块时，传入参数如果不对应，Quartus II 也不会报错，但是实现不了想要的功能，排查起来也比较麻烦。
- 注意要设置触发器，上升沿有效，才能把运行的结果锁定住，不然执行过快，还没来得及暂停就已经过去了。

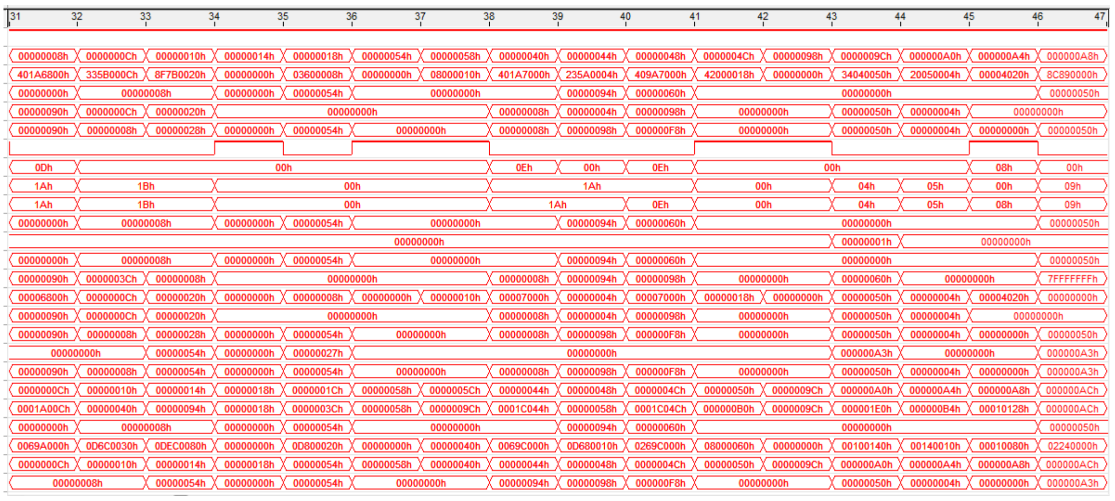
2、实验结果

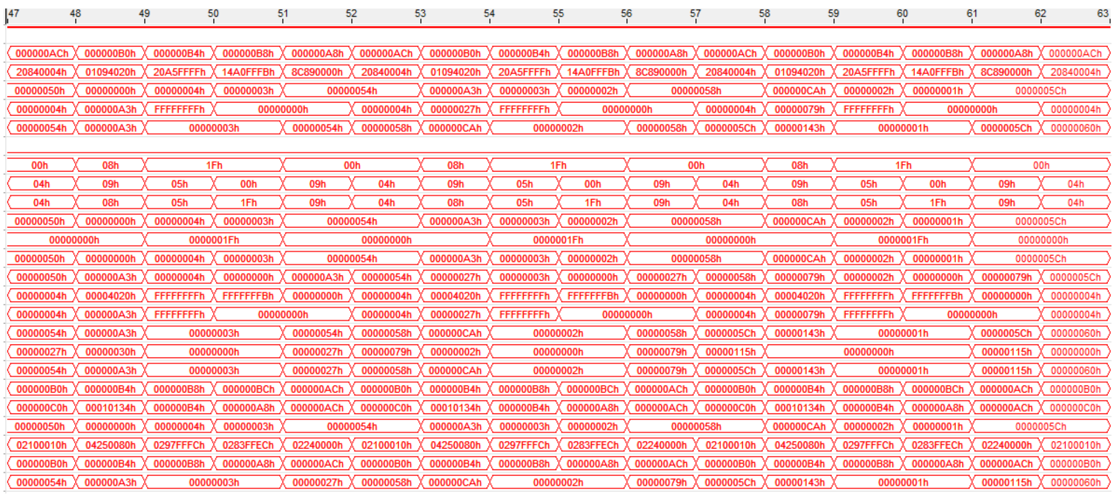
1.带异常和中断的 signaltap 仿真结果

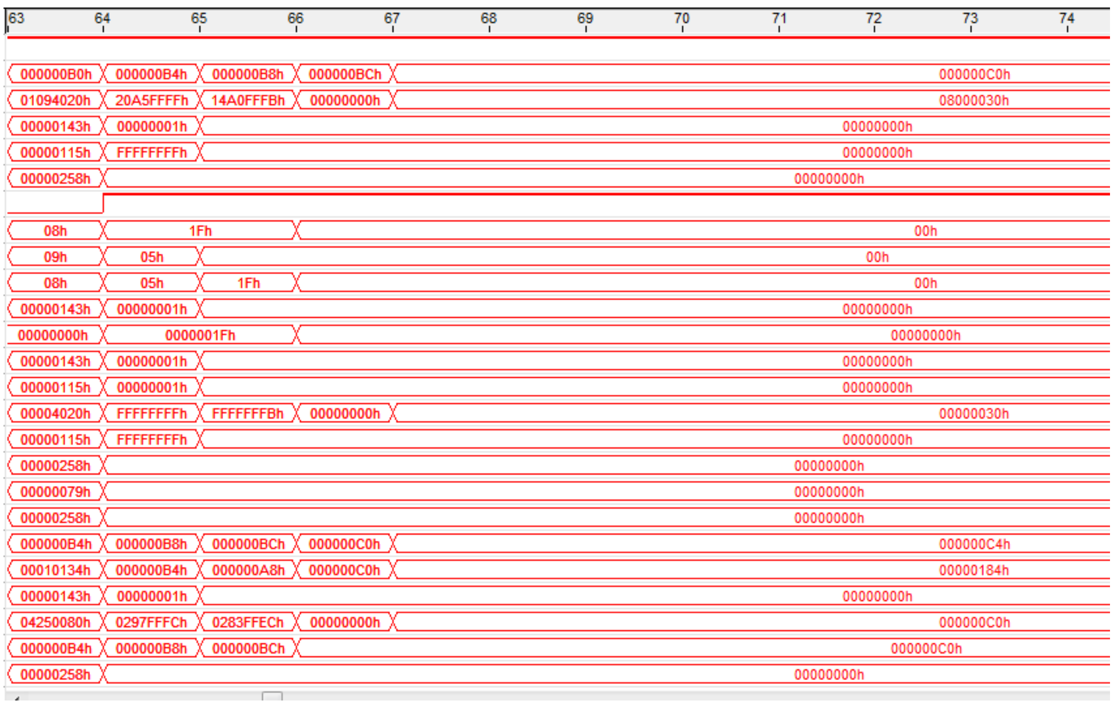
第一行是 pc 就是指令地址，可以看出在 00000084h 发生异常进入到总的异常程序处理入口 00000008h 然后在 000000018h 进入到处理异常程序的入口为 000000068h 最终在 0000004ch 返回主程序 00000088h，剩下的对照测试程序和处理异常程序的地址同理可以看出


Name
reset
main_dataflow:comb_4|pc[31..0]
main_dataflow:comb_4|inst[31..0]
main_dataflow:comb_4|alu:alu0|a_[31..0]
main_dataflow:comb_4|alu:alu0|b_[31..0]
main_dataflow:comb_4|alu:alu0|r[31..0]
main_dataflow:comb_4|alu:alu0|z
main_dataflow:comb_4|mux2x5:mux0|in0[4..0]
main_dataflow:comb_4|mux2x5:mux0|in1[4..0]
main_dataflow:comb_4|mux2x5:mux0|out[4..0]
main_dataflow:comb_4|mux2x32:mux1|in0[31..0]
main_dataflow:comb_4|mux2x32:mux1|in1[31..0]
main_dataflow:comb_4|mux2x32:mux1|out[31..0]
main_dataflow:comb_4|mux2x32:mux2|in0[31..0]
main_dataflow:comb_4|mux2x32:mux2|in1[31..0]
main_dataflow:comb_4|mux2x32:mux2|out[31..0]
main_dataflow:comb_4|mux2x32:mux3|in0[31..0]
main_dataflow:comb_4|mux2x32:mux3|in1[31..0]
main_dataflow:comb_4|mux2x32:mux3|out[31..0]
main_dataflow:comb_4|mux4x32:mux4|a0[31..0]
main_dataflow:comb_4|mux4x32:mux4|a1[31..0]
main_dataflow:comb_4|mux4x32:mux4|a2[31..0]
main_dataflow:comb_4|mux4x32:mux4|a3[31..0]
main_dataflow:comb_4|mux4x32:mux4|y[31..0]
main_dataflow:comb_4|mux2x32:mux5|in0[31..0]
00000000h 00000074h 00000078h 0000007Ch 00000080h 00000084h 00000008h 0000000Ch 00000010h 00000014h 00000018h 00000068h 0000006Ch 00000040h 00000044h


click to insert time bar
00000048h 0000004Ch 00000088h 0000008Ch 00000008h 0000000Ch 00000010h 00000014h 00000018h 0000003Ch 00000040h 00000044h 00000048h 0000004Ch 00000090h 00000094h

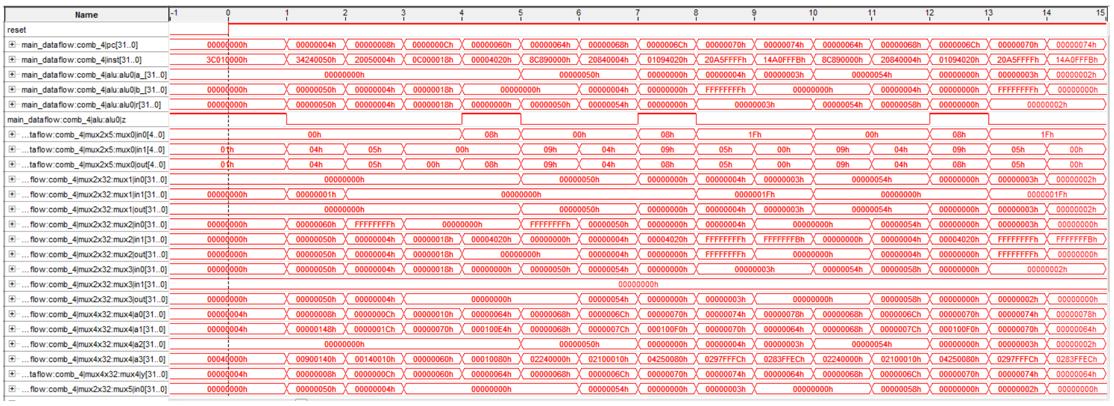


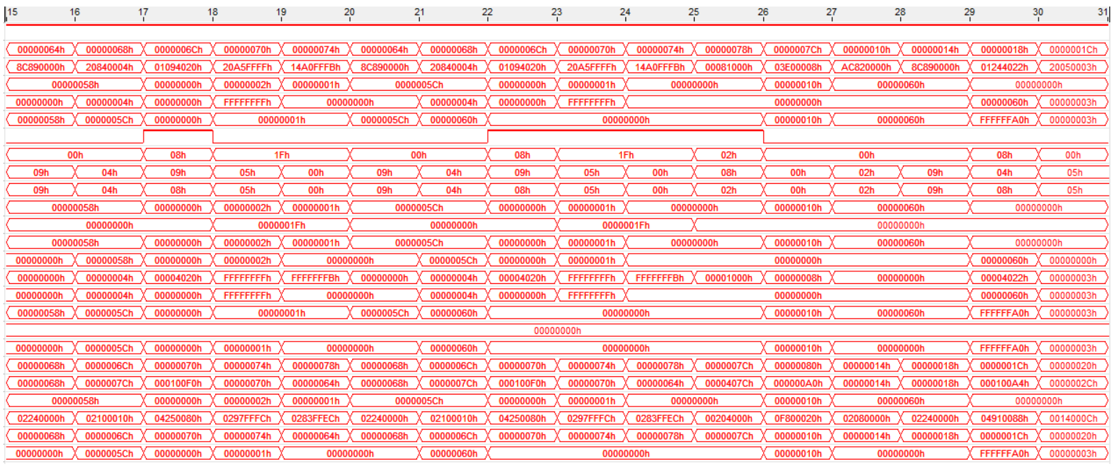


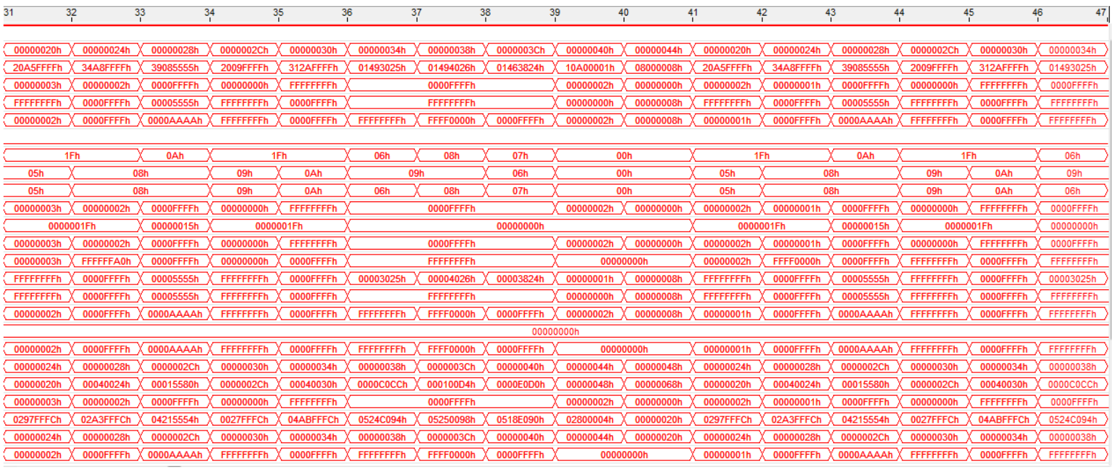


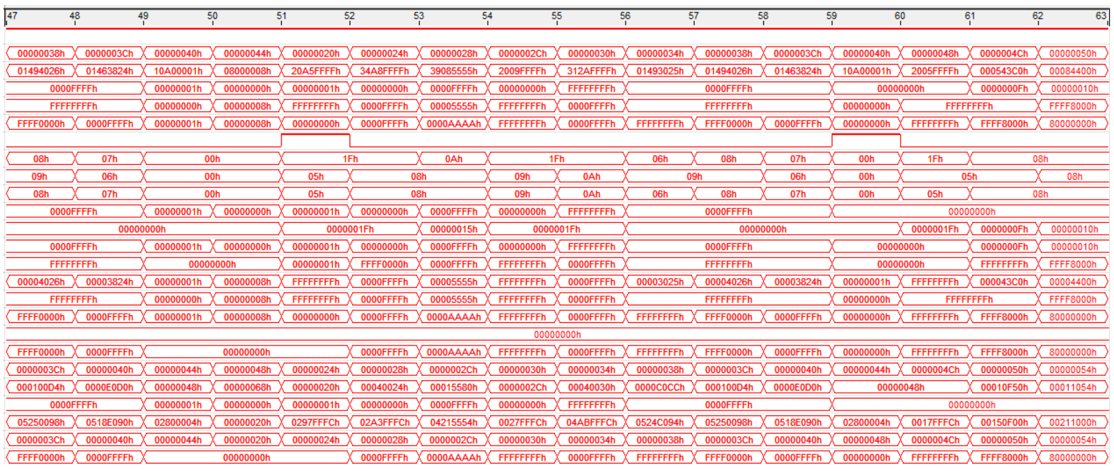
最终死循环在 finish

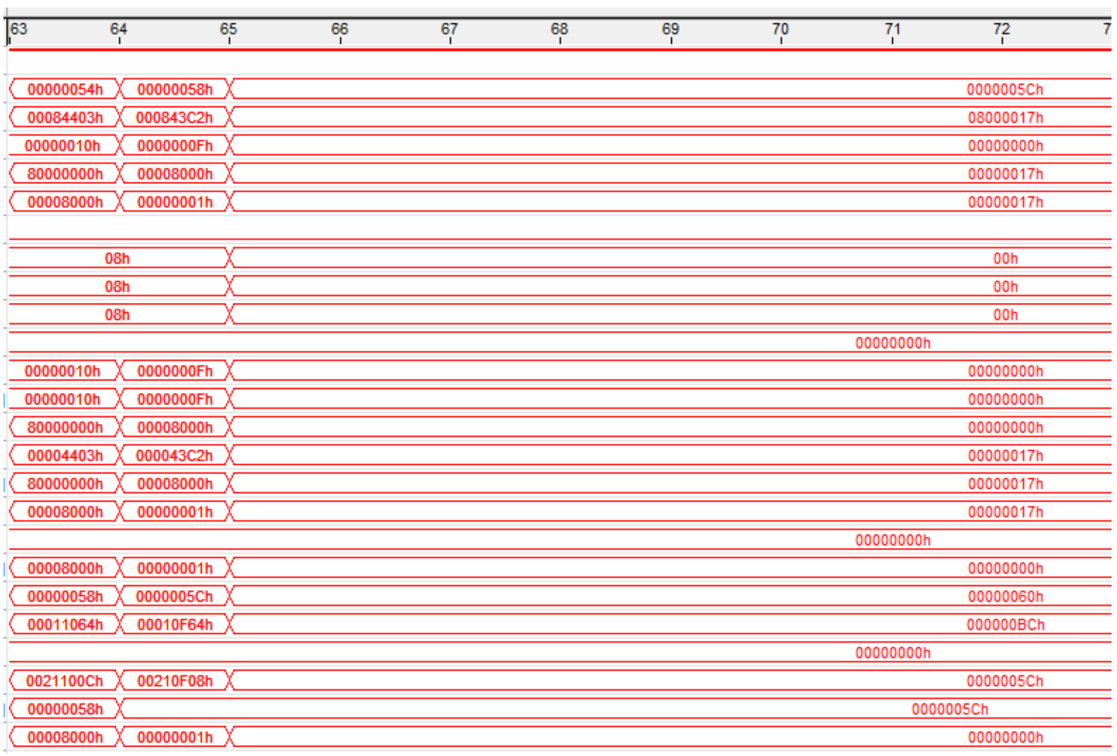
2. 不带异常和中断的 sinaltap 仿真结果


Name
reset
main_dataflow:comb_4|pc[31..0]
main_dataflow:comb_4|inst[31..0]
main_dataflow:comb_4|alu:alu0|a_[31..0]
main_dataflow:comb_4|alu:alu0|b_[31..0]
main_dataflow:comb_4|alu:alu0|r[31..0]
main_dataflow:comb_4|alu:alu0|z
...taflow:comb_4|mux2x5:mux0|in0[4..0]
...taflow:comb_4|mux2x5:mux0|in1[4..0]
...taflow:comb_4|mux2x5:mux0|out[4..0]
...flow:comb_4|mux2x32:mux1|in0[31..0]
...flow:comb_4|mux2x32:mux1|in1[31..0]
...flow:comb_4|mux2x32:mux1|out[31..0]
...flow:comb_4|mux2x32:mux2|in0[31..0]
...flow:comb_4|mux2x32:mux2|in1[31..0]
...flow:comb_4|mux2x32:mux2|out[31..0]
...flow:comb_4|mux2x32:mux3|in0[31..0]
...flow:comb_4|mux2x32:mux3|in1[31..0]
...flow:comb_4|mux2x32:mux3|out[31..0]
...flow:comb_4|mux4x32:mux4|a0[31..0]
...flow:comb_4|mux4x32:mux4|a1[31..0]
...flow:comb_4|mux4x32:mux4|a2[31..0]
...flow:comb_4|mux4x32:mux4|a3[31..0]
...taflow:comb_4|mux4x32:mux4|y[31..0]
...flow:comb_4|mux2x32:mux5|in0[31..0]








63 64 65 66 67 68 69 70 71 72
00000054h 00000058h 0000005Ch
00084403h 000843C2h 08000017h
00000010h 0000000Fh 00000000h
80000000h 00008000h 00000017h
00008000h 00000001h 00000017h
08h 00h
08h 00h
08h 00h
00000000h
00000010h 0000000Fh 00000000h
00000010h 0000000Fh 00000000h
80000000h 00008000h 00000000h
00004403h 000043C2h 00000017h
80000000h 00008000h 00000017h
00008000h 00000001h 00000017h
00000000h
00008000h 00000001h 00000000h
00000058h 0000005Ch 00000060h
00011064h 00010F64h 000000BCh
00000000h
0021100Ch 00210F08h 0000005Ch
00000058h 0000005Ch
00008000h 00000001h 00000000h
最后也是可以看到死循环在 finish 语句。

五、 总结与心得

不足：我们设计的 cpu 仍然还是很简陋的，仅仅局限在实现这个程度，而没有考虑效率和速度，而且还是单周期的，能处理的异常和中断也仅有四种。但是我相信我们已经具备了去学习这样的底层 cpu 的能力，如果日后需要我们会在次学习更高层次的知识，去实现更好更完善的更快的 cpu。

收获：本次实验虽然采用的书上的设计，但是我们自己动手实现更是加深了我们对 cpu 的记忆和理解，相信在很长一段时间内我们都不会忘记，并且用到的时候还会很快回忆起来，因为这个 cpu 确实花费了我们很多精力。在实现过程中对 bug 的调试也是锻炼了我们调试程序的能力，相信对我们的编程能力也有一定提升。最后就是增加了我们的信心，我们可以付出精力去改变我们想要改变的事情，并且逐渐变得熟练。

六、 参考文献

[1]计算机原理与设计：Verilog HDL 版