哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

课程报告

开课学期：2022 夏季

课程名称：计算机设计与实践

项目名称：基于 miniRV-1 的 SoC 设计

项目类型：综合设计型

课程学时：56 地点：T2605

学生班级：11 班

学生学号：200111102

学生姓名：张奕桢

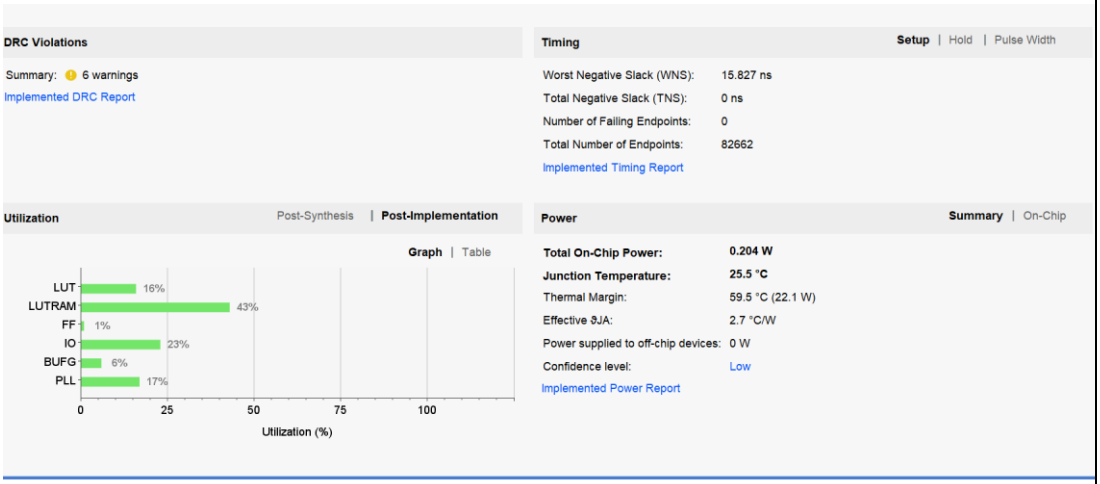
评阅教师：

报告成绩：

实验与创新实践教育中心制

2022 年 7 月

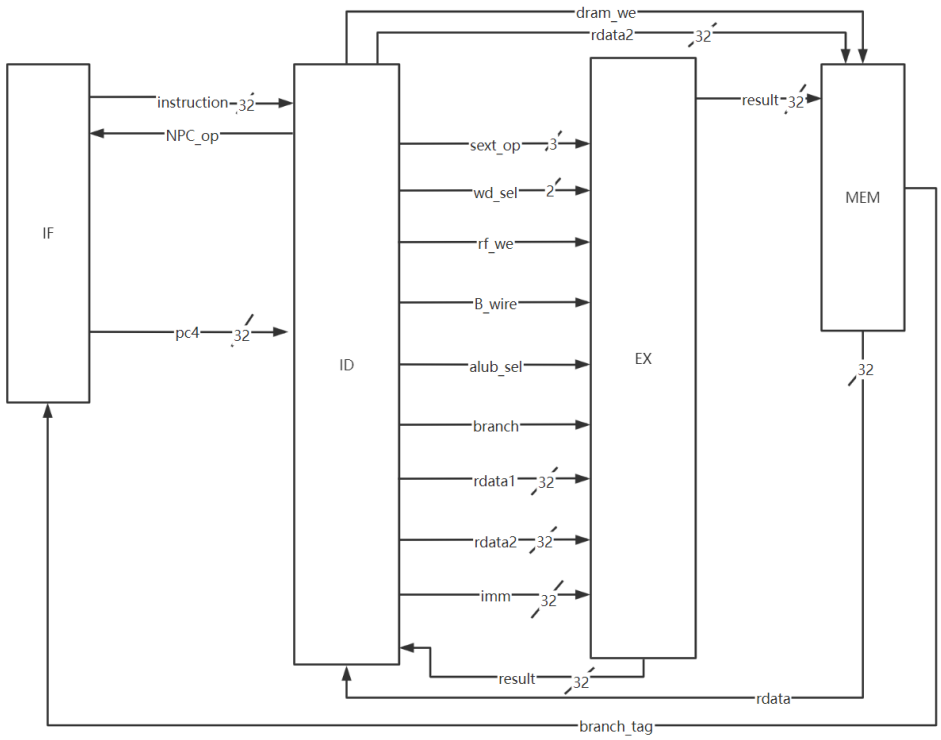
注：本设计报告中各个部分如果页数不够，请同学们自行扩页。原则上一定要把报告写详细，能说明设计的成果、特色和过程。报告应该详细叙述整体设计，以及设计中的每个模块。设计报告将是评定每个人成绩的重要组成部分（**设计内容及报告写作**都作为评分依据）。

设计的功能描述（含所有实现的指令描述，以及单周期/流水线 CPU 频率）
在 Minisys 开发板上实现了单周期 CPU，完成了 24 条指令 1. R 型指令：add, sub, and, or, xor, sll, srl, sra 2. I 型指令：addi, andi, ori, xori, slli, srli, srai, lw, jalr 3. S 型指令：sw 4. B 型指令：beq, bne, blt, bge 5. J 型指令：jal 单周期频率为 25MHZ
设计的主要特色（除基本要求以外的设计）
1. 在写回 PC 的 WB 阶段，对于各条指令的 PC 计算和写回放在最后执行，从而在流水线设计的时候能够实现更好地五级流水。 2. 在 B 型指令判定的时候，采用了单线判定，只根据一个信号给出是否进行地址跳转，而非多信号。
资源使用、功耗数据截图（Post Implementation；含单周期、流水线 2 个截图）
以下是示例，请贴自己的图。 

1 单周期 CPU 设计与实现

1.1 单周期 CPU 整体框图

要求：无需画出模块内的具体逻辑，但要标出模块的接口信号名、模块之间信号线的信号名和位宽，以及说明每个模块的功能含义。



分为五个模块,IF、ID、EX、MEM、WB(没有实体画出来，只有线连接)

IF:取指模块,根据内部 PC 给出 instruction 和 PC4(pc+4)(用于 jalr 的地址存储)。

ID:译码模块，根据 IF 模块传递过来的 instruction 给出控制信号：sext_op、wd_sel、rf_we、B_wire、alub_sel、branch、以及经过内部寄存器根据指令内寄存器的地址给出 rdata1、rdata2、imm

EX:执行模块，进行运算功能， 根据 rdata1、rdata2、imm 和相应的控制信号，运算得出 result 结果

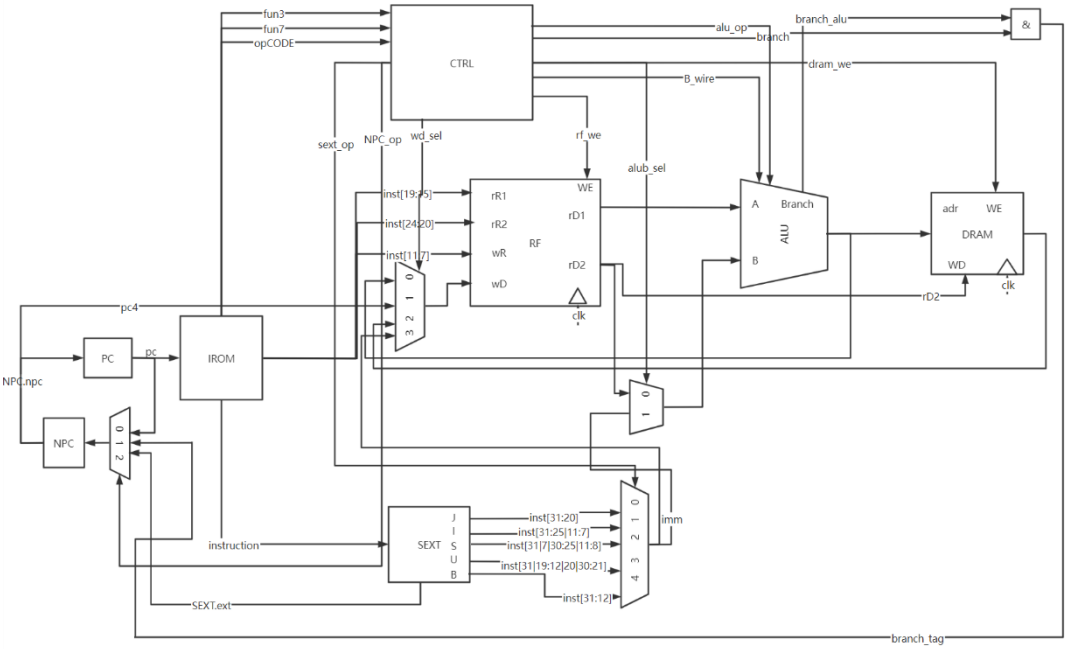
MEM:存储器模块，对应的是内存的存储和读取，接口对应地址，时钟，写使能，读数据，写数据，模块用 dram 实现

WB:写回模块，对应的是 RF 寄存器的写入包括计算后的新地址，立即数 imm，运算后的新结果等，还有另一部分是 IF 模块的地址写回。

1.2 单周期 CPU 模块详细设计

要求：画出各个模块的详细设计图，包含内部的子模块，以及关键性逻辑；标出子模块接口信号名、各信号线的信号名和位宽，并有详细的解释说明。

（除了 ALU 单独画了出来，剩下的模块细节在如下图中显示）



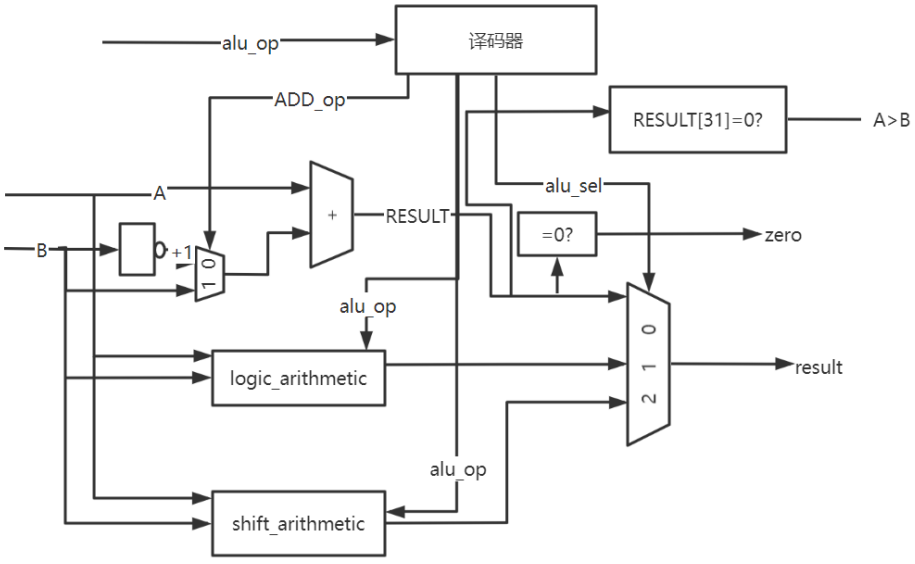
```
module IF_pc(
    input clk,
    input rst_n,
    input [31:0] npc,
    output [31:0] pc,
    output [31:0] pc4

    );

    reg [31:0]pc_reg;
    always@(posedge clk or negedge rst_n )begin
    if(~rst_n)begin
        pc_reg <= -4;

    end
    else begin
        begin
            pc_reg = npc;
        end
      end
    end
    assign pc=pc_reg;
    assign pc4=pc+4;

endmodule
```

其中的 IF 部分就是包括的 pc 部件和 IROM 部件，负责根据 clk 信号提取 IROM 中的 pc[31:0]地址以及指令 instruction[31:0]

```
module NPC(
    input [31:0] pc,
    input [31:0] pc_reg,
    input [1:0] npc_op,
    input branch_tag,
    input [31:0]rs1,
    input [31:0] imm,
    output [31:0] npc

    );
    reg [31:0] npc_reg;
    always@(*)begin

        case(npc_op)
            2'd0:begin
                npc_reg=pc+32'd4;
            end
            2'd1:begin
                case(branch_tag)
                1'd0:npc_reg=pc+32'd4;
                1'd1:npc_reg=pc_reg+imm;
                endcase
            end
            2'd2:begin
                npc_reg=pc_reg+imm;
            end
            2'd3:begin
                npc_reg=rs1+imm;
            end
```

然后就是 ID 模块，其中包括了 CTRL 部件、RF 部件以及 SEXT 部件，根据上一模块 IF 提供的 pc 地址和指令信息，解析出 fun3[2:0]、fun7[6:0]、opCODE[6:0]确定指令类型，CTRL 负责提供向后续模块提供 sext_op[2:0]，rf_we，wd_sel[1:0] NPC_op[2:0]、alub_op、B_wire、alu_op、branch，RF 则根据解析出来的 Rr1，Rr2，wr 还有多路选择器得出来的 wd 进行寄存器堆的相关逻辑，对应输出寄存器堆内存储的[31:0]rdata1,rdata2,SEXT 部件则根据从 instruction 得出来的 imm 根据对应的指令给出扩展后的立即数[31:0]imm

```
1  `timescale 1ns / 1ps
2
3
4  module idecode(
5      input clk,
6      input [4:0]raddr1,//rs1
7      input [4:0]raddr2,//rs2
8      input [4:0]waddr,//rd
9      input we,//д ??????
10     input [31:0]wdata,//д ????
11     output [31:0]rdata1,//rs1??????
12     output [31:0]rdata2//rs2???????
13
14     );
15     reg [31:0]reg_array[31:0];//???????
16     always @(posedge clk)begin
17         if(we)reg_array[waddr]<=wdata;
18     end
19     assign rdata1=(raddr1==5'b0)?32'b0:reg_array[raddr1];
20     assign rdata2=(raddr2==5'b0)?32'b0:reg_array[raddr2];
21
22
23 endmodule
```



H:/project_3/project_3.srcs/sources_1/new/execute.v

```
`timescale 1ns / 1ps

module execute(
    input [31:0] A,
    input [31:0] B,
    input [2:0] alu_op,
    input [1:0] B_wire,
    input branch,
    output  branch_alu_wire,
    output  [31:0] result_wire
    );
    reg branch_alu;
    reg [31:0] result;
    assign result_wire = result;
    assign branch_alu_wire = branch_alu;
    always@(*)begin
        case(alu_op)
            `ADD:begin
               result=A+B;
            end

            `SUB:begin
               result=A-B;
            end

            `AND:begin
               result=A&B;
            end
```

```
            if(branch)begin
              case(B_wire)
                2'b00:begin//beq
                  if(A==B)branch_alu=1;
                  else branch_alu=0;
                end
                2'b01:begin//bne
                  if(A==B)branch_alu=0;
                  else branch_alu=1;
                end
                2'b10:begin// blt
                   if(A[31]&&B[31]&&A<B)branch_alu=1;
                   else if(A[31]&&~B[31])branch_alu=1;
                   else if(~A[31]&&~B[31]&&A<B)branch_alu=1;
                   else branch_alu=0;
                end
                2'b11:begin// bge
                   if(A==B)branch_alu=1;
                   else if(A[31]&&B[31]&&A>B)branch_alu=1;
                   else if(~A[31]&&B[31]) branch_alu=1;
                   else if(~A[31]&&~B[31]&&A>B) branch_alu=1;
                   else branch_alu=0;
                 end
              endcase
            end
            else branch_alu=0;
        end
endmodule
```

EX 模块，则只有 ALU 部件，根据 alu_op 选择对应的 ALU 计算方式，而 A,B 则对应的 rdata1，rdata2，根据对应的选择传入 ALU，进行相关的计算，根据相

应的运算，给出是否跳转的信号 branch_alu,对应的是 B 型指令是否进行跳转的信号，外部接口引入 alu_op[2:0],根据 alu_op,选择进行运算的种类，对 A[31:0]，B[31:0]进行补码运算，特殊的有 B 型指令的相关赋值操作，要根据 B 型指令的判断条件给出是否跳转的信号 branch_alu，同时给出判定大小的逻辑条件判定

```
//DRAM
dram u_dram (
.clk    (clk),              // input wire clka
.a      (waddr_tmp[15:2]),      // input wire [13:0] addra
.spo    (rdata_dram),         // output wire [31:0] douta
.we     (dram_wen),           // input wire [0:0] wea
.d      (wdata)          // input wire [31:0] dina
);
```

MEM 模块则主要负责存取信息，也只有 DRAM 模块，由 IP 核实现，需要给出是 addr,we(是否写入内存)，WD[31:0]表示的是写入的数据，rdata[31:0]则表示的是读出的数据。

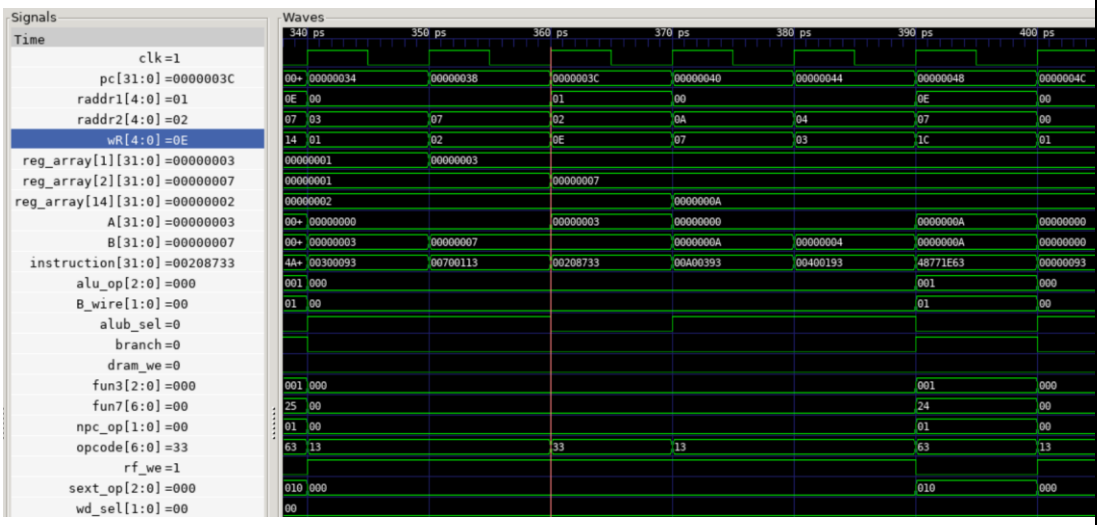
WB 模块负责写回寄存器堆和 PC 地址，因此没有具体的模块，更多的是信号的传导，在 NPC 模块和 Wd 处体现

1.3 单周期 CPU 仿真及结果分析

要求：包含逻辑运算、访存、分支跳转三类指令的仿真截图以及波形分析；每类指令的截图和分析中，至少包含 1 条具体指令；截图需包含信号名和关键信号。

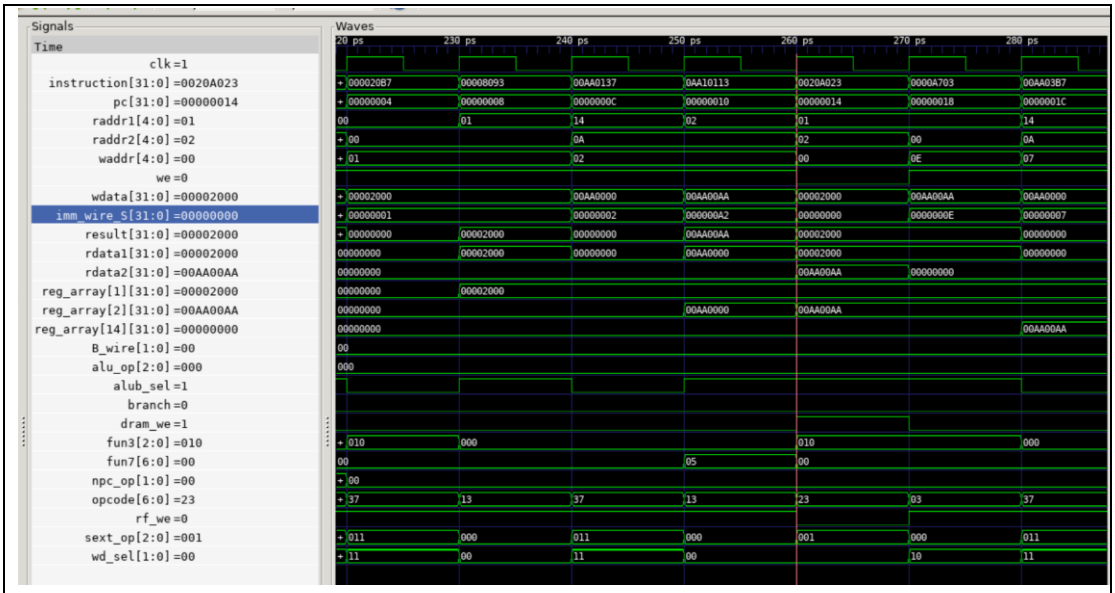
add.dump

```
  18: 4c771663            bne x14,x7,4e4 <fail>

0000001c <test_3>:
  1c: 00100093            addi  x1,x0,1
  20: 00100113            addi  x2,x0,1
  24: 00208733            add x14,x1,x2
  28: 00200393            addi  x7,x0,2
  2c: 00300193            addi  x3,x0,3
  30: 4a771a63            bne x14,x7,4e4 <fail>

00000034 <test_4>:
  34: 00300093            addi  x1,x0,3
  38: 00700113            addi  x2,x0,7
  3c: 00208733            add x14,x1,x2
```



根据 pc=0x0000003c，得出对应的 0x00208733 instruction 根据此进入到 ID 阶段，并根据指令分解得出相关的控制信号，以及 rd1=1,rd2=2,在寄存器堆 RF 内解析出对应寄存器堆内的数据分别为 3、7，根据指令信号 alu_op=add 到了下个时钟阶段进行加和运算，得出结果 10 也就是（0A），而由于 add 指令没有访存需求，因此到达 WB 阶段，根据 wr=14 也就是（0E）。

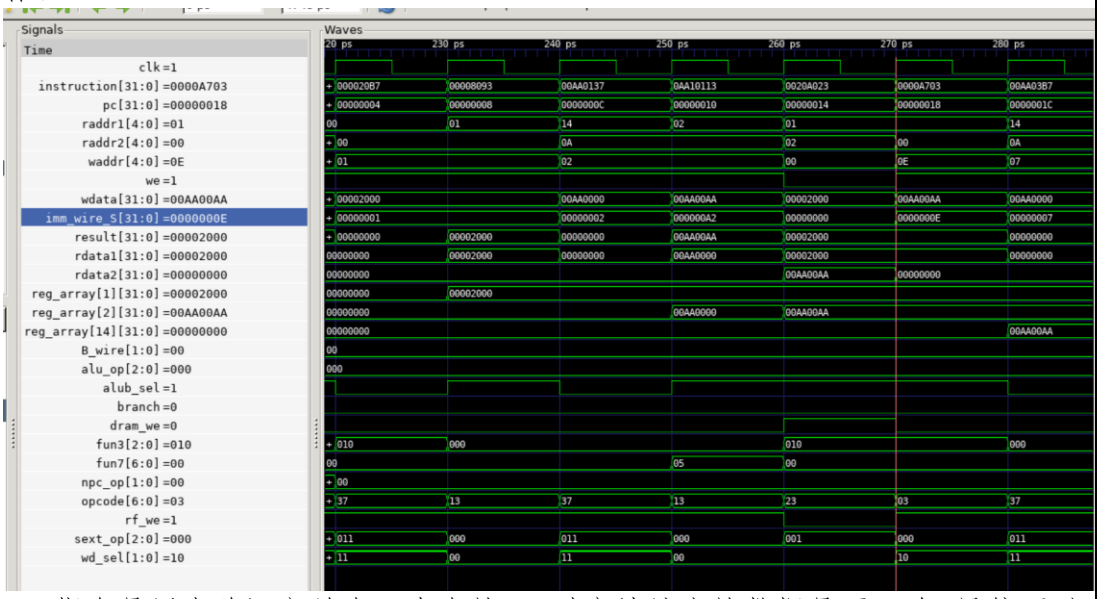
```
00000004 <reset_vector>:
   4: 000020b7            lui x1,0x2
   8: 00008093            addi  x1,x1,0 # 2000 <begin_signature>
   c: 00aa0137            lui x2,0xaa0
  10: 0aa10113            addi  x2,x2,170 # aa00aa <_end+0xa9e07a>
  14: 0020a023            sw  x2,0(x1)
  18: 0000a703            lw  x14,0(x1)
```



主要分析 sw 和 lw 两条涉及到存取的指令

分析得知 x2=aa00aa,当 pc=14 的时候，根据分析可以的出 rs2=2,rs1=1.

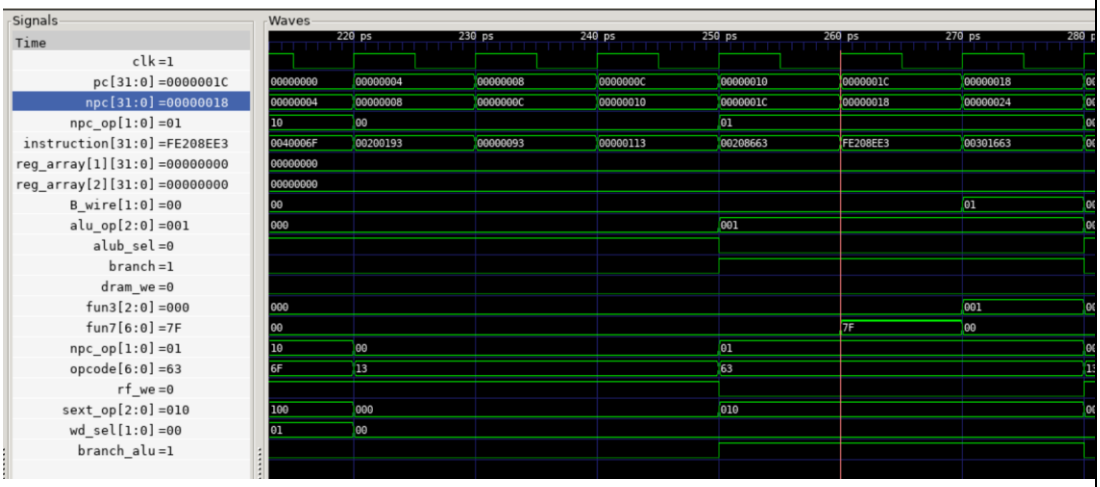
指令要将 x2 内的数据存储到 x1 寄存器内数据对应的内存地址当中去，当 clk 上升的时候实现,可以看到x1 内的数据为 2000,进入 alu 以后由于 alu_op=add，对应的是 rs1 对应的数据和 imm=0 相加和得到 result=2000 对应是否写入内存由给出的控制指令 dram_we=1 实现写入操作，在下一个时钟写入，最后由于不需要写回 RF 寄存器堆，因此 WB 阶段只有 pc 的写回，而且是正常+4 操作。



Lw 指令是用来验证之前存入内存的 x1 对应地址出的数据是否正确，显然可以解析指令为 0000a703,再根据 ID 阶段的指令拆解规则，得到 rd=14(0E),rs1=1,同样的是进入 alu 后的加和操作（ALU_op=add），得出依然要读取的是 rdata1+imm=2000,然后进入到 MEM 阶段，将最终结果存储到 RF(rf_we=1 表示写入寄存器 rd=14)可以看到最后 rd[14]内的数据为之前存入的数据 AA00AA

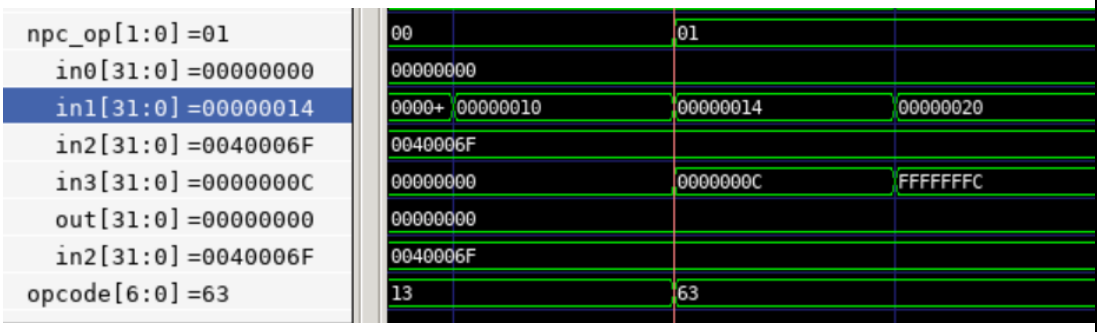
beq.dump

```
beq:     file format elf32-littleriscv


Disassembly of section .text.init:

00000000 <_start>:
   0: 0040006f            jal x0,4 <reset_vector>

00000004 <reset_vector>:
   4: 00200193            addi  x3,x0,2
   8: 00000093            addi  x1,x0,0
   c: 00000113            addi  x2,x0,0
  10: 00208663            beq x1,x2,1c <reset_vector+0x18>
  14: 2a301863            bne x0,x3,2c4 <fail>
  18: 00301663            bne x0,x3,24 <test_3>
  1c: fe208ee3            beq x1,x2,18 <reset_vector+0x14>
```



跳转指令以 beq 为例：分析 pc=14 时候的跳转

根据前面的指令可以看到 x1=x2=0，有用的信号为 beq 指令解析出来的 rs1=x1，rs2=x2,进入 alu 之后，进行比较得出的结果用 branch_alu 判别，可以看到此时为 1，由于是跳转指令因此在 ID 阶段将能跳转的信号 branch 赋值为 1，两者取并为 1，实现跳转跳转后的地址

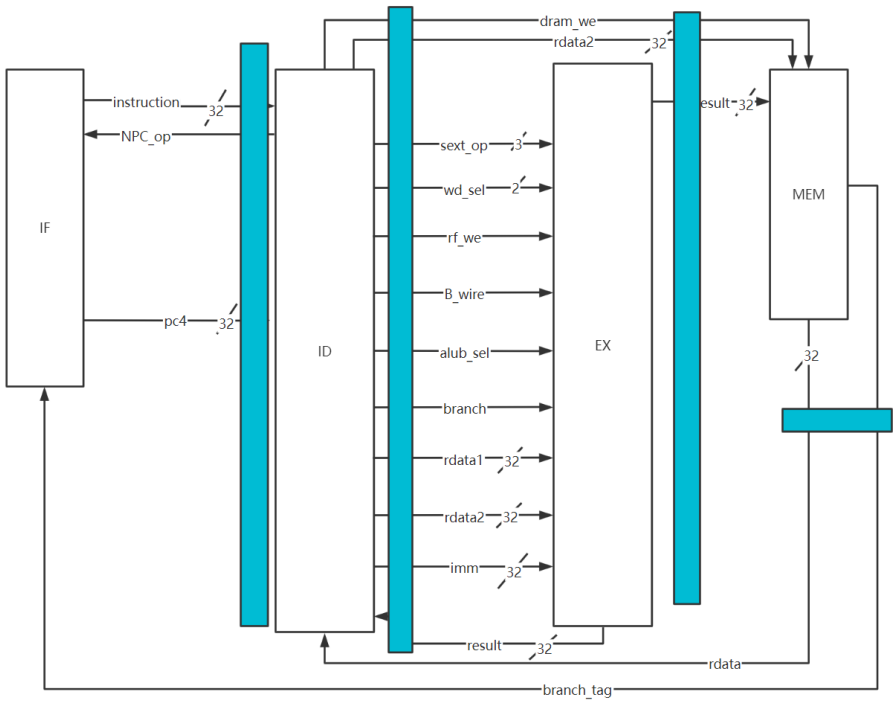


此时的 npc_op 为 1，与 offset 相加得到最终地址 18

2 流水线 CPU 设计与实现

2.1 流水线的划分

要求：画出流水线如何划分，说明每个流水级具备什么功能、需要完成哪些操作。



流水线分为 5 级，为 IF，ID,EX,MEM,WB

(1)IF 取指(insturction fetch),从 IROM 获取下一条指令

(2)RD 读取寄存器(read register),读取该指令的源寄存器域指定的 CPU 寄存器的内容。

(3)ALU 算术逻辑单元(arithmetic/logic unit)在一个时钟周期内完成算术或者逻辑操作。

(4)MEM 访问内存(memory),该阶段指令可以读写 dram 中的内存变量。

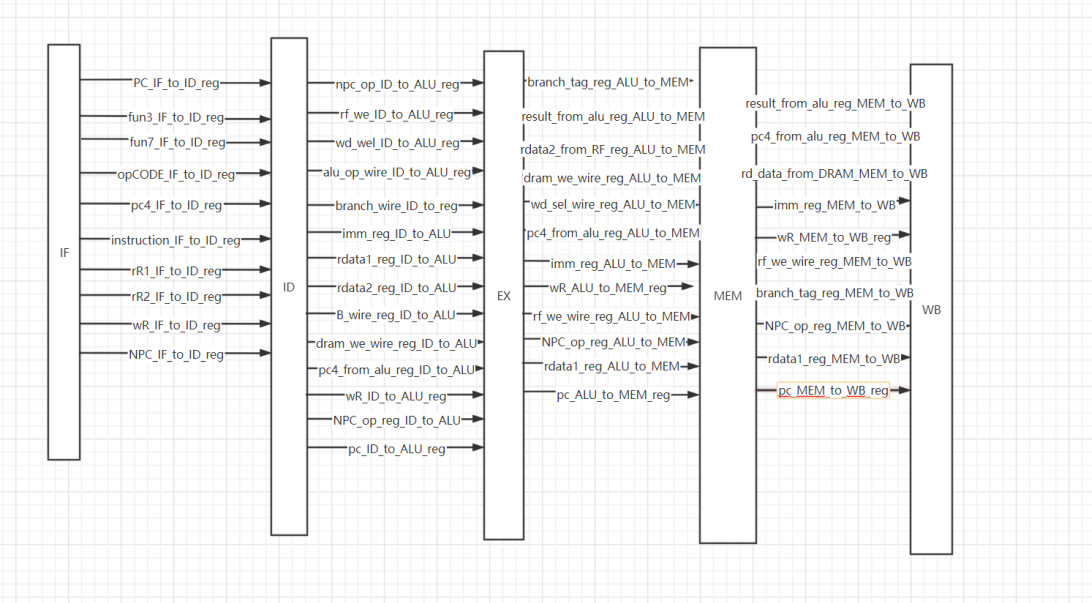
(5)WB 写回寄存器(write back),将操作结果值写道寄存器堆以及 pc 写回操作中。

2.2 流水线 CPU 整体框图

要求：无需画出模块内的具体逻辑，但要标出模块的接口信号名、模块之间信号线的信号名和位宽，以及说明每个模块的功能含义。

```
    reg [31:0] pc_IF_to_ID_reg;
    reg [2:0]  fun3_IF_to_ID_reg;
    reg [2:0]  fun7_IF_to_ID_reg;
    reg [6:0] opCODE_IF_to_ID_reg;
    reg  [31:0] pc4_IF_to_ID_reg;
    reg [31:0] instruction_IF_to_ID_reg;
    reg [4:0] rR1_IF_to_ID_reg;
    reg [4:0] rR2_IF_to_ID_reg;
    reg [4:0] wR_IF_to_ID_reg;
    reg [1:0]NPC_op_reg_IF_to_ID;

    reg [1:0] npc_op_wire_reg_ID_to_ALU;
    reg rf_we_wire_reg_ID_to_ALU;
    reg [1:0] wd_sel_wire_reg_ID_to_ALU;
//    reg [2:0] sext_op_wire_reg_ID_to_ALU;
    reg [2:0] alu_op_wire_reg_ID_to_ALU;
//    reg alub_sel_wire_reg_ID_to_ALU;
    reg branch_wire_reg_ID_to_ALU;
    reg [31:0] imm_reg_ID_to_ALU;
    reg [31:0] rdata1_reg_ID_to_ALU;
    reg [31:0] rdata2_reg_ID_to_ALU;
//    reg [31:0] rdata2_reg_ID_to_ALU;
    reg [31:0] rdata2_from_RF_reg_ID_to_ALU;
    reg [1:0]  B_wire_reg_ID_to_ALU;
    reg dram_we_wire_reg_ID_to_ALU;
    reg [31:0] pc4_from_alu_reg_ID_to_ALU;
    reg [4:0] wR_ID_to_ALU_reg;
    reg [1:0]NPC_op_reg_ID_to_ALU;
    reg [31:0] pc_ID_to_ALU_reg;

reg branch_tag_reg_ALU_to_MEM;
reg [31:0] result_from_alu_reg_ALU_to_MEM;
reg [31:0] rdata2_from_RF_reg_ALU_to_MEM;
reg dram_we_wire_reg_ALU_to_MEM;
reg [1:0] wd_sel_wire_reg_ALU_to_MEM;
reg [31:0] pc4_from_alu_reg_ALU_to_MEM;
reg [31:0] imm_reg_ALU_to_MEM;
reg [4:0] wR_ALU_to_MEM_reg;
reg rf_we_wire_reg_ALU_to_MEM;
reg [1:0]NPC_op_reg_ALU_to_MEM;
reg [31:0] rdata1_reg_ALU_to_MEM;
reg [31:0] pc_ALU_to_MEM_reg;
```

```
reg [31:0] result_from_alu_reg_MEM_to_WB;
reg [31:0] pc4_from_alu_reg_MEM_to_WB;
reg [31:0] rd_data_from_DRAM_MEM_to_WB;
reg [31:0] imm_reg_MEM_to_WB;
reg [31:0] wR_MEM_to_WB_reg;
reg  rf_we_wire_reg_MEM_to_WB;
reg  branch_tag_reg_MEM_to_WB;
reg [1:0]NPC_op_reg_MEM_to_WB;
reg [31:0]rdata1_reg_MEM_to_WB;
reg [31:0] pc_MEM_to_WB_reg;
```



2.3 流水线 CPU 模块详细设计

要求：画出各个模块的详细设计图，包含内部的子模块，以及关键性逻辑；标

出子模块接口信号名、各信号线的信号名和位宽，并有详细的解释说明；此外，必须结合模块图，详细说明数据冒险、控制冒险的解决方法。

2.4 流水线 CPU 仿真及结果分析

要求：包含控制冒险和数据冒险三种情形的仿真截图，以及波形分析。若仅实现了理想流水，则此处贴上理想流水的仿真截图及详细的波形分析。

3 设计过程中遇到的问题及解决方法

要求：包括设计过程中遇到的有价值的错误，或测试过程中遇到的有价值的问题。所谓有价值，指的是解决该错误或问题后，能够学到新的知识和技巧，或加深对已有知识的理解和运用。

首先是 trace 中 pc 地址的问题一开始没有初始化,导致一开始 cpu 无法工作，后来重新分析原理和过程才进行修正。
然后就是根据 trace 中的各种错误，一步步修正指令中的错误，有方向性的找到了自己写的内容中的各种问题
最后就是上版上的写外设，有很多原理上没有弄懂的问题，导致一开始并不知道如何下手，后来明白原理以后就会很快，还有就是实际上板的时候接口对错问题，也反映了一开始的时候没有想清楚哪个接口对应哪个接口，理论没有想清楚之前，就盲目下手，耽误了很多的时间。

4 总结

要求：谈谈学完本课程后的个人收获以及对本课程的建议和意见。请在认真总结和思考后填写总结。
首先是实际完全靠自己没有询问相关同学上手了 cpu 的实际操作，而不是只是停留在理论层面，另外也学习了代码层面的实践经验，反过来也对很多计算机结构和体系的内容也有了更为深刻的理解。课程建议的话，仇老师很认真负责，希望老师明年安排些助教，这样可以轻松些，然后就是实际的课程讲授的内容可以更多和大家的进度匹配些，受到期末考试的影响，一开始没有把重心放到上面，后来也只是写了理想流水，没有更多的时间去进一步的调试仿真。