浙江大學

# 本科实验报告

| | |
|---|---|
| 课程名称: | 计算机体系结构 |
| 姓　　名: | 王俊 |
| 学　　院: | 海洋学院 |
| 专　　业: | 海洋工程与技术 |
| 学　　号: | 3170100186 |
| 指导教师: | 翁恺 |

2020 年　11　月　13　日

# Lab3——简单流水线 CPU

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| 课程名称: | 计算机体系结构 |  |  | 实验类型: | 综合 |
| 实验项目名称: | 简单流水线 CPU |  |  |  |  |
| 学生姓名: | 王俊 | 专业: | 海洋工程与技术 | 学号: | 3170100186 |
| 同组学生姓名: | None |  |  | 指导老师: | 翁恺 |
| 实验地点: | 曹光彪西-301 |  |  | 实验日期: | 2020 年 11 月 13 日 |

## 一、实验目的和要求

**Experiment Purpose:**

•Understand the principles of Pipelined CPU

•Understand the basic units of Pipelined CPU

•Understand the working flow of 5-stages

•Master the method of simple Pipelined CPU

•master methods of program verification of simple Pipelined CPU

**Experiment Apparatus:**

- Computer (Intel Core i5 or higher, 4GB RAM or higher) system
- Sword-V4 development board
- Xilinx ISE 14.4 and above development tools

**Experimental Materials**:

No

二、实验内容和原理

**2.1.Experimental task:**

1.Design the CPU Controller

2.Based on the CPU Controller, design the Datapath of 5-stages Pipelined CPU: –5 Stages

–Register File

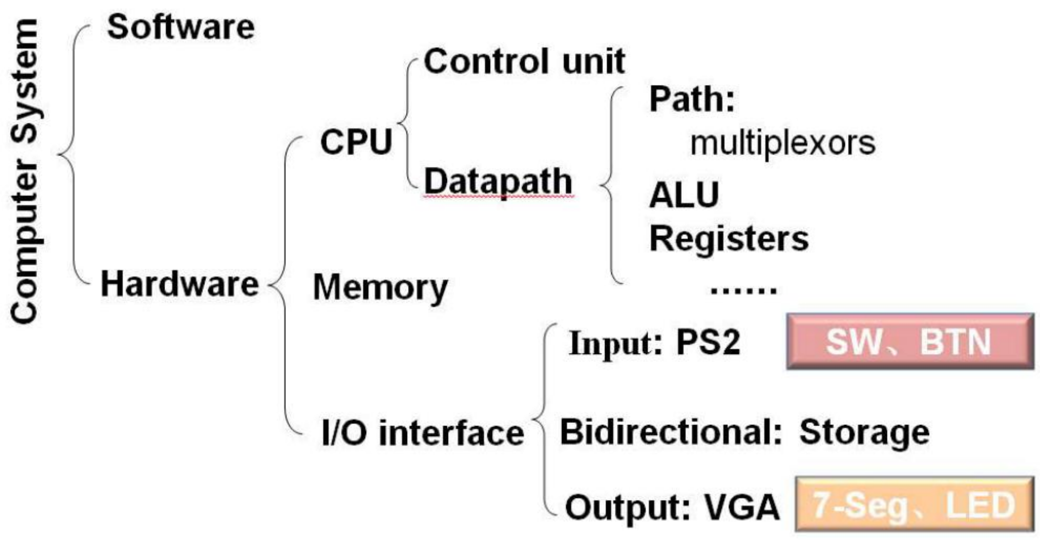
–Memory (Instruction and Data)

–other basic units

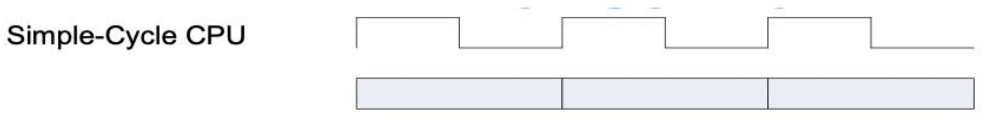
3.Verify the Pp. CPU with program and observe the execution of program

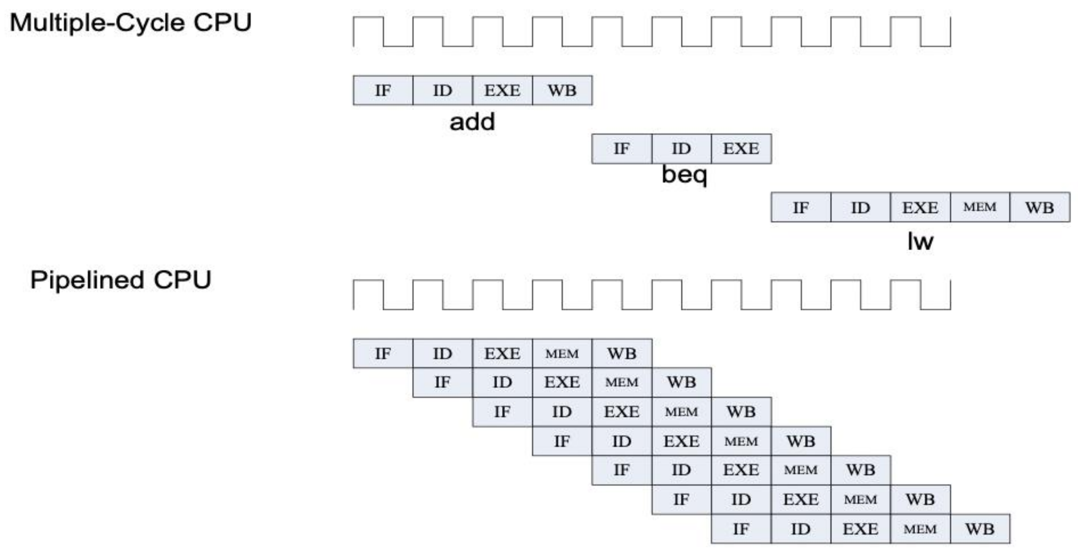
**2.2.Basic principle**

**2.2.1.Computer system decomposition**



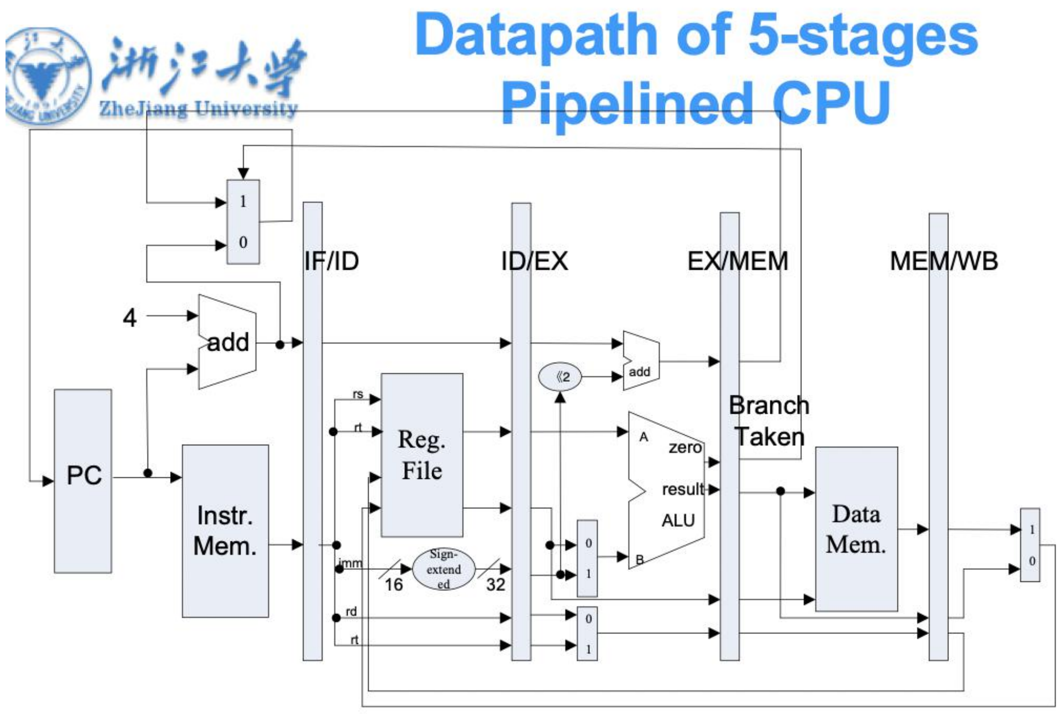
**2.2.2.Comparison of CPU's work**



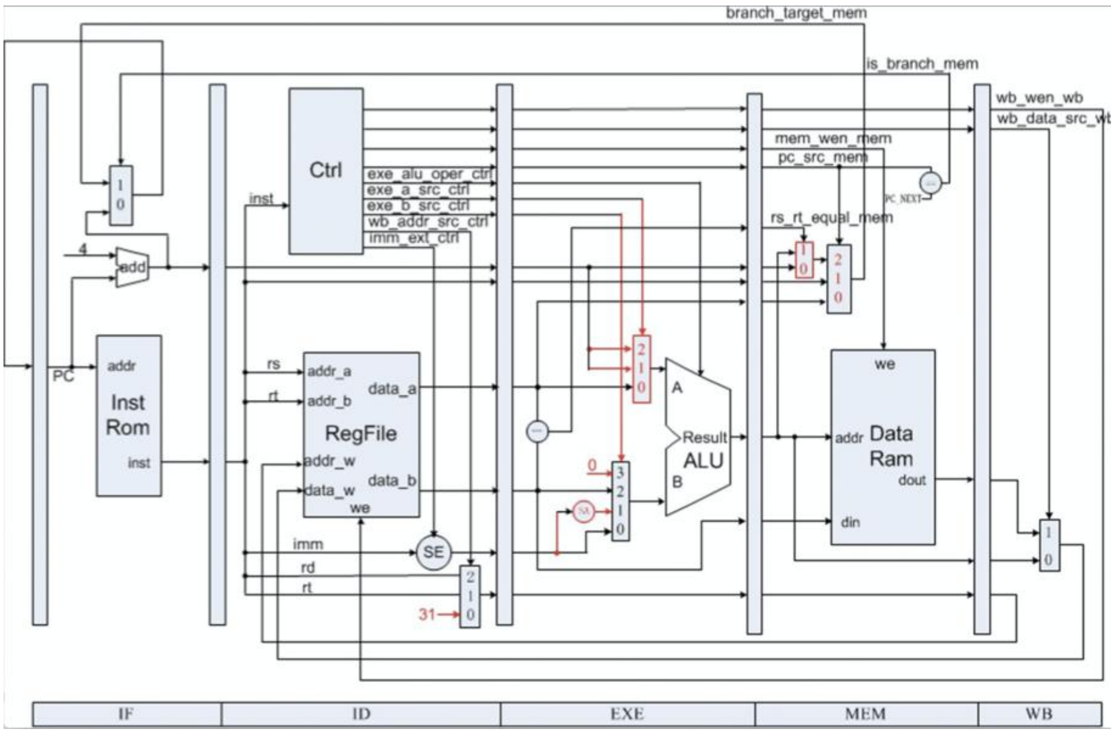


## 2.2.3.Datapath of 5-stages pipelined CPU

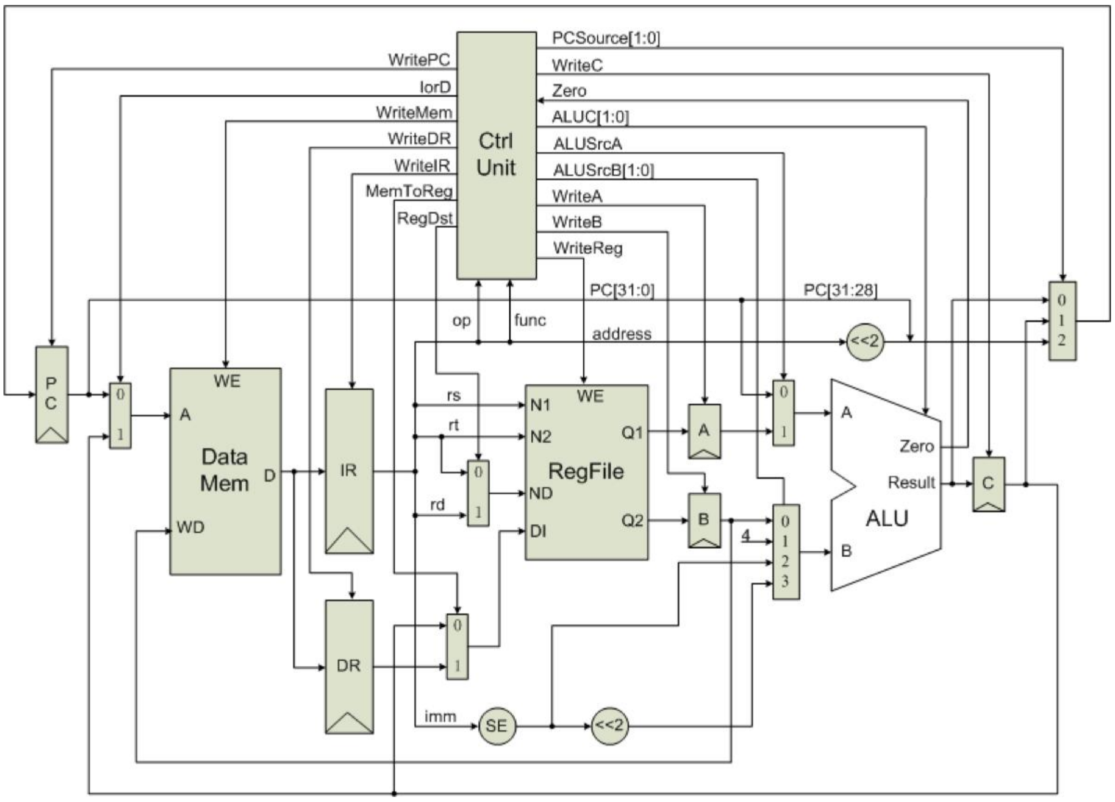
- **The datapath from the slides:**



- **My design:**



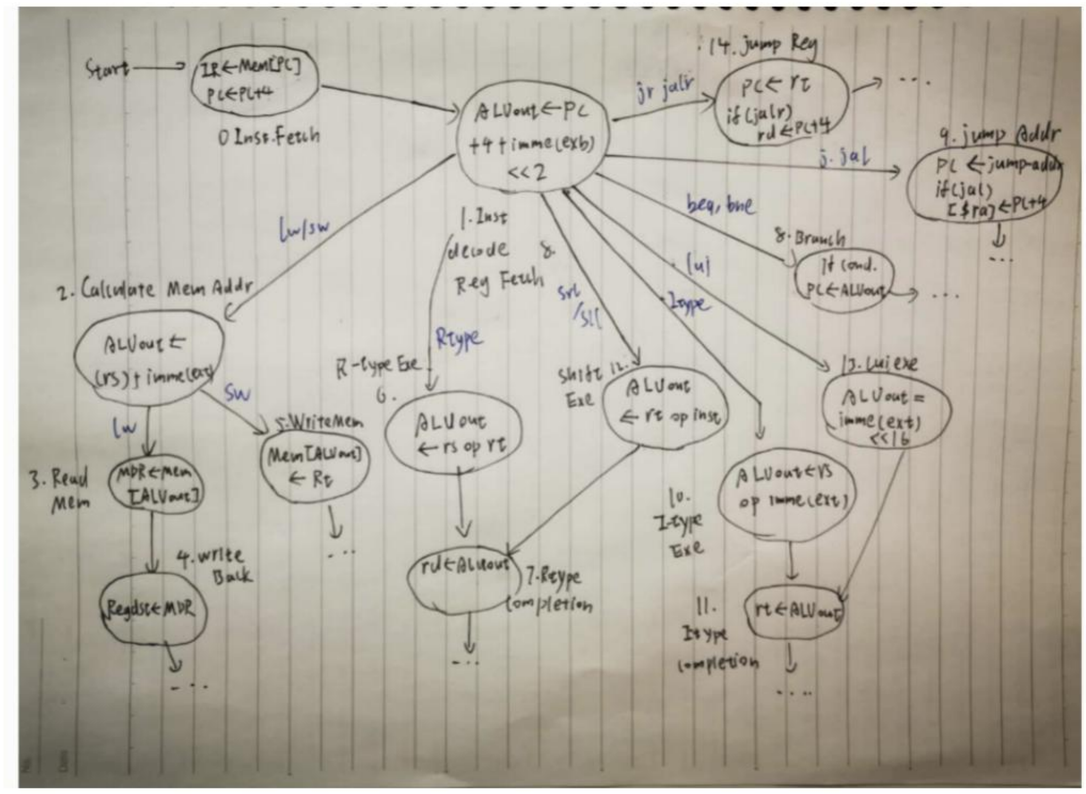
## ● The principle of Multiple-cycle CPU



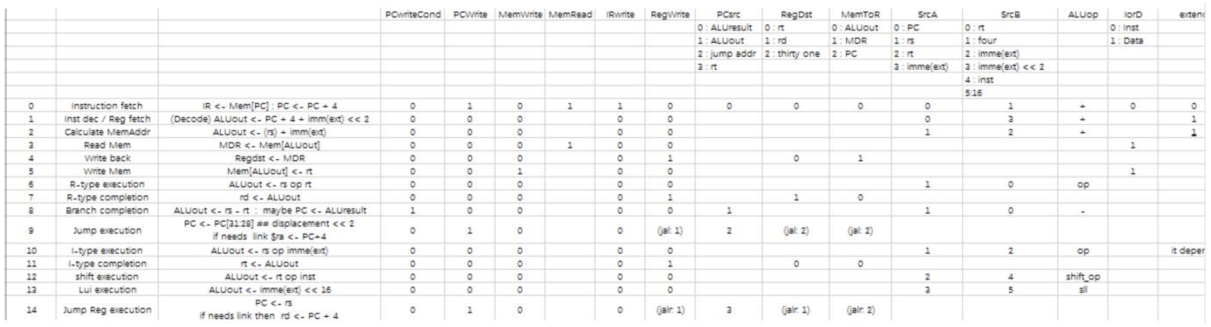
## State machine with 18+ instructions: drawn according to design instructions:

### State Figure

We will use finite state machine model to implement MCPU. And my design can be seen as follows:

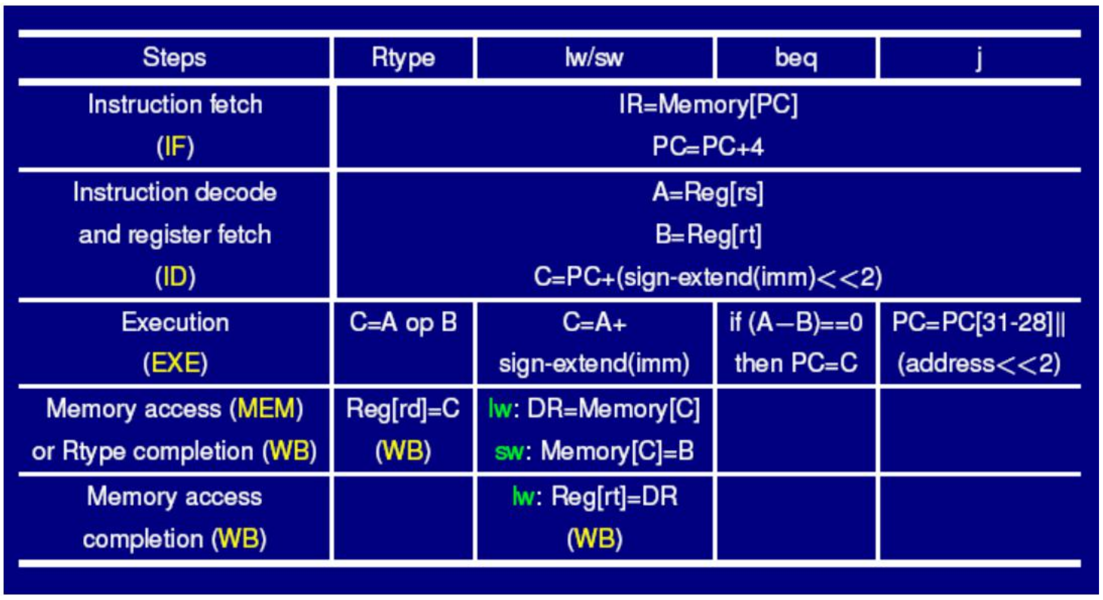


## True Table:

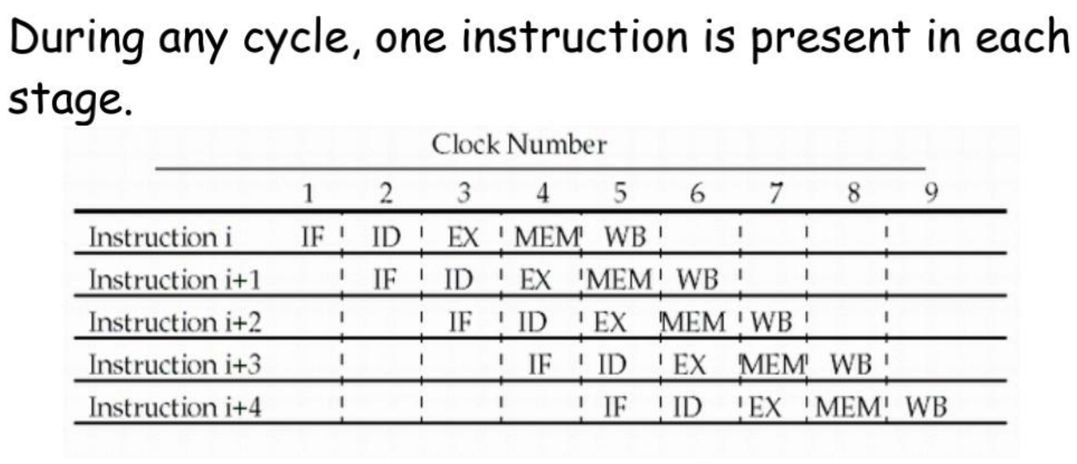


| | | | PCwriteCond | PCWrite | MemWrite | MemRead | IRwrite | RegWrite | PCsrc | RegDst | MemToR | SrcA | SrcB | ALUop | IorD | exten |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | 0 : ALUresult | 0 : rt | 0 : ALUout | 0 : PC | 0 : rt | | 0 : Inst | |
| | | | | | | | | | 1 : ALUout | 1 : rd | 1 : MDR | 1 : rs | 1 : four | | 1 : Data | |
| | | | | | | | | | 2 : jump addr | 2 : thirty one | 2 : PC | 2 : rt | 2 : imme(ext) | | | |
| | | | | | | | | | 3 : rt | | | 3 : imme(ext) | 3 : imme(ext) << 2 | | | |
| | | | | | | | | | | | | | 4 : inst | | | |
| | | | | | | | | | | | | | 5:16 | | | |
| 0 | Instruction fetch | IR <- Mem[PC] ; PC <- PC + 4 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | + | 0 | 0 |
| 1 | Inst dec / Reg fetch | (Decode) ALUout <- PC + 4 + imm(ext) << 2 | 0 | 0 | 0 | | 0 | 0 | | | | 0 | 3 | + | | 1 |
| 2 | Calculate MemAddr | ALUout <- (rs) + imm(ext) | 0 | 0 | 0 | | 0 | 0 | | | | 1 | 2 | + | | 1 |
| 3 | Read Mem | MDR <- Mem[ALUout] | 0 | 0 | 0 | 1 | 0 | 0 | | | | | | | 1 | |
| 4 | Write back | Regdst <- MDR | 0 | 0 | 0 | | 0 | 1 | | 0 | 1 | | | | | |
| 5 | Write Mem | Mem[ALUout] <- rt | 0 | 0 | 1 | | 0 | 0 | | | | | | | 1 | |
| 6 | R-type execution | ALUout <- rs op rt | 0 | 0 | 0 | | 0 | 0 | | | | 1 | 0 | op | | |
| 7 | R-type completion | rd <- ALUout | 0 | 0 | 0 | | 0 | 1 | | 1 | 0 | | | | | |
| 8 | Branch completion | ALUout <- rs - rt ; maybe PC <- ALUresult | 1 | 0 | 0 | | 0 | 0 | 1 | | | 1 | 0 | - | | |
| 9 | Jump execution | PC <- PC[31:28] ## displacement << 2<br>if needs link $ra <- PC+4 | 0 | 1 | 0 | | 0 | (jal: 1) | 2 | (jal: 2) | (jal: 2) | | | | | |
| 10 | I-type execution | ALUout <- rs op imme(ext) | 0 | 0 | 0 | | 0 | 0 | | | | 1 | 2 | op | | it deper |
| 11 | I-type completion | rt <- ALUout | 0 | 0 | 0 | | 0 | 1 | | 0 | 0 | | | | | |
| 12 | shift execution | ALUout <- rt op inst | 0 | 0 | 0 | | 0 | 0 | | | | 2 | 4 | shift_op | | |
| 13 | Lui execution | ALUout <- imme(ext) << 16 | 0 | 0 | 0 | | 0 | 0 | | | | 3 | 5 | sll | | |
| 14 | Jump Reg execution | PC <- rs<br>if needs link then rd <- PC + 4 | 0 | 1 | 0 | | 0 | (jalr: 1) | 3 | (jalr: 1) | (jalr: 2) | | | | | |

### 2.2.4.Design the 5 stages of the Pipelined CPU



| Steps | Rtype | lw/sw | beq | j |
|---|---|---|---|---|
| Instruction fetch (IF) | IR=Memory[PC] PC=PC+4 | | | |
| Instruction decode and register fetch (ID) | A=Reg[rs] B=Reg[rt] C=PC+(sign-extend(imm)<<2) | | | |
| Execution (EXE) | C=A op B | C=A+ sign-extend(imm) | if (A−B)==0 then PC=C | PC=PC[31-28]\|\| (address<<2) |
| Memory access (MEM) or Rtype completion (WB) | Reg[rd]=C (WB) | lw: DR=Memory[C] sw: Memory[C]=B | | |
| Memory access completion (WB) | | lw: Reg[rt]=DR (WB) | | |



During any cycle, one instruction is present in each stage.

| | Clock Number | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Instruction i | IF | ID | EX | MEM | WB | | | | |
| Instruction i+1 | | IF | ID | EX | MEM | WB | | | |
| Instruction i+2 | | | IF | ID | EX | MEM | WB | | |
| Instruction i+3 | | | | IF | ID | EX | MEM | WB | |
| Instruction i+4 | | | | | IF | ID | EX | MEM | WB |

### 2.2.5.Structural hazards – resource conflicts

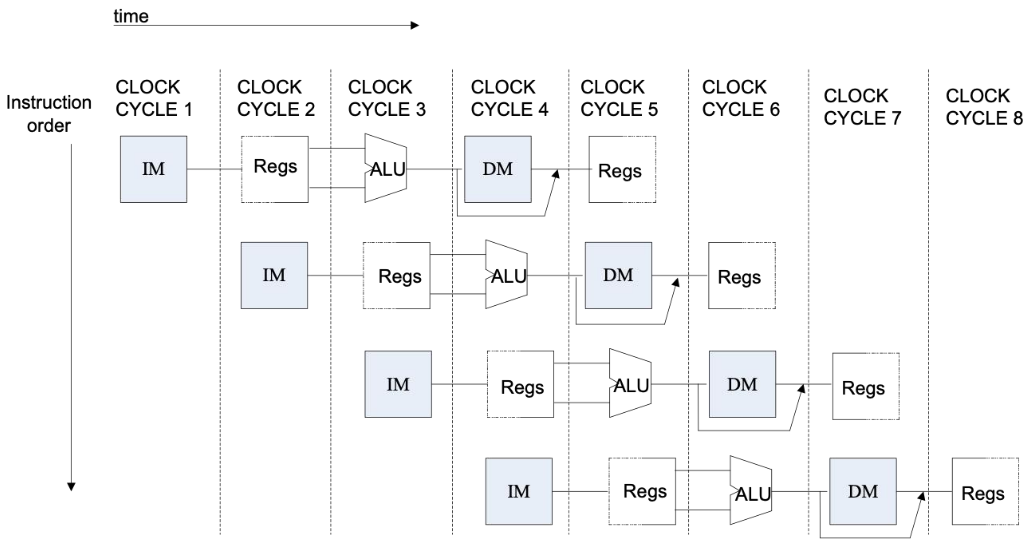
Structural hazards arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution:

–Memory conflicts

–Register File conflicts

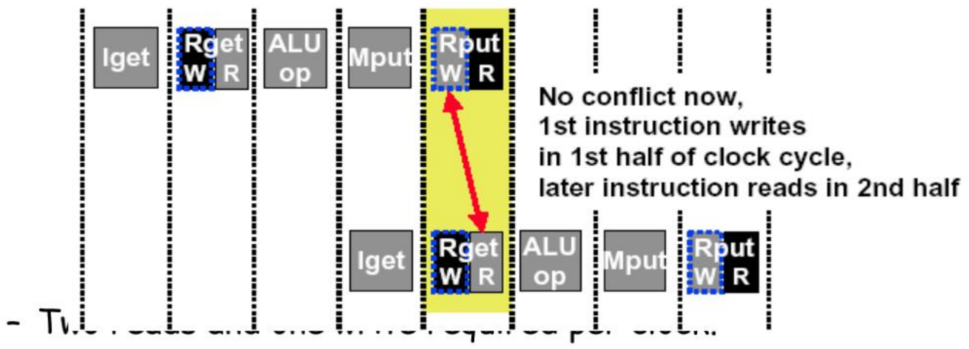
–Other units conflicts

- **And how to resolve Structural hazards?**



- **Use double bump :**

allow WRITE-then-READ in one clock cycle (double pump)



**In the experiments, it implement it in a different way:**

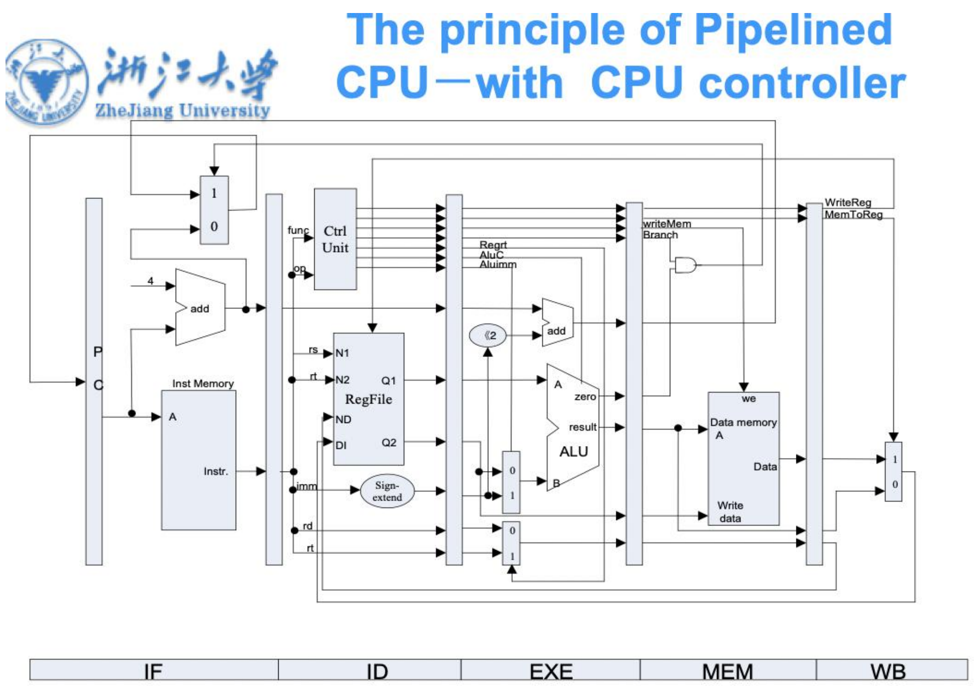
Register File

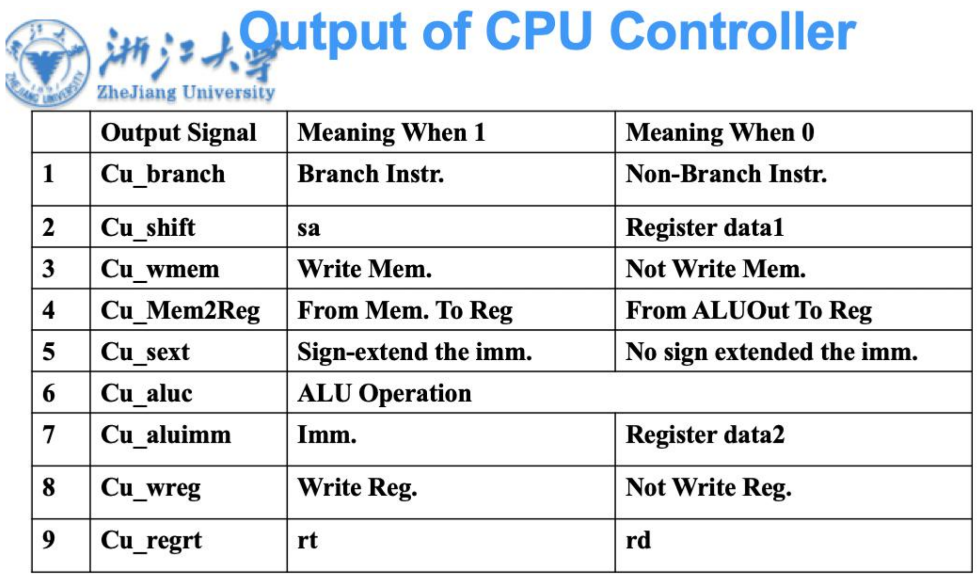
- Positive edge for transfer data for stages
- Negative edge for write operation
- Low level for read operation

**Memory:**

- Instruction Memory
  - Single Port Block Memory
  - Read only, Width:32
  - Falling Edge Triggered
- Data Memory
  - Single Port Block Memory
  - Read and write, Width:32
  - Falling Edge Triggered

## 2.2.6 CPU controller：

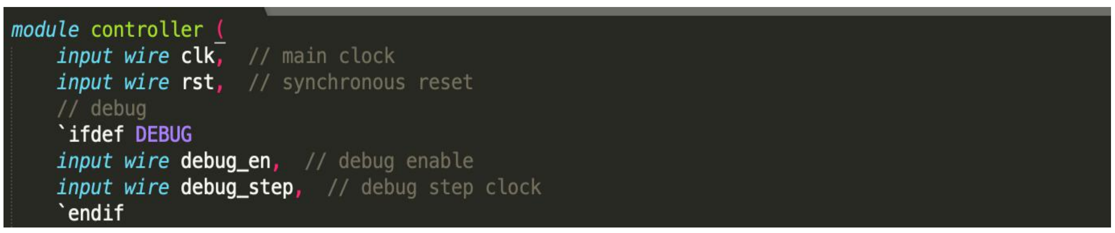

The principle of Pipelined CPU－with CPU controller


Output of CPU Controller
浙江大学
ZheJiang University

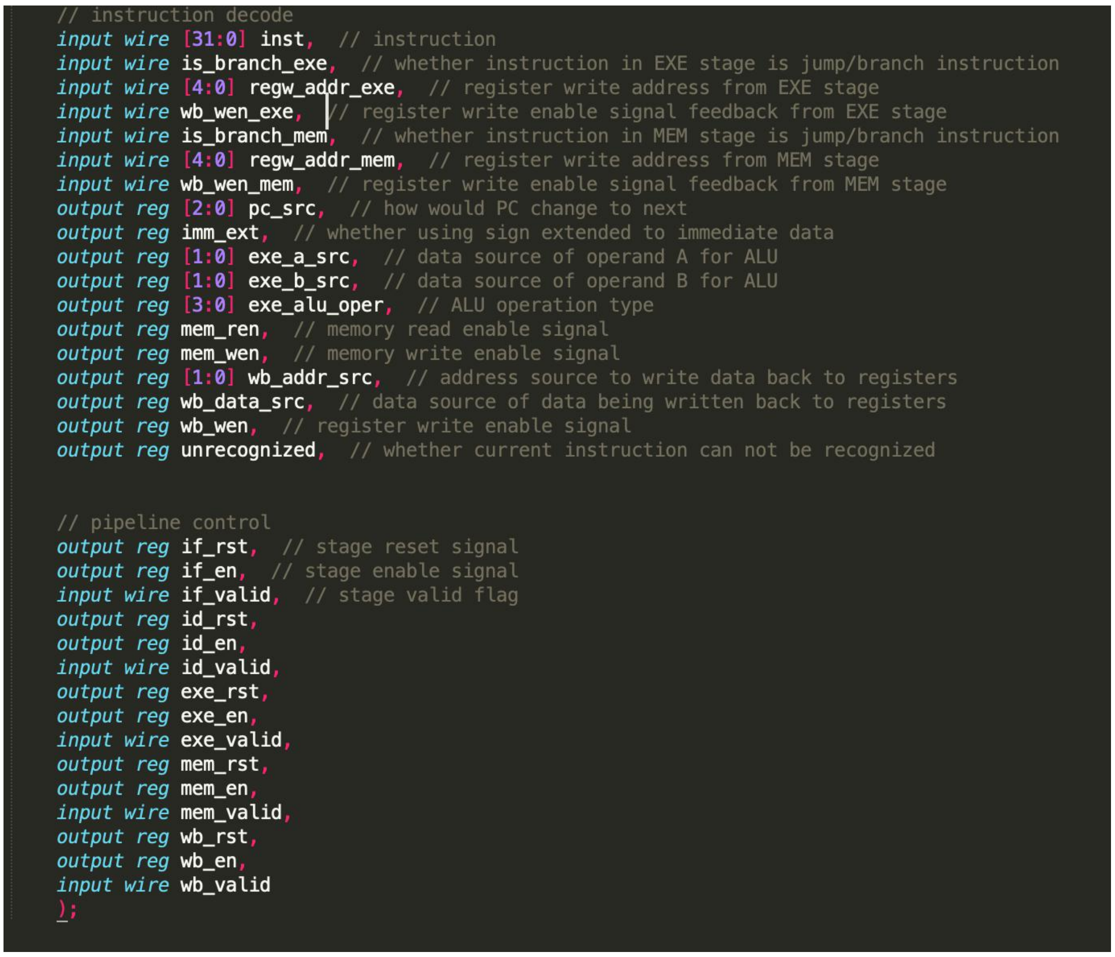
| | Output Signal | Meaning When 1 | Meaning When 0 |
|---|---|---|---|
| 1 | Cu_branch | Branch Instr. | Non-Branch Instr. |
| 2 | Cu_shift | sa | Register data1 |
| 3 | Cu_wmem | Write Mem. | Not Write Mem. |
| 4 | Cu_Mem2Reg | From Mem. To Reg | From ALUOut To Reg |
| 5 | Cu_sext | Sign-extend the imm. | No sign extended the imm. |
| 6 | Cu_aluc | ALU Operation | |
| 7 | Cu_aluimm | Imm. | Register data2 |
| 8 | Cu_wreg | Write Reg. | Not Write Reg. |
| 9 | Cu_regrt | rt | rd |

## 三、实验过程

### 3.1.Design the CPU Controller with debugger：

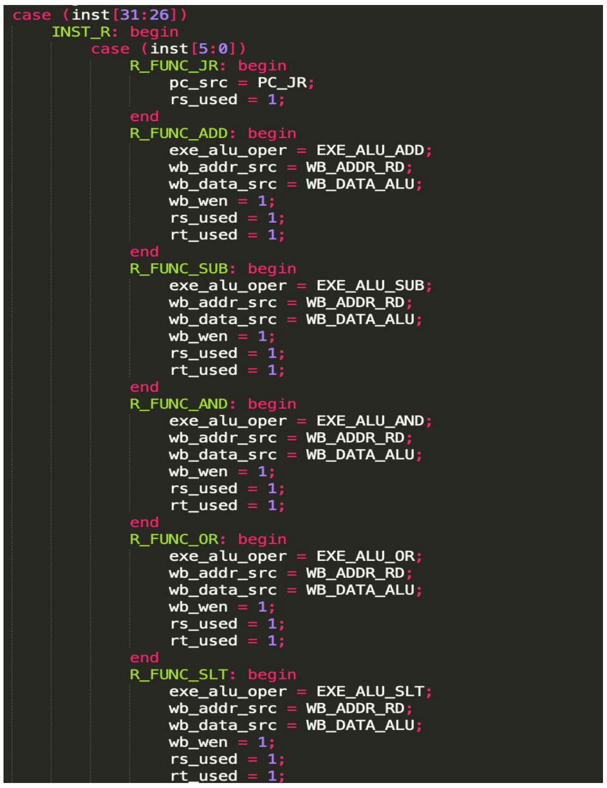


```
module controller (
    input wire clk,  // main clock
    input wire rst,  // synchronous reset
    // debug
    `ifdef DEBUG
    input wire debug_en,  // debug enable
    input wire debug_step,  // debug step clock
    `endif
```

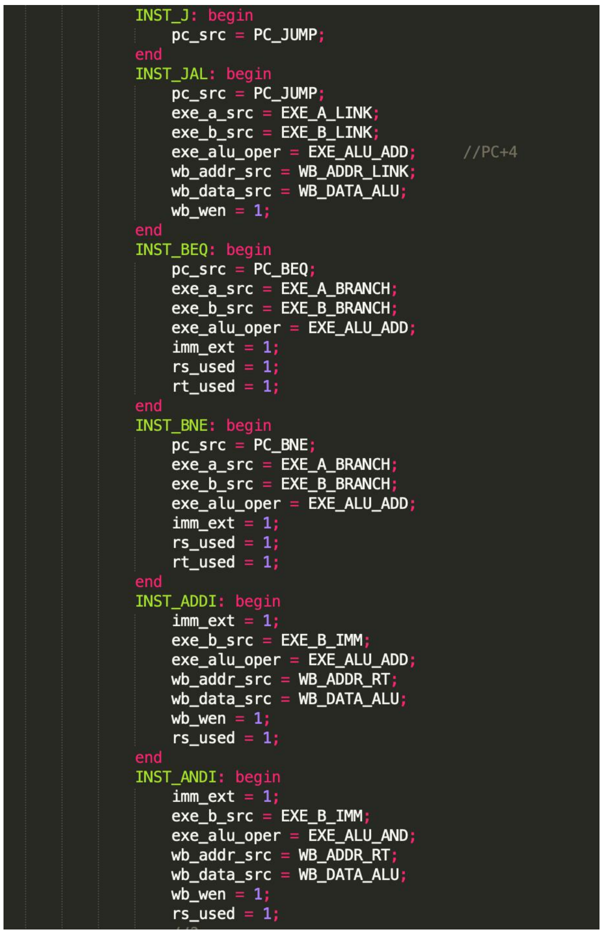


```
    // instruction decode
    input wire [31:0] inst,  // instruction
    input wire is_branch_exe,  // whether instruction in EXE stage is jump/branch instruction
    input wire [4:0] regw_addr_exe,  // register write address from EXE stage
    input wire wb_wen_exe,  // register write enable signal feedback from EXE stage
    input wire is_branch_mem,  // whether instruction in MEM stage is jump/branch instruction
    input wire [4:0] regw_addr_mem,  // register write address from MEM stage
    input wire wb_wen_mem,  // register write enable signal feedback from MEM stage
    output reg [2:0] pc_src,  // how would PC change to next
    output reg imm_ext,  // whether using sign extended to immediate data
    output reg [1:0] exe_a_src,  // data source of operand A for ALU
    output reg [1:0] exe_b_src,  // data source of operand B for ALU
    output reg [3:0] exe_alu_oper,  // ALU operation type
    output reg mem_ren,  // memory read enable signal
    output reg mem_wen,  // memory write enable signal
    output reg [1:0] wb_addr_src,  // address source to write data back to registers
    output reg wb_data_src,  // data source of data being written back to registers
    output reg wb_wen,  // register write enable signal
    output reg unrecognized,  // whether current instruction can not be recognized


    // pipeline control
    output reg if_rst,  // stage reset signal
    output reg if_en,  // stage enable signal
    input wire if_valid,  // stage valid flag
    output reg id_rst,
    output reg id_en,
    input wire id_valid,
    output reg exe_rst,
    output reg exe_en,
    input wire exe_valid,
    output reg mem_rst,
    output reg mem_en,
    input wire mem_valid,
    output reg wb_rst,
    output reg wb_en,
    input wire wb_valid
    );
```

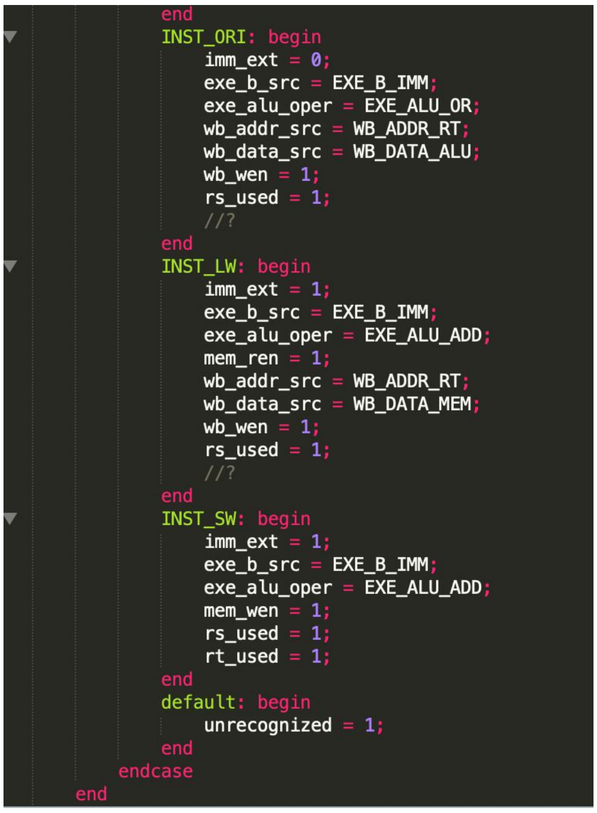
## Signals in the Controller:



```
case (inst[31:26])
    INST_R: begin
        case (inst[5:0])
            R_FUNC_JR: begin
                pc_src = PC_JR;
                rs_used = 1;
            end
            R_FUNC_ADD: begin
                exe_alu_oper = EXE_ALU_ADD;
                wb_addr_src = WB_ADDR_RD;
                wb_data_src = WB_DATA_ALU;
                wb_wen = 1;
                rs_used = 1;
                rt_used = 1;
            end
            R_FUNC_SUB: begin
                exe_alu_oper = EXE_ALU_SUB;
                wb_addr_src = WB_ADDR_RD;
                wb_data_src = WB_DATA_ALU;
                wb_wen = 1;
                rs_used = 1;
                rt_used = 1;
            end
            R_FUNC_AND: begin
                exe_alu_oper = EXE_ALU_AND;
                wb_addr_src = WB_ADDR_RD;
                wb_data_src = WB_DATA_ALU;
                wb_wen = 1;
                rs_used = 1;
                rt_used = 1;
            end
            R_FUNC_OR: begin
                exe_alu_oper = EXE_ALU_OR;
                wb_addr_src = WB_ADDR_RD;
                wb_data_src = WB_DATA_ALU;
                wb_wen = 1;
                rs_used = 1;
                rt_used = 1;
            end
            R_FUNC_SLT: begin
                exe_alu_oper = EXE_ALU_SLT;
                wb_addr_src = WB_ADDR_RD;
                wb_data_src = WB_DATA_ALU;
                wb_wen = 1;
                rs_used = 1;
                rt_used = 1;
```

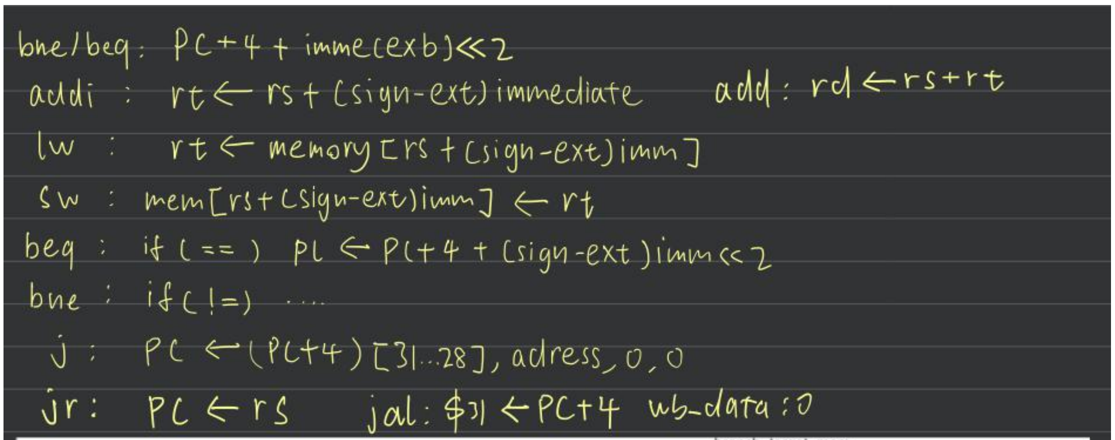


```
            INST_J: begin
                pc_src = PC_JUMP;
            end
            INST_JAL: begin
                pc_src = PC_JUMP;
                exe_a_src = EXE_A_LINK;
                exe_b_src = EXE_B_LINK;
                exe_alu_oper = EXE_ALU_ADD;      //PC+4
                wb_addr_src = WB_ADDR_LINK;
                wb_data_src = WB_DATA_ALU;
                wb_wen = 1;
            end
            INST_BEQ: begin
                pc_src = PC_BEQ;
                exe_a_src = EXE_A_BRANCH;
                exe_b_src = EXE_B_BRANCH;
                exe_alu_oper = EXE_ALU_ADD;
                imm_ext = 1;
                rs_used = 1;
                rt_used = 1;
            end
            INST_BNE: begin
                pc_src = PC_BNE;
                exe_a_src = EXE_A_BRANCH;
                exe_b_src = EXE_B_BRANCH;
                exe_alu_oper = EXE_ALU_ADD;
                imm_ext = 1;
                rs_used = 1;
                rt_used = 1;
            end
            INST_ADDI: begin
                imm_ext = 1;
                exe_b_src = EXE_B_IMM;
                exe_alu_oper = EXE_ALU_ADD;
                wb_addr_src = WB_ADDR_RT;
                wb_data_src = WB_DATA_ALU;
                wb_wen = 1;
                rs_used = 1;
            end
            INST_ANDI: begin
                imm_ext = 1;
                exe_b_src = EXE_B_IMM;
                exe_alu_oper = EXE_ALU_AND;
                wb_addr_src = WB_ADDR_RT;
                wb_data_src = WB_DATA_ALU;
                wb_wen = 1;
                rs_used = 1;
```



```
            end
            INST_ORI: begin
                imm_ext = 0;
                exe_b_src = EXE_B_IMM;
                exe_alu_oper = EXE_ALU_OR;
                wb_addr_src = WB_ADDR_RT;
                wb_data_src = WB_DATA_ALU;
                wb_wen = 1;
                rs_used = 1;
                //?
            end
            INST_LW: begin
                imm_ext = 1;
                exe_b_src = EXE_B_IMM;
                exe_alu_oper = EXE_ALU_ADD;
                mem_ren = 1;
                wb_addr_src = WB_ADDR_RT;
                wb_data_src = WB_DATA_MEM;
                wb_wen = 1;
                rs_used = 1;
                //?
            end
            INST_SW: begin
                imm_ext = 1;
                exe_b_src = EXE_B_IMM;
                exe_alu_oper = EXE_ALU_ADD;
                mem_wen = 1;
                rs_used = 1;
                rt_used = 1;
            end
            default: begin
                unrecognized = 1;
            end
        endcase
    end
```

## 3.2.Design of the datapath:



bne/beq: PC+4 + imme(exb)≪2

addi : rt ← rs + (sign-ext) immediate     add: rd ← rs+rt

lw : rt ← memory[rs + (sign-ext)imm]

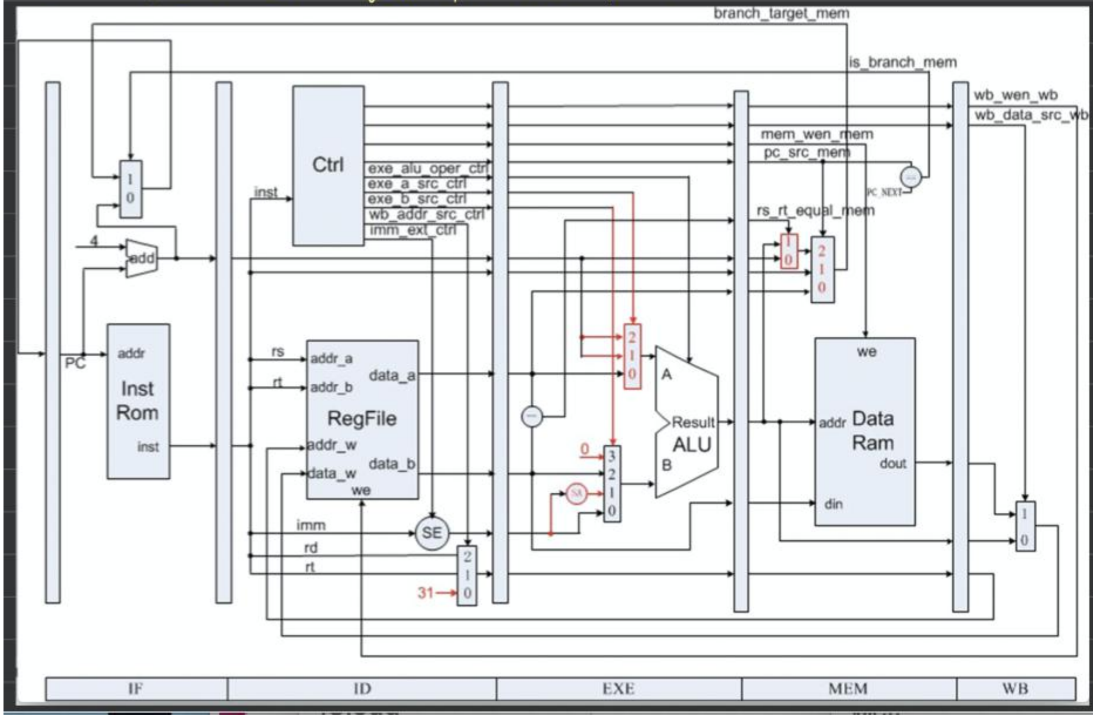
sw : mem[rs+(sign-ext)imm] ← rt

beq : if (==) PC ← PC+4 + (sign-ext)imm<<2

bne : if (!=) ....

j : PC ← (PC+4)[31..28], adress, 0, 0

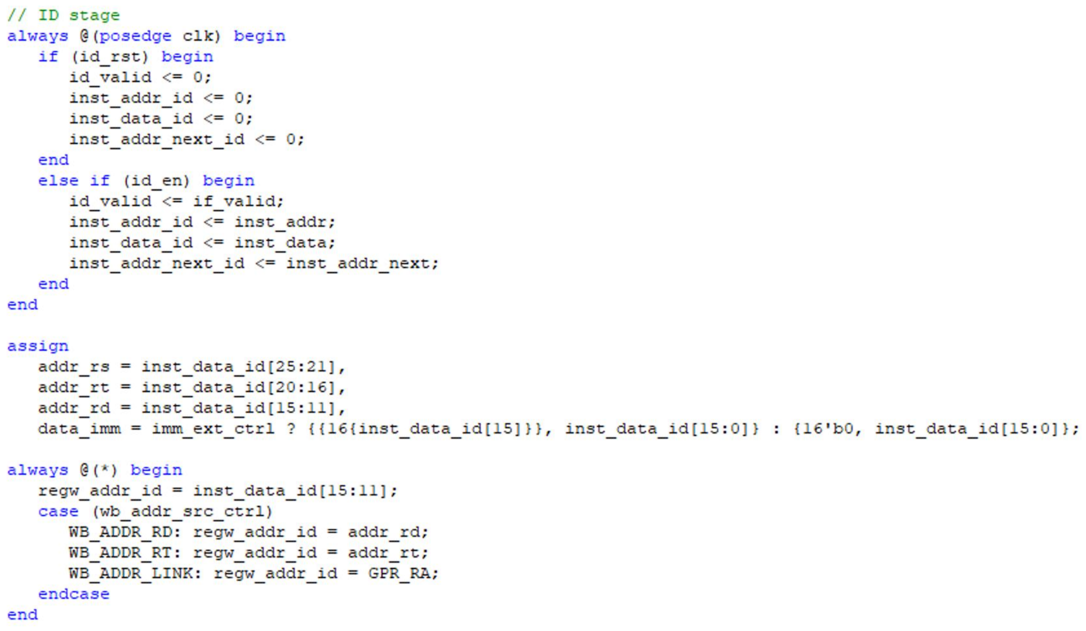
jr: PC ← rs     jal: $31 ← PC+4   wb_data: 0



IF stage:

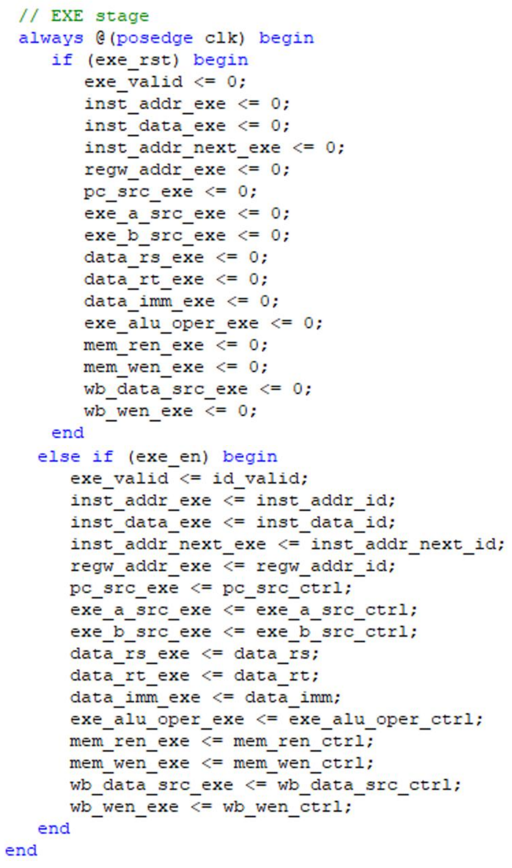
```
// IF stage
assign
   inst_addr_next = inst_addr + 4;

always @(*) begin
   if_valid = ~if_rst & if_en;
   inst_ren = ~if_rst;
end

always @(posedge clk) begin
   if (if_rst) begin
      inst_addr <= 0;
   end
   else if (if_en) begin
      if (is_branch_mem)
         inst_addr <= branch_target_mem;
      else
         inst_addr <= inst_addr_next;
   end
end
```

ID stage:



```
// ID stage
always @(posedge clk) begin
   if (id_rst) begin
      id_valid <= 0;
      inst_addr_id <= 0;
      inst_data_id <= 0;
      inst_addr_next_id <= 0;
   end
   else if (id_en) begin
      id_valid <= if_valid;
      inst_addr_id <= inst_addr;
      inst_data_id <= inst_data;
      inst_addr_next_id <= inst_addr_next;
   end
end

assign
   addr_rs = inst_data_id[25:21],
   addr_rt = inst_data_id[20:16],
   addr_rd = inst_data_id[15:11],
   data_imm = imm_ext_ctrl ? {{16{inst_data_id[15]}}, inst_data_id[15:0]} : {16'b0, inst_data_id[15:0]};

always @(*) begin
   regw_addr_id = inst_data_id[15:11];
   case (wb_addr_src_ctrl)
      WB_ADDR_RD: regw_addr_id = addr_rd;
      WB_ADDR_RT: regw_addr_id = addr_rt;
      WB_ADDR_LINK: regw_addr_id = GPR_RA;
   endcase
end
```

EXE stage with rst signal:



```
// EXE stage
always @(posedge clk) begin
   if (exe_rst) begin
      exe_valid <= 0;
      inst_addr_exe <= 0;
      inst_data_exe <= 0;
      inst_addr_next_exe <= 0;
      regw_addr_exe <= 0;
      pc_src_exe <= 0;
      exe_a_src_exe <= 0;
      exe_b_src_exe <= 0;
      data_rs_exe <= 0;
      data_rt_exe <= 0;
      data_imm_exe <= 0;
      exe_alu_oper_exe <= 0;
      mem_ren_exe <= 0;
      mem_wen_exe <= 0;
      wb_data_src_exe <= 0;
      wb_wen_exe <= 0;
   end
  else if (exe_en) begin
     exe_valid <= id_valid;
     inst_addr_exe <= inst_addr_id;
     inst_data_exe <= inst_data_id;
     inst_addr_next_exe <= inst_addr_next_id;
     regw_addr_exe <= regw_addr_id;
     pc_src_exe <= pc_src_ctrl;
     exe_a_src_exe <= exe_a_src_ctrl;
     exe_b_src_exe <= exe_b_src_ctrl;
     data_rs_exe <= data_rs;
     data_rt_exe <= data_rt;
     data_imm_exe <= data_imm;
     exe_alu_oper_exe <= exe_alu_oper_ctrl;
     mem_ren_exe <= mem_ren_ctrl;
     mem_wen_exe <= mem_wen_ctrl;
     wb_data_src_exe <= wb_data_src_ctrl;
     wb_wen_exe <= wb_wen_ctrl;
  end
end
```

MEM stage with rst signal:

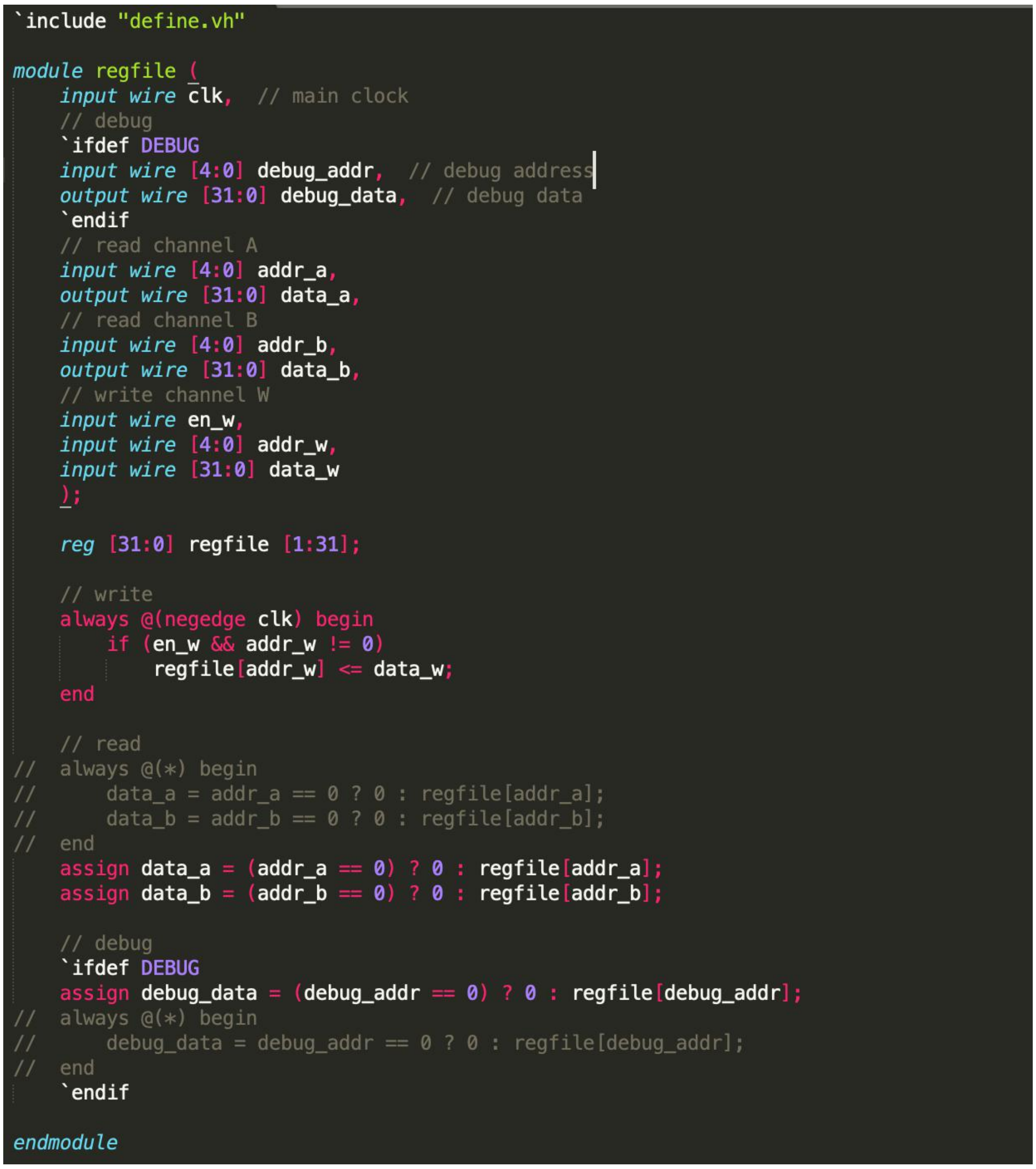
```
// MEM stage
always @(posedge clk) begin
   if (mem_rst) begin
      mem_valid <= 0;
      pc_src_mem <= 0;
      inst_addr_mem <= 0;
      inst_data_mem <= 0;
      inst_addr_next_mem <= 0;
      regw_addr_mem <= 0;
      data_rs_mem <= 0;
      data_rt_mem <= 0;
      alu_out_mem <= 0;
      mem_ren_mem <= 0;
      mem_wen_mem <= 0;
      wb_data_src_mem <= 0;
      wb_wen_mem <= 0;
      rs_rt_equal_mem <= 0;
   end

  else if (mem_en) begin
     mem_valid <= exe_valid;
     pc_src_mem <= pc_src_exe;
     inst_addr_mem <= inst_addr_exe;
     inst_data_mem <= inst_data_exe;
     inst_addr_next_mem <= inst_addr_next_exe;
     regw_addr_mem <= regw_addr_exe;
     data_rs_mem <= data_rs_exe;
     data_rt_mem <= data_rt_exe;
     alu_out_mem <= alu_out_exe;
     mem_ren_mem <= mem_ren_exe;
     mem_wen_mem <= mem_wen_exe;
     wb_data_src_mem <= wb_data_src_exe;
     wb_wen_mem <= wb_wen_exe;
     rs_rt_equal_mem <= rs_rt_equal_exe;
  end
end
```

WB stage:

```
// WB stage
always @(posedge clk) begin
   if (wb_rst) begin
      wb_valid <= 0;
      wb_wen_wb <= 0;
      wb_data_src_wb <= 0;
      regw_addr_wb <= 0;
      alu_out_wb <= 0;
      mem_din_wb <= 0;
   end
   else if (wb_en) begin
      wb_valid <= mem_valid;
      wb_wen_wb <= wb_wen_mem;
      wb_data_src_wb <= wb_data_src_mem;
      regw_addr_wb <= regw_addr_mem;
      alu_out_wb <= alu_out_mem;
      mem_din_wb <= mem_din;
   end
end

always @(*) begin
   regw_data_wb = alu_out_wb;
   case (wb_data_src_wb)
      WB_DATA_ALU: regw_data_wb = alu_out_mem;
      WB_DATA_MEM: regw_data_wb = mem_din;
   endcase
end
```

## 3.3.Register file:

### Register with double bump



```
`include "define.vh"

module regfile (
    input wire clk,  // main clock
    // debug
    `ifdef DEBUG
    input wire [4:0] debug_addr,  // debug address
    output wire [31:0] debug_data,  // debug data
    `endif
    // read channel A
    input wire [4:0] addr_a,
    output wire [31:0] data_a,
    // read channel B
    input wire [4:0] addr_b,
    output wire [31:0] data_b,
    // write channel W
    input wire en_w,
    input wire [4:0] addr_w,
    input wire [31:0] data_w
    );

    reg [31:0] regfile [1:31];

    // write
    always @(negedge clk) begin
        if (en_w && addr_w != 0)
            regfile[addr_w] <= data_w;
    end

    // read
//  always @(*) begin
//      data_a = addr_a == 0 ? 0 : regfile[addr_a];
//      data_b = addr_b == 0 ? 0 : regfile[addr_b];
//  end
    assign data_a = (addr_a == 0) ? 0 : regfile[addr_a];
    assign data_b = (addr_b == 0) ? 0 : regfile[addr_b];

    // debug
    `ifdef DEBUG
    assign debug_data = (debug_addr == 0) ? 0 : regfile[debug_addr];
//  always @(*) begin
//      debug_data = debug_addr == 0 ? 0 : regfile[debug_addr];
//  end
    `endif

endmodule
```

### Detail:

**Double bump:**

**There are two ways to implement it :**

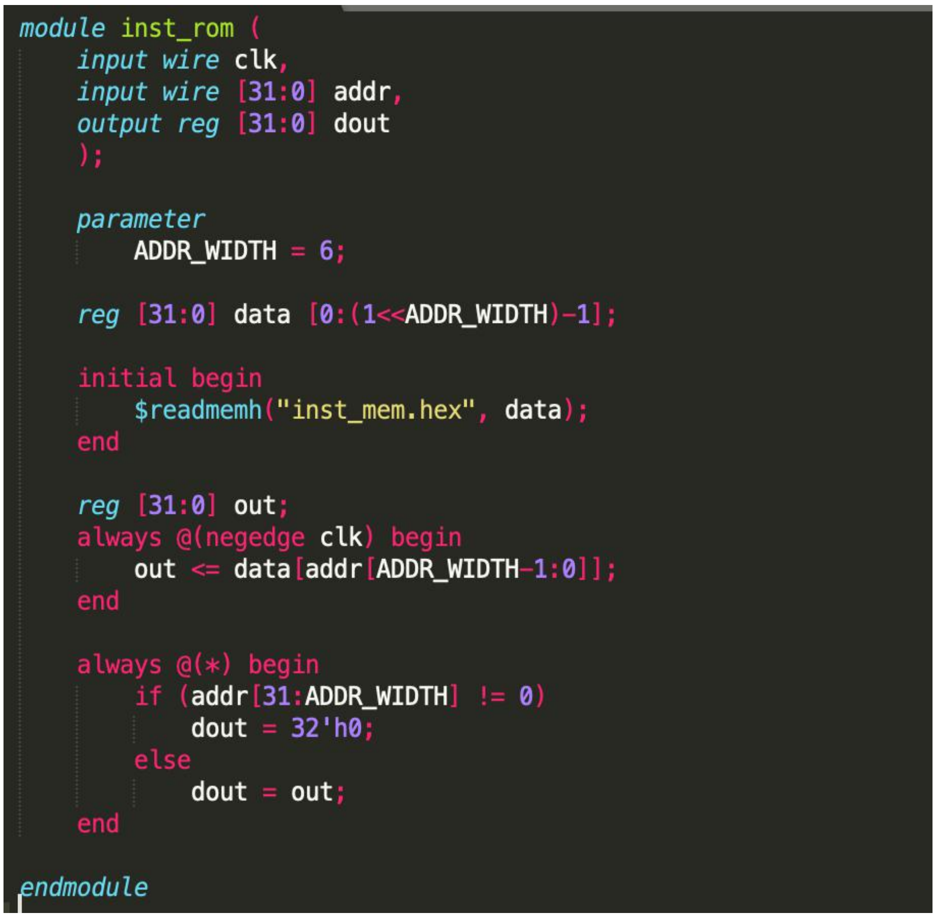
**And here all the signal here in the latches read at the positive edge, so the register file write at the negative edge**

## 3.4.Memory:

**In order to resolve the structure hazard, we need to divide the memory into two parts: one for the inst, and one for the data**

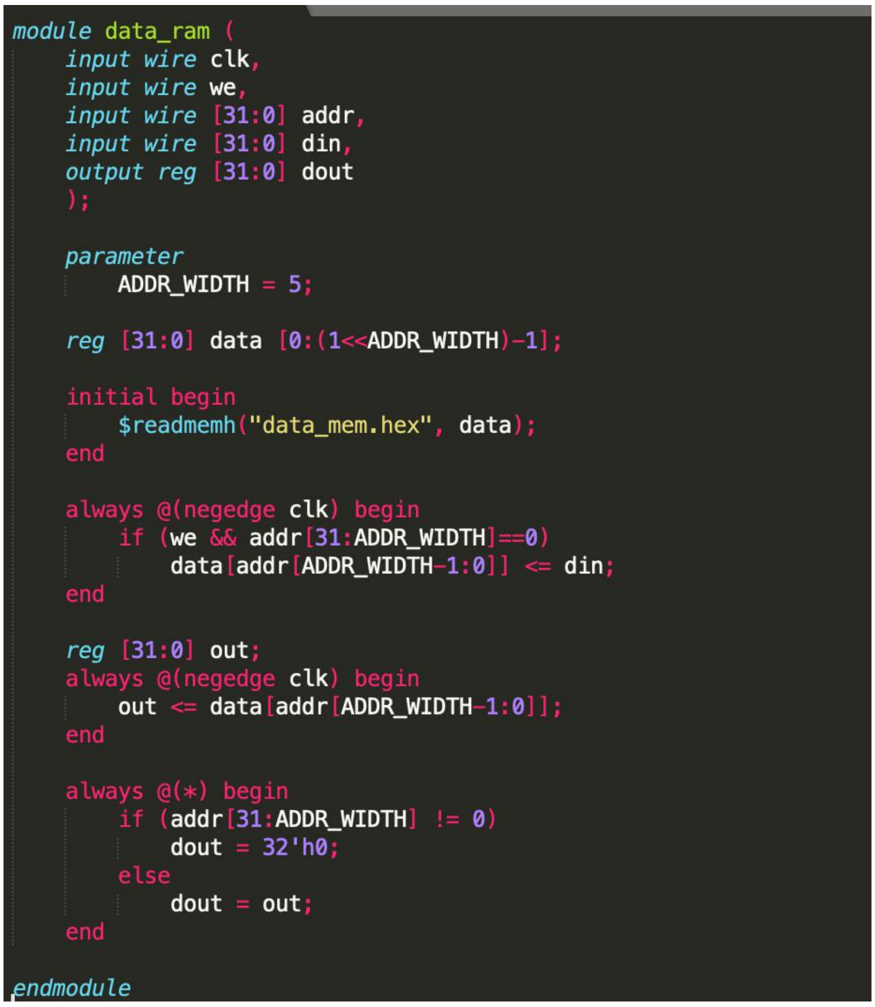
**(Read the hex file in the current folder)**

- **Inst_rom:**



```
module inst_rom (
    input wire clk,
    input wire [31:0] addr,
    output reg [31:0] dout
    );

    parameter
        ADDR_WIDTH = 6;

    reg [31:0] data [0:(1<<ADDR_WIDTH)-1];

    initial begin
        $readmemh("inst_mem.hex", data);
    end

    reg [31:0] out;
    always @(negedge clk) begin
        out <= data[addr[ADDR_WIDTH-1:0]];
    end

    always @(*) begin
        if (addr[31:ADDR_WIDTH] != 0)
            dout = 32'h0;
        else
            dout = out;
    end

endmodule
```

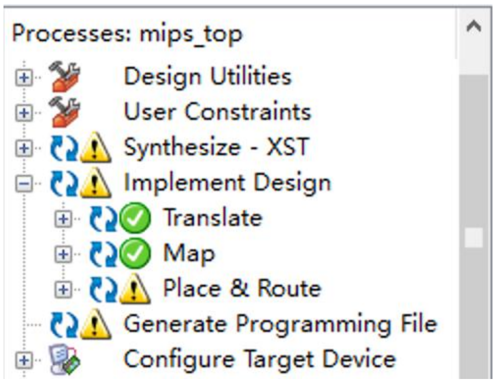
- **data_ram:**



```
module data_ram (
    input wire clk,
    input wire we,
    input wire [31:0] addr,
    input wire [31:0] din,
    output reg [31:0] dout
    );

    parameter
        ADDR_WIDTH = 5;

    reg [31:0] data [0:(1<<ADDR_WIDTH)-1];

    initial begin
        $readmemh("data_mem.hex", data);
    end

    always @(negedge clk) begin
        if (we && addr[31:ADDR_WIDTH]==0)
            data[addr[ADDR_WIDTH-1:0]] <= din;
    end

    reg [31:0] out;
    always @(negedge clk) begin
        out <= data[addr[ADDR_WIDTH-1:0]];
    end

    always @(*) begin
        if (addr[31:ADDR_WIDTH] != 0)
            dout = 32'h0;
        else
            dout = out;
    end

endmodule
```

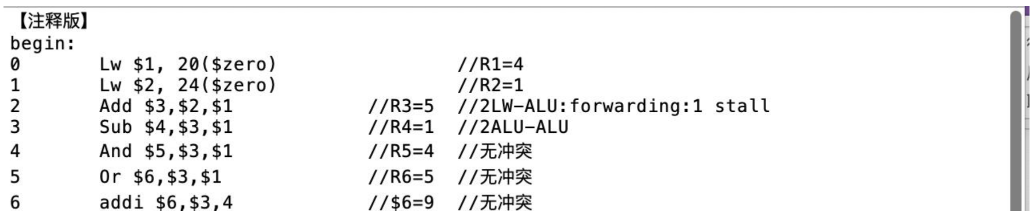
## 四、实验结果分析

编译:



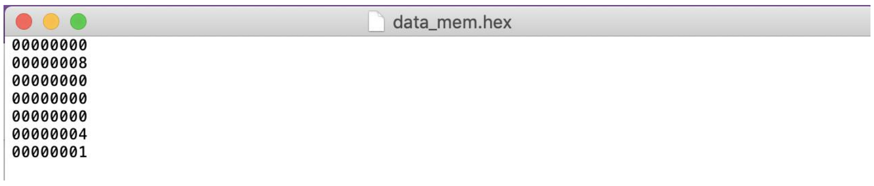
```
Processes: mips_top
⊞ Design Utilities
⊞ User Constraints
⊞ Synthesize - XST
⊟ Implement Design
    ⊞ Translate
    ⊞ Map
    ⊞ Place & Route
  Generate Programming File
⊞ Configure Target Device
```

Use the code given:

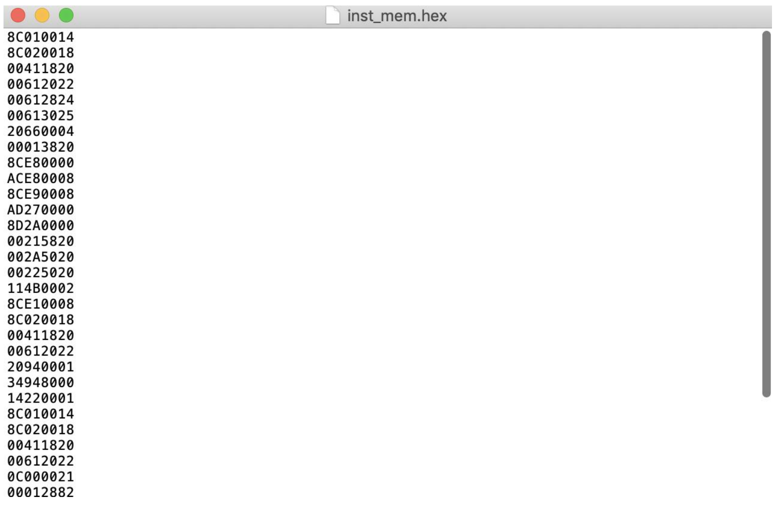


```
【注释版】
begin:
0       Lw $1, 20($zero)                //R1=4
1       Lw $2, 24($zero)                //R2=1
2       Add $3,$2,$1            //R3=5  //2LW-ALU:forwarding:1 stall
3       Sub $4,$3,$1            //R4=1  //2ALU-ALU
4       And $5,$3,$1            //R5=4  //无冲突
5       Or $6,$3,$1             //R6=5  //无冲突
6       addi $6,$3,4            //$6=9  //无冲突
```

Put the hex file in the current folder and have a test:

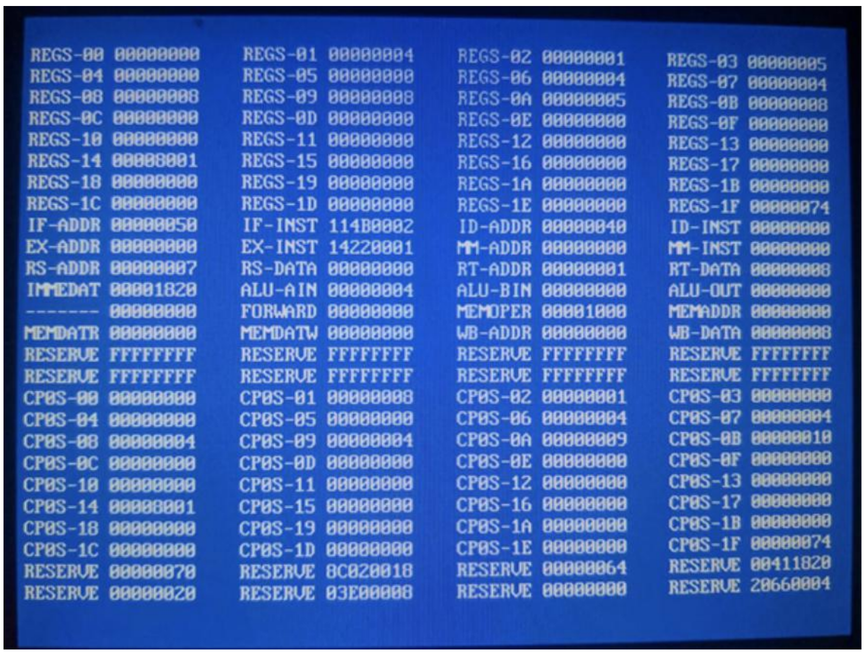


```
00000000
00000008
00000000
00000000
00000000
00000004
00000001
```



```
8C010014
8C020018
00411820
00612022
00612824
00613025
20660004
00013820
8CE80000
ACE80008
8CE90008
AD270000
8D2A0000
00215820
002A5020
00225020
114B0002
8CE10008
8C020018
00411820
00612022
20940001
34948000
14220001
8C010014
8C020018
00411820
00612022
0C000021
00012882
```

**Vga:**

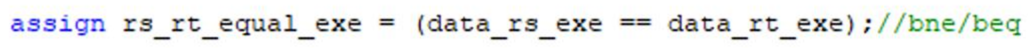


```
REGS-00  00000000   REGS-01  00000004   REGS-02  00000001   REGS-03  00000005
REGS-04  00000000   REGS-05  00000000   REGS-06  00000004   REGS-07  00000004
REGS-08  00000008   REGS-09  00000008   REGS-0A  00000005   REGS-0B  00000008
REGS-0C  00000000   REGS-0D  00000000   REGS-0E  00000000   REGS-0F  00000000
REGS-10  00000000   REGS-11  00000000   REGS-12  00000000   REGS-13  00000000
REGS-14  00008001   REGS-15  00000000   REGS-16  00000000   REGS-17  00000000
REGS-18  00000000   REGS-19  00000000   REGS-1A  00000000   REGS-1B  00000000
REGS-1C  00000000   REGS-1D  00000000   REGS-1E  00000000   REGS-1F  00000074
IF-ADDR  00000050   IF-INST  114B0002   ID-ADDR  00000040   ID-INST  00000000
EX-ADDR  00000000   EX-INST  14220001   MM-ADDR  00000000   MM-INST  00000000
RS-ADDR  00000007   RS-DATA  00000000   RT-ADDR  00000001   RT-DATA  00000008
IMMEDAT  00001820   ALU-AIN  00000004   ALU-BIN  00000000   ALU-OUT  00000000
-------  00000000   FORWARD  00000000   MEMOPER  00001000   MEMADDR  00000000
MEMDATR  00000000   MEMDATW  00000000   WB-ADDR  00000000   WB-DATA  00000008
RESERVE  FFFFFFFF   RESERVE  FFFFFFFF   RESERVE  FFFFFFFF   RESERVE  FFFFFFFF
RESERVE  FFFFFFFF   RESERVE  FFFFFFFF   RESERVE  FFFFFFFF   RESERVE  FFFFFFFF
CP0S-00  00000000   CP0S-01  00000008   CP0S-02  00000001   CP0S-03  00000000
CP0S-04  00000000   CP0S-05  00000000   CP0S-06  00000004   CP0S-07  00000004
CP0S-08  00000004   CP0S-09  00000004   CP0S-0A  00000009   CP0S-0B  00000010
CP0S-0C  00000000   CP0S-0D  00000000   CP0S-0E  00000000   CP0S-0F  00000000
CP0S-10  00000000   CP0S-11  00000000   CP0S-12  00000000   CP0S-13  00000000
CP0S-14  00008001   CP0S-15  00000000   CP0S-16  00000000   CP0S-17  00000000
CP0S-18  00000000   CP0S-19  00000000   CP0S-1A  00000000   CP0S-1B  00000000
CP0S-1C  00000000   CP0S-1D  00000000   CP0S-1E  00000000   CP0S-1F  00000074
RESERVE  00000070   RESERVE  8C020018   RESERVE  00000064   RESERVE  00411820
RESERVE  00000020   RESERVE  03E00008   RESERVE  00000000   RESERVE  20660004
```

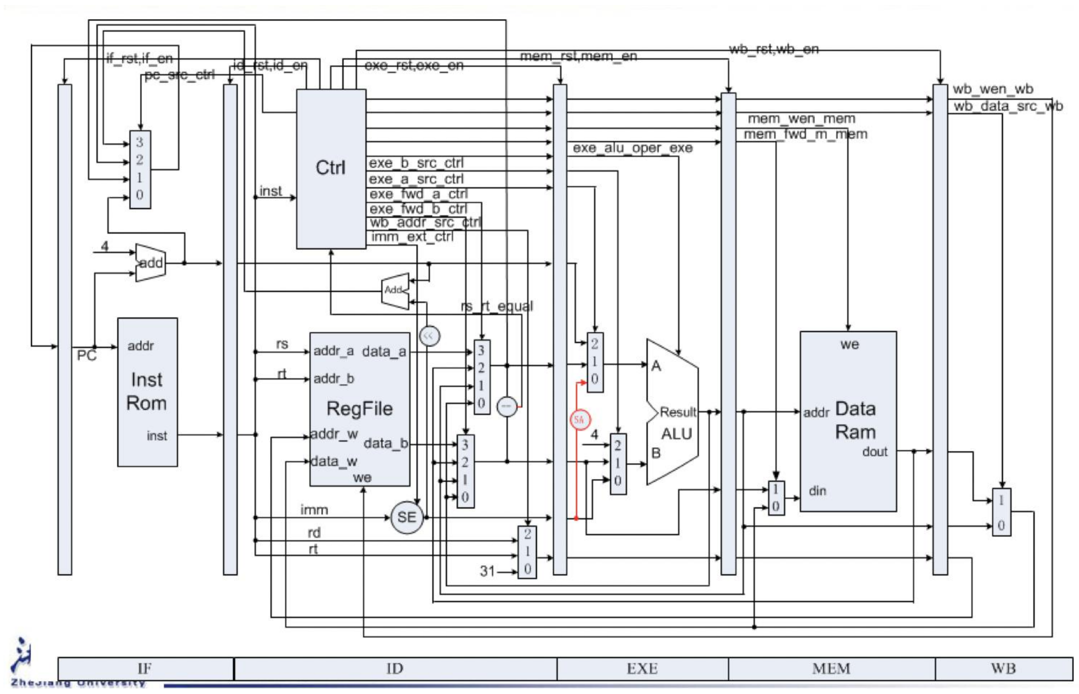
## 五、实验总结与反思

1.Specifically, in the implementation of the bne code, the calculation of the address in the EXE stage and the judgment of whether to jump are selected in this stage. In order to facilitate future experiments, the steps of judgment and address calculation should be moved forward. Moreover, in order to advance the ALU module used to calculate the address, an adder should be added.

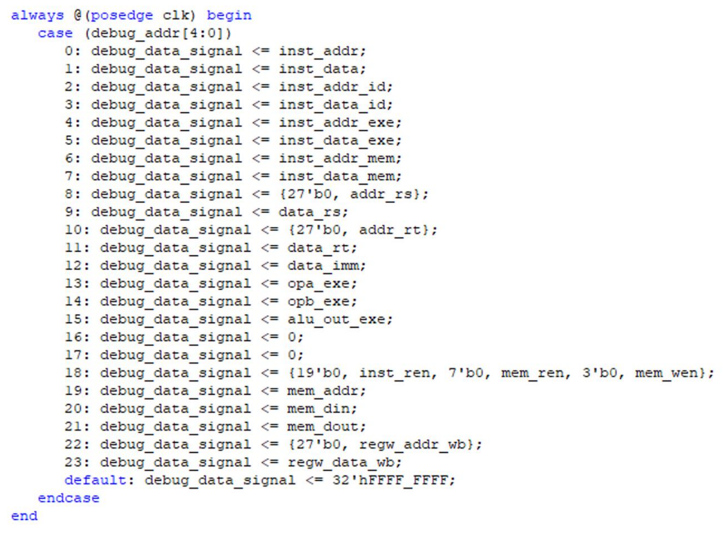
Code: in the EXE stage:



```
assign rs_rt_equal_exe = (data_rs_exe == data_rt_exe);//bne/beq
```



2.The vga debugger is need to be added into every module that has the signals in the following code, and be careful not to miss any:



```
always @(posedge clk) begin
    case (debug_addr[4:0])
        0: debug_data_signal <= inst_addr;
        1: debug_data_signal <= inst_data;
        2: debug_data_signal <= inst_addr_id;
        3: debug_data_signal <= inst_data_id;
        4: debug_data_signal <= inst_addr_exe;
        5: debug_data_signal <= inst_data_exe;
        6: debug_data_signal <= inst_addr_mem;
        7: debug_data_signal <= inst_data_mem;
        8: debug_data_signal <= {27'b0, addr_rs};
        9: debug_data_signal <= data_rs;
        10: debug_data_signal <= {27'b0, addr_rt};
        11: debug_data_signal <= data_rt;
        12: debug_data_signal <= data_imm;
        13: debug_data_signal <= opa_exe;
        14: debug_data_signal <= opb_exe;
        15: debug_data_signal <= alu_out_exe;
        16: debug_data_signal <= 0;
        17: debug_data_signal <= 0;
        18: debug_data_signal <= {19'b0, inst_ren, 7'b0, mem_ren, 3'b0, mem_wen};
        19: debug_data_signal <= mem_addr;
        20: debug_data_signal <= mem_din;
        21: debug_data_signal <= mem_dout;
        22: debug_data_signal <= {27'b0, regw_addr_wb};
        23: debug_data_signal <= regw_data_wb;
        default: debug_data_signal <= 32'hFFFF_FFFF;
    endcase
end
```

For example in the Reg:

```
`ifdef DEBUG
.debug_addr(debug_addr[4:0]),
.debug_data(debug_data_reg),
`endif
```