计算机系统 I 实验报告 6

Project: 单周期 CPU 设计

姓 名: 洪 奕 迅

学 号: 3230102930

班 号: 信安 2301

计算机系统 I

(春夏, 2024)

浙江大学

计算机学院

2024 年 6 月 9 日

目录

1 单周期 CPU 设计

1.1 Instruction Fetch

1.1.1 InstSel

这个模块存在的原因在于我们输入的数据同指令的宽度并不相同，因此我们需要根据 PC 来正确选择高位或低位来确定指令。

由于我们的步进总是指令，且我们总是使用 *Byte* 作为一个单元，因此对于一条 32 位的指令和相应的程序计数器应当满足：

$$WIDTH = 32bits = 8Bytes$$

$$PC \equiv 0 \pmod 4 \leftrightarrow \mathrm{PC}[0] = \mathrm{PC}[1] = 0$$

因此我们去考虑 $PC[2]$ 的值，不难发现其值正好可以代表一个被读入数据的两个 32 位单元，因此该模块代码也相应完成了，在 $PC[2] = 0$ 时我们取低位，反之取高位：

Code Snippet 1.1.1 ▶ InstSel

```
//Instruction selection
always_comb begin
    if(pc[2] == 1'b1)begin
        inst = imem_ift.r_reply_bits.rdata[63:32];
    end else begin
        inst = imem_ift.r_reply_bits.rdata[31:0];
    end
end
```

1.1.2 PC

PC 的设计可以分为两个部分，一部分是 PC 本身的变动，另一部分则是对下一个 PC 信号的选取。前者是本次设计中少有的时序电路，同时相对较为简单：

Code Snippet 1.1.2 ▶ PC

```
always@(posedge clk or posedge rst) begin
    if(rst) bgein
        pc <= 64'b0;
    end else begin
        pc <= next_pc
    end
end
```

而对于 NextPC，我们需要考虑执行的指令类型，在所有指令中 J 型指令和 B 型指令涉及到可能的对 PC 的修改，而其他情况 PC 只需要执行 $PC = PC + 4$ 即可。其中 J 型指令我们需要直接跳转到目标地址而 B 型指令则还需要考虑是否满足跳转条件，因此我们有以下代码：

Code Snippet 1.1.3 ▶ NPC

```
assign npc_sel = (if_jump & is_b) | is_j;
//指令类型信号将由译码器给出
//而判断是否满足跳转条件信号的计算将由后续的比较运算器完成。
assign next_pc = npc_sel ?  alu_res : pc + 4;
```

1.2 Instruction Decode

1.2.1 Controller

Theorem 1.2.1 ▶ 如何区分不同指令

在 RISC-V 中我们有一个相对规整的指令格式，我们将通过 *opcode*, *FUNCT_3*, *FUNCT_7* 来判断具体指令，对于不同类型的指令满足以下规范：

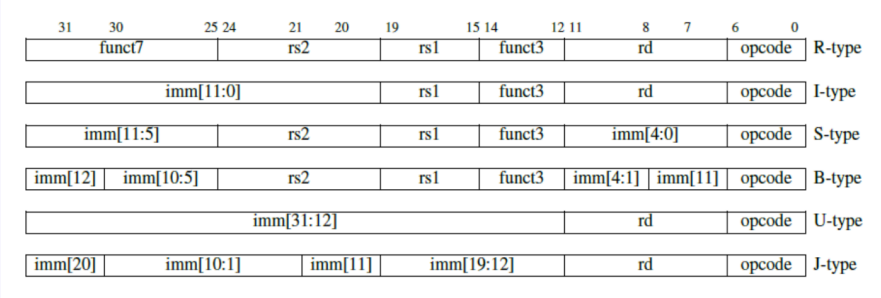

图 1.1: 格式规范

对于所有的指令，其 *opcode* 位置都是固定的，我们的译码器将部分采用部分二段译码的方式，先通过 *opcode* 进行指令的分类并直接确定一些信号，而需要根据 $FUNCT3$, $FUNCT7$ 来确定的信号则直接使用多路选择器，因为我在写代码的时候觉得对于这些信号二段译码可能并不如直接的选择器容易维护，因为需要同时涉及 *opcode*, $func\_3$, $func\_7$ 的一次译码和组合后进行二段译码。

首先我们进行第一次译码初步确定指令类型。

Code Snippet 1.2.2 ▶ Decoder-Part1

```
assign ins_load = (inst[6:0]==LOAD_OPCODE);
assign ins_imm = (inst[6:0]==IMM_OPCODE);
assign ins_auipc = (inst[6:0]==AUIPC_OPCODE);
assign ins_immw = (inst[6:0]==IMMW_OPCODE);
assign ins_store = (inst[6:0]==STORE_OPCODE);
assign ins_reg = (inst[6:0]=== REG_OPCODE);
assign ins_lui = (inst[6:0]=== LUI_OPCODE);
assign ins_regw = (inst[6:0]=== REGW_OPCODE);
assign ins_branch = (inst[6:0]=== BRANCH_OPCODE);
assign ins_jalr = (inst[6:0]=== JALR_OPCODE);
assign ins_jal = (inst[6:0]=== JAL_OPCODE);
```

然后我们将根据第一次译码得到的结果来选择正确的输出信号与输出相连：

Code Snippet 1.2.3 ▶ Decoder-Part2

```
assign we_reg = ins_reg | ins_imm | ins_load | ins_auipc | ins_lui | ins_regw |
 ↪  ins_immw | ins_jalr | ins_jal;
assign we_mem = ins_store;
assign re_mem = ins_load;
assign is_b = ins_branch;
assign is_j = ins_jalr | ins_jal;
assign immgen_op = (ins_load | ins_imm | ins_immw | ins_jalr) ? I_IMM :
((ins_auipc | ins_lui) ? U_IMM : ((ins_reg | ins_regw) ? IMM0 :
(ins_store ? S_IMM : (ins_branch ? B_IMM : UJ_IMM))));
assign alu_asel = (ins_reg | ins_imm | ins_immw | ins_store | ins_load | ins_regw |
 ↪  ins_jalr) ? ASEL_REG : ((ins_auipc | ins_jal | ins_branch) ? ASEL_PC : ASEL0);
assign alu_bsel = (ins_load | ins_imm | ins_auipc | ins_immw | ins_lui | ins_store
 ↪  | ins_branch | ins_jal) ? BSEL_IMM : ((ins_reg | ins_regw) ? BSEL_REG : BSEL0);
assign wb_sel = ins_load ? WB_SEL_MEM : ((ins_jal | ins_jalr | ins_branch) ?
```

```
WB_SEL_PC : (ins_load | ins_imm | ins_immw | ins_auipc | ins_reg |ins_regw |
 ↪  ins_lui)
? WB_SEL_ALU : WB_SEL0);
```

最后再判断 $func\_3$, $func\_7$ 来确定所有运算符和数据类型：

Code Snippet 1.2.4 ▶ Decoder-Part3

```
always_comb begin
    case(opcode)
        LOAD_OPCODE:begin
            alu_op=ALU_ADD;
            cmp_op=CMP_NO;
            case(func3)
                LB_FUNCT3:mem_op=MEM_B;
                LH_FUNCT3:mem_op=MEM_H;
                LW_FUNCT3:mem_op=MEM_W;
                LD_FUNCT3:mem_op=MEM_D;
                LBU_FUNCT3:mem_op=MEM_UB;
                LHU_FUNCT3:mem_op=MEM_UH;
                LWU_FUNCT3:mem_op=MEM_UW;
                default: mem_op=MEM_NO;
            endcase
        end
        IMM_OPCODE:begin
            case(func3)
                ADD_FUNCT3:alu_op=ALU_ADD;
                SLL_FUNCT3:alu_op=ALU_SLL;
                SLT_FUNCT3:alu_op=ALU_SLT;
                SLTU_FUNCT3:alu_op=ALU_SLTU;
                XOR_FUNCT3:alu_op=ALU_XOR;
                SRL_FUNCT3:alu_op=(func7==7'b0000000)?ALU_SRL:ALU_SRA;
                OR_FUNCT3:alu_op=ALU_OR;
                AND_FUNCT3:alu_op=ALU_AND;
                default:alu_op=ALU_DEFAULT;
            endcase
            cmp_op=CMP_NO;
            mem_op=MEM_NO;
        end
```

```
32
33          AUIPC_OPCODE:begin
34              alu_op=ALU_ADD;
35              cmp_op=CMP_NO;
36              mem_op=MEM_NO;
37          end
38          IMMW_OPCODE:begin
39              case (func3)
40                  ADDW_FUNCT3: alu_op = (func7 == 7'b0100000) ? ALU_SUBW : ALU_ADDW;
41                  SLLW_FUNCT3: alu_op = ALU_SLLW;
42                  SRLW_FUNCT3: alu_op = (func7 == 7'b0000000) ? ALU_SRLW : ALU_SRAW;
43                  default: alu_op = ALU_DEFAULT;
44              endcase
45              cmp_op = CMP_NO;
46              mem_op = MEM_NO;
47          end
48          STORE_OPCODE:begin
49              alu_op = ALU_ADD;
50              cmp_op = CMP_NO;
51              case (func3)
52                  SB_FUNCT3: mem_op = MEM_B;
53                  SH_FUNCT3: mem_op = MEM_H;
54                  SW_FUNCT3: mem_op = MEM_W;
55                  SD_FUNCT3: mem_op = MEM_D;
56                  default: mem_op = MEM_NO;
57              endcase
58          end
59          REG_OPCODE:begin
60              case(func3)
61                  ADD_FUNCT3:alu_op=(func7==7'b0000000)?ALU_ADD:ALU_SUB;
62                  SLL_FUNCT3:alu_op=ALU_SLL;
63                  SLT_FUNCT3:alu_op=ALU_SLT;
64                  SLTU_FUNCT3:alu_op=ALU_SLTU;
65                  XOR_FUNCT3:alu_op=ALU_XOR;
66                  SRL_FUNCT3:alu_op=(func7==7'b0000000)?ALU_SRL:ALU_SRA;
67                  OR_FUNCT3:alu_op=ALU_OR;
68                  AND_FUNCT3:alu_op=ALU_AND;
69                  default:alu_op=ALU_DEFAULT;
```

```
            endcase
            cmp_op = CMP_NO;
            mem_op = MEM_NO;
        end
        LUI_OPCODE:begin
            alu_op = ALU_ADD;
            cmp_op = CMP_NO;
            mem_op = MEM_NO;
        end
        REGW_OPCODE:begin
            case (func3)
                ADDW_FUNCT3: alu_op = (func7 == 7'b0000000) ? ALU_ADDW : ALU_SUBW;
                SLLW_FUNCT3: alu_op = ALU_SLLW;
                SRLW_FUNCT3: alu_op = (func7 == 7'b0000000) ? ALU_SRLW : ALU_SRAW;
                default: alu_op = ALU_DEFAULT;
            endcase
            cmp_op = CMP_NO;
            mem_op = MEM_NO;
        end
        BRANCH_OPCODE:begin
            alu_op = ALU_ADD;
            case (func3)
                BEQ_FUNCT3: cmp_op = CMP_EQ;
                BNE_FUNCT3: cmp_op = CMP_NE;
                BLT_FUNCT3: cmp_op = CMP_LT;
                BGE_FUNCT3: cmp_op = CMP_GE;
                BLTU_FUNCT3: cmp_op = CMP_LTU;
                BGEU_FUNCT3: cmp_op = CMP_GEU;
                default: cmp_op = CMP_NO;
            endcase
        end
        JALR_OPCODE:begin
            alu_op = ALU_ADD;
            cmp_op = CMP_NO;
            mem_op = MEM_NO;
        end
        JAL_OPCODE:begin
            alu_op = ALU_ADD;
```

```
108             cmp_op = CMP_NO;
109             mem_op = MEM_NO;
110         end
111         default:begin
112             alu_op = ALU_DEFAULT;
113             cmp_op = CMP_NO;
114             mem_op = MEM_NO;
115         end
116     endcase
117 end
```

现在我们得到了一个完善的译码器，能够将指令正确拆分并给出对应的操作类型。接下来则要考虑数据读写和数据处理的问题。

1.2.2 Regfile

Regfile 能够将给出的数据写入目标地址，或从给出的地址中读取数据，这同样是我们所有设计中少有的时序部件：

Code Snippet 1.2.5 ▶ Regfile

```
1   always @(posedge clk or posedge rst) begin
2   if(rst) begin
3     for(i = 1; i < 32; i++) begin
4       register[i] <= 0;
5     end
6   end else begin
7     if(we & write_addr != 0) begin
8       register[write_addr] <= write_data;
9     end
10  end
11 end
12
13 assign read_data_1 = (read_addr_1 == 0) ? 0 : register[read_addr_1];
14 assign read_data_2 = (read_addr_2 == 0) ? 0 : register[read_addr_2];
```

这里需要注意的是我们需要把读写分开，因为读并不涉及使能信号，其并不需要时序控制。而写则涉及重置和写使能信号，这都依赖时序运行。

此外在实验过程中发现对于非法地址 0（对于我们声明的寄存器向量 $reg[31:1]$）而言的特判是必

须的，这当然是一件想起来非常自然的事，但是在实验过程中发现缺少这一判断并不影响我们的程序通过仿真，但是在上板时将陷入内存死循环，关于为什么非法的读能够通过仿真，目前原因未知。

1.3 Execution

1.3.1 ALU

ALU 的作用是完成算术运算，在这个单元中会充分使用 SystemVerilog 的各种内置算术运算符来综合电路：

Code Snippet 1.3.1 ▶ ALU

```
import CorePack::*;
logic [31:0] extension1, extension2, extension3, extension4, extension5;
assign extension1 = a[31:0] + b[31:0];
assign extension2 = a[31:0] - b[31:0];
assign extension3 = a[31:0] << b[4:0];
assign extension4 = a[31:0] >> b[4:0];
assign extension5 = $signed(a[31:0]) >>> b[4:0];

always_comb begin
  case (alu_op)
    ALU_ADD: res = a + b;
    ALU_SUB: res = a - b;
    ALU_AND: res = a & b;
    ALU_OR: res = a | b;
    ALU_XOR: res = a ^ b;
    ALU_SLT: res = $signed(a) < $signed(b) ? 1 : 0;
    ALU_SLTU: res = a < b ? 1 : 0;
    ALU_SLL: res = a << b[5:0];
    ALU_SRL: res = a >> b[5:0];
    ALU_SRA: res = $signed(a) >>> b[5:0];
    ALU_ADDW: res = {{32{extension1[31]}}, a[31:0] + b[31:0]};
    ALU_SUBW: res = {{32{extension2[31]}}, a[31:0] - b[31:0]};
    ALU_SLLW: res = {{32{extension3[31]}}, a[31:0] << b[4:0]};
    ALU_SRLW: res = {{32{extension4[31]}}, a[31:0] >> b[4:0]};
    ALU_SRAW: res = {{32{extension5[31]}}, $signed(a[31:0]) >>> b[4:0]};
    ALU_DEFAULT: res = 0;
  endcase
```

```
28 end
```

这里需要注意对于所有的移位操作我们都只在可能的有效范围内进行，而不是粗暴地使用类似 $a << b$ 的操作，这会导致综合中出现问题。

另外注意，对于单字运算，我们需要进行符号扩展，皮卡丘如是说：

Integer Register-Register Operations

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SRA	dest	OP
0000000	src2	src1	ADDW	dest	OP-32
0000000	src2	src1	SLLW/SRLW	dest	OP-32
0100000	src2	src1	SUBW/SRAW	dest	OP-32

ADDW and SUBW are RV64I-only instructions that are defined analogously to ADD and SUB but operate on 32-bit values and produce signed 32-bit results. Overflows are ignored, and the low 32-bits of the result is sign-extended to 64-bits and written to the destination register.

图 1.2: RV64i 对于单字运算的手册说明

只是有了 ALU 并不能解决所有问题，因为对于不同类型的指令其输入 ALU 进行操作的数的类型是不同的，除开常规的寄存器还会涉及对于立即数和 PC 的操作，我们需要进行数据选择，首先是对于第一个操作数：

Code Snippet 1.3.2 ▸ ASel

```
1 always_comb begin
2     case(alu_asel)
3         ASEL_REG: alu_a = read_data_1;
4         ASEL_PC: alu_a = pc;
5         default: alu_a = 64'b0;
6     endcase
7 end
```

在进行第二个操作数选择前我们还需要完成一个额外的模块 ImmGen，因为在遇到需要向第二个数据口输入立即数的指令类型时，我们需要根据输入数据生成用于操作的立即数，我们会对按照规范生成的 32 位数据进行符号扩展，32 位立即数满足以下格式：

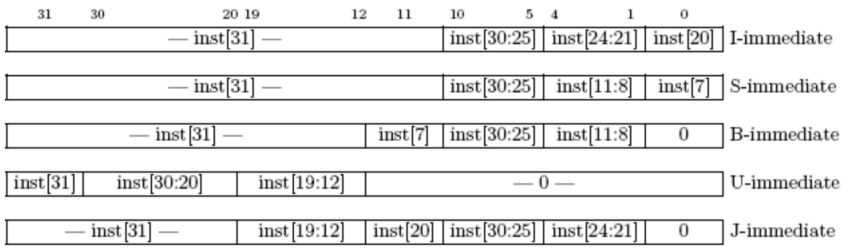

图 1.3: 立即数变体情况

我们对此进行符号位扩展得到我们想要的用于计算的立即数，并且参照 a 口选择数据的代码我们也可以写出 b 口的选择器代码，区别只在于程序计数器部分被立即数所替代：

Code Snippet 1.3.3 ▸ ImmGen

```
always_comb begin
    case(immgen_op)
        I_IMM: imm = {{53{inst[31]}}, inst[30:20]};
        S_IMM: imm = {{53{inst[31]}}, inst[30:25], inst[11:7]};
        B_IMM: imm = {{52{inst[31]}}, inst[7], inst[30:25], inst[11:8], 1'b0};
        U_IMM: imm = {{33{inst[31]}}, inst[30:12], 12'b0};
        UJ_IMM: imm = {{44{inst[31]}}, inst[19:12], inst[20], inst[30:21], 1'b0};
        default: imm = 64'b0;
    endcase
end
always_comb begin
    case(alu_bsel)
        BSEL_REG: alu_b = read_data_2;
        BSEL_IMM: alu_b = imm;
        default: alu_b = 64'b0;
    endcase
end
```

1.3.2 Cmp

Cmp 模块会负责所有可能的分支跳转条件的计算，思路基本同 ALU：

Code Snippet 1.3.4 ▶ Cmp

```
always_comb begin
    case (cmp_op)
        CMP_NO: cmp_res = 0;
        CMP_EQ: cmp_res = (a == b);
        CMP_NE: cmp_res = (a != b);
        CMP_LT: cmp_res = ($signed(a) < $signed(b));
        CMP_GE: cmp_res = ($signed(a) >= $signed(b));
        CMP_LTU: cmp_res = (a < b);
        CMP_GEU: cmp_res = (a >= b);
        CMP7: cmp_res = 0;
    endcase
end
```

1.4 MemoryAccess and WriteBack

1.4.1 DataProcess

数据的处理和整个数据通路的设计是单周期 CPU 的核心部分之一，正如文档中提到的多路选择器总是会造成最大的电路消耗，我们在接下来的设计中除了必要的 Core 包涉及的状态外，尽可能使用移位运算符和向量组合取代所有的 case 和可能需要使用的 Verilog 内置算术运算符：

DataPackage

我们设计的单周期 CPU 能够对许多类型的数据进行处理，但是我们的数据宽度总是 64 位，因此在写入前总是需要提前将数据进行内存对齐，我们以需要处理的最简单的单字为例，数据本身在输入的低 32 位，其处理操作类似于此前 InsSel 的逆操作，若 $alu\_res[2:0] = 0$ 则不移位，反之移到高位，其他类型数据操作思路相同：

Code Snippet 1.4.1 ▶ DataPackage

```
always_comb begin
    case(mem_op)
        MEM_B: begin
            dmem_ift.w_request_bits.wdata[7:0] = read_data_2[7:0];
            dmem_ift.w_request_bits.wdata = dmem_ift.w_request_bits.wdata <<
             ↪ {alu_res[2:0], 3'b0};
        end
```

```
        MEM_H: begin
            dmem_ift.w_request_bits.wdata[15:0] = read_data_2[15:0];
            dmem_ift.w_request_bits.wdata = dmem_ift.w_request_bits.wdata <<
             ↪ {alu_res[2:1], 4'b0};
        end
        MEM_W: begin
            dmem_ift.w_request_bits.wdata[31:0] = read_data_2[31:0];
            dmem_ift.w_request_bits.wdata = dmem_ift.w_request_bits.wdata <<
             ↪ {alu_res[2], 5'b0};
        end
        MEM_D: dmem_ift.w_request_bits.wdata = read_data_2;
        default: dmem_ift.w_request_bits.wdata = 64'b0;
    endcase
end
```

MaskGemration

我们在生成了写入数据后，还需要说明我们数据的有效位，这通过掩码表示 - 1 代表有效字节，0 代表无效字节，我们采用一样的方式，先设置低位有效，再根据 *alu_res* 的值进行移位;

Code Snippet 1.4.2 ▶ MaskGeneration

```
always_comb begin
    case(mem_op)
        MEM_B: begin
            dmem_ift.w_request_bits.wmask = {7'b0,1'b1};
            dmem_ift.w_request_bits.wmask = dmem_ift.w_request_bits.wmask <<
             ↪ alu_res[2:0];
        end
        MEM_H: begin
            dmem_ift.w_request_bits.wmask = {6'b0,2'b11};
            dmem_ift.w_request_bits.wmask = dmem_ift.w_request_bits.wmask <<
             ↪ {alu_res[2:1], 1'b0};
        end
        MEM_W: begin
            dmem_ift.w_request_bits.wmask = {4'b0,4'b1111};
            dmem_ift.w_request_bits.wmask = dmem_ift.w_request_bits.wmask <<
             ↪ {alu_res[2], 2'b0};
        end
```

```
        MEM_D: dmem_ift.w_request_bits.wmask = 8'b11111111;
        default: dmem_ift.w_request_bits.wmask = 8'b00000000;
    endcase
end
```

DataTrunction

这一部分与此前的区别是数据不再固定在低位，我们要根据 *alu_res* 来确定我们需要的数据将其作为低位并进行符号扩展。因此我们的思路是先通过移位将目标数据置于低位，再根据需要进行扩展：

Code Snippet 1.4.3 ▶ DataTrunction

```
always_comb begin
    case(mem_op)
        MEM_B:  begin
            temp = dmem_ift.r_reply_bits.rdata >> {alu_res[2:0],3'b0};
            mem = {{56{temp[7]}}, temp[7:0]};
        end
        MEM_UB: begin
            temp = dmem_ift.r_reply_bits.rdata >> {alu_res[2:0],3'b0};
            mem = {{56{1'b0}}, temp[7:0]};
        end
        MEM_H:  begin
            temp = dmem_ift.r_reply_bits.rdata >> {alu_res[2:1],4'b0};
            mem = {{48{temp[15]}}, temp[15:0]};
        end
        MEM_UH: begin
            temp = dmem_ift.r_reply_bits.rdata >> {alu_res[2:1],4'b0};
            mem = {{48{1'b0}}, temp[15:0]};
        end
        MEM_W:  begin
            temp = dmem_ift.r_reply_bits.rdata >> {alu_res[2],5'b0};
            mem = {{32{temp[31]}}, temp[31:0]};
        end
        MEM_UW: begin
            case (alu_res[2])
                1'b0: mem = {{32{1'b0}}, dmem_ift.r_reply_bits.rdata[31:0]};
                1'b1: mem = {{32{1'b0}}, dmem_ift.r_reply_bits.rdata[63:32]};
                default: mem = 64'b0;
```

```
            endcase
        end
        MEM_D:  mem = dmem_ift.r_reply_bits.rdata;
        MEM_NO: mem = dmem_ift.r_reply_bits.rdata;
    endcase
end
```

至此数据相关的处理完成了，当然我们还是不可避免的使用了多路选择器（我其实不太明白文档中说的在 ALU 中怎么避开多路选择器），但最起码在每个 case 内避开了列举状态和乘法器而是用移位和向量组合解决了问题。然后最后我们还需要将处理好的数据与 RAM 连线，在原有的 CPU 连线上加上以下内容：

Code Snippet 1.4.4 ▶ RAM wires

```
assign dmem_ift.r_request_bits.raddr = alu_res;
assign dmem_ift.w_request_bits.waddr = alu_res;
assign imem_ift.r_request_bits.raddr = pc;
```

1.4.2 WriteBack

在完成所有运算后，我们还需要将处理好的数据与 RAM 连线，在原有的 CPU 连线上加上以下内容：

Code Snippet 1.4.5 ▶ RAM wires

```
assign dmem_ift.r_request_bits.raddr = alu_res;
assign dmem_ift.w_request_bits.waddr = alu_res;
assign imem_ift.r_request_bits.raddr = pc;
```

另外对于原有 CPU 连线，需要写入的值 *wb_val* 还是未知的，我们仍需要通过一个选择器根据类型载入目标值：

Code Snippet 1.4.6 ▶ WbSel

```
always_comb begin
    case(wb_sel)
        WB_SEL_ALU: wb_val = alu_res;
        WB_SEL_MEM: wb_val = mem;
        WB_SEL_PC: wb_val = pc + 4;
        default: wb_val = 64'b0;
```

```
    endcase
end
```

至此，所有模块设计完成，我们接下来进行仿真和上板测试。

2 仿真与上板

2.1 仿真

限于截图篇幅问题，我们这里只展示 $make\ TESTCASE = full$ 的成功截图：

```
core   0: 3 0x0000000000000980 (0x3450809b) x1  0xfffffffffff12345
core   0: 0x0000000000000984 (0x00c09093) slli    ra, ra, 12
core   0: 3 0x0000000000000984 (0x00c09093) x1  0xffffffff12345000
core   0: 0x0000000000000988 (0x67808093) addi    ra, ra, 1656
core   0: 3 0x0000000000000988 (0x67808093) x1  0xffffffff12345678
core   0: 0x000000000000098c (0x00400113) li      sp, 4
core   0: 3 0x000000000000098c (0x00400113) x2  0x0000000000000004
core   0: 0x0000000000000990 (0x0020973b) sllw    a4, ra, sp
core   0: 3 0x0000000000000990 (0x0020973b) x14 0x0000000023456780
core   0: 0x0000000000000994 (0x234563b7) lui     t2, 0x23456
core   0: 3 0x0000000000000994 (0x234563b7) x7  0x0000000023456000
core   0: 0x0000000000000998 (0x7803839b) addiw   t2, t2, 1920
core   0: 3 0x0000000000000998 (0x7803839b) x7  0x0000000023456780
core   0: 0x000000000000099c (0x00771463) bne     a4, t2, pc + 8
core   0: 3 0x000000000000099c (0x00771463)
core   0: 0x00000000000009a0 (0x00301663) bne     zero, gp, pc + 12
core   0: 3 0x00000000000009a0 (0x00301663)
core   0: >>>>  pass
core   0: 0x00000000000009ac (0x00000093) li      ra, 0
core   0: 3 0x00000000000009ac (0x00000093) x1  0x0000000000000000
core   0: 0x00000000000009b0 (0xc0001073) unimp
core   0: exception trap_illegal_instruction, epc 0x00000000000009b0
core   0:           tval 0x00000000c0001073
core   0: >>>>
core   0: 0x0000000000000000 (0x00100193) li      gp, 1
core   0: 3 0x0000000000000000 (0x00100193) x3  0x0000000000000001
[error] PC SIM 0000000000000000, DUT 00000000000009b0
[error] INSN SIM 00100193, DUT c0001073
```

图 2.1: 仿真成功截图

2.2 上板

同样限于篇幅问题，这里只展示正确到达目标地址的图片；

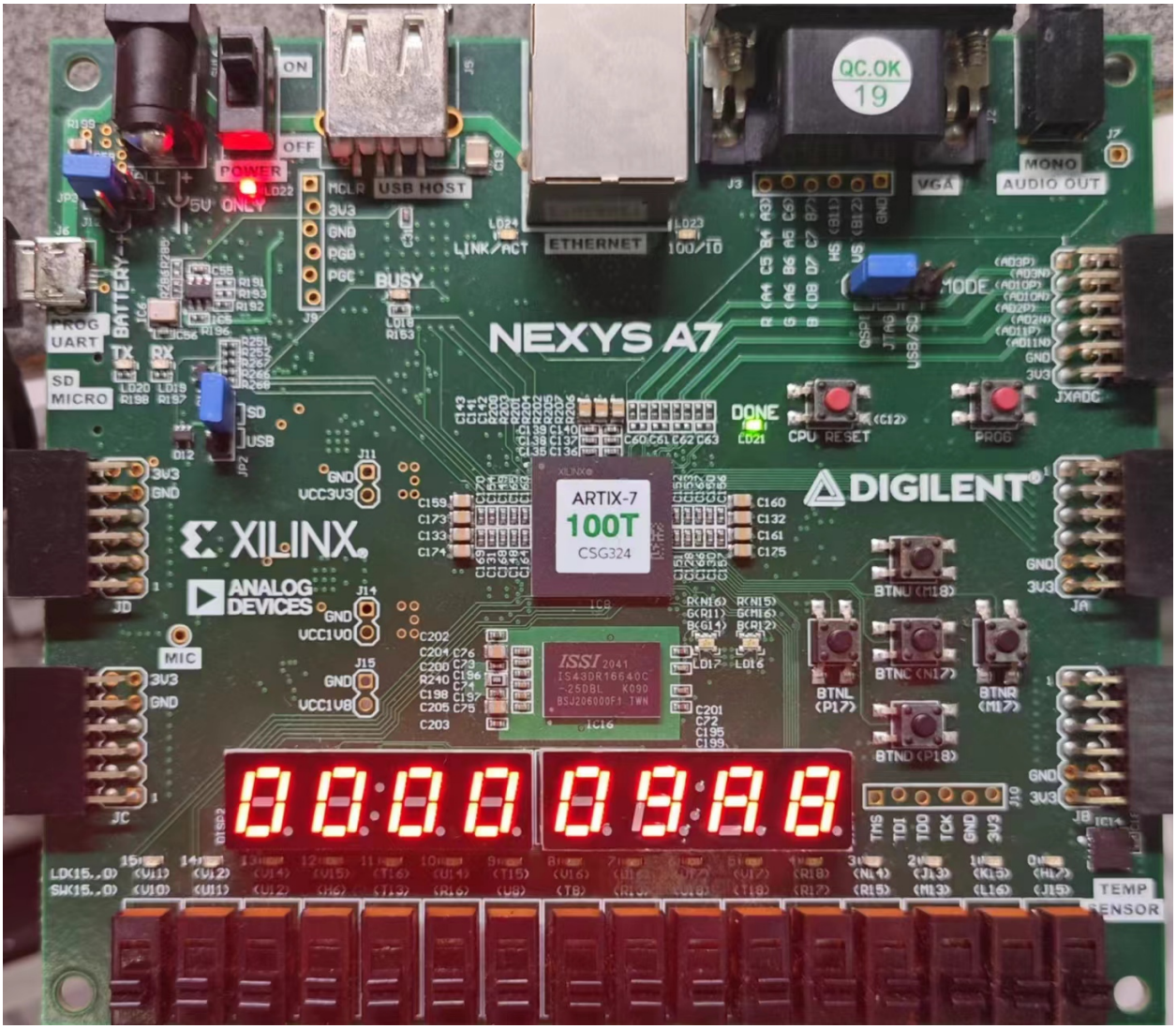

图 2.2: 到达目标地址

有一点注意由于写法的问题代码在生成比特流时可能报错，但是实测不影响正确性，只需要按照报错提示调整约束文件即可。

3 意见和建议

作为年轻人的第一门硬件课程，在学习过程中碰到了相当多的困难，Lab 的设计非常有趣和有质量但是确实是不小的挑战，因此特别感谢助教们不辞辛苦回答我的各种抽象问题。系统一让我学到了非常多东西，两位助教也是我希望成为的目标，如果未来我也有机会成为助教，我也会把这份意志传递下去的（中二音）。

然后以下是这学期实验中碰到的一些问题，主要是 project：

1. **血泪教训：不要试图用 Mac 学习系统一，尤其是 project，谁也不知道实验用的框架在 LinuxOn-Arm 上有什么奇怪的问题**。
2. 感觉 project 的安排不太合理，应该先让我们设计译码器再设计数据通路吧，不然就干脆把数据通路设计放在最前面，然后给出所有需要使用模块的模块定义，然后在第二部分进行完善。现在的 projet 很多模块需要自己设计，而数据通路作为串联所有模块的存在却被放在译码器说明之前，阅读的时候很奇怪（当然后来也说了第一部分第二部分要一起看，所以其实无伤大雅）。
3. project 文档疑似有历史遗留痕迹，前文提到 *Mux* 和 *ImmGen* 是组合电路，但是后文只字未提这些模块。并且对于新加入的 *DataTrunc*，*DataPkg*，*MaskGen* 模块说明有点过于简略了，完全摸不着头脑。