浙江大學
ZHEJIANG UNIVERSITY

Lab 5 Dynamically Scheduled Pipelines using Scoreboarding

2024-2025 春夏学期 计算机体系结构

课程实验报告

姓名 王浩雄

学号 3230106032

年级 2023 级

专业 混合班（计算机科学与技术）

班级 混合 2303 班

2025 年 4 月 24 日

Lab 5 Report

1 实验目的

本实验要求在前期实现的流水线 CPU 基础上，使用 IF/ID/FU/WB 四阶段重新对流水线进行设计，使得流水线支持乘法、除法、内存访问等多周期操作。具体要求如下：

1. 理解 CPU 支持多周期操作的基本原理；
2. 为多周期流水线 CPU 的 FU 阶段各模块进行设计，包括 ALU 模块、乘法运算模块、除法运算模块等；
3. 为多周期流水线 CPU 设计控制器 CtrlUnit。

2 多周期流水线 CPU 的结构

多周期流水线 CPU 采用 IF（取指）、ID（译码）、FU（功能单元执行）、WB（写回）四阶段结构，使得流水线支持乘法、除法、内存访问等多周期操作。其整体架构如图所示：

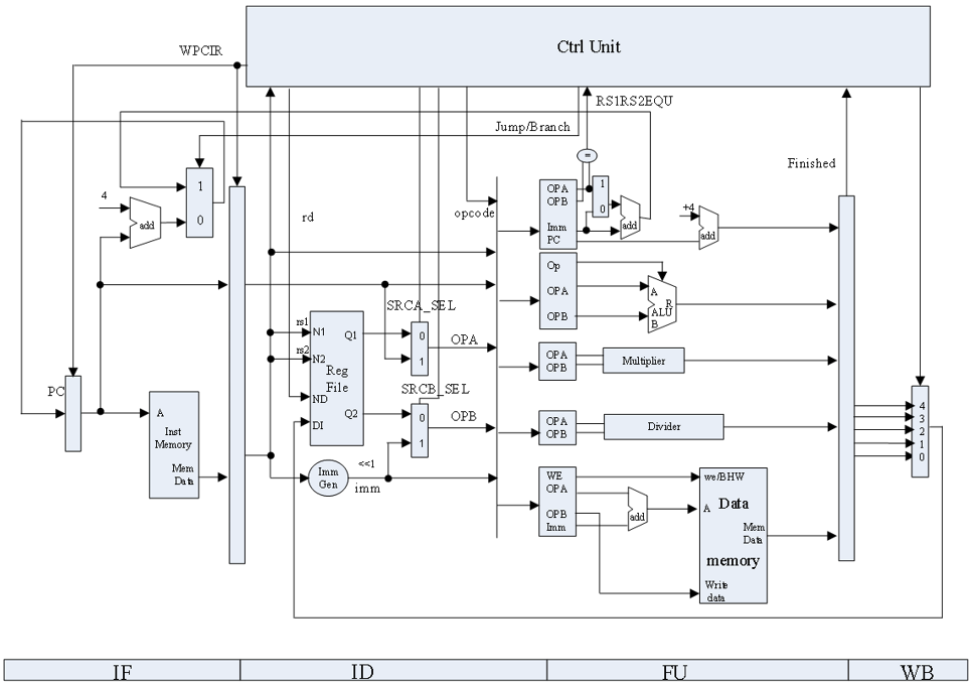


图 1: 多周期流水线 CPU 的结构

- **IF 阶段**：负责从指令存储器（ROM）中取指令，并更新 PC 寄存器。该阶段的器件包括 PC 寄存器、加法器（PC+4）、多路选择器（选择下一条指令地址）和指令存储器。

- **ID 阶段**：译码指令，读取寄存器堆数据，生成立即数，并通过控制单元生成控制信号。
- **FU 阶段**：功能单元执行阶段，支持多周期操作，包括 ALU、乘法、除法、内存访问和跳转地址计算等模块。
- **WB 阶段**：将 FU 阶段的计算结果（内存读取结果）写回寄存器堆。

为支持多周期操作，流水线通过计分板技术动态调度指令，解决数据冒险（包括 RAW、WAR、WAW 三种类型）和控制冒险。此部分将在下一次实验中实现。

3 FU 阶段各模块设计

FU 阶段包含多个功能单元模块，通过状态机控制多周期执行。以下是模块的详细设计：

3.1 ALU 模块：FU_ALU

ALU 模块负责执行一般的算术和逻辑运算，其为单周期运算。当 EN 信号为 1 且状态为 0 时，表示需要启动 ALU 操作。此时，ALU 模块的控制信号被设置为输入的 ALUControl，操作数被设置为传入的 ALU_A 和 ALU_B，并将状态置为 1 表示 ALU 模块正在执行。操作完成后，结果将被送出，且将状态恢复为 0。

```
always @(posedge clk) begin
        if (EN & state) begin
                Control <= ALUControl;
                A <= ALUA;
                B <= ALUB;
                state <= 1;
        end
        else state <= 0;
end
```

3.2 除法运算模块：FU_DIV

除法模块为多周期运算，但不进行流水化实现。当 EN 信号为 1 且状态为 0 时，表示需要启动除法操作。此时，除法模块的操作数被设置为传入的 A 和 B，并将状态置为 1 表示除法模块正在执行。操作完成后，结果将被送出，且将状态恢复为 0。

```
always@(posedge clk) begin
        if(EN & ~state) begin
```

```
                A_reg <= A;
                B_reg <= B;
                A_valid <= 1;
                B_valid <= 1;
                state <= 1;
        end
        else if(res_valid) begin
                A_valid <= 0;
                B_valid <= 0;
                state <= 0;
        end
end
```

3.3 乘法运算模块：FU_MUL

乘法模块为多周期运算，且通过流水化分为 8 个阶段。当 EN 信号为 1 且状态为 0 时，表示需要启动乘法操作。此时，乘法模块的操作数被设置为传入的 A 和 B，并将状态置为非 0 表示乘法模块正在执行。操作完成后，结果将被送出，且将状态恢复为 0。

注意:这里的状态使用独热码进行分配,状态切换顺序为:0000000->1000000->0100000->0010000->0001000->0000100->0000010->0000001->0000000

```
always@(posedge clk) begin
        if(EN & ~|state) begin
                A_reg <= A;
                B_reg <= B;
                state <= 7'b100_0000;
        end
        else state <= {1'b0, state[6:1]};
end
```

3.4 内存读写模块：FU_MEM

内存读写模块为多周期运算，且通过流水化分为 3 个阶段。当 EN 信号为 1 且状态为 0 时，表示需要启动内存读或写操作。此时，为内存控制器输入读写类型、数据长度、读写地址、立即数等信息，并将状态置为非 0 表示内存读写模块正在执行。操作完成后，结果将被送出，且将状态恢复为 0。

注意：状态切换顺序为：00->10->01->00

```
always@(posedge clk) begin
        if(EN & ~|state) begin
```

```
                mem_w_reg <= mem_w;
                bhw_reg <= bhw;
                rs1_data_reg <= rs1_data;
                rs2_data_reg <= rs2_data;
                imm_reg <= imm;
                state <= 2'b10;
        end
        else state <= {1'b0, state[1]};
end
```

3.5 分支跳转地址计算模块：FU_JUMP

分支跳转地址计算模块负责为跳转指令计算目标地址，其为单周期运算。当 EN 信号为 1 且状态为 0 时，表示需要启动该模块。此时，该模块将两个操作数按照指定的规则进行比较，结合输入的立即数得出跳转地址，并将状态置为 1 表示模块正在执行。操作完成后，比较结果、目标地址将被送出，且将状态恢复为 0。

```
always@(posedge clk) begin
        if(EN & ~state) begin
                JALR_reg <= JALR;
                cmp_ctrl_reg <= cmp_ctrl;
                rs1_data_reg <= rs1_data;
                rs2_data_reg <= rs2_data;
                imm_reg <= imm;
                PC_reg <= PC;
                state <= 1;
        end
        else state <= 0;
end

cmp_32 cmp(.a(rs1_data_reg),.b(rs2_data_reg),.ctrl(cmp_ctrl_reg),.c(
    cmp_res));
add_32 a(.a(JALR_reg ? rs1_data_reg : PC_reg),.b(imm_reg),.c(PC_jump)
    );
add_32 b(.a(PC_reg),.b(32'd4),.c(PC_wb));
```

4 控制器 CtrlUnit 设计：RV32Core

控制器位于 ID 阶段进行工作。我对控制器代码进行了如下完善：

4.1 立即数生成器（ImmGen）

```
ImmGen imm_gen(.ImmSel(ImmSel_ctrl), .inst_field(inst_ID), .Imm_out(
    Imm_out_ID));
```

根据控制信号 ImmSel_ctrl 从指令 inst_ID 中提取立即数字段，生成 32 位立即数 Imm_out_ID，用于后续计算。

4.2 ALU 第一个操作数选择（mux_imm_ALU_ID_A）

```
MUX2T1_32 mux_imm_ALU_ID_A(.I0(rs1_data_ID), .I1(PC_ID), .s(
    ALUSrcA_ctrl), .o(ALUA_ID));
```

根据 ALUSrcA_ctrl 选择 ALU 第一个操作数 ALUA_ID。具体地，当 ALUSrcA_ctrl=0 时选择来自寄存器的数据 rs1_data_ID，当 ALUSrcA_ctrl=1 时选择 PC_ID。

4.3 ALU 第二个操作数选择（mux_imm_ALU_ID_B）

```
MUX2T1_32 mux_imm_ALU_ID_B(.I0(rs2_data_ID), .I1(Imm_out_ID), .s(
    ALUSrcB_ctrl), .o(ALUB_ID));
```

根据 ALUSrcB_ctrl 选择 ALU 第二个操作数 ALUB_ID。具体地，当 ALUSrcB_ctrl=0 时选择来自寄存器的数据 rs2_data_ID，当 ALUSrcB_ctrl=1 时选择立即数 Imm_out_ID。

4.4 写回数据选择器（mux_DtR）

```
MUX8T1_32 mux_DtR(.s(DatatoReg_ctrl),.I0(32'd0),.I1(ALUout_WB),.I2(
    mem_data_WB),.I3(mulres_WB),.I4(divres_WB),.I5(PC_wb_WB),.I6(32'd0
    ),.I7(32'd0),.o(wt_data_WB));
```

根据 DatatoReg_ctrl 选择写回寄存器堆的数据 wt_data_WB，支持以下来源：

- ALUout_WB：ALU 运算结果。
- mem_data_WB：内存读取数据。
- mulres_WB：乘法运算结果。
- divres_WB：除法运算结果。
- PC_wb_WB：跳转返回地址。
- 32'd0：占位输入，无特定含义。

5 实验结果

ROM 初始化内容如下：

NO.	Instruction	Addr.	Label	ASM
0	00000013	0	__start:	addi x0, x0, 0
1	00402103	4		lw x2, 4(x0)
2	00802203	8		lw x4, 8(x0)
3	004100b3	C		add x1, x2, x4
4	fff08093	10		addi x1, x1, -1
5	00c02283	14		lw x5, 12(x0)
6	01002303	18		lw x6, 16(x0)
7	01402383	1C		lw x7, 20(x0)
8	402200b3	20		sub x1,x4,x2
9	ffd50093	24		addi x1,x10,-3
10	00520c63	28		beq x4,x5,label0
11	00420a63	2C		beq x4,x4,label0
12	00000013	30		addi x0,x0,0
13	00000013	34		addi x0,x0,0
14	00000013	38		addi x0,x0,0
15	00000013	3C		addi x0,x0,0
16	000040b7	40	label0:	lui x1,4
17	00c000ef	44		jal x1,12
18	00000013	48		addi x0,x0,0
19	00000013	4C		addi x0,x0,0
20	00000013	50		addi x0,x0,0
21	00000013	54		addi x0,x0,0
22	ffff0097	58		auipc x1, 0xffff0
23	0223c433	5C		div x8, x7, x2
24	025204b3	60		mul x9, x4, x5
25	022404b3	64		mul x9, x8, x2
26	00400113	68		addi x2, x0, 4
27	000000e7	6C		jalr x1,0(x0)

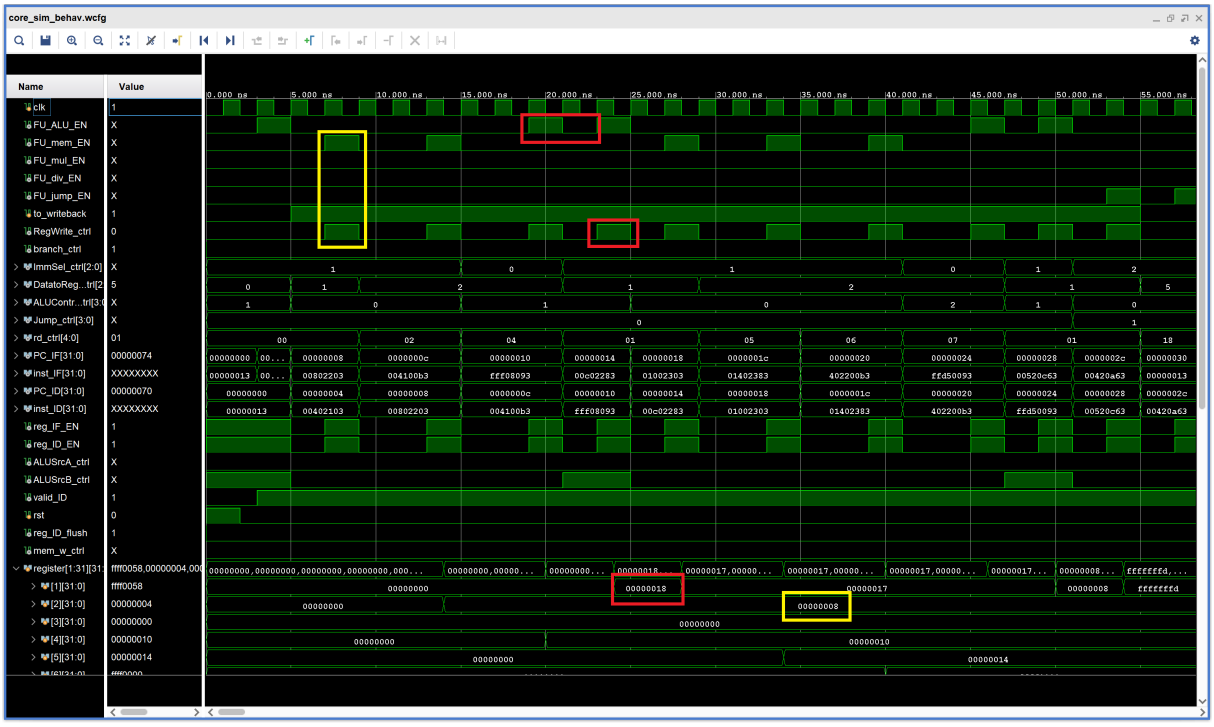
RAM 初始化内容如下：

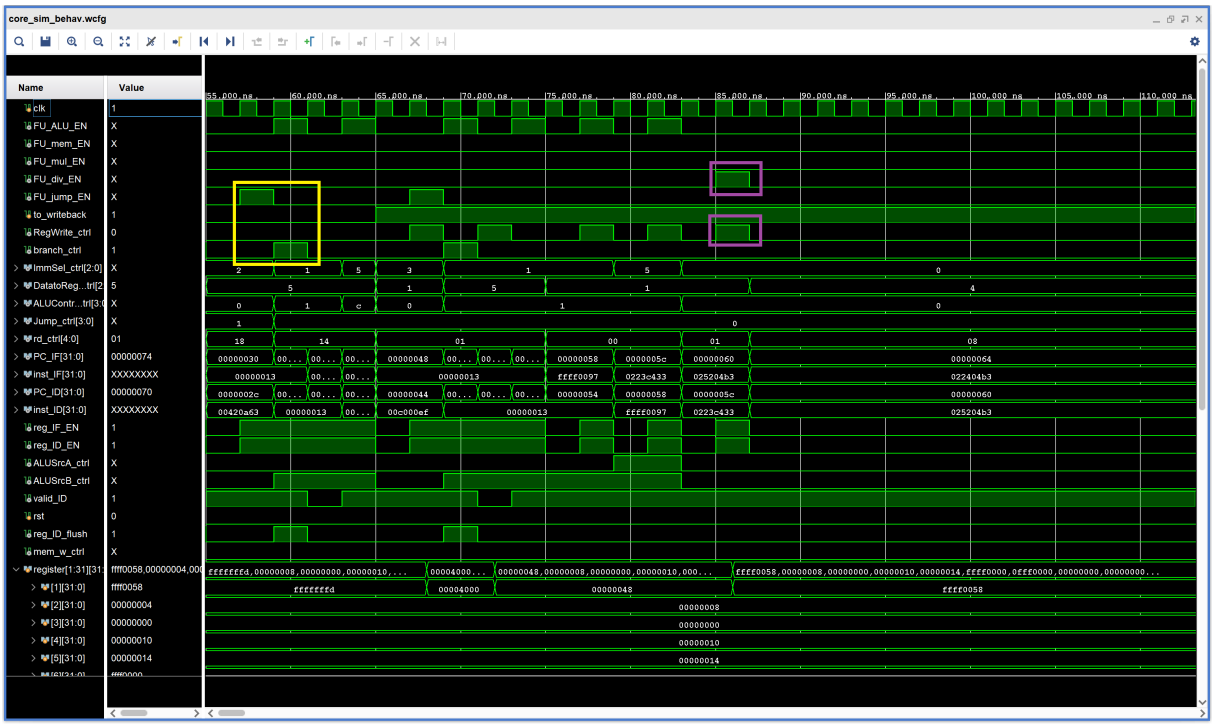
NO.	Data	Addr.
0	000080BF	0
1	00000008	4
2	00000010	8
3	00000014	C
4	FFFF0000	10
5	0FFF0000	14
6	FF000F0F	18
7	F0F0F0F0	1C
8	00000000	20
9	00000000	24
10	00000000	28
11	00000000	2C
12	00000000	30
13	00000000	34
14	00000000	38
15	00000000	3C

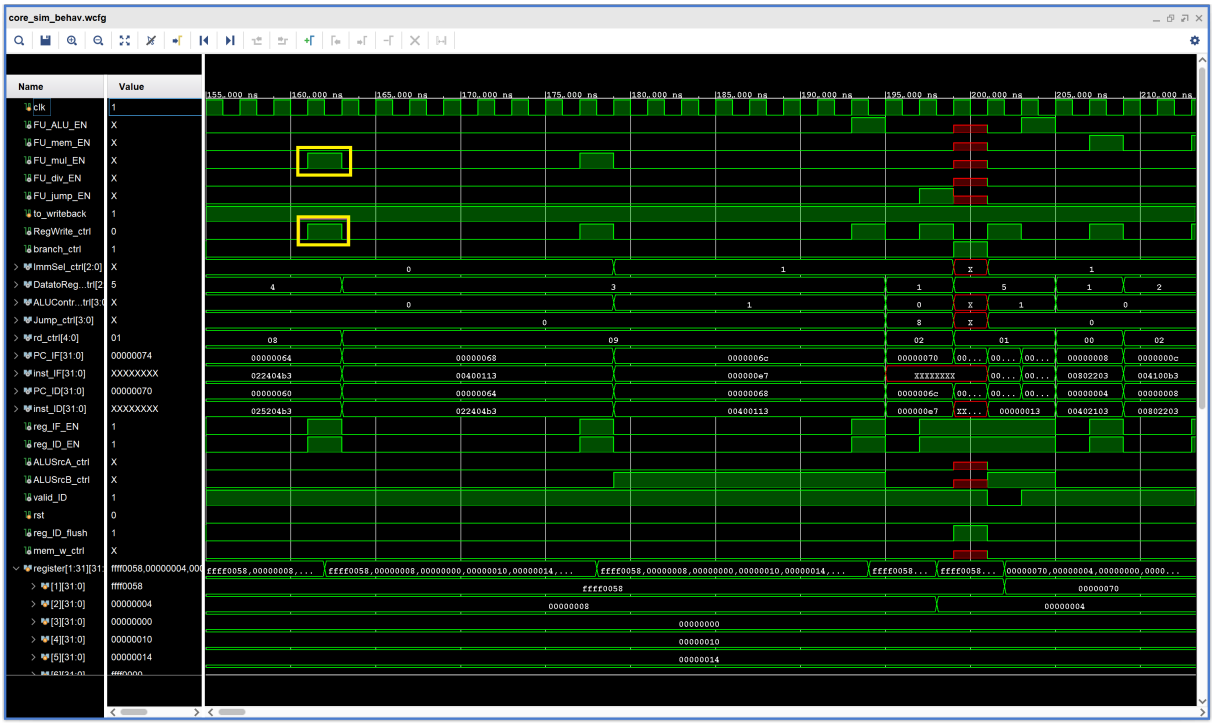
NO.	Instruction	Addr.
16	00000000	40
17	00000000	44
18	00000000	48
19	00000000	4C
20	A3000000	50
21	27000000	54
22	79000000	58
23	15100000	5C
24	00000000	60
25	00000000	64
26	00000000	68
27	00000000	6C
28	00000000	70
29	00000000	74
30	00000000	78
31	00000000	7C

5.1 仿真测试点分析

仿真波形如下：




core_sim_behav.wcfg
Name
Value
clk
FU_ALU_EN
FU_mem_EN
FU_mul_EN
FU_div_EN
FU_jump_EN
to_writeback
RegWrite_ctrl
branch_ctrl
ImmSel_ctrl[2:0]
DatatoReg...trl[2
ALUContr...trl[3:0
Jump_ctrl[3:0]
rd_ctrl[4:0]
PC_IF[31:0]
inst_IF[31:0]
PC_ID[31:0]
inst_ID[31:0]
reg_IF_EN
reg_ID_EN
ALUSrcA_ctrl
ALUSrcB_ctrl
valid_ID
rst
reg_ID_flush
mem_w_ctrl
register[1:31][31:
[1][31:0]
[2][31:0]
[3][31:0]
[4][31:0]
[5][31:0]


core_sim_behav.wcfg
Name
Value
clk
FU_ALU_EN
FU_mem_EN
FU_mul_EN
FU_div_EN
FU_jump_EN
to_writeback
RegWrite_ctrl
branch_ctrl
ImmSel_ctrl[2:0]
DatatoReg...trl[2
ALUContr...trl[3:0
Jump_ctrl[3:0]
rd_ctrl[4:0]
PC_IF[31:0]
inst_IF[31:0]
PC_ID[31:0]
inst_ID[31:0]
reg_IF_EN
reg_ID_EN
ALUSrcA_ctrl
ALUSrcB_ctrl
valid_ID
rst
reg_ID_flush
mem_w_ctrl
register[1:31][31:
[1][31:0]
[2][31:0]
[3][31:0]
[4][31:0]
[5][31:0]

下面选取部分关键测试点进行详细说明：

5.1.1 Load 操作

```
// PC = 0x4
lw x2, 4(x0)
```

解释：

1. 执行 Load 操作时，处理器暂停 2 个时钟周期以完成对内存的读取，并在第 3 个时钟周期恢复流水线，同时将读取数据写回寄存器。
2. 在仿真波形中，观察到 `FU_mem`、`RegWrite_Ctrl` 均为 1，x2 寄存器的值更新为 0x8，表示 Load 操作执行正确。相关位置在第一张波形图中使用黄色矩形标注。

5.1.2 ALU 操作

```
// PC = 0xC
add x1, x2, x4
```

解释：

1. 执行 ALU 操作时，处理器先将 `FU_ALU_EN` 置为 1。
2. `FU_ALU_EN` 置为 0，一个周期后，ALU 操作执行完成，`RegWrite_Ctrl` 置为 1，表示将计算结果写回寄存器。
3. 在仿真波形中，观察到 x1 寄存器的值更新为 0x18（x1=x2+x4=0x4+0x10=0x18），表示 ALU 操作执行正确。相关位置在第一张波形图中使用红色矩形标注。

5.1.3 DIV 操作

```
// PC = 0x5C
div x8, x7, x2
```

解释：

1. 执行 DIV 操作时，处理器先将 `FU_DIV_EN` 置为 1。
2. `FU_DIV_EN` 置为 0，若干周期后，DIV 操作执行完成，`RegWrite_Ctrl` 置为 1，表示将计算结果写回寄存器。
3. 在仿真波形中,观察到 x8 寄存器的值更新为 0x（x8=x7/x2=0x0FFF0000/0x8=0x01FFE000），表示 DIV 操作执行正确。相关位置在第二张波形图中使用紫色矩形标注。

5.1.4 MUL 操作

```
// PC = 0x64
mul x9, x8, x2
```

解释：

1. 执行 MUL 操作时，处理器先将 `FU_MUL_EN` 置为 1。
2. `FU_MUL_EN` 置为 0，若干周期后，MUL 操作执行完成，`RegWrite_Ctrl` 置为 1，表示将计算结果写回寄存器。
3. 在仿真波形中,观察到 x9 寄存器的值更新为 0x18（x1=x8*x2=0x01FFE000*0x8=0x0FFF0000），表示 MUL 操作执行正确。相关位置在第三张波形图中使用黄色矩形标注。

5.1.5 Branch 指令

```
// PC = 0x2C
beq  x4,x4,label0
```

解释：

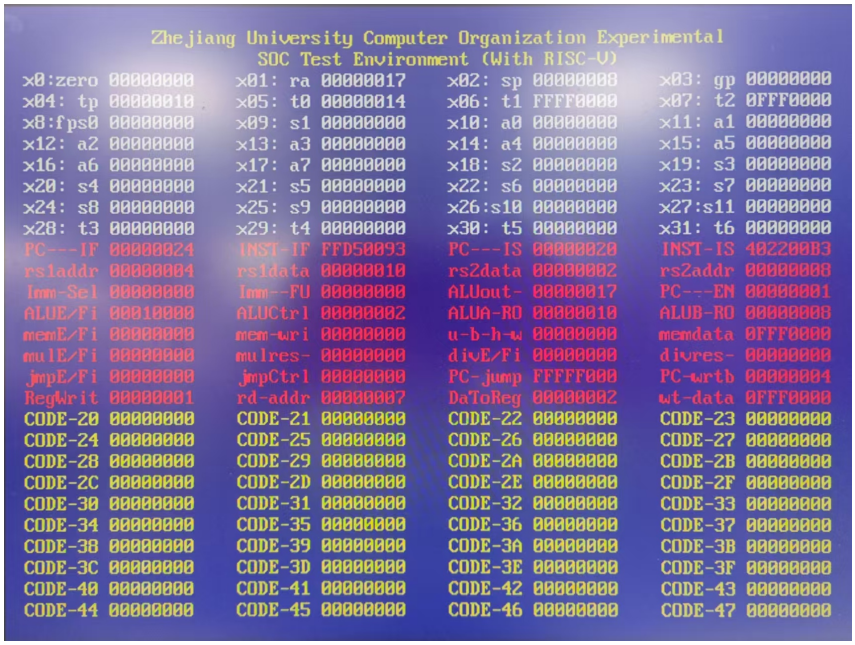
1. 执行 Branch 指令时，处理器先将 `FU_Jump_EN` 置为 1，以对操作数进行比较，决定是否跳转。
2. 若比较结果为需要跳转，则将 `branch_ctrl` 置为 1。
3. 在仿真波形中，观察到上述信号变化，表示 Branch 指令执行正确。相关位置在第二张波形图中使用黄色矩形标注。

5.2 上板测试结果分析

上板测试结果如下，均与预期相符：

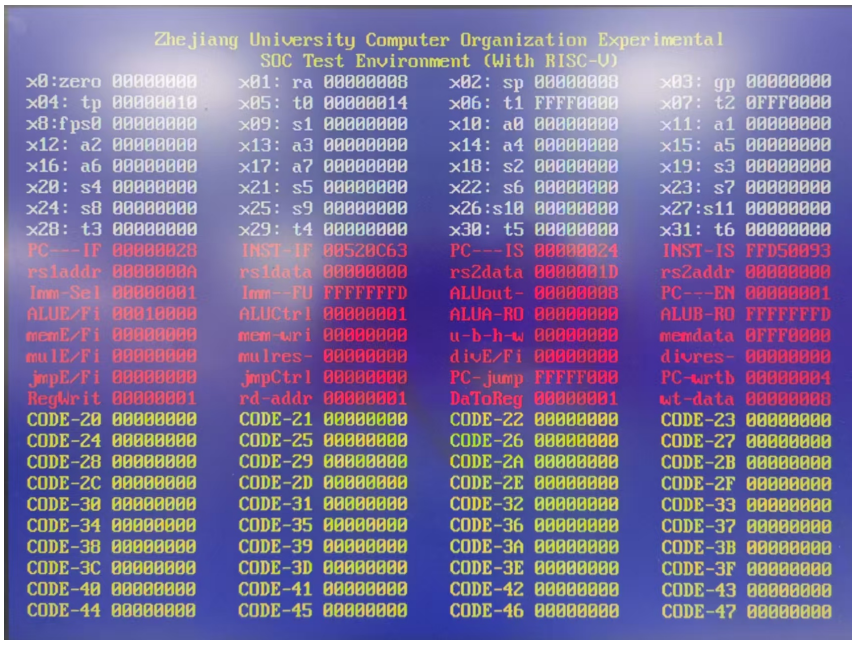
测试点 1：x2，x4，x5，x6，x7 数据载入正确

（PC：0x0-0x1C）



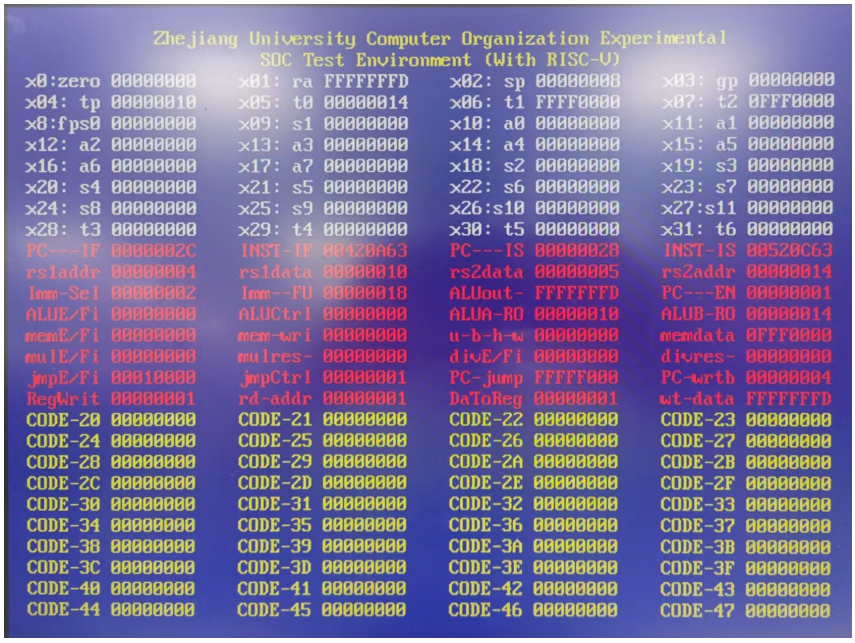
测试点 2：SUB 指令执行正确

（PC：0x20，x1=x4-x2=0x10-0x8=0x8）



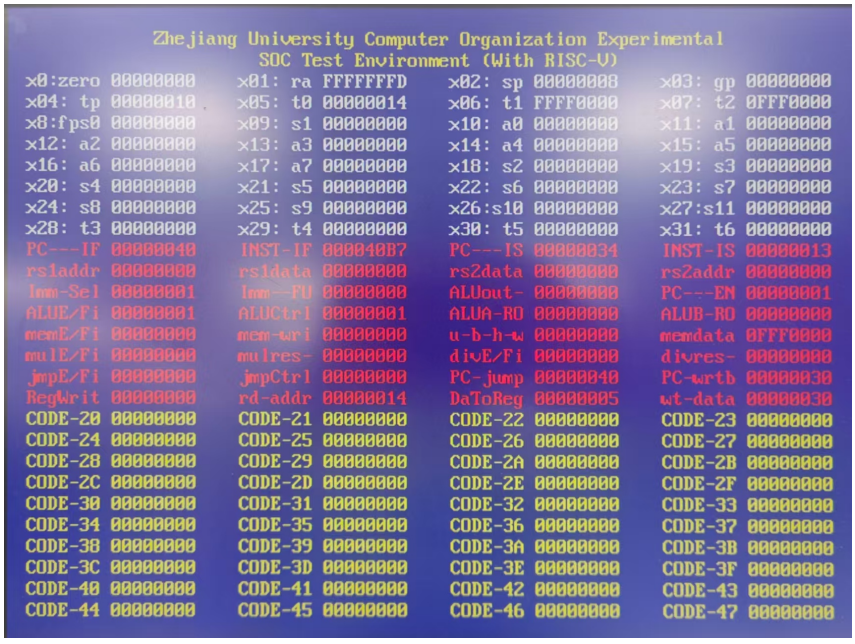
测试点 3：ADDI 指令执行正确

（PC：0x24，x1=x10-3=0x0-3=0xFFFFFFFD）



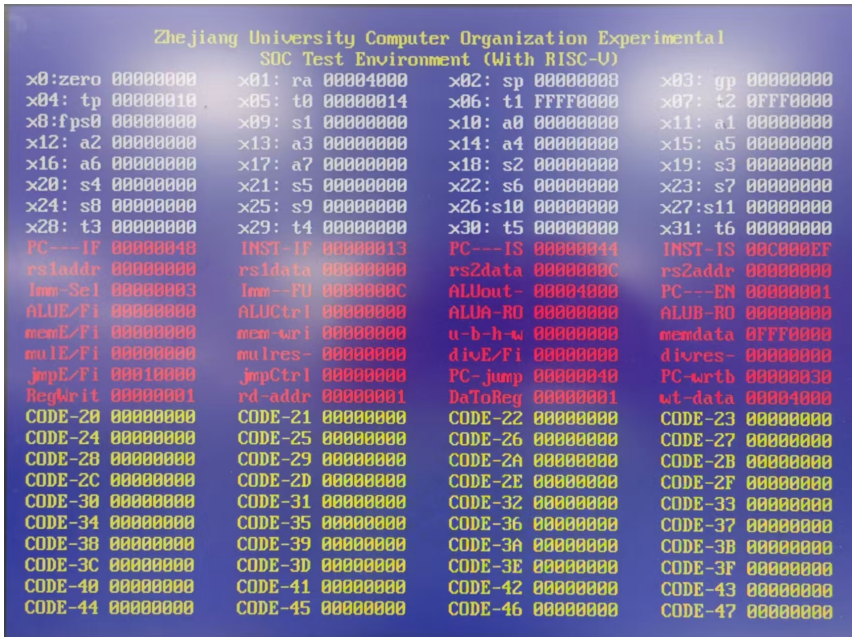
测试点 4：BEQ 指令跳转正确

（PC：0x2C，跳转至 0x40）



测试点 5：LUI 指令执行正确

（PC：0x40，x1=0x00004000）



测试点 6：MUL、DIV 指令执行正确

（PC：0x5C-0x64，x8=0x01FFE000，x9=0x0FFF0000）