浙江大學
ZHEJIANG UNIVERSITY

Lab 6 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 5 月 15 日

Lab 6 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/IS/RO/FU/WB 五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，并使用 ScoreBoard 实现流水线的动态调度。

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 重新设计控制器 CtrlUnit，使用 ScoreBoard 实现流水线的动态调度。

2 多周期流水线 CPU 的结构

多周期流水线 CPU 采用 IF（取指）、IS（译码）、RO（读操作数）、FU（功能单元执行）、WB（写回）五阶段结构，使得流水线支持 ScoreBoard 调度的乘法、除法、内存访问等多周期操作。其整体架构如图所示：

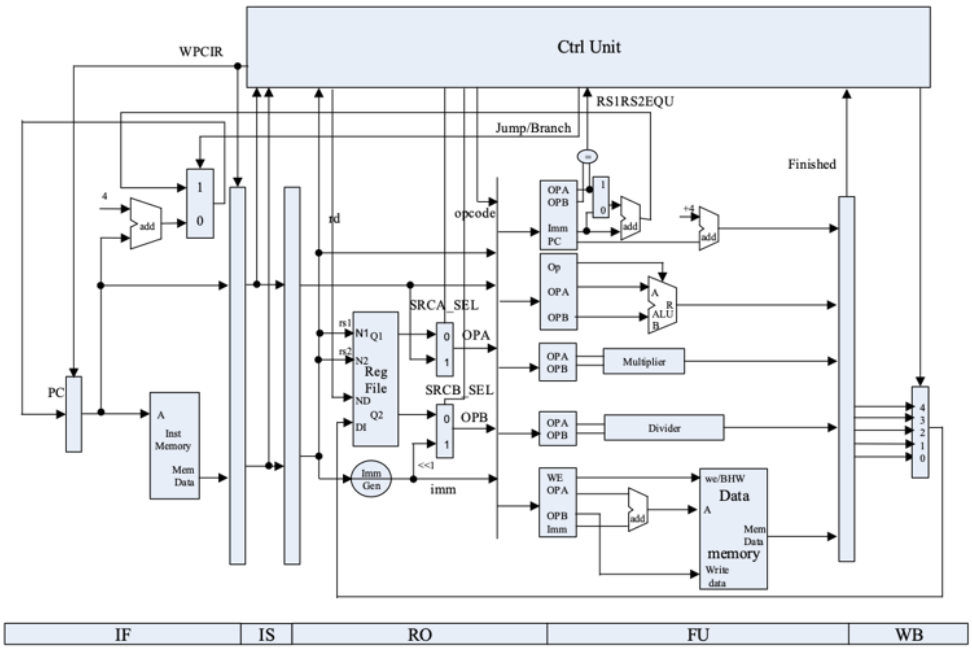


图 1: 多周期流水线 CPU 的结构

- **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。
- **IS 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。实现 ScoreBoard 后，控制单元在该阶段中进行指令的发射和跟踪。
- **RO 阶段**：读操作数，指令在该阶段等待其所有操作数可用，待操作数可用且完成冒险检测后进入 FU 阶段。
- **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。

3 ScoreBoard 记录结构

3.1 功能单元状态（FUS）

FUS 通过跟踪各功能单元的的忙闲状态 （BUSY）、当前执行的操作 （OP）、源/目标寄存器 （SRC1、SRC2、DST）等，为控制指令发射与执行提供信息，从而解决数据冒险的问题。每个功能单元有一条 FUS 记录，每条 FUS 记录的格式形如下表。

表 1: FUS 的各位含义

Bits	**名称**	**描述**
0	BUSY	指示功能单元是否正在执行指令。
1–5	OP	记录功能单元正在执行的指令类型。
6–10	DST	记录指令的目标寄存器号。
11–15	SRC1	记录指令的第一个源寄存器号。
16–20	SRC2	记录指令的第二个源寄存器号。
21–23	FU1	记录指令的第一个源寄存器来自的功能单元。
24–26	FU2	记录指令的第二个源寄存器来自的功能单元。
27	RDY1	指示指令的第一个源寄存器是否准备就绪。
28	RDY2	指示指令的第二个源寄存器是否准备就绪。
29	FU_DONE	指示功能单元是否完成指令的执行。

3.2 寄存器结果状态（RRS）

每个寄存器有一个 RRS 记录，RRS 记录了哪一功能单元正在执行的指令将该寄存器作为其目标。

4 控制单元 CtrlUnit 设计

4.1 处理结构冒险和 WAW 冒险

`normal_stall` 信号用于检测当前是否存在结构冒险和 WAW 冒险。若存在这两种情况，指令不能发射，必须暂停。

为此，新定义 `structural_hazard` 和 `waw` 两个信号。其中，`structural_hazard` 信号用于判断是否存在结构冒险，只需检查当前所需的功能单元是否处于 BUSY 状态；`waw` 信号用于判断是否存在 WAW 冒险，只需判断当前 IS 阶段的指令目标寄存器是否和 FUS 中已存在的目标寄存器重复。

相关代码如下：

```
wire structural_hazard = (
        use_ALU & FUS[`FU_ALU][`BUSY] |
        use_MEM & FUS[`FU_MEM][`BUSY] |
        use_MUL & FUS[`FU_MUL][`BUSY] |
        use_DIV & FUS[`FU_DIV][`BUSY] |
        use_JUMP & FUS[`FU_JUMP][`BUSY]);

wire waw = dst &    (
        dst == FUS[`FU_ALU][`DST_H:`DST_L] |
        dst == FUS[`FU_MEM][`DST_H:`DST_L] |
        dst == FUS[`FU_MUL][`DST_H:`DST_L] |
        dst == FUS[`FU_DIV][`DST_H:`DST_L] |
        dst == FUS[`FU_JUMP][`DST_H:`DST_L]);

assign normal_stall = structural_hazard | waw;
```

4.2 处理 WAR 冒险

在指令执行完成准备写回目标寄存器时，需要判断是否存在 WAR 冒险，若存在则需暂停指令的写回。

以 ALU 指令为例，判断是否可以写回时，需要查看除 ALU 外其余所有功能单元的源寄存器是否与 ALU 的目标寄存器相同。

- 若不同，则不会出现 WAR 冒险，可以写回。
- 若相同，则需要查看该源寄存器是否处于就绪状态：
  - 若处于就绪状态，说明指令尚未进行读操作数（RO），不能写回。
  - 若未就绪，说明指令已完成 RO 或正等待 ALU 写回，可以写回。

FU_ALU 单元的相关代码如下，其中 `ALU_WAR` 信号表示 ALU 指令可以写回目标寄存器，其它功能单元的代码与该单元类似，不再重复给出：

```
wire ALU_WAR = (
(FUS[`FU_MEM][`SRC1_H:`SRC1_L]  != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS
   [`FU_MEM][`RDY1])  &
(FUS[`FU_MEM][`SRC2_H:`SRC2_L]  != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS
   [`FU_MEM][`RDY2])  &
(FUS[`FU_MUL][`SRC1_H:`SRC1_L]  != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS
   [`FU_MUL][`RDY1])  &
(FUS[`FU_MUL][`SRC2_H:`SRC2_L]  != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS
   [`FU_MUL][`RDY2])  &
(FUS[`FU_DIV][`SRC1_H:`SRC1_L]  != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS
   [`FU_DIV][`RDY1])  &
(FUS[`FU_DIV][`SRC2_H:`SRC2_L]  != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS
   [`FU_DIV][`RDY2])  &
(FUS[`FU_JUMP][`SRC1_H:`SRC1_L] != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS
   [`FU_JUMP][`RDY1])  &
(FUS[`FU_JUMP][`SRC2_H:`SRC2_L] != FUS[`FU_ALU][`DST_H:`DST_L] | ~FUS
   [`FU_JUMP][`RDY2])
);
```

4.3 维护 ScoreBoard 记录

4.3.1 IS 阶段

在 IS 阶段，如果指令可以被发射，就需要将相关信息写入 FUS 和 RRS。

相关代码如下：

```
if (RO_en) begin
        if (|dst) RRS[dst] <= use_FU;

        FUS[use_FU][`BUSY] <= 1'b1;
        FUS[use_FU][`OP_H:`OP_L] <= op;
        FUS[use_FU][`DST_H:`DST_L] <= dst;
```

```
        FUS[use_FU][`SRC1_H:`SRC1_L] <= src1;
        FUS[use_FU][`SRC2_H:`SRC2_L] <= src2;
        FUS[use_FU][`FU1_H:`FU1_L] <= fu1;
        FUS[use_FU][`FU2_H:`FU2_L] <= fu2;
        FUS[use_FU][`RDY1] <= rdy1;
        FUS[use_FU][`RDY2] <= rdy2;
        FUS[use_FU][`FU_DONE] <= 1'b0;
        IMM[use_FU] <= imm;
        PCR[use_FU] <= PC;
end
```

4.3.2 RO 阶段

在 RO 阶段，如果指令的两个源寄存器都准备就绪，则需清除其 RDY 标志位。相关代码如下：

```
if (FUS[`FU_JUMP][`RDY1] & FUS[`FU_JUMP][`RDY2]) begin
        // JUMP
        FUS[`FU_JUMP][`RDY1] <= 1'b0;
        FUS[`FU_JUMP][`RDY2] <= 1'b0;
end
else if (FUS[`FU_ALU][`RDY1] & FUS[`FU_ALU][`RDY2]) begin
        // ALU
        FUS[`FU_ALU][`RDY1] <= 1'b0;
        FUS[`FU_ALU][`RDY2] <= 1'b0;
end
else if (FUS[`FU_MEM][`RDY1] & FUS[`FU_MEM][`RDY2]) begin
        // MEM
        FUS[`FU_MEM][`RDY1] <= 1'b0;
        FUS[`FU_MEM][`RDY2] <= 1'b0;
end
else if (FUS[`FU_MUL][`RDY1] & FUS[`FU_MUL][`RDY2]) begin
        // MUL
        FUS[`FU_MUL][`RDY1] <= 1'b0;
        FUS[`FU_MUL][`RDY2] <= 1'b0;
end
else if (FUS[`FU_DIV][`RDY1] & FUS[`FU_DIV][`RDY2]) begin
        // DIV
        FUS[`FU_DIV][`RDY1] <= 1'b0;
        FUS[`FU_DIV][`RDY2] <= 1'b0;
```

```
end
```

4.3.3 FU 阶段

在 FU 阶段，若指令执行完毕，则需要相应地更新其 FU_DONE 标记位为 1。

相关代码如下：

```
FUS[`FU_ALU][`FU_DONE] <= ALU_done;
FUS[`FU_MEM][`FU_DONE] <= MEM_done;
FUS[`FU_MUL][`FU_DONE] <= MUL_done;
FUS[`FU_DIV][`FU_DONE] <= DIV_done;
FUS[`FU_JUMP][`FU_DONE] <= JUMP_done;
```

4.3.4 WB 阶段

在 WB 阶段，需要清空当前指令的 FUS、RRS 信息。此外，若其它指令正在等待该寄存器，需对应地更新这些指令的 RDY 标记位。

FU_JUMP 单元的相关代码如下，其它功能单元的代码与该单元类似，不再重复给出：

```
if (FUS[`FU_JUMP][`FU_DONE] & JUMP_WAR) begin

        FUS[`FU_JUMP] <= 32'b0;
        RRS[FUS[`FU_JUMP][`DST_H:`DST_L]] <= 3'b0;

        if (FUS[`FU_ALU][`FU1_H:`FU1_L] == `FU_JUMP)
                FUS[`FU_ALU][`RDY1] <= 1'b1;
        if (FUS[`FU_MEM][`FU1_H:`FU1_L] == `FU_JUMP)
                FUS[`FU_MEM][`RDY1] <= 1'b1;
        if (FUS[`FU_MUL][`FU1_H:`FU1_L] == `FU_JUMP)
                FUS[`FU_MUL][`RDY1] <= 1'b1;
        if (FUS[`FU_DIV][`FU1_H:`FU1_L] == `FU_JUMP)
                FUS[`FU_DIV][`RDY1] <= 1'b1;
        if (FUS[`FU_ALU][`FU2_H:`FU2_L] == `FU_JUMP)
                FUS[`FU_ALU][`RDY2] <= 1'b1;
        if (FUS[`FU_MEM][`FU2_H:`FU2_L] == `FU_JUMP)
                FUS[`FU_MEM][`RDY2] <= 1'b1;
        if (FUS[`FU_MUL][`FU2_H:`FU2_L] == `FU_JUMP)
                FUS[`FU_MUL][`RDY2] <= 1'b1;
        if (FUS[`FU_DIV][`FU2_H:`FU2_L] == `FU_JUMP)
                FUS[`FU_DIV][`RDY2] <= 1'b1;
```

```
22
23 end
24 else if ...
```

5 实验结果

ROM 初始化内容如下：

NO.	Instruction	Addr.	Label	ASM	Comment
0	00000013	0	__start:	addi x0, x0, 0	
1	00402103	4		lw x2, 4(x0)	
2	00802203	8		lw x4, 8(x0)	Structural Hazard
3	004100b3	C		add x1, x2, x4	
4	fff08093	10		addi x1, x1, -1	WAW
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	

NO.	Instruction	Addr.	Label	ASM	Comment
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	ffff0097	50		auipc x1, 0xffff0	
21	0223c433	54		div x8, x7, x2	
22	025204b3	58		mul x9, x4, x5	St. Ha./RAW/WAW
23	022404b3	5C		mul x9, x8, x2	WAR
24	00400113	60		addi x2, x0, 4	
25	000000e7	64		jalr x1,0(x0)	
26	00000013	68		addi x0,x0,0	
27	00000013	6C		addi x0,x0,0	

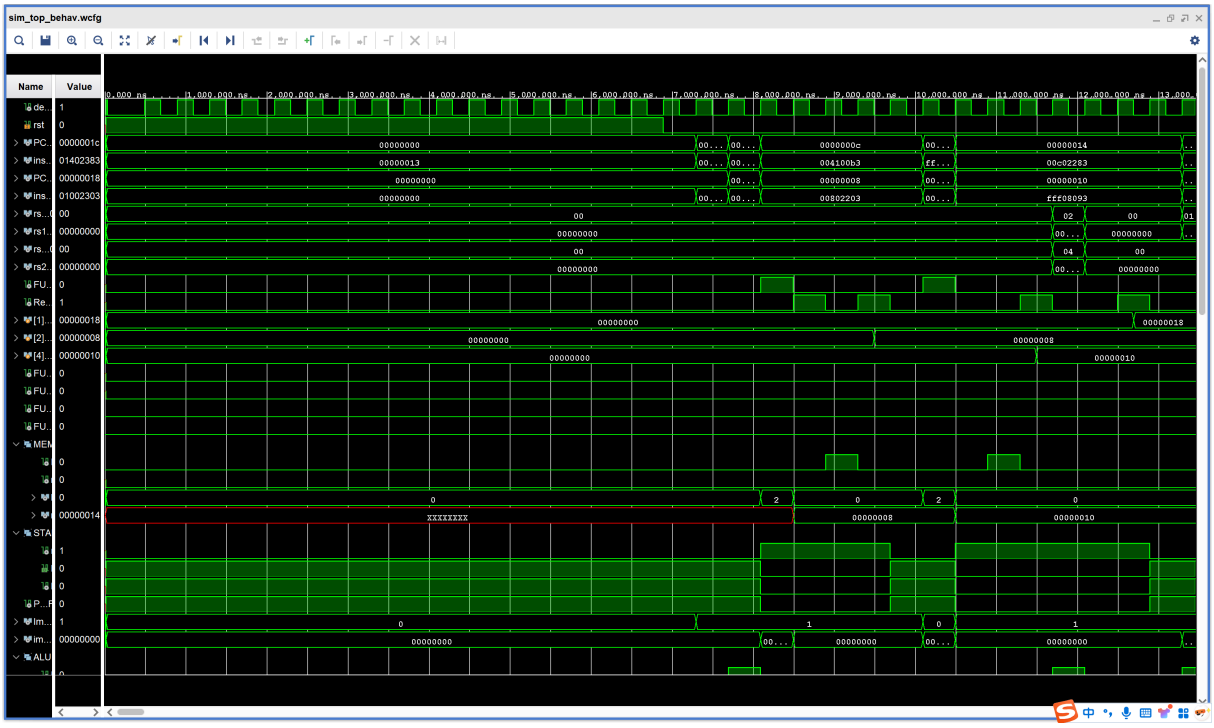
RAM 初始化内容如下：

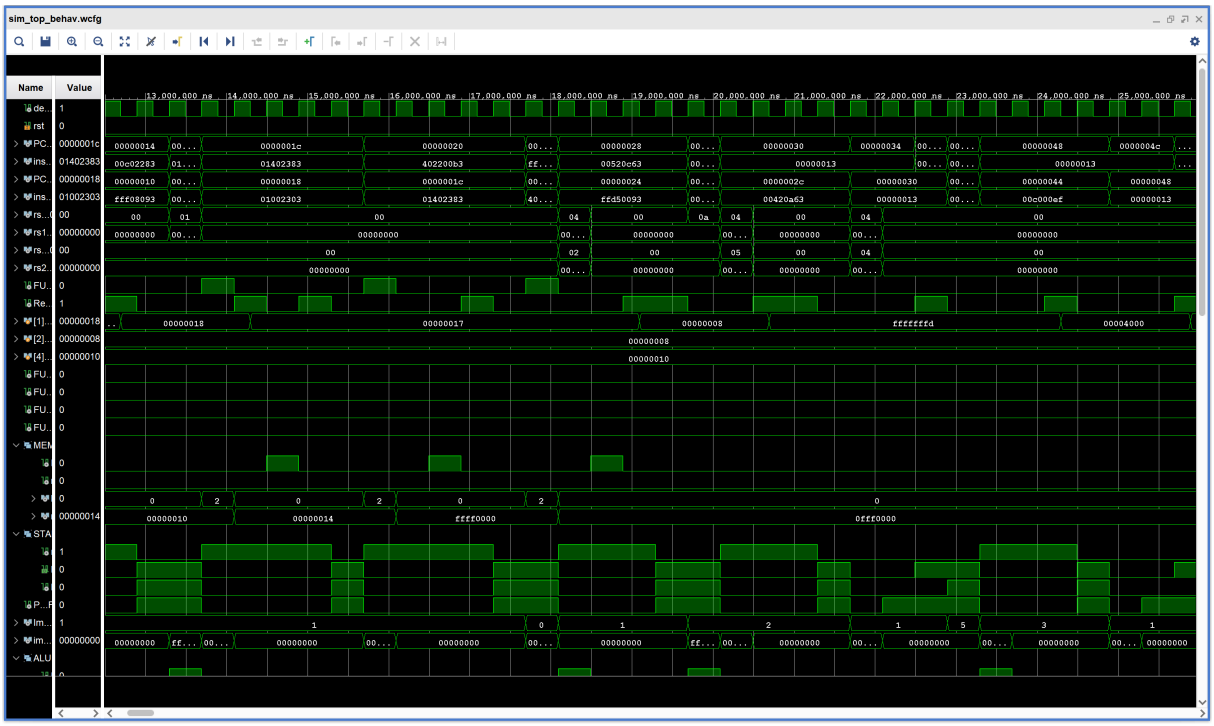
NO.	Data	Addr.
0	000080BF	0
1	00000008	4
2	00000010	8
3	00000014	C
4	FFFF0000	10
5	0FFF0000	14
6	FF000F0F	18
7	F0F0F0F0	1C
8	00000000	20
9	00000000	24
10	00000000	28
11	00000000	2C
12	00000000	30
13	00000000	34
14	00000000	38
15	00000000	3C

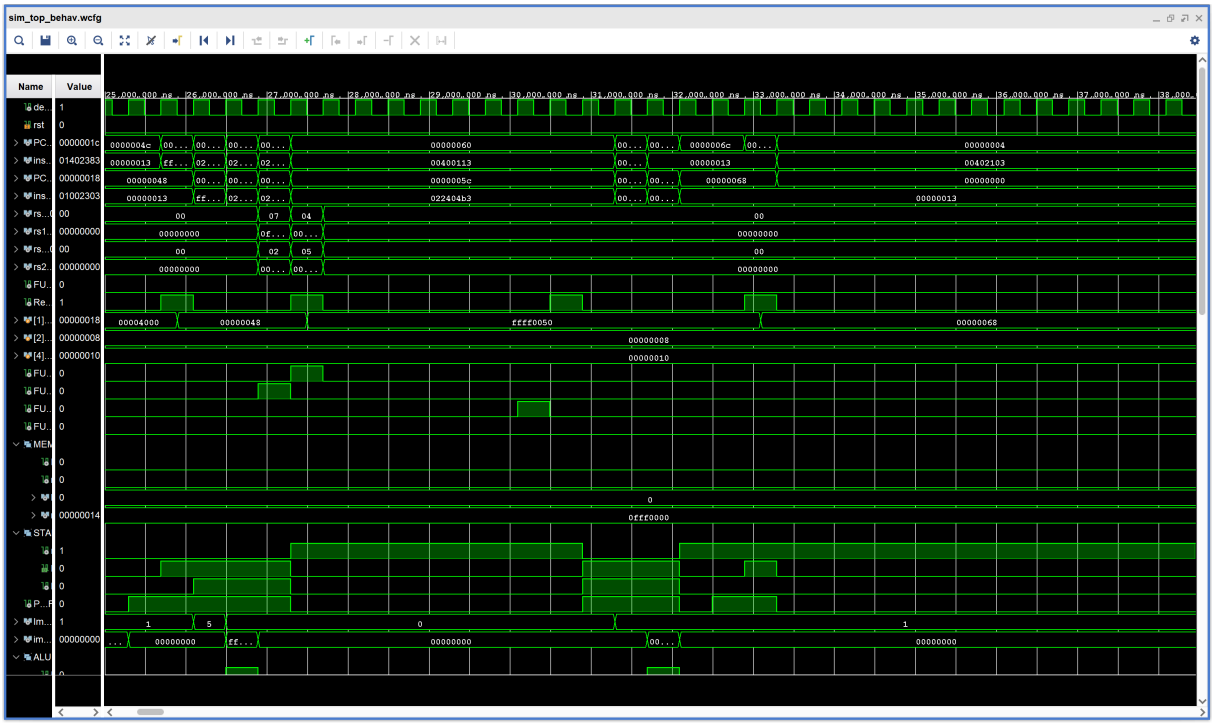
NO.	Instruction	Addr.
16	00000000	40
17	00000000	44
18	00000000	48
19	00000000	4C
20	A3000000	50
21	27000000	54
22	79000000	58
23	15100000	5C
24	00000000	60
25	00000000	64
26	00000000	68
27	00000000	6C
28	00000000	70
29	00000000	74
30	00000000	78
31	00000000	7C

5.1 仿真测试点分析

仿真波形如下：




sim_top_behav.wcfg
Name
Value


sim_top_behav.wcfg
Name
Value

下面选取部分关键情况进行详细说明：

5.1.1 处理结构冒险

```
// PC = 0x4-0x8
lw x2, 4(x0)
lw x4, 8(x0)
```

解释：

1. 执行第二条 Load 指令时，由于功能单元 FU_MEM 正在被前一条 Load 指令占用，发生结构冒险，控制单元发出 `normal_stall` 信号。
2. 结构冒险消除后，第二条 Load 指令继续执行，最终向 x4 寄存器写入了正确的值。

5.1.2 处理 WAW 冒险

```
// PC = 0xC-0x10（第一处）
add x1, x2, x4
addi x1, x1, -1

// PC = 0x40-0x44（第二处）
lui  x1,4
jal  x1,12
```

解释：

1. 由于前后两条指令的目标寄存器均为x1,存在 WAW 冒险,控制单元发出 `normal_stall` 信号。
2. 冒险消除后，第二条 ALU 指令继续执行，最终向 x1 寄存器写入正确的值。

5.1.3 处理 WAR 冒险

```
// PC = 0x5C-0x60
mul x9, x8, x2
addi x2, x0, 4
```

解释：

1. 由于后一条指令的目标寄存器和前一条指令的源寄存器相同，存在 WAR 冒险。

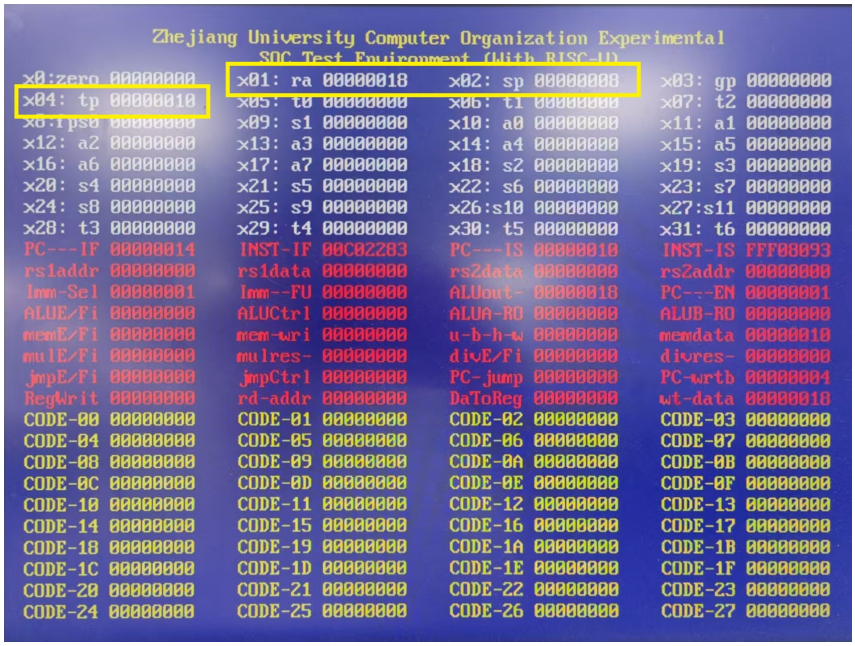
2. 在 ADDI 指令的 WB 阶段，控制单元将检查是否存在指令的源寄存器与 ADDI 指令的目标寄存器相同。检查到 WAR 冒险后，处理器进行停顿以等待 MUL 指令完成对源寄存器的读取。

3. 冒险消除后，ADDI 指令继续执行 WB 阶段，最终寄存器 x2、x9 均写入正确的值。

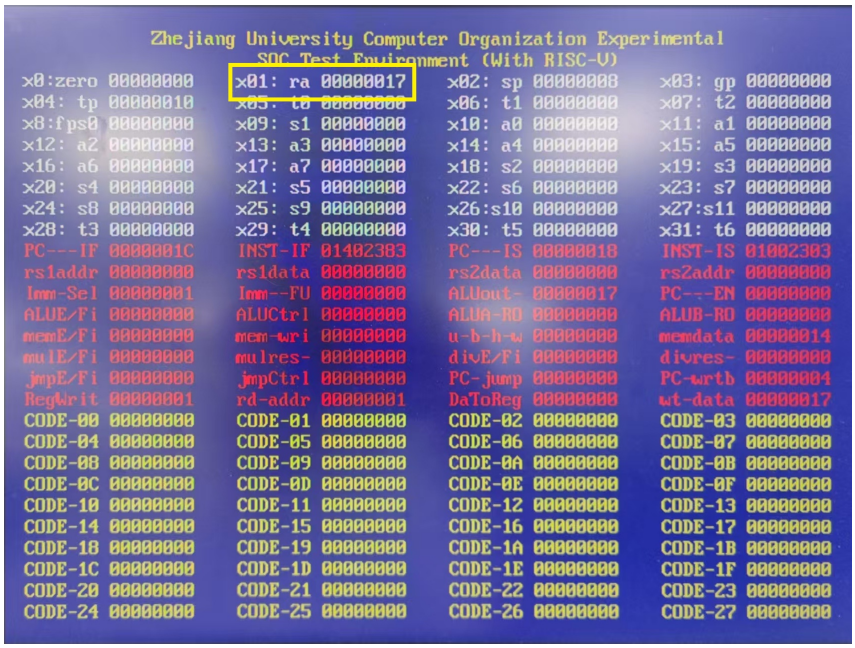
5.2 上板测试结果分析

上板测试结果如下，均与预期相符：

测试点 1：x2，x4 数据载入正确，ADD 指令正确，成功处理结构冒险（PC：0x0-0xC）

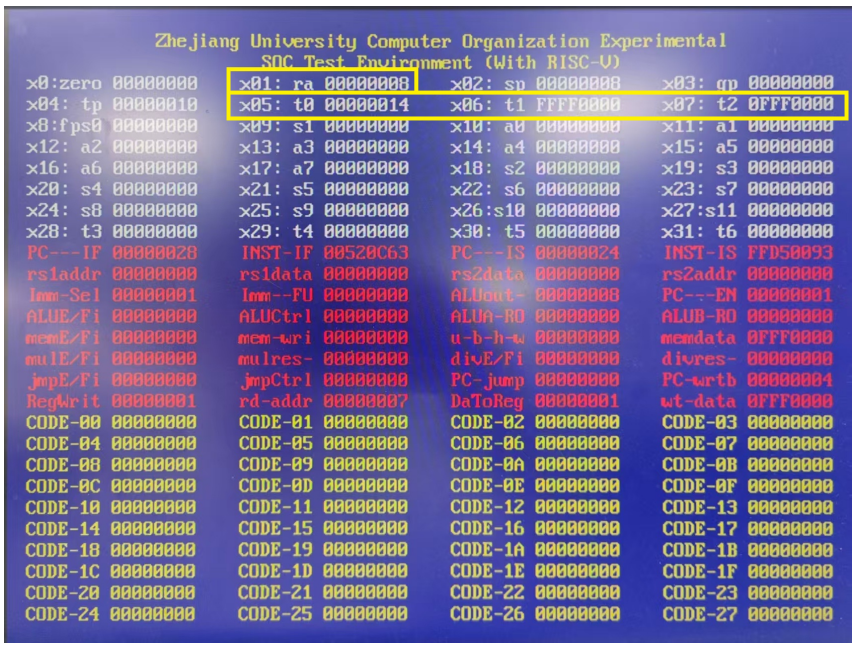


测试点 2：ADDI 指令执行正确，成功处理 WAW 冒险（PC：0x10）



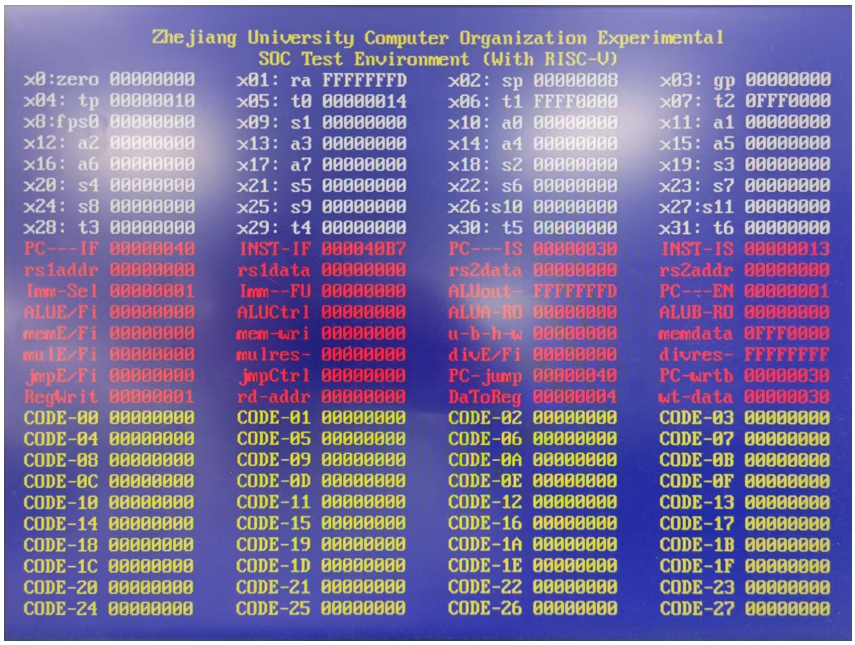
测试点 3：x5，x6，x7 数据载入正确，SUB 指令执行正确

（PC：0x14-0x20）



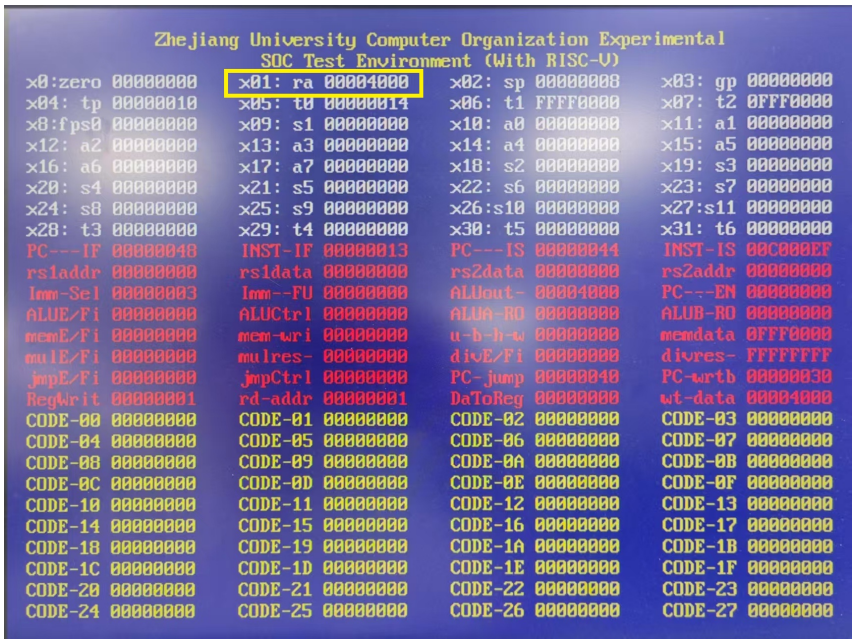
测试点 4：BEQ 指令跳转正确

（PC：0x2C，跳转至 0x40）



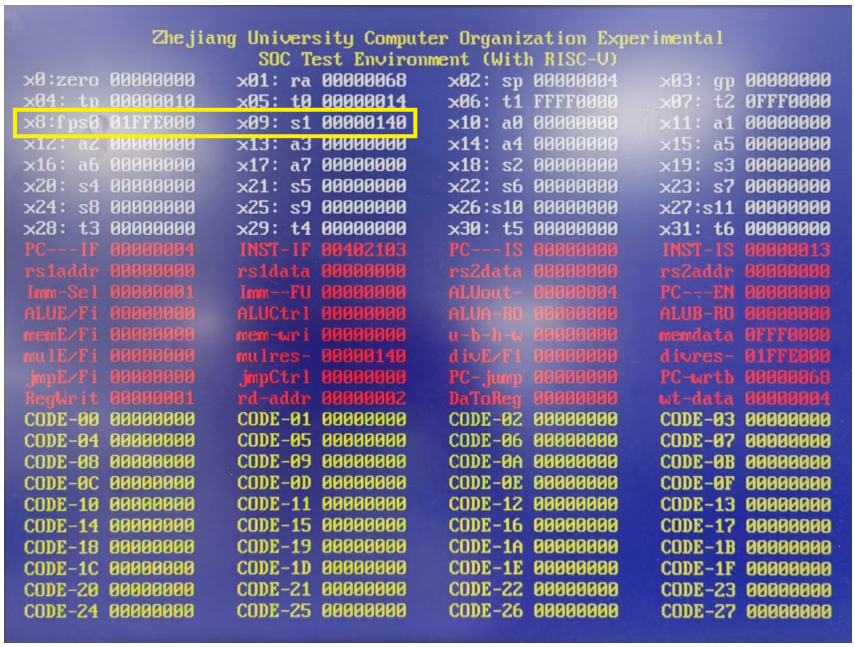
测试点 5：LUI 指令执行正确

（PC：0x40，x1=0x00004000）



测试点 6：MUL、DIV 指令执行正确，成功处理结构冒险和三种数据冒险

（PC：0x54-0x5C）


Zhejiang University Computer Organization Experimental
SOC Test Environment (With RISC-V)
x0:zero 00000000 x01: ra 00000068 x02: sp 00000004 x03: gp 00000000
x04: tp 00000010 x05: t0 00000014 x06: t1 FFFF0000 x07: t2 0FFF0000
x08:fps0 01FFE000 x09: s1 00000140 x10: a0 00000000 x11: a1 00000000
x12: a2 00000000 x13: a3 00000000 x14: a4 00000000 x15: a5 00000000
x16: a6 00000000 x17: a7 00000000 x18: s2 00000000 x19: s3 00000000
x20: s4 00000000 x21: s5 00000000 x22: s6 00000000 x23: s7 00000000
x24: s8 00000000 x25: s9 00000000 x26:s10 00000000 x27:s11 00000000
x28: t3 00000000 x29: t4 00000000 x30: t5 00000000 x31: t6 00000000
PC---IF 00000004 INST-IF 00402103 PC---IS 00000000 INST-IS 00000013
rs1addr 00000000 rs1data 00000000 rs2data 00000000 rs2addr 00000000
Imm-Sel 00000001 Imm--FU 00000000 ALUout- 00000004 PC---EN 00000000
ALUE/Fi 00000000 ALUCtrl 00000000 ALUA-RO 00000000 ALUB-RO 00000000
memE/Fi 00000000 mem-wri 00000000 u-b-h-w 00000000 memdata 0FFF0000
mulE/Fi 00000000 mulres- 00000140 divE/Fi 00000000 divres- 01FFE000
jmpE/Fi 00000000 jmpCtrl 00000000 PC-jump 00000000 PC-wrtb 00000068
RegWrit 00000001 rd-addr 00000002 DaToReg 00000000 wt-data 00000004
CODE-00 00000000 CODE-01 00000000 CODE-02 00000000 CODE-03 00000000
CODE-04 00000000 CODE-05 00000000 CODE-06 00000000 CODE-07 00000000
CODE-08 00000000 CODE-09 00000000 CODE-0A 00000000 CODE-0B 00000000
CODE-0C 00000000 CODE-0D 00000000 CODE-0E 00000000 CODE-0F 00000000
CODE-10 00000000 CODE-11 00000000 CODE-12 00000000 CODE-13 00000000
CODE-14 00000000 CODE-15 00000000 CODE-16 00000000 CODE-17 00000000
CODE-18 00000000 CODE-19 00000000 CODE-1A 00000000 CODE-1B 00000000
CODE-1C 00000000 CODE-1D 00000000 CODE-1E 00000000 CODE-1F 00000000
CODE-20 00000000 CODE-21 00000000 CODE-22 00000000 CODE-23 00000000
CODE-24 00000000 CODE-25 00000000 CODE-26 00000000 CODE-27 00000000