

实验八 流水线 MIPS 处理器设计

23342107 徐睿琳

1 实验要求

使用流水线技术，设计一个 MIPS 处理器，能够正确执行冒泡排序、64 位数据加法和最大公约数及最小公倍数求解程序。

2 实验目的

- 1、了解提高处理器性能的方法；
- 2、学习并掌握流水线 MIPS 处理器设计；
- 3、理解数据冒险、控制冒险的概念以及流水线冲突的解决方法；
- 4、掌握流水线 MIPS 处理器的测试方法。

3 实验内容

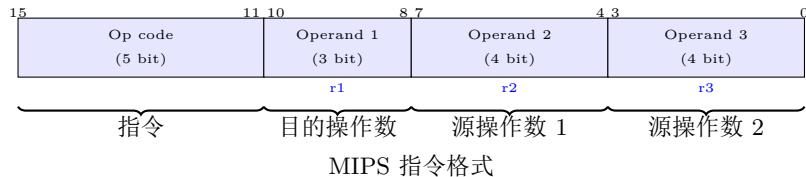
- 1、将 CPU 中 ID 的部分代码及 ALU 代码补充完整 (start 到 end 部分)；
- 2、编写 testbench，验证所补充的部分功能是否正确；

3.1 FPGA 验证

4 实验步骤

4.1 实验一 代码补全

4.1.1 ID 模块代码补全



(1) 若 ID 需要 EX 段结果

```

1 // 若 EX 段指令是运算类指令(不含LOAD)，且目的寄存器等于当前指令源寄存器
2 if ((ex_ir[15:11] == `ADD || ex_ir[15:11] == `LDIH || ex_ir[15:11] == `ADDI
3   || ex_ir[15:11] == `SUB || ex_ir[15:11] == `SUBI || ex_ir[15:11] == `ADDC
4   || ex_ir[15:11] == `SUBC || ex_ir[15:11] == `AND || ex_ir[15:11] == `OR
5   || ex_ir[15:11] == `XOR || ex_ir[15:11] == `SLL || ex_ir[15:11] == `SRL
6   || ex_ir[15:11] == `SLA || ex_ir[15:11] == `SRA)
7     && ex_ir[10:8] == id_ir[2:0])
8     reg_B <= ALUo;

```

在 EX 段实现前递判定以消除数据冒险: 当 EX 段指令属于在 EX 端即可产生最终运算结果的算术/移位/立即数类指令 (排除了 LOAD 指令), 且该指令的目的寄存器 `ex_ir[10:8]` 与当前 ID 段待译指令的源寄存器 `id_ir[2:0]` 匹配时, 将 EX 段的 ALU 输出 ALUo 直接前递赋给 `reg_B` (或相应源操作数)。这样可以在不引入气泡的情况下保证数据正确性并提高流水线吞吐率。

(2) 若 ID 需要 MEM 段结果

```

1 // 若 MEM 段指令是运算类或LOAD指令, 且目的寄存器等于当前指令源寄存器
2 else if ((mem_ir[15:11] == `ADD || mem_ir[15:11] == `LDIH || mem_ir[15:11] == `ADDI
3     || mem_ir[15:11] == `SUB || mem_ir[15:11] == `SUBI || mem_ir[15:11] == `ADDC
4     || mem_ir[15:11] == `SUBC || mem_ir[15:11] == `AND || mem_ir[15:11] == `OR
5     || mem_ir[15:11] == `XOR || mem_ir[15:11] == `SLL || mem_ir[15:11] == `SRL
6     || mem_ir[15:11] == `SLA || mem_ir[15:11] == `SRA || mem_ir[15:11] == `LOAD)
7     && mem_ir[10:8] == id_ir[2:0]) begin
8     if (mem_ir[15:11] == `LOAD)
9         reg_B <= d_datain; // 若是 LOAD 指令, 数据来自外部内存
10    else
11        reg_B <= reg_C; // 若是运算指令, 数据来自 EX 段计算结果
12 end

```

在 ID 级对来自 MEM 级的结果进行前递判定, 以尽早消除数据冒险并降低流水线停顿:

- 当 MEM 指令属于运算类或 LOAD, 且其目的寄存器 `mem_ir[10:8]` 与当前 ID 指令的源寄存器匹配时, 优先选择 MEM 级的数据作为该源操作数的供给;
- 若匹配且为 LOAD, 数据来自外部存储输入 `d_datain`;
- 否则由流水线寄存器 (如 `reg_C`) 提供 EX 级计算结果。

保证前递多路选择器和相关控制信号与 MEM 级数据到达时序一致, 防止毛刺或竞争条件: 对于无法通过前递解决的 load-use 情形, 仍需通过插入气泡或暂停取指来保证正确性。

(3) 若 ID 需要 WB 段结果

```

1 // 若 WB 段指令是运算类或LOAD指令, 且目的寄存器等于当前指令源寄存器
2 else if ((wb_ir[15:11] == `ADD || wb_ir[15:11] == `LDIH || wb_ir[15:11] == `ADDI
3     || wb_ir[15:11] == `SUB || wb_ir[15:11] == `SUBI || wb_ir[15:11] == `ADDC
4     || wb_ir[15:11] == `SUBC || wb_ir[15:11] == `AND || wb_ir[15:11] == `OR
5     || wb_ir[15:11] == `XOR || wb_ir[15:11] == `SLL || wb_ir[15:11] == `SRL
6     || wb_ir[15:11] == `SLA || wb_ir[15:11] == `SRA || wb_ir[15:11] == `LOAD)
7     && wb_ir[10:8] == id_ir[2:0])
8     reg_B <= reg_C1; // 数据来自写回阶段的结果

```

当写回指令为算术/移位/立即数类或 LOAD (即可在 WB 得到最终写回值), 且其目的寄存器 `wb_ir[10:8]` 与当前 ID 指令的源寄存器匹配时, 优先使用写回数据 `reg_C1` 作为该源操作数的供给。

该判定位于 EX/MEM 前递逻辑之后作为保底路径, 以保证在 EX/MEM 未命中但 WB 已有最新结果的情况下仍能消除依赖并避免不必要的气泡。

4.1.2 ALU 模块代码补全

```

1 if(ex_ir[15:11] == `ADD || ex_ir[15:11] == `LDIH || ex_ir[15:11] == `ADDI)
2     {cf,ALUo} <= reg_A + reg_B;
3 else if(ex_ir[15:11] == `CMP || ex_ir[15:11] == `SUB || ex_ir[15:11] == `SUBI)
4     {cf,ALUo} <= reg_A - reg_B;
5 else if(ex_ir[15:11] == `ADDC )
6     {cf,ALUo} <= reg_A + reg_B + cin;
7 else if(ex_ir[15:11] == `SUBC )
8     {cf,ALUo} <= reg_A - reg_B - cin;
9 else if(ex_ir[15:11] == `OR )
10    {cf,ALUo} <= reg_A | reg_B;
11 else if(ex_ir[15:11] == `AND)
12    {cf,ALUo} <= reg_A & reg_B;
13 else if(ex_ir[15:11] == `XOR)
14    {cf,ALUo} <= reg_A ^ reg_B;
15 else if(ex_ir[15:11] == `SLL)
16    {cf,ALUo} <= reg_A << reg_B;
17 else if(ex_ir[15:11] == `SRL)
18    {cf,ALUo} <= reg_A >> reg_B;
19 else if(ex_ir[15:11] == `SLA)
20    {cf,ALUo} <= reg_A << reg_B;
21 else if(ex_ir[15:11] == `SRA)
22    {cf,ALUo} <= reg_A >>> reg_B;
23 else if(ex_ir[15:11] == `LOAD || ex_ir[15:11] == `STORE || ex_ir[15:11] == `BN
24 || ex_ir[15:11] == `BNN || ex_ir[15:11] == `BZ || ex_ir[15:11] == `BNZ
25 || ex_ir[15:11] == `BC || ex_ir[15:11] == `BNC || ex_ir[15:11] == `JMPR)
26     {cf,ALUo} <= reg_A + reg_B;

```

逻辑较为简单，根据指令类型对 ALU 执行相应的算术或逻辑运算，并将结果存储在 ALUo 中，同时更新进位标志 cf。对于分支和跳转指令，ALU 计算目标地址或偏移量。

4.2 testbench 补全与验证

testbench 设计与验证流程如算法 1 所示，完整代码见附录部分。

Algorithm 1: Testbench 设计与验证流程

```

输入: clk, enable, reset, start, read, address, clkreset
输出: light, an, light2, an2, 控制台输出
变量: 整数 x, 模块实例 uut

// 时钟生成进程 (并发)

1 进程 1: clk 每 1 个时间单位翻转;
2 进程 2: clkreset 每 20 个时间单位翻转;

// 主仿真序列

3 初始块

// 步骤 0: 初始化所有信号以避免 X (未知) 状态
4   clk ← 0; clkreset ← 0;
5   enable ← 0; start ← 0; read ← 0;
6   address ← 8'h00; x ← 0;

// 步骤 1: 系统复位 (低电平有效)
7   reset ← 0;
8   等待 5 个时钟周期 (clk 上升沿);
9   reset ← 1; // 释放复位

// 步骤 2: 使能并启动 CPU
10  等待 2 个时钟周期 (clk 上升沿);
11  enable ← 1;
12  start ← 1;

// 步骤 3: 执行阶段
// 等待指令执行完成 (含循环/跳转)
13  等待 2000 个时钟周期 (clk 上升沿);

// 步骤 4: 结果校验 (内存转储)
14  read ← 0;
15  x ← 0;
16  while x < 65 do
    // 直接访问被测单元 (UUT) 内部的内存数组
17    显示 " 地址 x: uut.d_mem[x] ";
18    x ← x + 1;

19 结束;

```

5 板级验证结果

5.1 管脚约束设置

在 Vivado 开发环境中，物理管脚的映射通过 **xdc** 文件实现。本实验中，我们需要将 Verilog 程序中定义的顶层输入输出端口与 EGO1 开发板上的真实 FPGA 引脚建立物理连接。首先在工程管理器中新建约束文件，选择文件类型为.xdc，用于存储后续的引脚分配和时序定义指令（完整约束代码见附录部分）。

表 1: FPGA 管脚约束与端口分配映射表

信号名称	端口方向	FPGA 管脚	硬件模块	电平标准
clk	Input	P17	系统时钟 (100MHz)	LVCMOS33
clkreset	Input	R3	拨码开关	LVCMOS33
reset	Input	T3	拨码开关	LVCMOS33
enable	Input	T5	拨码开关	LVCMOS33
start	Input	R17	按键开关	LVCMOS33
read	Input	R11	按键开关	LVCMOS33
adress[7:0]	Input	P5, P4, P3, P2, R2, M4, N4, R1	拨码开关 (SW7--SW0)	LVCMOS33
light[6:0]	Output	D4, D3, E3, F4, F3, E2, D2	数码管段选 (下半片)	LVCMOS33
an[3:0]	Output	G1, F1, E1, G6	数码管位选 (下半片)	LVCMOS33
light2[6:0]	Output	B4, A4, A3, B1, A1, B3, B2	数码管段选 (上半片)	LVCMOS33
an2[3:0]	Output	G2, C2, C1, H1	数码管位选 (上半片)	LVCMOS33

5.2 用十进制显示数码管

为便于板级观察，本实验在数码管端实现 **十进制显示**。

i. D_mem 读取并十进制输出

D_mem 为 16 位数据，最大为 65535_{10} ，需 **5 位十进制数**表示。因此采用上下两片数码管联合显示：**下半片显示低 4 位十进制，上半片最低位显示最高 1 位十进制**。

ii. PC 码读取并十进制输出

PC 最大不超过 255_{10} ，仅需 **3 位十进制数**表示，使用 **上半片高 3 位数码管**显示。

iii. 译码器设计

需要两个译码模块完成“二进制 \rightarrow BCD \rightarrow 七段码”的显示链路（完整代码见附录部分）：

- (1) **下半片译码器**：输入 D_mem (16 位二进制)，转换得到 5 个 BCD 数字；其中低 4 位 BCD 在下半片完成 $0 \sim 9$ 七段译码输出，最高位 BCD 转交上半片显示。
- (2) **上半片译码器**：同时接收 D_mem 的最高位 BCD，以及 PC 的低 8 位二进制输入；PC 转换为 3 个 BCD 数字后进行 $0 \sim 9$ 七段译码，分别驱动上半片高 3 位；上半片最低位用于显示 D_mem 的最高位。

5.3 实验结果

5.3.1 实验用开发板概述

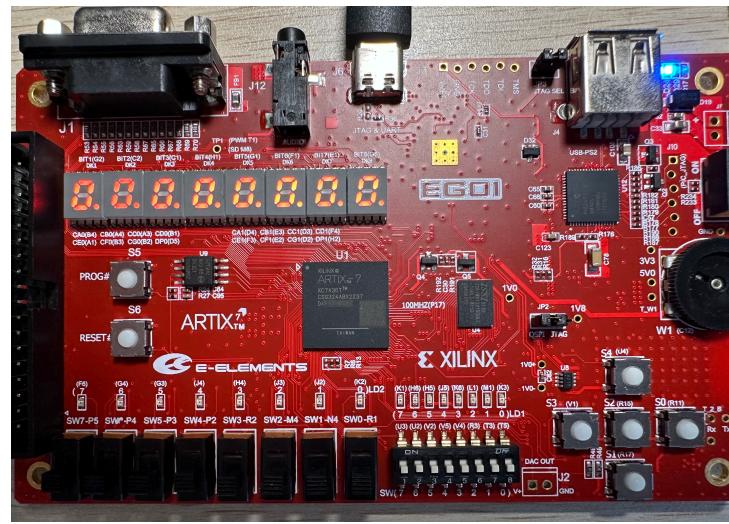
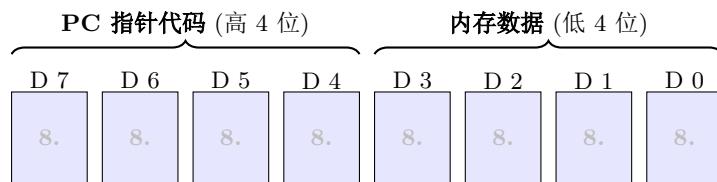


图 1：实验所用 FPGA 开发板示意图

本实验采用 EGO1 开发板进行板级验证，通过管脚约束将硬件 I/O 端口与逻辑信号建立映射：输入端利用板载拨码开关设定内存地址，

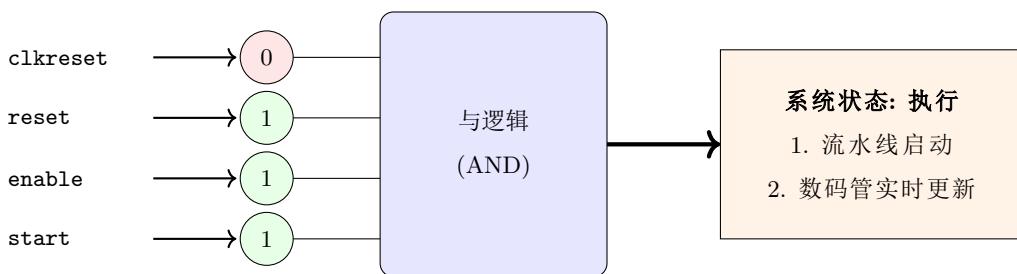
控制信号 clkreset、reset 及 enable 被分配至指定拨码开关，而 start 与 read 信号则由按键触发；

输出显示逻辑分配 (十六进制模式)

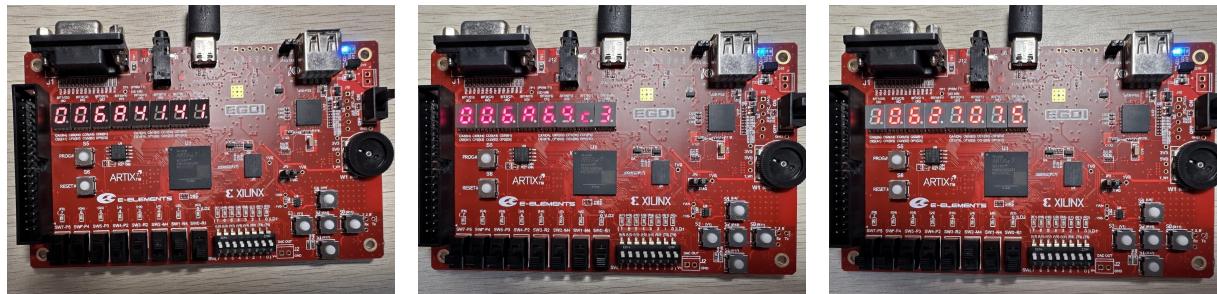


输出端利用 8 位七段数码管实时监测系统状态，并根据实验需求设计了十六进制与十进制两种显示模式，在十六进制模式下，数码管高四位用于显示 PC 码，低四位显示内存数据，十进制模式则相应调整位宽分配以呈现对应的十进制数值；系统设有严格的启动时序保护，当且仅当 `clkreset` 信号置为低电平 (0)，且 `reset`、`enable` 及 `start` 信号均置为高电平 (1) 时，流水线处理器方进入执行状态，数码管随即启动并实时更新显示当前的 PC 地址与内存数据。

系统启动时序保护逻辑



5.3.2 全指令测试



ADD:00000016,CELL:4141

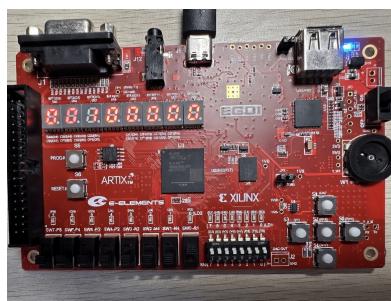
ADD:00000004,CELL:69c3

PC:6AH=106D; 数据 69c3H=27075D

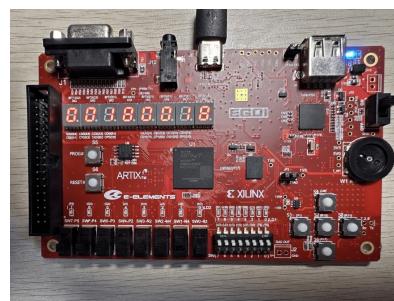
PC=106D 对应 I_mem 中 HALT 的下一条指令地址：由于 HALT 进入 id_ir 时，PC 自增并指向下一条指令（即 $105 + 1 = 106$ ）

1 ADD:00000000,CELL:ffffd	1 ADD:00000016,CELL:4141	1 ADD:0000002d,CELL:69c3
2 ADD:00000001,CELL:0004	2 ADD:00000017,CELL:eb eb	2 ADD:0000002e,CELL:34e1
3 ADD:00000002,CELL:0005	3 ADD:00000018,CELL:aaaa	3 ADD:0000002f,CELL:0069
4 ADD:00000003,CELL:c369	4 ADD:00000019,CELL:c369	4 ADD:00000030,CELL:0000
5 ADD:00000004,CELL:69c3	5 ADD:0000001a,CELL:86d2	5 ADD:00000031,CELL:xxxx
6 ADD:00000005,CELL:0041	6 ADD:0000001b,CELL:3690	6 ADD:00000032,CELL:xxxx
7 ADD:00000006,CELL:fffff	7 ADD:0000001c,CELL:8000	7 ADD:00000033,CELL:xxxx
8 ADD:00000007,CELL:0001	8 ADD:0000001d,CELL:c369	8 ADD:00000034,CELL:xxxx
9 ADD:00000008,CELL:0000	9 ADD:0000001e,CELL:61b4	9 ADD:00000035,CELL:xxxx
10 ADD:00000009,CELL:0000	10 ADD:0000001f,CELL:00c3	10 ADD:00000036,CELL:xxxx
11 ADD:0000000a,CELL:0000	11 ADD:00000020,CELL:0001	11 ADD:00000037,CELL:xxxx
12 ADD:0000000b,CELL:xxxx	12 ADD:00000021,CELL:c369	12 ADD:00000038,CELL:0030
13 ADD:0000000c,CELL:xxxx	13 ADD:00000022,CELL:86d2	13 ADD:00000039,CELL:xxxx
14 ADD:0000000d,CELL:xxxx	14 ADD:00000024,CELL:8000	14 ADD:0000003a,CELL:xxxx
15 ADD:0000000e,CELL:xxxx	15 ADD:00000025,CELL:69c3	15 ADD:0000003b,CELL:xxxx
16 ADD:0000000f,CELL:xxxx	16 ADD:00000026,CELL:d386	16 ADD:0000003c,CELL:xxxx
17 ADD:00000010,CELL:b600	17 ADD:00000027,CELL:c300	17 ADD:0000003d,CELL:xxxx
18 ADD:00000011,CELL:0001	18 ADD:00000028,CELL:8000	18 ADD:0000003e,CELL:xxxx
19 ADD:00000012,CELL:0005	19 ADD:00000029,CELL:c369	19 ADD:0000003f,CELL:xxxx
20 ADD:00000013,CELL:0001	20 ADD:0000002a,CELL:e1b4	20 ADD:00000040,CELL:xxxx
21 ADD:00000014,CELL:ffff	21 ADD:0000002b,CELL:ffc3	

5.3.3 最大公约数、最小公倍数算法验证



ADD:00000003,CELL:0008



ADD:00000002,CELL:0018



PC:18H=24D; 数据 0018H=0024D

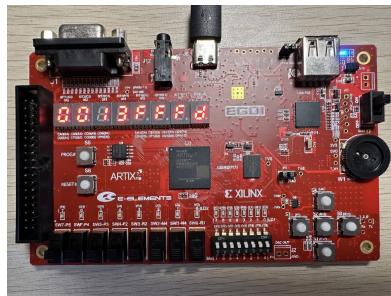
PC=24D 对应着 I_mem 中 HALT 指令下一条指令的地址，即下一条指令的地址 24D

1 ADD:00000000,CELL:0000
2 ADD:00000001,CELL:0020
3 ADD:00000002,CELL:0018
4 ADD:00000003,CELL:0008
5 ADD:00000004,CELL:0060

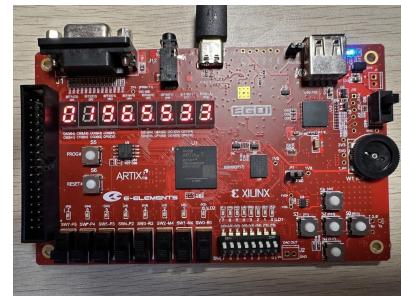
5.3.4 64 位加法验证



ADD:00000002, CELL:ffffe



ADD:00000008, CELL:ffffd



PC:13H=19D; 数据 fffdH=65533D

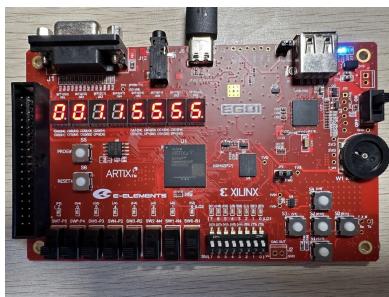
PC=19D 对应着 I_mem 中 HALT 指令下一条指令的地址 19D

```

1 ADD:00000000,CELL:ffffe
2 ADD:00000001,CELL:ffffe
3 ADD:00000002,CELL:ffffe
4 ADD:00000003,CELL:0000
5 ADD:00000004,CELL:fffff
6 ADD:00000005,CELL:fffff
7 ADD:00000006,CELL:fffff
8 ADD:00000007,CELL:0000
9 ADD:00000008,CELL:ffffd
10 ADD:00000009,CELL:ffffe
11 ADD:0000000a,CELL:ffffe
12 ADD:0000000b,CELL:0001

```

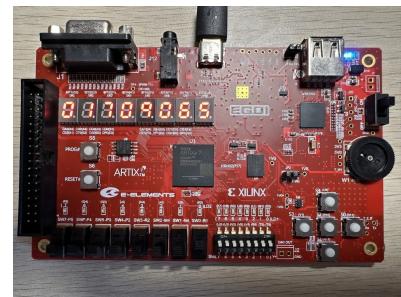
5.3.5 冒泡排序算法测试



ADD:00000002, CELL:ffffe



ADD:00000008, CELL:ffffd



PC:13H=19D; 数据 fffdH=65533D

PC=17D 对应着 I_mem 中 HALT 指令下一条指令的地址 17D

```

1 ADD:00000000,CELL:000a
2 ADD:00000001,CELL:69c3
3 ADD:00000002,CELL:6152
4 ADD:00000003,CELL:5555
5 ADD:00000004,CELL:2895
6 ADD:00000005,CELL:2369

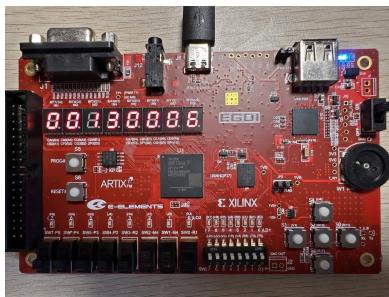
```

```

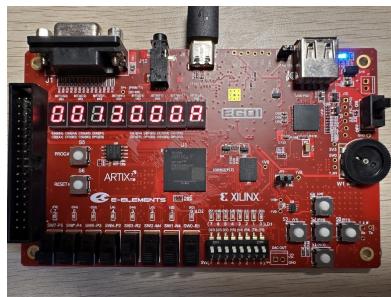
7 ADD:00000006,CELL:1057
8 ADD:00000007,CELL:0fff
9 ADD:00000008,CELL:0060
10 ADD:00000009,CELL:0005
11 ADD:0000000a,CELL:0004
12 ADD:0000000b,CELL:0000

```

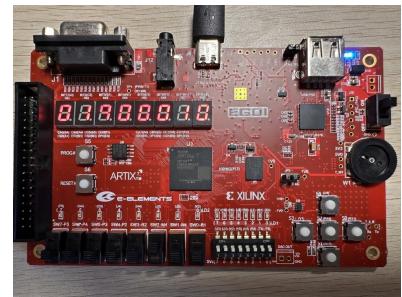
5.3.6 sort 排序算法测试



ADD:00000002,CELL:ffffe



ADD:00000008,CELL:fffd



PC:13H=19D; 数据 fffdH=65533D

PC=19D 也正好对应着 I_mem 中 HALT 指令下一条指令的地址 19D

```

1 ADD:00000000,CELL:0000
2 ADD:00000001,CELL:0000
3 ADD:00000002,CELL:0001
4 ADD:00000003,CELL:0003
5 ADD:00000004,CELL:0004
6 ADD:00000005,CELL:0005
7 ADD:00000006,CELL:0006
8 ADD:00000007,CELL:0009
9 ADD:00000008,CELL:000a
10 ADD:00000009,CELL:0011

```

6 总结

本实验完成了流水线 MIPS 处理器的关键模块补全与系统级验证。首先在 ID 阶段加入基于 EX/MEM/WB 三级的前递判定逻辑，实现对算术/逻辑/移位/立即数以及 LOAD 等指令的数据相关处理，尽量在不插入气泡的情况下消除数据冒险；随后补全 ALU 的各类运算与地址计算路径，保证控制信号与运算结果在流水线各级之间传递一致。

在验证环节，编写 testbench 生成时钟与复位序列，按照启动时序对 CPU 进行使能运行，并通过转储 d_mem 的方式对执行结果进行对比检查。仿真结果表明处理器能够正确执行全指令集测试、最大公约数/最小公倍数求解、64 位加法以及冒泡排序等程序，说明前递逻辑与 ALU 运算功能满足设计要求。

在板级验证部分，完成 XDC 管脚约束与数码管显示链路设计，将 PC 与数据存储器内容以十进制方式输出，便于观察运行状态与最终结果。总体上，本实验加深了对流水线结构、数据/控制冒险来源及其解决方法（前递、必要时停顿）的理解，并熟悉了从模块实现、仿真验证到 FPGA 约束与板级观测的完整硬件开发流程。

附录 A test.v 代码

```
1 `timescale 1ps / 1ps
2 module test;
3
4 // inputs
5 reg clk;
6 reg enable;
7 reg reset;
8 reg start;
9 reg read;
10 reg [7:0] adress;
11 reg clkreset;
12
13 // outputs
14 wire [6:0] light;
15 wire [3:0] an;
16 wire [6:0] light2;
17 wire [3:0] an2;
18
19 integer x;
20
21 // UUT
22 cpu uut (
23     .clk(clk),
24     .enable(enable),
25     .reset(reset),
26     .read(read),
27     .clkreset(clkreset),
28     .start(start),
29     .adress(adress),
30     .light(light),
31     .an(an),
32     .light2(light2),
33     .an2(an2)
34 );
35
36 // clocks
37 always #1 clk      = ~clk;
38 always #20 clkreset = ~clkreset;
39
40 initial begin
```

```

41 // init
42 clk      = 1'b0;
43 clkreset = 1'b0;
44 enable   = 1'b0;
45 start    = 1'b0;
46 read     = 1'b0;
47 adress   = 8'h00;
48 x        = 0;
49
50 // reset (active low)
51 reset = 1'b0;
52 repeat (5) @(posedge clk);
53 reset = 1'b1;
54
55 // enable + start
56 repeat (2) @(posedge clk);
57 enable = 1'b1;
58 start  = 1'b1;
59
60 // run
61 repeat (2000) @(posedge clk);
62
63 // dump d_mem
64 read = 1'b0;
65 x   = 0;
66 repeat (65) begin
67   $display("ADD:%h,CELL:%h", x, uut.d_mem.d_mem[x]);
68   x = x + 1;
69 end
70
71 $finish;
72 end
73
74 endmodule

```

附录 B 管脚约束代码

```

1 # 给下半片4个数码管的段选输出信号分配电平
2 set_property IOSTANDARD LVCMOS33 [get_ports {light[6]}]
3 set_property IOSTANDARD LVCMOS33 [get_ports {light[5]}]
4 set_property IOSTANDARD LVCMOS33 [get_ports {light[4]}]

```

```

5  set_property IOSTANDARD LVCMOS33 [get_ports {light[3]}]
6  set_property IOSTANDARD LVCMOS33 [get_ports {light[2]}]
7  set_property IOSTANDARD LVCMOS33 [get_ports {light[1]}]
8  set_property IOSTANDARD LVCMOS33 [get_ports {light[0]}]

9
10 # 给下半片4个数码管的段选输出信号分配接口
11 set_property PACKAGE_PIN D4 [get_ports {light[6]}]
12 set_property PACKAGE_PIN D3 [get_ports {light[4]}]
13 set_property PACKAGE_PIN E3 [get_ports {light[5]}]
14 set_property PACKAGE_PIN F4 [get_ports {light[3]}]
15 set_property PACKAGE_PIN F3 [get_ports {light[2]}]
16 set_property PACKAGE_PIN E2 [get_ports {light[1]}]
17 set_property PACKAGE_PIN D2 [get_ports {light[0]}]

18
19 # 给上半片4个数码管的段选输出信号分配电平
20 set_property IOSTANDARD LVCMOS33 [get_ports {light2[6]}]
21 set_property IOSTANDARD LVCMOS33 [get_ports {light2[5]}]
22 set_property IOSTANDARD LVCMOS33 [get_ports {light2[4]}]
23 set_property IOSTANDARD LVCMOS33 [get_ports {light2[3]}]
24 set_property IOSTANDARD LVCMOS33 [get_ports {light2[2]}]
25 set_property IOSTANDARD LVCMOS33 [get_ports {light2[1]}]
26 set_property IOSTANDARD LVCMOS33 [get_ports {light2[0]}]

27
28 # 给上半片4个数码管的段选输出信号分配接口
29 set_property PACKAGE_PIN B4 [get_ports {light2[6]}]
30 set_property PACKAGE_PIN A4 [get_ports {light2[5]}]
31 set_property PACKAGE_PIN A3 [get_ports {light2[4]}]
32 set_property PACKAGE_PIN B1 [get_ports {light2[3]}]
33 set_property PACKAGE_PIN A1 [get_ports {light2[2]}]
34 set_property PACKAGE_PIN B3 [get_ports {light2[1]}]
35 set_property PACKAGE_PIN B2 [get_ports {light2[0]}]

```

```

1 # 给下半片4个数码管的位选输出信号分配电平
2 set_property IOSTANDARD LVCMOS33 [get_ports {an[3]}]
3 set_property IOSTANDARD LVCMOS33 [get_ports {an[2]}]
4 set_property IOSTANDARD LVCMOS33 [get_ports {an[1]}]
5 set_property IOSTANDARD LVCMOS33 [get_ports {an[0]}]

6
7 # 给下半片4个数码管的位选输出信号分配接口
8 set_property PACKAGE_PIN G1 [get_ports {an[3]}]
9 set_property PACKAGE_PIN F1 [get_ports {an[2]}]
10 set_property PACKAGE_PIN E1 [get_ports {an[1]}]

```

```
11 set_property PACKAGE_PIN G6 [get_ports {an[0]}]
12
13 # 给上半片4个数码管的位选输出信号分配电平
14 set_property IOSTANDARD LVCMOS33 [get_ports {an2[3]}]
15 set_property IOSTANDARD LVCMOS33 [get_ports {an2[2]}]
16 set_property IOSTANDARD LVCMOS33 [get_ports {an2[1]}]
17 set_property IOSTANDARD LVCMOS33 [get_ports {an2[0]}]
18
19 # 给上半片4个数码管的位选输出信号分配接口
20 set_property PACKAGE_PIN G2 [get_ports {an2[3]}]
21 set_property PACKAGE_PIN C2 [get_ports {an2[2]}]
22 set_property PACKAGE_PIN C1 [get_ports {an2[1]}]
23 set_property PACKAGE_PIN H1 [get_ports {an2[0]}]
24
25 # 给D_mem的输入地址信号分配电平
26 set_property IOSTANDARD LVCMOS33 [get_ports {address[7]}]
27 set_property IOSTANDARD LVCMOS33 [get_ports {address[6]}]
28 set_property IOSTANDARD LVCMOS33 [get_ports {address[5]}]
29 set_property IOSTANDARD LVCMOS33 [get_ports {address[4]}]
30 set_property IOSTANDARD LVCMOS33 [get_ports {address[3]}]
31 set_property IOSTANDARD LVCMOS33 [get_ports {address[2]}]
32 set_property IOSTANDARD LVCMOS33 [get_ports {address[1]}]
33 set_property IOSTANDARD LVCMOS33 [get_ports {address[0]}]
```

```
1 # 给D_mem的输入地址信号分配接口
2 set_property PACKAGE_PIN P5 [get_ports {adress[7]}]
3 set_property PACKAGE_PIN P4 [get_ports {adress[6]}]
4 set_property PACKAGE_PIN P3 [get_ports {adress[5]}]
5 set_property PACKAGE_PIN P2 [get_ports {adress[4]}]
6 set_property PACKAGE_PIN R2 [get_ports {adress[3]}]
7 set_property PACKAGE_PIN M4 [get_ports {adress[2]}]
8 set_property PACKAGE_PIN N4 [get_ports {adress[1]}]
9 set_property PACKAGE_PIN R1 [get_ports {adress[0]}]
10
11 # 给全局时钟输入信号分配分配电平和接口
12 set_property IOSTANDARD LVCMOS33 [get_ports clk]
13 set_property PACKAGE_PIN P17 [get_ports clk]
14
15 # 给enable输入信号分配分配电平和接口
16 set_property IOSTANDARD LVCMOS33 [get_ports enable]
17 set_property PACKAGE_PIN T5 [get_ports enable]
18
```

```

19 # 给reset输入信号分配分配电平和接口
20 set_property IOSTANDARD LVCMOS33 [get_ports reset]
21 set_property PACKAGE_PIN T3 [get_ports reset]
22
23 # 给read输入信号分配分配电平和接口
24 set_property IOSTANDARD LVCMOS33 [get_ports read]
25 set_property PACKAGE_PIN R11 [get_ports read]

```

```

1 # 给reset输入信号分配配电平和接口
2 set_property IOSTANDARD LVCMOS33 [get_ports clkreset]
3 set_property PACKAGE_PIN R3 [get_ports clkreset]
4
5 # 给start输入信号分配配电平和接口
6 set_property IOSTANDARD LVCMOS33 [get_ports start]
7 set_property PACKAGE_PIN R17 [get_ports start]
8
9 # 给出综合需要用到的时钟信号
10 create_clock -period 11.000 -name clk -waveform {0.000 5.500} [get_ports clk]
11 set_input_delay -clock [get_clocks clk] -min -add_delay 0.000 [get_ports {adress[*]}]
12 set_input_delay -clock [get_clocks clk] -max -add_delay 0.000 [get_ports {adress[*]}]
13 set_input_delay -clock [get_clocks clk] -min -add_delay 0.000 [get_ports clkreset]
14 set_input_delay -clock [get_clocks clk] -max -add_delay 0.000 [get_ports clkreset]
15 set_input_delay -clock [get_clocks clk] -min -add_delay 0.000 [get_ports read]
16 set_input_delay -clock [get_clocks clk] -max -add_delay 0.000 [get_ports read]
17 set_output_delay -clock [get_clocks clk] -min -add_delay 0.000 [get_ports {an[*]}]
18 set_output_delay -clock [get_clocks clk] -max -add_delay 0.000 [get_ports {an[*]}]
19 set_output_delay -clock [get_clocks clk] -min -add_delay 0.000 [get_ports {an2[*]}]
20 set_output_delay -clock [get_clocks clk] -max -add_delay 0.000 [get_ports {an2[*]}]
21 set_output_delay -clock [get_clocks clk] -min -add_delay 0.000 [get_ports {light[*]}]
22 set_output_delay -clock [get_clocks clk] -max -add_delay 0.000 [get_ports {light[*]}]
23 set_output_delay -clock [get_clocks clk] -min -add_delay 0.000 [get_ports {light2[*]}]
24 set_output_delay -clock [get_clocks clk] -max -add_delay 0.000 [get_ports {light2[*]}]

```

附录 C 十进制显示译码器代码

```

1 module decoder10(
2     input clock,
3     input reset,
4     input [15:0] y,      // 接dmen四位16进制输入
5     output reg [6:0] select_segment,
6     output reg [3:0] select_bit,

```

```
7     output [3:0] BCDout // 输出BCD码最高位
8 );
9
10    wire [1:0] s;
11    reg [3:0] digit;
12    reg [20:0] clkdiv;
13    wire [15:0] BCD;
14
15    assign s = clkdiv[20:19];
16    assign BCDout[3:0] = (y/10000)%10;      // D_mem码万位数
17    assign BCD[15:12] = (y/1000)%10;       // D_mem码千位数
18    assign BCD[11:8] = (y/100)%10;         // D_mem码百位数
19    assign BCD[7:4] = (y/10)%10;           // D_mem码十位数
20    assign BCD[3:0] = y%10;                // D_mem码个位数
21
22    always @(posedge clock) begin
23        case(s)
24            0: digit = BCD[3:0];
25            1: digit = BCD[7:4];
26            2: digit = BCD[11:8];
27            3: digit = BCD[15:12];
28            default: digit = BCD[3:0];
29        endcase
30    end
31
32    always @(posedge clock) begin
33        case(digit) // 十六进制输出
34            0: select_segment = 7'b1111_110; // 0
35            1: select_segment = 7'b0110_000; // 1
36            2: select_segment = 7'b1101_101; // 2
37            3: select_segment = 7'b1111_001; // 3
38            4: select_segment = 7'b0110_011; // 4
39            5: select_segment = 7'b1011_011; // 5
40            6: select_segment = 7'b1011_111; // 6
41            7: select_segment = 7'b1110_000; // 7
42            8: select_segment = 7'b1111_111; // 8
43            9: select_segment = 7'b1111_011; // 9
44        endcase
45    end
46
47    always @(posedge clock) begin
48        case(clkdiv[1:0])
```

```
49      0: select_bit = 4'b0001; // 消除显影
50      1: select_bit = 4'b0010; // 指定s位位选
51      2: select_bit = 4'b0100;
52      3: select_bit = 4'b1000;
53
54      endcase
55
56  end
57
58
59 always @ (posedge clock or posedge reset) begin
60   if (reset) begin
61     clkdiv <= 0;
62   end
63   else begin
64     clkdiv <= clkdiv + 1;
65   end
66 end
67
68 endmodule
```

```
1 module decoder10_2(
2   input  clock,
3   input  reset,
4   input [15:0] y,           // 接PC码输入
5   input [3:0] BCDin,        // 接dmen最高位BCD码
6   output reg [6:0] select_segment,
7   output reg [3:0] select_bit
8 );
9
10 wire [1:0] s;
11 reg [3:0] digit;
12 reg [20:0] clkdiv;
13 wire [11:0] BCD;
14
15 assign s = clkdiv[20:19];
16 assign BCD[11:8] = y/100;    // PC码百位数
17 assign BCD[7:4] = (y/10)%10; // PC码十位数
18 assign BCD[3:0] = y%10;     // PC码个位数
19
20 always @ (posedge clock) begin
21   case(s)
22     0: digit = BCDin[3:0];
23     1: digit = BCD[3:0];
24     2: digit = BCD[7:4];
```

```
25      3: digit = BCD[11:8];
26      default: digit = BCD[3:0];
27
28  endcase
29
30 always @ (posedge clock) begin
31   case(digit) // 十六进制输出
32     0: select_segment = 7'b1111110; // 0
33     1: select_segment = 7'b0110000; // 1
34     2: select_segment = 7'b1101101; // 2
35     3: select_segment = 7'b1111001; // 3
36     4: select_segment = 7'b0110011; // 4
37     5: select_segment = 7'b1011011; // 5
38     6: select_segment = 7'b1011111; // 6
39     7: select_segment = 7'b1110000; // 7
40     8: select_segment = 7'b1111111; // 8
41     9: select_segment = 7'b1111011; // 9
42     default: select_segment = 7'b1111110; // 0
43   endcase
44 end
45
46 endmodule
```