Lab 5 流水线 CPU 设计

姓名:傅申 学号: PB20000051 实验日期: 2022-5-11

1 实验题目

流水线 CPU 设计

2 实验目的

- 理解流水线CPU的结构和工作原理
- 掌握流水线CPU的设计和调试方法，特别是流水线中数据相关和控制相关的处理
- 熟练掌握数据通路和控制器的设计和描述方法

3 实验平台

- Xilinx Vivado v2019.1
- Microsoft Visual Studio Code
- FPGAOL

4 实验过程

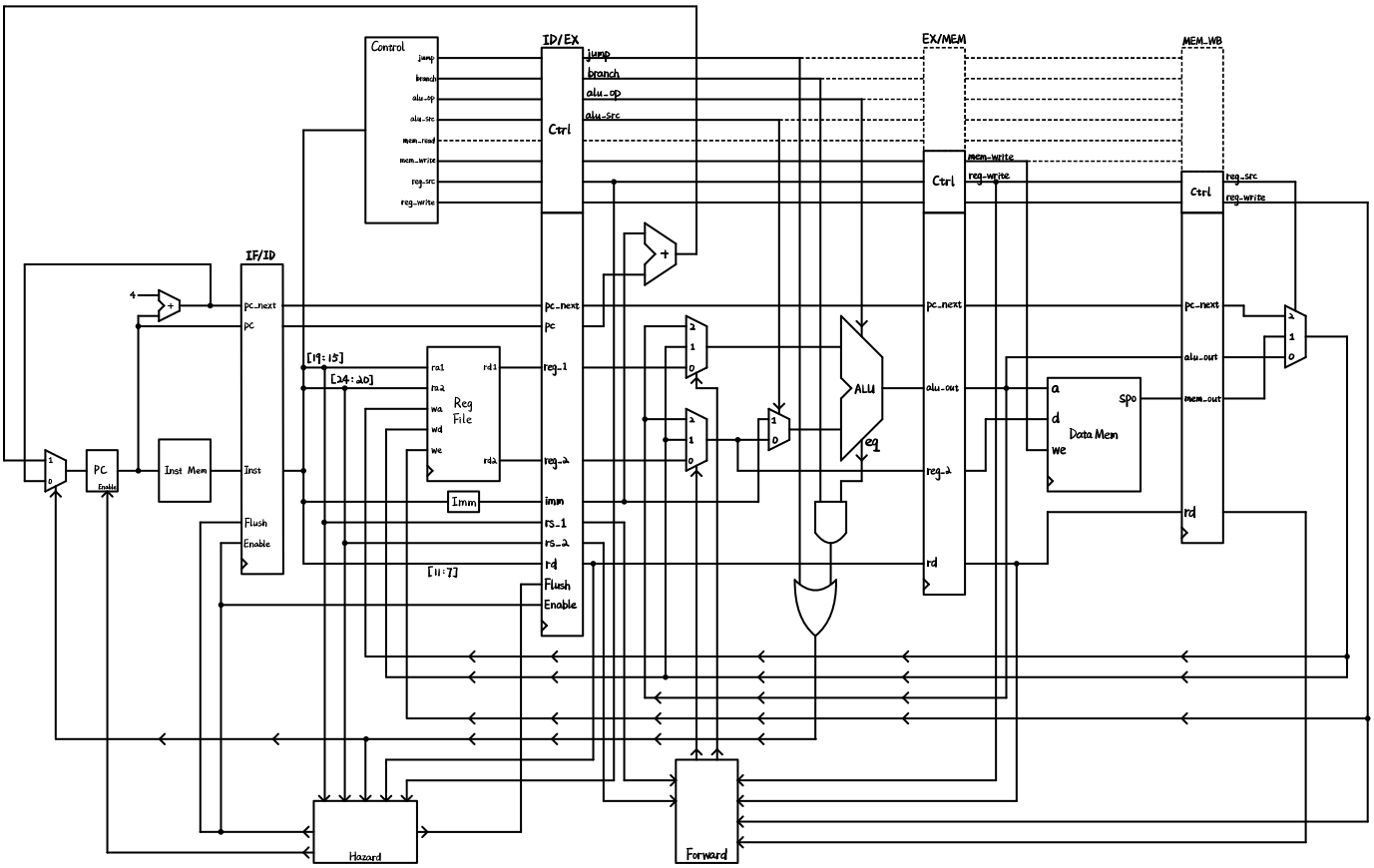
本次实验实现了一个五级流水线 RISC-V CPU, 并且将其与提供的 PDU 连接以实现 I/O 操作. CPU 可执行的指令有: `add`, `addi`, `lw`, `sw`, `beq`, `jal`.

整个 Vivado 项目的层次结构如下:

```
main (Main.v)
├── PDU:  pdu  (PDU.v)
└── CPU:  cpu  (CPU.v)
      ├── Inst_Mem:  inst_mem  (inst_mem.xci)
      ├── Data_Mem:  data_mem  (data_mem.xci)
      ├── Control:   control   (Ctrl.v)
      ├── Reg_File:  regfile   (Reg_File.v)
      ├── Imm_Gen:   imm_gen   (Imm_Gen.v)
      ├── ALU:       alu       (ALU.v)
      ├── Forward:   forward   (forward.v)
      ├── Hazard:    hazard    (hazard.v)
      ├── IF_ID:     if_id     (IF_ID.v)
      ├── ID_EX:     id_ex     (ID_EX.v)
      ├── EX_MEM:    ex_mem    (EX_MEM.v)
      └── MEM_WB:    mem_wb    (MEM_WB.v)
```

4.1 CPU 数据通路

不含 I/O 的数据通路如下, 其中控制信号被集成在了 32 位的 `ctrl` 线中, 所以图中的虚线部分为在之后的流水段没有用的控制信号部分, 但是由于其在控制线中, 所以仍会被逐级传递.



4.2 Control 模块

Control 模块输入指令的低 7 位 (opcode 段), 输出一系列控制信号如下, 其中 `mem_read` 信号仅集成于 `ctrl` 线中输出, 并不参与流水线的控制.

输出信号	位数	含义	在 ctrl 线中的位置
jump	1	是否跳转指令	ctrl[9]
branch	1	是否分支指令	ctrl[8]
mem_read	1	指令是否需要读取内存	ctrl[13]
mem_write	1	指令是否需要写入内存	ctrl[12]
alu_op	1	ALU 操作码 (是否需要 ALU 输出)	ctrl[0]
alu_src	1	ALU 第二操作数选择信号	ctrl[4]
reg_src	2	寄存器写回值选择信号	ctrl[17:16]
reg_write	1	指令是否需要写回寄存器	ctrl[18]

具体的 Verilog 代码如下

```
module control (
    input   [6:0] inst,
    output        jump,
    output        branch,
    output        mem_read,
    output        mem_write,
    output        alu_op,
    output        alu_src,
    output  [1:0] reg_src,
    output        reg_write
);
    wire add;
    wire addi;
    wire lw;
    wire sw;
    wire beq;
    wire jal;
    assign add  = (inst == 7'b0110011);
    assign addi = (inst == 7'b0010011);
    assign lw   = (inst == 7'b0000011);
    assign sw   = (inst == 7'b0100011);
    assign beq  = (inst == 7'b1100011);
    assign jal  = (inst == 7'b1101111);

    assign jump      = jal;
    assign branch    = beq;
    assign mem_read  = lw;
    assign mem_write = sw;
    assign alu_op    = add | addi | lw | sw | beq;
    assign alu_src   = addi | lw | sw;
    assign reg_src   = {jal, lw};
    assign reg_write = add | addi | lw;
endmodule
```

4.3 寄存器堆模块

寄存器堆为 32 × 32 位的写优先寄存器堆, 且 0 号寄存器始终为 0. 具体的 Verilog 代码如下

```
module regfile (
    input             clk,
    input             we,
    input      [4:0]  ra1,
    input      [4:0]  ra2,
    input      [4:0]  ra3,
    input      [4:0]  wa,
    input      [31:0] wd,
    output reg [31:0] rd1,
    output reg [31:0] rd2,
    output reg [31:0] rd3
);
    reg [31:0] rf [31:0];

    always @(posedge clk) begin
        if (we) rf[wa] <= wd;
    end
```

```
    always @(*) begin
        if (ra1 == 5'h0)    rd1 = 32'h0;
        else if (ra1 == wa) rd1 = wd;
        else                rd1 = rf[ra1];
    end

    always @(*) begin
        if (ra2 == 5'h0)    rd2 = 32'h0;
        else if (ra2 == wa) rd2 = wd;
        else                rd2 = rf[ra2];
    end

    always @(*) begin
        if (ra3 == 5'h0)    rd3 = 32'h0;
        else if (ra3 == wa) rd3 = wd;
        else                rd3 = rf[ra3];
    end
endmodule
```

4.4 立即数生成模块与 ALU 模块

立即数生成模块输入指令的全 32 位, 根据指令的 opcode 段输出相应的立即数, 具体的 Verilog 代码如下

```
module imm_gen(
    input      [31:0] inst,
    output reg [31:0] imm
);
    always @(*) begin
        case (inst[6:0])
            7'b0000011,
            7'b0010011: // lw, addi
                imm = {{21{inst[31]}}, inst[30:20]};
            7'b0100011: // sw
                imm = {{21{inst[31]}}, inst[30:25], inst[11:7]};
            7'b1100011: // beq
                imm = {{20{inst[31]}}, inst[7], inst[30:25], inst[11:8], 1'b0};
            7'b1101111: // jal
                imm = {{12{inst[31]}}, inst[19:12], inst[20], inst[30:21], 1'b0};
            default:
                imm = 31'h0;
        endcase
    end
endmodule
```

而 ALU 只需要进行加法操作和比较操作, **alu_op** 控制其是否输出, 输出信号 **out** 即为两输入值相加, **eq** 位两输入值是否相等, 具体的 Verilog 代码如下

```
module alu(
    input  [31:0] in_1,
    input  [31:0] in_2,
    input         op,
    output [31:0] out,
    output        eq
```

```
);
    assign out = (op == 1'b0) ? 32'h0 : in_1 + in_2;
    assign eq  = (op == 1'b0) ? 1'b0  : (in_1 == in_2);
endmodule
```

4.5 Forward 模块

前一条指令的目标寄存器是当前 EX 阶段指令的源寄存器时, 需要将数据前递给等待该数据的单元, 对应的逻辑如下 forward 模块的 Verilog 代码, 即

- 当 MEM 阶段的目标寄存器为当前 EX 阶段的源寄存器 (不为 `x0`) 时, 发生 EX 冒险, 需要将上一阶段 ALU 的输出前递给 EX 阶段, 对应的 `forward` 信号值为 `2'b10`.
- 当 WB 阶段的目标寄存器为当前 EX 阶段的源寄存器 (不为 `x0`) 时, 发生 MEM 冒险, 需要将寄存器写回值前递给 MEM 阶段, 对应的 `forward` 信号值为 `2'b01`.
- 如果两个冒险都发生, 则处理 EX 冒险, 即 `forward` 信号值为 `2'b10`.
- 如果不发生数据冒险, 则 `forward` 信号值为 `2'b00`, 不需要前递

```
module forward(
    input      [4:0]  IDEX_rs_1,
    input      [4:0]  IDEX_rs_2,
    input             EXMEM_reg_write,
    input      [4:0]  EXMEM_rd,
    input             MEMWB_reg_write,
    input      [4:0]  MEMWB_rd,
    output reg [1:0]  forward_1,
    output reg [1:0]  forward_2
);
    always @(*) begin
        if (EXMEM_reg_write & (EXMEM_rd != 5'h0) & (EXMEM_rd == IDEX_rs_1))
            forward_1 = 2'b10;
        else if (MEMWB_reg_write & (MEMWB_rd != 5'h0) & (MEMWB_rd == IDEX_rs_1))
            forward_1 = 2'b01;
        else
            forward_1 = 2'b00;
    end
    always @(*) begin
        if (EXMEM_reg_write & (EXMEM_rd != 5'h0) & (EXMEM_rd == IDEX_rs_2))
            forward_2 = 2'b10;
        else if (MEMWB_reg_write & (MEMWB_rd != 5'h0) & (MEMWB_rd == IDEX_rs_2))
            forward_2 = 2'b01;
        else
            forward_2 = 2'b00;
    end
endmodule
```

在 CPU 内对应的选择器部分如下

```
always @(*) begin
    case (forward_1)
        2'b00:   alu_in_1 = IDEX_reg_1;
        2'b01:   alu_in_1 = WB_data;
        2'b10:   alu_in_1 = EXMEM_alu_out;
        default: alu_in_1 = 32'h0;
```

```
    endcase
end
always @(*) begin
    case (forward_2)
        2'b00:   EX_reg_2 = IDEX_reg_2;
        2'b01:   EX_reg_2 = WB_data;
        2'b10:   EX_reg_2 = EXMEM_alu_out;
        default: EX_reg_2 = 32'h0;
    endcase
end
assign alu_in_2 = IDEX_ctrl[4] ? IDEX_imm : EX_reg_2;
```

4.6 Hazard 模块

当 EX 阶段指令需要从内存中读取数据写回寄存器, 且写回的寄存器为当前 ID 阶段指令的源寄存器时, 发生 load-use 数据冒险, 该冒险无法被前递解决, 这时需要在流水线中插入停顿, 即暂停 PC, IF/ID 寄存器的更新并冲刷 ID/EX 寄存器.
当分支预测失败时, 发生控制冒险, 这时需要冲刷流水线, 即丢弃 IF/ID, ID/EX 寄存器中的值, PC 将变为分支的地址. 在本次实验中, 由于采用假设分支不发生的策略, 所以冲刷流水线的条件为 `pc_src = jump | (branch & eq)` 有效.
对应的 Verilog 代码如下

```
module hazard(
    input [4:0] rs_1,
    input [4:0] rs_2,
    input       pc_src,
    input [4:0] IDEX_rd,
    input [1:0] IDEX_reg_src,
    output reg  enable,
    output      IFID_flush,
    output      IDEX_flush
);
    always @(*) begin
        if ((IDEX_reg_src == 2'b01) &
            (IDEX_rd != 5'h0) &
            (IDEX_rd == rs_1 | IDEX_rd == rs_2)
        )
            enable = 1'b0;
        else
            enable = 1'b1;
    end
    assign IFID_flush = pc_src;
    assign IDEX_flush = ~enable | pc_src;
endmodule
```

在 CPU 内, `enable` 作为 PC, IF/ID, ID/EX 寄存器的有效信号, `flush` 作为对应流水段寄存器的同步清除信号.

4.7 流水段寄存器

各个流水段寄存器拥有的信号如下

信号	IF/ID	ID/EX	EX/MEM	MEM/WB
clk	√	√	√	√
rst	√	√	√	√
en	√	√	×	×
clr	√	√	×	×
pc	√	√	×	×
pc_next	√	√	√	√
inst	√	×	×	×
reg_1	×	√	×	×
reg_2	×	√	√	×
imm	×	√	×	×
rs_1	×	√	×	×
rs_2	×	√	×	×
rd	×	√	√	√
ctrl	×	√	√	√
alu_out	×	×	√	√
mem_out	×	×	×	√

对于拥有 **en** 和 **clr** 的寄存器, 以 IF/ID 段为例, Verilog 代码如下

```
module if_id(
    input             clk,
    input             rst,
    input             en,
    input             clr,
    input      [31:0] pc,
    input      [31:0] pc_next,
    input      [31:0] inst,
    output reg [31:0] IFID_pc,
    output reg [31:0] IFID_pc_next,
    output reg [31:0] IFID_inst
);
    always @(posedge clk or posedge rst) begin
        if (rst | clr) begin
            IFID_pc      <= 32'h0;
            IFID_pc_next <= 32'h0;
            IFID_inst    <= 32'h0;
        end
        else if (en) begin
            IFID_pc      <= pc;
            IFID_pc_next <= pc_next;
            IFID_inst    <= inst;
        end
    end
endmodule
```

对于没有 **en** 和 **clr** 的寄存器, 以 EX/MEM 段为例, Verilog 代码如下

```
module ex_mem(
    input              clk,
    input              rst,
    input      [31:0] alu_out,
    input      [31:0] IDEX_pc_next,
    input      [31:0] EX_reg_2,
    input      [4:0]  IDEX_rd,
    input      [31:0] IDEX_ctrl,
    output reg [31:0] EXMEM_alu_out,
    output reg [31:0] EXMEM_pc_next,
    output reg [31:0] EXMEM_reg_2,
    output reg [4:0]  EXMEM_rd,
    output reg [31:0] EXMEM_ctrl
);
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            EXMEM_alu_out <= 32'h0;
            EXMEM_pc_next <= 32'h0;
            EXMEM_reg_2   <= 32'h0;
            EXMEM_rd      <= 5'h0;
            EXMEM_ctrl    <= 32'h0;
        end
        else begin
            EXMEM_alu_out <= alu_out;
            EXMEM_pc_next <= IDEX_pc_next;
            EXMEM_reg_2   <= EX_reg_2;
            EXMEM_rd      <= IDEX_rd;
            EXMEM_ctrl    <= IDEX_ctrl;
        end
    end
endmodule
```

4.8 CPU 中的 MUX 与寄存器

对于 CPU 中的部分 MUX 和寄存器, 项目中没有将其单独写成模块, 它们的 Verilog 代码如下:

- PC 寄存器

```
assign pc_next = pc + 32'h4;
assign pc_in   = (pc_src == 1'b1) ? pc_add : pc_next;
assign pc_src  = IDEX_ctrl[9] | (IDEX_ctrl[8] & eq);
assign pc_add  = IDEX_pc + IDEX_imm;
always @(posedge clk or posedge rst) begin
    if (rst)         pc <= 32'h0;
    else if (enable) pc <= pc_in;
end
```

- ALU 前的 MUX

```
always @(*) begin
    case (forward_1)
        2'b00:   alu_in_1 = IDEX_reg_1;
        2'b01:   alu_in_1 = WB_data;
        2'b10:   alu_in_1 = EXMEM_alu_out;
        default: alu_in_1 = 32'h0;
```

```
    endcase
end
always @(*) begin
    case (forward_2)
        2'b00:   EX_reg_2 = IDEX_reg_2;
        2'b01:   EX_reg_2 = WB_data;
        2'b10:   EX_reg_2 = EXMEM_alu_out;
        default: EX_reg_2 = 32'h0;
    endcase
end
assign alu_in_2 = IDEX_ctrl[4] ? IDEX_imm : EX_reg_2;
```

- I/O 内存映射

```
assign io_we   = EXMEM_ctrl[12] & EXMEM_alu_out[10];
assign io_addr = EXMEM_alu_out[7:0];
assign io_dout = EXMEM_reg_2;
assign mem_we  = EXMEM_ctrl[12] & (~EXMEM_alu_out[10]);
assign mem_out = (EXMEM_alu_out[10]) ? io_din : mem_data;
```

- WB 阶段 MUX

```
always @(*) begin
    case (MEMWB_ctrl[17:16])
        2'b00:   WB_data = MEMWB_alu_out;
        2'b01:   WB_data = MEMWB_mem_out;
        2'b10:   WB_data = MEMWB_pc_next;
        default: WB_data = 32'h0;
    endcase
end
```

具体的 CPU 代码与顶层模块 Main 代码见附录: CPU, Main

5 实验结果

5.1 CPU 仿真

对于下面的 RISC-V 汇编程序,

```
start:
sw   x0, 0x408(x0) #out1=0

#test data hazards
addi x1, x0, 1     #x1=1
addi x2, x1, 1     #x2=2
add  x3, x1, x2    #x3=3
add  x4, x1, x3    #x4=4
add  x5, x1, x4    #x5=5
sw   x5, 0x408(x0) #out1=5

add  x6, x1, x2    #x6=3
add  x6, x6, x3    #x6=6
add  x6, x6, x4    #x6=10
```

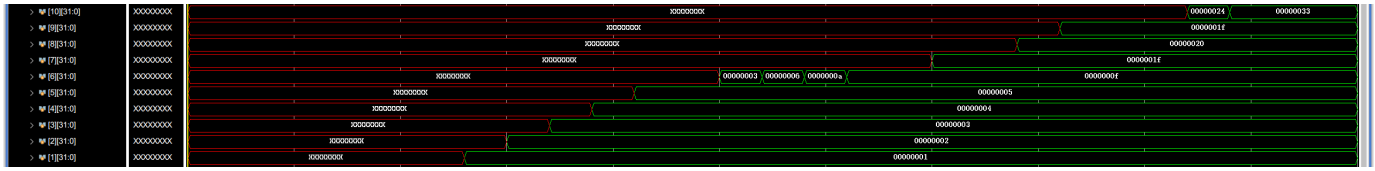
```
add  x6, x6, x5    #x6=15
sw   x6, 0x408(x0) #out1=15

#test load-use hazard
lw   x7, 0x40C(x0) #x7=in
addi x8, x7, 1     #x8=in+1
addi x9, x8, -1    #x9=in
sw   x9, 0x408(x0) #out1=in

#test control hazard
beq  x9, x0, start  #if (in==0) start
add  x10, x9, x5
add  x10, x10, x6
sw   x10, 0x408(x0) #out1=in+20
stop:
jal  x0, stop

#do not execute
add  x11, x9, x10
add  x12, x10, x11
add  x13, x11, x12
```

设置输入值 (地址为 `0x408`) 为 `0x1F`, 仿真结果如下图

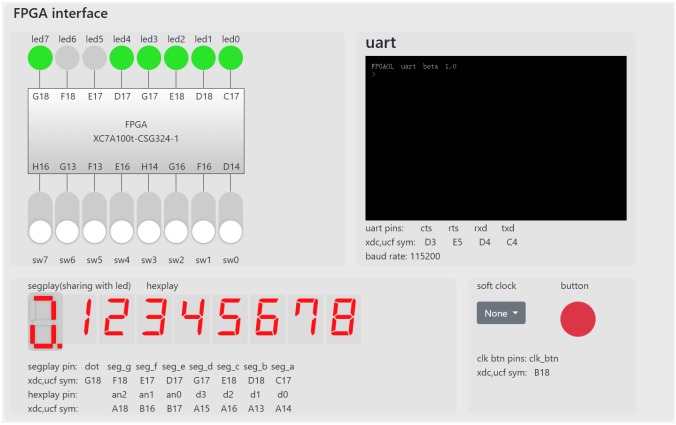


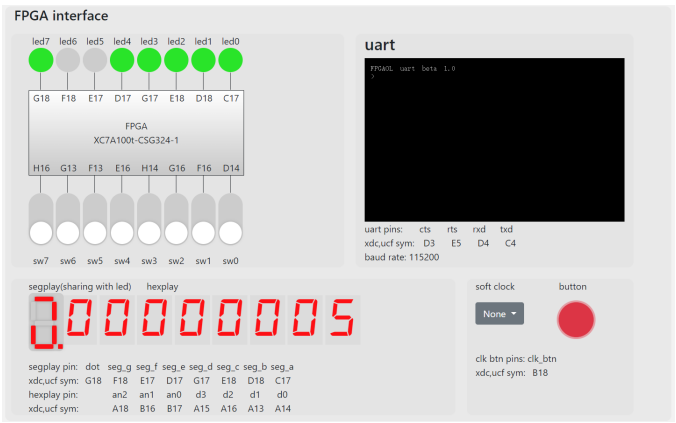
观察第三张图片 (寄存器堆) 的信号可知 CPU 运行正常.

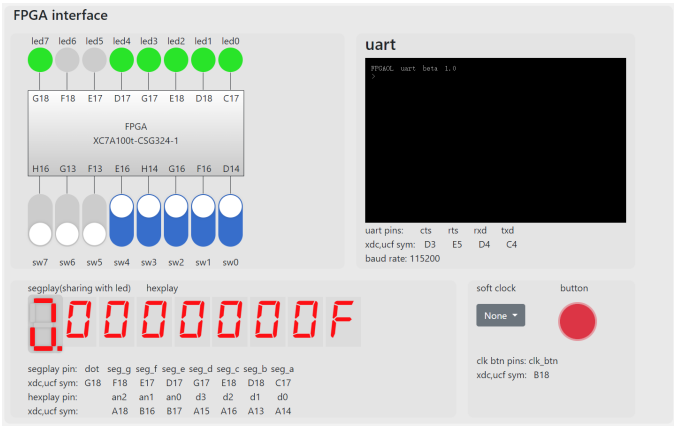
5.2 上板

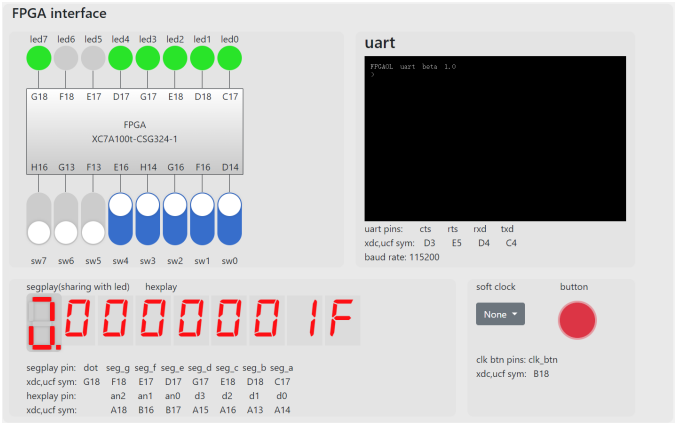
5.2.1 hazards_test.s

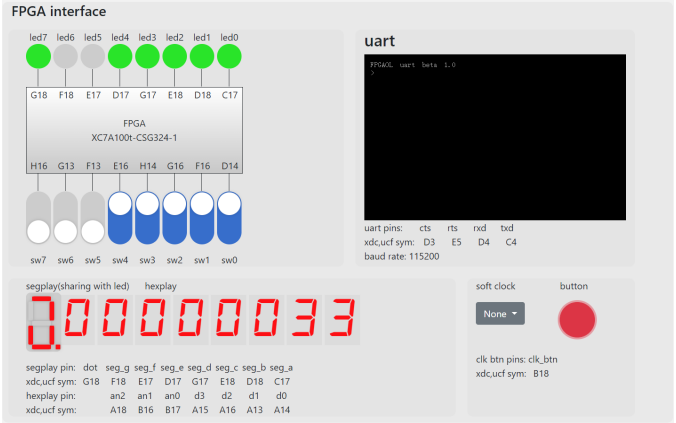
同样使用上面的汇编代码进行上板测试, 在 FPGAOL 上的一系列输出如下











5.2.2 fib_test.s

对 `fib_test.s` 程序进行上板测试, 在 FPGAOL 上的一系列输出如下

6 心得体会

本次实验难度较大, 但是能帮助深入理解流水线的细节.

7 附录: Verilog 代码

7.1 CPU

```
module cpu(
    input clk,
    input rst,

    // IO_BUS
    input      [31:0] io_din,     // 来自 sw 的输入数据
```

```
    output      [7:0]  io_addr,   // led 和 seg 的地址
    output      [31:0] io_dout,   // 输出 led 和 seg 的数据
    output             io_we,     // 输出 led 和 seg 数据时的使能信号

    // Debug_BUS
    input       [7:0]  dbg_addr,  // 存储器 (MEM) 或寄存器堆 (RF) 的调试读口地址
    output      [31:0] rf_data,   // 从RF读取的数据
    output      [31:0] m_data,    // 从MEM读取的数据

    // PC/IF/ID 流水段寄存器
    output reg [31:0] pc,
    output      [31:0] pc_in,
    output      [31:0] IFID_pc,
    output      [31:0] IFID_inst,

    // ID/EX 流水段寄存器
    output      [31:0] IDEX_pc,
    output      [31:0] IDEX_reg_1,
    output      [31:0] IDEX_reg_2,
    output      [31:0] IDEX_imm,
    output      [4:0]  IDEX_rd,
    output      [31:0] IDEX_ctrl,

    // EX/MEM 流水段寄存器
    output      [31:0] EXMEM_alu_out,
    output      [31:0] EXMEM_reg_2,
    output      [4:0]  EXMEM_rd,
    output      [31:0] EXMEM_ctrl,

    // MEM/WB 流水段寄存器
    output      [31:0] MEMWB_alu_out,
    output      [31:0] MEMWB_mem_out,
    output      [4:0]  MEMWB_rd,
    output      [31:0] MEMWB_ctrl
);
    // Control Signal
    wire        enable;
    // IF
    wire        pc_src;
    wire [31:0] pc_next;
    wire [31:0] pc_add;
    wire [31:0] inst;
    // IF/ID
    wire        IFID_flush;
    wire [31:0] IFID_pc_next;
    // ID
    wire [31:0] ctrl;
    wire        jump;       // ctrl[9]
    wire        branch;     // ctrl[8]
    wire        mem_read;   // ctrl[13]
    wire        mem_write;  // ctrl[12]
    wire        alu_op;     // ctrl[0]
    wire        alu_src;    // ctrl[4]
    wire [1:0]  reg_src;    // ctrl[17:16]
    wire        reg_write;  // ctrl[18]
    wire [31:0] reg_1;
    wire [31:0] reg_2;
    wire [4:0]  rs_1;
    wire [4:0]  rs_2;
    wire [4:0]  rd;
```

```
wire [31:0] imm;
// ID/EX
wire        IDEX_flush;
wire [31:0] IDEX_pc_next;
wire [4:0]  IDEX_rs_1;
wire [4:0]  IDEX_rs_2;
// EX
wire [1:0]  forward_1;
wire [1:0]  forward_2;
reg  [31:0] EX_reg_2;
reg  [31:0] alu_in_1;
wire [31:0] alu_in_2;
wire [31:0] alu_out;
wire        eq;
// EX/MEM
wire [31:0] EXMEM_pc_next;
// MEM
wire        mem_we;
wire [31:0] mem_out;
wire [31:0] mem_data;
// MEM/WB
wire [31:0] MEMWB_pc_next;
// WB
reg  [31:0] WB_data;
// IF: pc, inst_mem
assign pc_next = pc + 32'h4;
assign pc_in   = (pc_src == 1'b1) ? pc_add : pc_next;
always @(posedge clk or posedge rst) begin
    if (rst)         pc <= 32'h0;
    else if (enable) pc <= pc_in;
end
inst_mem Inst_Mem (
    .a   (pc[9:2]),
    .spo (inst)
);
// IF/ID Register
if_id IF_ID (
    .clk          (clk),
    .rst          (rst),
    .en           (enable),
    .clr          (IFID_flush),
    .pc           (pc),
    .pc_next      (pc_next),
    .inst         (inst),
    .IFID_pc      (IFID_pc),
    .IFID_pc_next (IFID_pc_next),
    .IFID_inst    (IFID_inst)
);
// ID: hazard, ctrl, regfile, imm_gen
assign ctrl = {~enable, ~enable, IFID_flush, IDEX_flush, 2'b00, forward_1,
               2'b00, forward_2, 1'b0, reg_write, reg_src, 2'b00, mem_read,
               mem_write, 2'b00, jump, branch, 2'b00, 1'b0, alu_src,
               3'b000, alu_op}; // bunch of fking wires
assign rs_1 = IFID_inst[19:15];
assign rs_2 = IFID_inst[24:20];
assign rd   = IFID_inst[11:7];

hazard Hazard (
    .rs_1         (rs_1),
    .rs_2         (rs_2),
```

```
    .pc_src       (pc_src),
    .IDEX_rd      (IDEX_rd),
    .IDEX_reg_src (IDEX_ctrl[17:16]),
    .enable       (enable),
    .IFID_flush   (IFID_flush),
    .IDEX_flush   (IDEX_flush)
);
control Control (
    .inst      (IFID_inst[6:0]),
    .jump      (jump),
    .branch    (branch),
    .mem_read  (mem_read),
    .mem_write (mem_write),
    .alu_op    (alu_op),
    .alu_src   (alu_src),
    .reg_src   (reg_src),
    .reg_write (reg_write)
);
regfile Reg_File (
    .clk (clk),
    .we  (MEMWB_ctrl[18]),
    .ra1 (rs_1),
    .ra2 (rs_2),
    .ra3 (dbg_addr[4:0]),
    .wa  (MEMWB_rd),
    .wd  (WB_data),
    .rd1 (reg_1),
    .rd2 (reg_2),
    .rd3 (rf_data)
);
imm_gen Imm_Gen (
    .inst (IFID_inst),
    .imm  (imm)
);
// ID/EX Register
id_ex ID_EX (
    .clk          (clk),
    .rst          (rst),
    .clr          (IDEX_flush),
    .IFID_pc      (IFID_pc),
    .IFID_pc_next (IFID_pc_next),
    .reg_1        (reg_1),
    .reg_2        (reg_2),
    .imm          (imm),
    .rs_1         (rs_1),
    .rs_2         (rs_2),
    .rd           (rd),
    .ctrl         (ctrl),
    .IDEX_pc      (IDEX_pc),
    .IDEX_pc_next (IDEX_pc_next),
    .IDEX_reg_1   (IDEX_reg_1),
    .IDEX_reg_2   (IDEX_reg_2),
    .IDEX_imm     (IDEX_imm),
    .IDEX_rs_1    (IDEX_rs_1),
    .IDEX_rs_2    (IDEX_rs_2),
    .IDEX_rd      (IDEX_rd),
    .IDEX_ctrl    (IDEX_ctrl)
);
// EX: forward, alu, pc_adder
assign pc_src  = IDEX_ctrl[9] | (IDEX_ctrl[8] & eq);
```

```
    assign pc_add  = IDEX_pc + IDEX_imm;
    always @(*) begin
        case (forward_1)
            2'b00:   alu_in_1 = IDEX_reg_1;
            2'b01:   alu_in_1 = WB_data;
            2'b10:   alu_in_1 = EXMEM_alu_out;
            default: alu_in_1 = 32'h0;
        endcase
    end
    always @(*) begin
        case (forward_2)
            2'b00:   EX_reg_2 = IDEX_reg_2;
            2'b01:   EX_reg_2 = WB_data;
            2'b10:   EX_reg_2 = EXMEM_alu_out;
            default: EX_reg_2 = 32'h0;
        endcase
    end
    assign alu_in_2 = IDEX_ctrl[4] ? IDEX_imm : EX_reg_2;
    forward Forward (
        .IDEX_rs_1       (IDEX_rs_1),
        .IDEX_rs_2       (IDEX_rs_2),
        .EXMEM_reg_write (EXMEM_ctrl[18]),
        .EXMEM_rd        (EXMEM_rd),
        .MEMWB_reg_write (MEMWB_ctrl[18]),
        .MEMWB_rd        (MEMWB_rd),
        .forward_1       (forward_1),
        .forward_2       (forward_2)
    );
    alu ALU (
        .in_1 (alu_in_1),
        .in_2 (alu_in_2),
        .op   (IDEX_ctrl[0]),
        .out  (alu_out),
        .eq   (eq)
    );
    // EX/MEM Register
    ex_mem EX_MEM (
        .clk           (clk),
        .rst           (rst),
        .alu_out       (alu_out),
        .IDEX_pc_next  (IDEX_pc_next),
        .EX_reg_2      (EX_reg_2),
        .IDEX_rd       (IDEX_rd),
        .IDEX_ctrl     (IDEX_ctrl),
        .EXMEM_alu_out (EXMEM_alu_out),
        .EXMEM_pc_next (EXMEM_pc_next),
        .EXMEM_reg_2   (EXMEM_reg_2),
        .EXMEM_rd      (EXMEM_rd),
        .EXMEM_ctrl    (EXMEM_ctrl)
    );
    // MEM: data_mem
    assign io_we   = EXMEM_ctrl[12] & EXMEM_alu_out[10];
    assign io_addr = EXMEM_alu_out[7:0];
    assign io_dout = EXMEM_reg_2;
    assign mem_we  = EXMEM_ctrl[12] & (~EXMEM_alu_out[10]);
    assign mem_out = (EXMEM_alu_out[10]) ? io_din : mem_data;
    data_mem Data_Mem (
        .clk  (clk),
        .we   (mem_we),
        .a    (EXMEM_alu_out[9:2]),
```

```
        .dpra (dbg_addr),
        .d    (EXMEM_reg_2),
        .spo  (mem_data),
        .dpo  (m_data)
    );
    // MEM/WB Register
    mem_wb MEM_WB (
        .clk           (clk),
        .rst           (rst),
        .mem_out       (mem_out),
        .EXMEM_pc_next (EXMEM_pc_next),
        .EXMEM_alu_out (EXMEM_alu_out),
        .EXMEM_rd      (EXMEM_rd),
        .EXMEM_ctrl    (EXMEM_ctrl),
        .MEMWB_mem_out (MEMWB_mem_out),
        .MEMWB_pc_next (MEMWB_pc_next),
        .MEMWB_alu_out (MEMWB_alu_out),
        .MEMWB_rd      (MEMWB_rd),
        .MEMWB_ctrl    (MEMWB_ctrl)
    );
    // WB
    always @(*) begin
        case (MEMWB_ctrl[17:16])
            2'b00:   WB_data = MEMWB_alu_out;
            2'b01:   WB_data = MEMWB_mem_out;
            2'b10:   WB_data = MEMWB_pc_next;
            default: WB_data = 32'h0;
        endcase
    end
endmodule
```

7.2 Main

```
module main(
    input        clk,    // clk_100MHz
    input        step,   // btn
    input        rst,    // sw[7]
    input        run,    // sw[6]
    input        valid,  // sw[5]
    input  [4:0] in,     // sw[4:0]
    output       ready,  // led[7]
    output [1:0] check,  // led[6:5]
    output [4:0] out_0,  // led[4:0]
    output [2:0] an,     // segplay_an
    output [3:0] seg     // segplay_data
);
    wire clk_cpu;

    // IO_BUS
    wire        io_we;
    wire [7:0]  io_addr;
    wire [31:0] io_din;
    wire [31:0] io_dout;

    // Debug_BUS
    wire [31:0] rf_data;
    wire [31:0] m_data;
```

```
    wire [7:0]  dbg_addr;

    // Pipeline Register
    wire [31:0] pc_in;
    wire [31:0] pc;
    wire [31:0] IFID_pc;
    wire [31:0] IFID_inst;
    wire [31:0] IDEX_pc;
    wire [31:0] IDEX_reg_1;
    wire [31:0] IDEX_reg_2;
    wire [31:0] IDEX_imm;
    wire [4:0]  IDEX_rd;
    wire [31:0] IDEX_ctrl;
    wire [31:0] EXMEM_alu_out;
    wire [31:0] EXMEM_reg_2;
    wire [4:0]  EXMEM_rd;
    wire [31:0] EXMEM_ctrl;
    wire [31:0] MEMWB_alu_out;
    wire [31:0] MEMWB_mem_out;
    wire [4:0]  MEMWB_rd;
    wire [31:0] MEMWB_ctrl;

    // Wire Mapping
    cpu CPU(
        .clk           (clk_cpu),
        .rst           (rst),
        .io_we         (io_we),
        .io_addr       (io_addr),
        .io_din        (io_din),
        .io_dout       (io_dout),
        .rf_data       (rf_data),
        .m_data        (m_data),
        .dbg_addr      (dbg_addr),
        .pc_in         (pc_in),
        .pc            (pc),
        .IFID_pc       (IFID_pc),
        .IFID_inst     (IFID_inst),
        .IDEX_pc       (IDEX_pc),
        .IDEX_reg_1    (IDEX_reg_1),
        .IDEX_reg_2    (IDEX_reg_2),
        .IDEX_imm      (IDEX_imm),
        .IDEX_rd       (IDEX_rd),
        .IDEX_ctrl     (IDEX_ctrl),
        .EXMEM_alu_out (EXMEM_alu_out),
        .EXMEM_reg_2   (EXMEM_reg_2),
        .EXMEM_rd      (EXMEM_rd),
        .EXMEM_ctrl    (EXMEM_ctrl),
        .MEMWB_alu_out (MEMWB_alu_out),
        .MEMWB_mem_out (MEMWB_mem_out),
        .MEMWB_rd      (MEMWB_rd),
        .MEMWB_ctrl    (MEMWB_ctrl)
    );

    pdu PDU(
        .clk       (clk),
        .rst       (rst),
        .run       (run),
        .step      (step),
        .clk_cpu   (clk_cpu),
        .valid     (valid),
```

```
        .in         (in),
        .ready      (ready),
        .check      (check),
        .out0       (out_0),
        .an         (an),
        .seg        (seg),
        .io_we      (io_we),
        .io_addr    (io_addr),
        .io_din     (io_din),
        .io_dout    (io_dout),
        .rf_data    (rf_data),
        .m_data     (m_data),
        .m_rf_addr  (dbg_addr),
        .pcin       (pc_in),
        .pc         (pc),
        .pcd        (IFID_pc),
        .pce        (IDEX_pc),
        .ir         (IFID_inst),
        .imm        (IDEX_imm),
        .mdr        (MEMWB_mem_out),
        .a          (IDEX_reg_1),
        .b          (IDEX_reg_2),
        .y          (EXMEM_alu_out),
        .bm         (EXMEM_reg_2),
        .yw         (MEMWB_alu_out),
        .rd         (IDEX_rd),
        .rdm        (MEMWB_rd),
        .rdw        (EXMEM_rd),
        .ctrl       (IDEX_ctrl),
        .ctrlm      (EXMEM_ctrl),
        .ctrlw      (MEMWB_ctrl)
    );
endmodule
```