计算机组成实验报告

实验 3：5 段流水线 CPU 设计

姓 名: 郭 嘉 宋
学 号: 517030910374
班 号: F1703015

上海交通大学
IEEE 试点班

2019 年 6 月 13 日

1 实验目的

1. 理解计算机指令流水线的协调工作原理，初步掌握流水线的设计和实现原理。
2. 深刻理解流水线寄存器在流水线实现中所起的重要作用。
3. 理解和掌握流水段的划分、设计原理及其实现方法原理。
4. 掌握运算器、寄存器堆、存储器、控制器在流水工作方式下，有别于单周期 CPU 的设计和实现方法。
5. 掌握流水方式下，通过 I/O 端口与外部设备进行信息交互的方法。

2 实验所用的仪器以及元器件

Intel/DE1-SOC 实验板 1 套
示波器 1 台
数字万用表 1 台

3 实验内容和任务

1. 采用 Verilog 在 quartus II 中实现基本的具有 20 条 MIPS 指令的 5 段流水 CPU 设计。
2. 利用实验提供的标准测试程序代码，完成仿真测试。
3. 采用 I/O 统一编址方式，即将输入输出的 I/O 地址空间，作为数据存取空间的一部分，实现 CPU 与外部设备的输入输出端口设计。实验中可采用高端地址。
4. 利用设计的 I/O 端口，通过 lw 指令，输入 DE2 实验板上的按键等输入设备信息。即将外部设备状态，读到 CPU 内部寄存器。
5. 利用设计的 I/O 端口，通过 sw 指令，输出对 DE2 实验板上的 LED 灯等输出设备的控制信号（或数据信息）。即将对外部设备的控制数据，从 CPU 内部的寄存器，写入到外部设备的相应控制寄存器（或可直接连接至外部设备的控制输入信号）。
6. 利用自己编写的程序代码，在自己设计的 CPU 上，实现对板载输入开关或按键的状态输入，并将判别或处理结果，利用板载 LED 灯或 7 段 LED 数码管显示出来。
7. 例如，将一路 4bit 二进制输入与另一路 4bit 二进制输入相加，利用两组分别 2 个 LED 数码管以 10 进制形式显示“被加数”和“加数”，另外一组 LED 数码管以 10 进制形式显示“和”等）。

8. 在实现 MIPS 基本 20 条指令的基础上，掌握新指令的扩展方法。
9. 在实验报告中，汇报自己的设计思想和方法；并以汇编语言的形式，提供以上两种指令集（MIPS 和 Y86）应用功能的程序设计代码，并提供程序主要流程图。

4 实验原理和结果展示

4.1 5 级流水线 CPU 设计原理

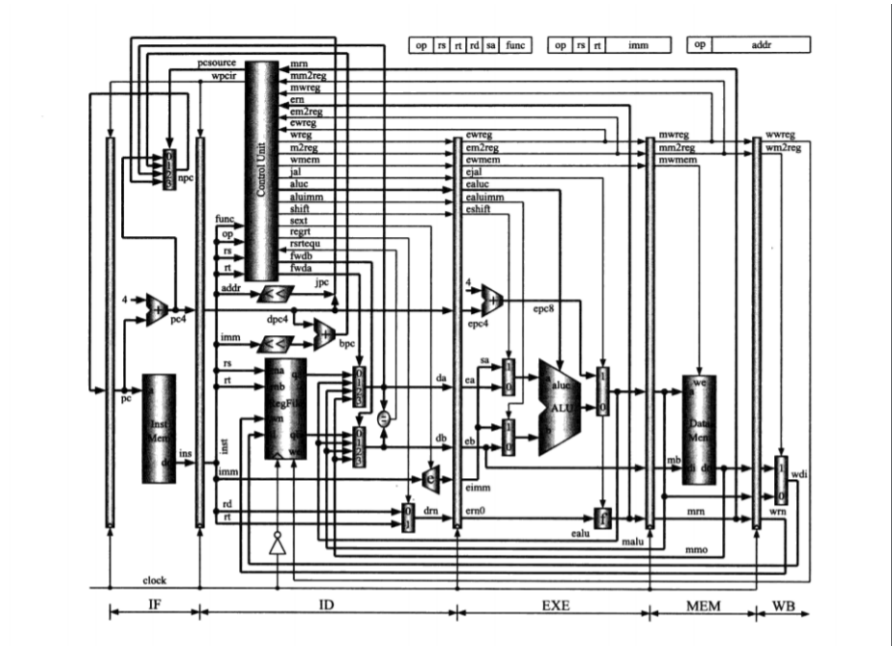

图 1: 5 级流水线 CPU 原理图

5 级流水线分为:
IF - Instruction fetch
ID - Instruction decode
EXE - Execution
MEM - Memory
WB - Write back
因此按照这 5 个模块进行设计，需要在每两端之间插入段寄存器用来保存信息。

4.2 顶层文件设计

```
module pipelined_computer(resetn,clock,mem_clock,in_port0,in_port1,pc,inst,
    ealu,malu,walu,hex0,hex1,hex2,hex3,hex4,hex5);
  input resetn,clock,mem_clock;
  output [31:0] pc,inst,ealu,malu,walu;

  input [3:0] in_port0, in_port1;
  output[6:0] hex0,hex1,hex2,hex3,hex4,hex5;
  wire [31:0] out_port0, out_port1, out_port2;

  //IF
  wire [31:0] bpc,jpc,npc,pc4,ins,inst;
  //ID
  wire [31:0] dpc4,da,db,dimm;
  //EX
  wire [31:0] epc4,ea,eb,eimm;
  //MEM
  wire [31:0] mb,mmo;
  //WB
  wire [31:0] wmo,wdi;
  //register number
  wire [4:0] drn,ern0,ern,mrn,wrn;
  //ALUC
  wire [3:0] daluc,ealuc;
  wire [1:0] pcsource;
  wire wpcir;
  //IF/ID
  wire dwreg,dm2reg,dwmem,daluimm,dshift,djal;
  //ID/EX
  wire ewreg,em2reg,ewmem,ealuimm,eshift,ejal;
  //EX/MEM
  wire mwreg,mm2reg,mwmem;
  //MEM/WB
  wire wwreg,wm2reg;
  pipepc prog_cnt(npc,wpcir,clock,resetn,pc);

  pipeif if_stage(pcsource,pc,bpc,da,jpc,npc,pc4,ins,mem_clock);

  pipeir inst_reg(pc4,ins,wpcir,clock,resetn,dpc4,inst);

  pipeid id_stage(mwreg,mrn,ern,ewreg,em2reg,mm2reg,
  dpc4,inst,wrn,wdi,ealu,malu,mmo,wwreg,clock,resetn,bpc,
  jpc,pcsource,wpcir,dwreg,dm2reg,dwmem,daluc,daluimm,
  da,db,dimm,drn,dshift,djal);

  pipedereg de_reg(dwreg,dm2reg,dwmem,daluc,daluimm,da,db,dimm,
  drn,dshift,djal,dpc4,clock,resetn,ewreg,em2reg,ewmem,ealuc,ealuimm,
  ea,eb,eimm,ern0,eshift,ejal,epc4);

```

```
48   pipeexe exe_stage(ealuc,ealuimm,ea,eb,eimm,eshift,ern0,epc4,ejal,ern,ealu)
         ;
49
50   pipeemreg em_reg(ewreg,em2reg,ewmem,ealu,eb,ern,clock,resetn,mwreg,
51   mm2reg,mwmem,malu,mb,mrn);
52
53   pipemem mem_stage(mwmem,malu,mb,wmo,wm2reg,in_port0,in_port1,
54   clock,mem_clock,resetn,mmo,out_port0,out_port1,out_port2);
55
56   pipemwreg mw_reg(mwreg,mm2reg,mmo,malu,mrn,clock,resetn,wwreg,
57   wm2reg,wmo,walu,wrn);
58
59   mux2x32 wb_stage(walu,wmo,wm2reg,wdi);
60
61   sevenseg trans0(out_port0[3:0],hex4);
62   sevenseg trans1(out_port0[7:4],hex5);
63   sevenseg trans2(out_port1[3:0],hex2);
64   sevenseg trans3(out_port1[7:4],hex3);
65   sevenseg trans4(out_port2[3:0],hex0);
66   sevenseg trans5(out_port2[7:4],hex1);
67 endmodule
```

需要注意的是，流水线的 PC 与单周期 PC 不同的是，流水线 PC 很有可能因为加入了 NOP 即气泡指令而导致 PC 维持原状不随时钟更新而更新，因此需要如代码所示增加 wpcir 信号 (write-pc-instruction-register)。wpcir 相当于使能端，用于控制 PC 是否更新。

之后对每一个模块进行连接。分别是：PC 模块 pipepc，IF 模块 pipeif，IR 寄存器 pipeir，ID 模块 pipeid，ID/EXE 寄存器 pipedereg，EXE/MEM 寄存器 pipeemreg，MEM 模块 pipemem，MEM/WB 寄存器 pipemwreg，WB 模块 mux2x32(WB 模块只需要一个复用器选择输入数据即可)。

4.3 3 类冒险的解决

4.3.1 结构冒险

FPGA 板上的寄存器允许同时读写，且实现过程采用哈佛结构，即指令 MEM 与数据 MEM 分离，因此不存在结构冒险。

4.3.2 数据冒险

1. 没有 lw 指令参与的数据冒险

分为两种情况：此次 EXE 级的操作数是上一条指令的结果，或此次 EXE 级的操作数是再之前一条指令的结果。

这时候需要对 fwda 和 fwdb 两条控制线进行约束。代码如下，分 4 种情况

进行控制。

```
always @ (ewreg, mwreg, ern, mrn, em2reg, mm2reg, rs, rt) begin
  fwda = 2'b00; // no hazard
  if (ewreg & (ern != 0) & (ern == rs) & ~mm2reg) begin
  fwda = 2'b01; // select exe_alu
  end
  else begin
    if (mwreg & ( mrn !=0 )&(mrn == rs) & ~mm2reg) begin
      fwda = 2'b10; //select mem_alu
    end
    else begin
      if (mwreg & (mrn!=0) &(mrn == rs) & mm2reg) begin
        fwda = 2'b11; //select mem_lw
      end
    end
  end
  fwdb = 2'b00; // no hazard
  if (ewreg & (ern != 0) & (ern == rt) & ~mm2reg) begin
  fwdb = 2'b01; // select exe_alu
  end
  else begin
    if (mwreg & ( mrn !=0 )&(mrn == rt) & ~mm2reg) begin
      fwdb = 2'b10; //select mem_alu
    end
    else begin
      if (mwreg & (mrn!=0) &(mrn == rt) & mm2reg) begin
        fwdb = 2'b11; //select mem_lw
      end
    end
  end
  end
```

2. 有 lw 指令参与的数据冒险

这个时候不可避免地需要暂停一次流水线，因此需要使用控制信号 wpcir 控制状态锁存器。

因此 wpcir 信号控制了 wreg 和 wmem 两根线，达到了不让新指令修改原来状态的作用，即增加了 1 个 Bubble。

```
  wreg = wreg_o & wpcir;
  wmem = wmen_o & wpcir;
```

4.3.3 控制冒险

对于控制冒险，我全部采用了 flush 的方式重新对 PC 进行更新，将多执行出来的指令作废处理。代码仅需对使能端 wpcir 控制即可，如下。

```
assign wpcir = ~(em2reg & ((ern==rs)|(ern==rt)) & ~dwmem_tmp)
```

4.4 I/O 模块原理

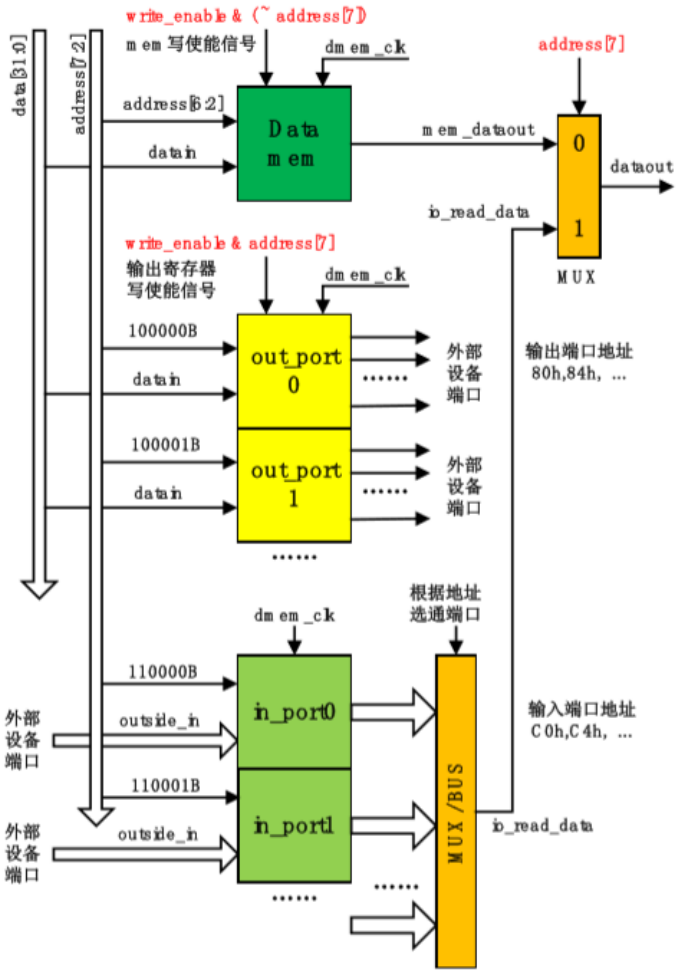

图 2: I/O 接口原理图

I/O 设备与内存并列，通过地址的第 7 位进行判断数据属于 I/O 还是内存。

4.5 I/O 模块实现

```
使用io\_input\_reg和io\_output\_reg可对两个I/O寄存器进行控制。
module io_input_reg (addr,io_clk,io_read_data,in_port0,in_port1);
//inport:  外部直接输入进入ioreg
    input  [31:0]  addr;
    input          io_clk;
    input  [31:0]  in_port0,in_port1;
    output [31:0]  io_read_data;

    reg    [31:0]  in_reg0;     // input port0
    reg    [31:0]  in_reg1;     // input port1

    io_input_mux io_imput_mux2x32(in_reg0,in_reg1,addr[7:2],io_read_data);

    always @(posedge io_clk)
    begin
        in_reg0 <= in_port0;   // 输入端口在 io_clk 上升沿时进行数据锁存
        in_reg1 <= in_port1;   // 输入端口在 io_clk 上升沿时进行数据锁存

            // more ports, 可根据需要设计更多的输入端口。

    end
endmodule



module io_input_mux(a0,a1,sel_addr,y);
    input   [31:0]  a0,a1;
    input   [ 5:0]  sel_addr;
    output  [31:0]  y;
    reg     [31:0]  y;
    always @ *
    case (sel_addr)

       6'b110000: y = a0;
       6'b110001: y = a1;
       // more ports, 可根据需要设计更多的端口。
       endcase
endmodule
```

```
module io_output_reg(addr,datain,write_io_enable,io_clk,clrn,out_port0,
    out_port1,out_port2);

  input [31:0] addr, datain;
  input write_io_enable, io_clk;
  input clrn;
  //reset signal. if necessary,can use this signal to reset the output to 0.
  output [31:0] out_port0, out_port1, out_port2;
  reg [31:0] out_port0, out_port1, out_port2;

```

```
  always @(posedge io_clk or negedge clrn)
  begin
    if(clrn == 0)
    begin
      out_port0 <= 0;
      out_port1 <= 0;
      out_port2 <= 0;
    end
    else
    begin
      if(write_io_enable == 1)
      case(addr[7:2])
        6'b100000: out_port0 <= datain;//80h
        6'b100001: out_port1 <= datain;//84h
        6'b100010: out_port2 <= datain;//88h
      endcase
    end
  end
endmodule
```

其中输出端口即最后需要烧在 FPGA 上的输出 HEX 端口。输入端口这里只指定了两个寄存器，随后会在指令中调用定义的两个寄存器。

4.6 测试指令

指令从 sc_instmem.mif 文件中调用，因为 quartus 存在类似缓存的机制，因此不同的指令文件需要使用不同的名字，文件夹里我写了 sc_instmemADD,sc_instmenSUB 等等。

这里仅拿 sc_instmenSUB 作为展示。

```
DEPTH = 64;           % Memory depth and width are required %
WIDTH = 32;           % Enter a decimal number %
ADDRESS_RADIX = HEX;  % Address and value radixes are optional %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, or OCT; unless %
% otherwise specified, radixes = HEX %
CONTENT
BEGIN
0 : 200100c0;        % (00)       addi $1,$0,192 #  %
1 : 20020080;        % (04)       addi $2,$0,128 #  %
2 : 8c230000;        % (08) loop: lw $3,0($1)    #  %
3 : 8c240004;        % (0c)       lw $4,4($1)    #  %
4 : ac430000;        % (10)       sw $3,0($2)    #  %
5 : ac440004;        % (14)       sw $4,4($2)    #  %
6 : 00642822;        % (18)       sub $5,$3,$4  #  %
7 : ac450008;        % (1c)       sw $5,8($2)    #  %
8 : 08000000;        % (20)       j loop         #  %
END ;
```

其中 192 和 128 号寄存器即是之前 input_reg 指定的输入端口。

5 实验总结

这次实验是继单周期 CPU 后的一次提升，我是在单周期基础上对照着 Verilog 教材写的程序。感悟就是一定要对整体框架非常熟悉，尤其是整体的电路框图，哪个接口对应哪个模块之类的一定要了然于胸。

首先，和单周期时遇到的问题相同，更改指令无效非常头疼，还是没解决同名 mif 无法更改的问题。更改了 mif 文件按理说重新编译会读入新的指令，可是后来发现 quartus 有类似 cache 的机制，即同一个文件读取一次之后，第二次 quartus 会自动判断，如果同名那么就不再读取一次。因此每次读取时都要更改 mif 的文件名，非常恶心。

其次，很多指令的调用关系较为复杂，需要慢慢耐下心来研究，比如 pc 调用指令的时候明明已经 PC+4 了为何还要从倒数第二位开始读取，因为 PC 最后两位永远是 0，读取浪费了时间降低了速度。再比如为什么读数据是直接从 addr 的倒数第 3 位开始取，因为数据宽度为 32，因此后 2 个 bit 仅表示每一位里的偏移量，不做考虑。

再次，流水线相比单周期又多了一个时钟同步的问题，一定要注意什么时候，哪个时间点数据向前移位，只有把握住了数据流的时刻才不会有数据冲突导致结果错误。还要注意 3 种冒险的处理，控制冒险我采用了相对简单的 flush 方式，可是数据冒险一定要注意判断的顺序，判断前 2 条的时候一定要判断在前 1 条有没有更改，否则可能读入了前 2 条的结果却没有在前 1 条进行更新。

最后，制作流水线 CPU 的过程相对单周期更加困难，研究了很长时间才得以完成，希望之后继续努力，对 Verilog 和硬件技术有更深的了解！

517030910374 郭嘉宋