# 实验报告

## 5 段流水线 CPU 设计

王 谱越 516030910462

2018-12-6

# 目录



# 一 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，有别于实验一的设计和实现方法。
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。

# 二 实验内容：

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。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。
3. 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。
4. 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）。
5. 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。

# 三 预习内容：

1. 实验前仔细阅读 DE1-SOC User Manual 及相关用户应用数据手册，学习并掌握其板载相关资源的工作原理、连接方式、和应用注意事项。
2. 根据课程所讲 5 段流水 CPU 设计原理，提前设计并仿真实现相关设计代码。

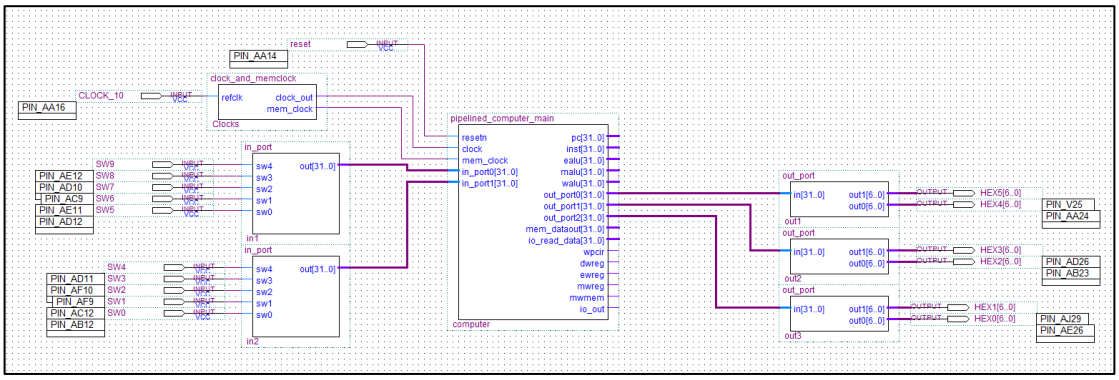
# 四 实验器材：

- 硬件： DE1-SoC 实验板
- 软件： Altera Quartus II 13.1、 Altera ModelSim 10.1d

# 五 实验过程：

## 1. 流水线 CPU 设计：

### 1.1 顶层模块




### 1.2 流水线寄存器

要确保在流水线中不同阶段中的指令不能相互影响，我们需要加入一个新的模块来将各个阶段区分开来，因此引入了流水线寄存器。使得每个周期结束之后，该阶段的执行结果都被保存在流水线寄存器中，在下一个时钟周期写入到下一个阶段。其主要模块如下：

```
module pipeir( pc4,ins,wpcir,clock,resetn,dpc4,inst );
   input [31:0] pc4,ins;
   input        wpcir,clock,resetn;
   output [31:0] dpc4,inst;
   dffe32 pc_puls4 (pc4,clock,resetn,wpcir,dpc4);
   dffe32 dinst (ins,clock,resetn,wpcir,inst);
endmodule
```

对于 IF/ID 以及 WB/IF 流水线寄存器，为了实现指令流水线 Stall，需要加入一个使能信号，用于标识寄存器中的值是否向下一阶段传递。

### 1.3 阶段模块

#### 1.3.1 IF 模块

IF 阶段主要有 3 个模块，负责读取指令以及生成下一条指令的 PC（npc）。需要注意 pc4 以及指令需传到 ID 阶段前的流水线寄存器，但 npc 应当传入 IF 阶段前的流水线寄存器。

```
module pipeif ( pcsource,pc,bpc,da,jpc,npc,pc4,ins,mem_clock );
   input mem_clock;
   input [1:0] pcsource;
   input [31:0] pc,bpc,da,jpc;
   output [31:0] npc,pc4,ins;

   cla32 pc_plus4(pc,32'h4,32'h0,pc4);
   mux4x32 next_pc_switch(pc4,bpc,da,jpc,pcsource,npc);
   sc_instmem inst_mem (pc,ins,mem_clock);
endmodule
```

#### 1.3.2 ID 模块

ID 阶段的主要功能模块有控制单元以及寄存器文件。

```
sc_cu control_unit (mwreg,mrn,ern,ewreg,em2reg,mm2reg,rsrtequ,func,op,rs,rt,
            dwreg,dm2reg,dwmem,daluc,regrt,daluimm,fwda,fwdb,wpcir,sext,
            pcsource,dshift,djal);

regfile reg_file(rs,rt,wdi,wrn,wwreg,clock,resetn,qa,qb);
```

控制单元，负责产生流水线中的控制号，并随着流水线寄存器传递到各个阶段。其中 rsrtequ 表示 da 与 db 是否相等。由 ID 阶段产生，用来表示分支语句的判断条件是否满足。

模块从寄存器文件中读取指令相关寄存器中的值，以及之后的写回到目标寄存器的操作。

为了解决控制冲突，在这个阶段中，模块需要产生 3 个跳转操作的目的地址 bpc, jpc, da，具体的冲突解决方法将在 1.4 节中详细介绍。

```
cla32 bpc_pluse(dpc4,offset,32'h0,bpc); //bpc
assign jpc = {dpc4[31:28],(inst[25:0]<<2)}; //jpc
mux4x32 da_switch (qa,ealu,malu,mmo,fwda,da);//da
```

### 1.3.3 EXE 模块

EXE 模块主要任务是进行 ALU 计算，以及 jal 指令的返回地址的生成。

```
module pipeexe( ealuc,ealuimm,ea,eb,eimm,eshift,ern0,epc4,ejal,ern,ealu );
   input       ealuimm,eshift,ejal;
   input [3:0] ealuc;
   input [31:0] ea,eb,eimm,epc4;
   input [4:0] ern0;
   output [31:0] ealu;
   output [4:0] ern;
   wire [31:0] epc8,sa,ealua,ealub,ealu0;
   wire z;
   assign sa={27'b0,eimm[10:6]};
   cla32 pc4_pluse4(epc4,32'h4,32'h0,epc8);
   mux2x32 e_alua (ea,sa,eshift,ealua);
   mux2x32 e_alub (eb,eimm,ealuimm,ealub);
   alu E_ALU(ealua,ealub,ealuc,ealu0,z);
   mux2x32 e_alu (ealu0,epc8,ejal,ealu);
   assign ern = ern0 | {5{ejal}};


endmodule
```

### 1.3.4 MEM 模块

MEM 模块的主要任务是将 ALU 结果或寄存器数据通过数据单元写入到储存器中。数据单元将在 1.5 节详细介绍。

```
module pipemem( mwmem,malu,mb,clock,mem_clock,mmo ,out_port0,out_port1,out_port2,in_port0,in_port1,mem_dataout,io_read_data,io_out);
   input       mwmem,clock,mem_clock;
   input[31:0] malu,mb,in_port0,in_port1;
   output [31:0] mmo,out_port0,out_port1,out_port2,mem_dataout,io_read_data;
   output io_out;
   sc_datamem datamem(malu,mb,mmo,mwmem,clock,mem_clock,out_port0,out_port1,out_port2,in_port0,in_port1,mem_dataout,io_read_data,io_out);

endmodule
```

### 1.3.5 WB 模块

WB 模块比较简单，只有一个多路选择器，用于选择写回到寄存器的数据。

## 1.4 控制单元

### 1.4.1 处理流程：

控制单元是这次实验中最复杂的模块，负责控制信号的接受、处理以

及发送，来控制整个流水线的行为。

控制单元首先需要解析指令，来识别各种操作，基于解析的指令，控制单元会产生多个控制信号：

- regrt:为 1 时，以 rt 为目标寄存器。为 0 时，以 rd 为目标寄存器。
- aluimm: 为 1 时，将立即数传入 ALU 的 b 输入端。为 0 时，将 rt 寄存器的数据传入 ALU 的 b 输入端。
- aluc:4 位，表示 ALU 模块进行的操作。
- wmem: 为 1 时写储存器，为 0 时不写。
- m2reg: 为 1 时选择储存器输出写入寄存器文件 d 端，为 0 时将 ALU 写入寄存器文件 d 端。
- wreg: 为 1 时将 d 端数据写入目的寄存器，为 0 时不写入。
- jal: 为 1 时，将 pc+8 替换 alu 输出，为 0 时不替换。
- shift:为 1 时，对 rt 进行 shift 操作,为 0 时不进行。
- sext:为 1 时，对立即数进行符号扩展，为 0 时逻辑扩展。
- pcsource:2 位，表示 nextPC 的选择。
- wpcir:为解决冲突引入，若为 0 则流水线 stall.
- fwda,fwdb:2 位，为解决冲突引入，进行数据的旁路直通。

其中大部分的信号作用与单周期时相同，区别在于，ID 阶段控制单元产生的信号，会存到流水线寄存器中，随着阶段向后传递。这些向后传递的信号也会输入到控制单元中用于检测冲突。

### 1.4.2 冲突解决

#### 1.4.2.1数据冲突

##### 1.4.2.1.1 Forwarding

为了解决，第二条指令的需要使用第一条指令尚未写会寄存器的结果的问题，我们可以采用内部旁路的方法。将前一条指令 ALU 结果传送到 ID 阶段。使用这样的方法，可以把 EXE、MEM 阶段的 alu 结果，以及 MEM 阶段从储存器中读出的结果推到 ID 阶段。

```
always @ (ewreg or mwreg or ern or mrn or em2reg or mm2reg or rs or rt )
   begin
   //rd
   fwda =2'b00;
   if (ewreg & (ern != 0 ) & (ern == rs) & ~em2reg)//alu need the result of prev-inst rd = rs
      begin
      fwda = 2'b01;
      end
   else
      begin
      if (mwreg & (mrn != 0 ) & (mrn == rs) & ~mm2reg)//alu need the result of the second inst before it   rd = rs
         begin
         fwda = 2'b10;
         end
      else
         begin
         if (mwreg & (mrn != 0) & (mrn == rs) & mm2reg) //alu need the memery output of the second inst before it   rd = rs
            begin
            fwda = 2'b11;
            end
         end
      end
   //.....
```

但是，对于 lw 的之后指令需要 lw 的结果，则无法通过这样的方式解决。

##### 1.4.2.1.2 Stall

为了解决上述问题，我们需要在遇到 lw 的冲突时将流水线暂停一个周期。为实现这样的目的，我们只需要不修改 PC 和 IR 即可，因此对 IR 及 IF 的流水线寄存器引入使能信号 wpcir.

```
assign wpcir = ~(ewreg & em2reg & (ern != 0) & (i_rs &(ern == rs)| i_rt & (ern == rt)));//STALL
```

同时为了防止 IR 阶段的指令被执行两次，需要将以此 IR 指令废弃掉，也即使得其不改变 CPU 状态：

```
assign wreg = (i_add | i_sub | i_and | i_or   | i_xor  |
               i_sll | i_srl | i_sra | i_addi | i_andi |
               i_ori | i_xori | i_lw | i_lui  | i_jal) & wpcir;
assign wmem  = i_sw & wpcir;
```

#### 1.4.2.2控制冲突

流水线 CPU 在执行转移和跳转相关指令时，会出现控制相关问题，在实际转向目标地址之前，转移或跳转指令的后续指令已经取到流水线里来了。

解决控制冲突的方法有很多种，这里采用的是和 MIPS 指令一致的延迟转移的方式，即在跳转和转移指令后加入一个延迟槽，使得跳转一个周期后生效。这样只要在 ID 阶段，确定跳转地址即可正确的完成跳转，因此在 ID 阶段引入了多个地址候选项的计算，同时在控制单元中产生地址选择的信号。

```
//00:pc+4 01:bpc 10:da 11:jpd
assign pcsource[1] = i_jr | i_j | i_jal;
assign pcsource[0] = ( i_beq & rsrtequ ) | (i_bne & ~rsrtequ) | i_j | i_jal ;
```

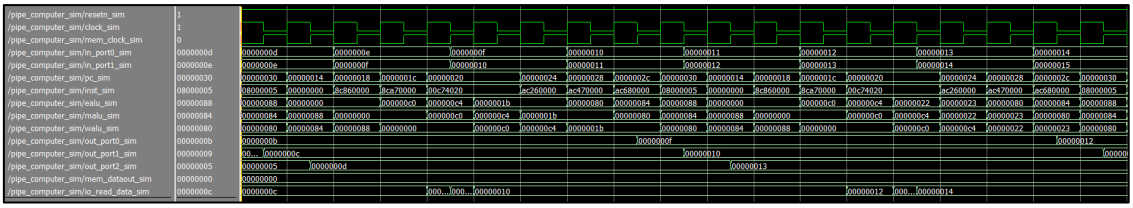
### 1.5 数据单元

数据单元主要负责内存数据以及 I/O 数据的读写。利用数据单元的处理，将指令中的地址分为两段：“1”开头的被分配为 IO 地址，而“0”开头的地址被分配为内存地址，这样就完成了对 IO 与内存的读写操作的兼容。

```
assign write_datamem_enable = write_enable & ( ~ addr[7]); //mem
assign write_io_output_reg_enable = write_enable & ( addr[7]); //IO
```

## 2. 仿真模拟验证

### 2.1 流程验证



可以看到 IO 与指令执行都有正常的输出，整个程序的逻辑是将两个输入相加并输出到输出端口，可以看到结果也符合预期。

### 2.2 冲突解决验证

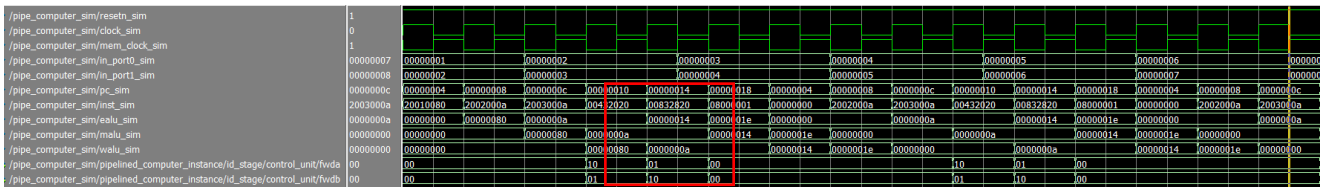
#### 2.2.1 Forwarding

i. 测试代码：

```
BEGIN
[0..7] : 00000000;   % Range--Every address from 0 to 1F = 00000000 %

0 : 20010080;        % (00) main: addi  $1, $0, 128 # outport0                %
1 : 2002000a;        % (04) loop: addi   $2, $0,10  # input inport0 to $4      %
2 : 2003000a;        % (08)       addi $3, $0,10    # input inport1 to $5      %
3 : 00432020;        % (0c)       add  $4, $2, $3   # need forwarding 3        %
4 : 00832820;        % (10)       add $5,$4,$3      # need forwarding 4        %
5 : 08000001;        % (14)       j loop            #                         %
END ;
```

ii. 测试结果




可以看到 ID 阶段有生成准确的 fwd 信号。

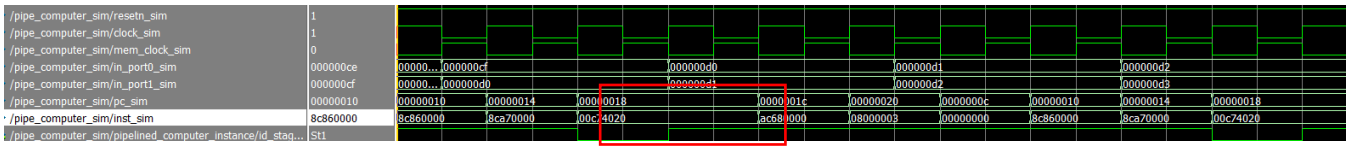
### 2.2.2 Stall

i. 测试代码：

```
BEGIN
[0..F] : 00000000;   % Range--Every address from 0 to 1F = 00000000 %

0 : 20030088;        % (00) main: addi $3, $0, 136 # outport2                    %
1 : 200400c0;        % (04)       addi $4, $0, 192 # inport0                     %
2 : 200500c4;        % (08)       addi $5, $0, 196 #inport1                      %
3 : 8c860000;        % (0c) loop: lw   $6, 0($4)   # input inport0 to $4         %
4 : 8ca70000;        % (10)       lw   $7, 0($5)   # input inport1 to $5         %
5 : 00c74020;        % (14)       add  $8, $6, $7  # need stall here             %
6 : ac680000;        % (18)       sw   $8, 0($3)   # output result to outport2   %
7 : 08000003;        % (1c)       j loop           #                            %
END ;
```

ii. 测试结果



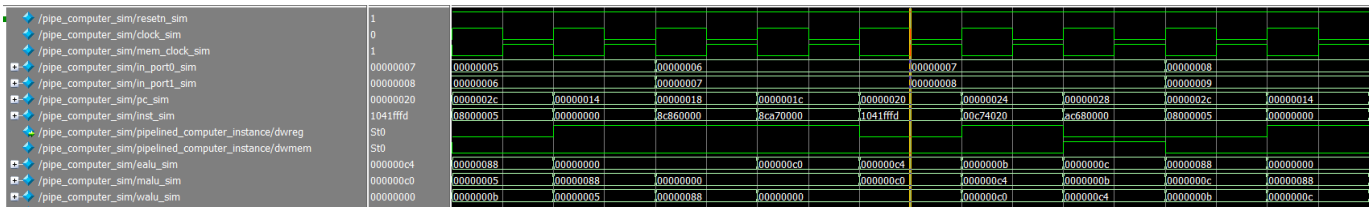

可以看到在第 5 条指令处流水线按照预期暂停了一个周期。

### 2.2.3 控制冲突

i. 测试代码：

```
0 : 2001000a;        % (00) main: addi $1,$0,10    #                          %
1 : 20020001;        % (04)       addi $2,$0,1     #                          %
2 : 20030088;        % (08)       addi $3, $0, 136 # outport2                 %
3 : 200400c0;        % (0c)       addi $4, $0, 192 # inport0                  %
4 : 200500c4;        % (10)       addi $5, $0, 196 #inport1                   %
5 : 8c860000;        % (14) loop: lw   $6, 0($4)   # input inport0 to $4      %
6 : 8ca70000;        % (18)       lw   $7, 0($5)   # input inport1 to $5      %
7 : 1041fffd;        % (1c)       beq $2,$1,loop   # control hazard           %
8 : 00c74020;        % (20)       add  $8, $6, $7  # add inport0 with inport1 to $6 %
9 : ac680000;        % (24)       sw   $8, 0($3)   # output result to outport2   %
A : 08000005;        % (28)       j loop           #                          %
END ;
```

ii. 测试结果




可以看到第十条指令后，有一个 NOP 指令，作为延迟转移槽。对于 beq 指令，wreg 和 wmem 的信号值为 0，保证之后的指令不会执行。

### 3. 实际验证

实际验证采用与之前单周期相同的加法器测试代码，烧录后可正常运行。

## 六 实验结果与感想：

### 1. 实验结果

经过之前的验证，流水线可以正常运行，多种冲突都可以正常处理，烧录到硬件后也运行正常，完成实验要求。

## 2. 遇到的问题

- **Memclok 与 Cpuclock 的相位关系：**

  在单周期 CPU 中，为了保证多个过程的顺序，因此采用了 2 倍频于 CPUclock 的 Memclock。

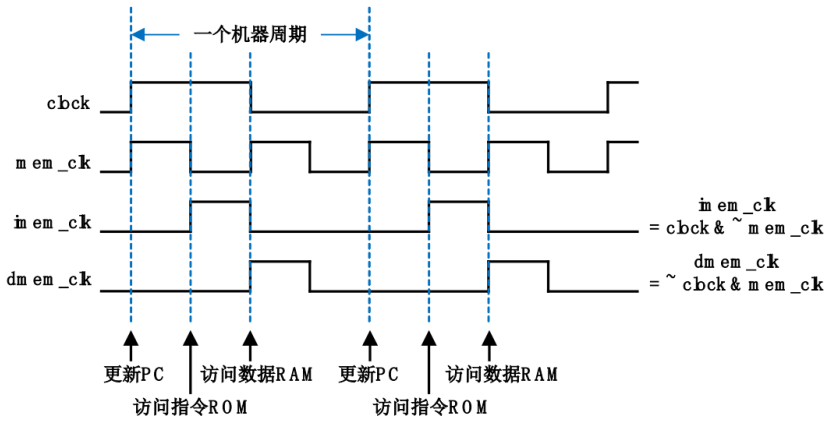



图 2-1-15. 一个机器周期内各执行步骤的执行触发顺序

在流水线 CPU 中，由于有了流水线寄存器，执行顺序有了保证，Memclock 与 CPUclock 的关系也有了改变，是一个与 CPUclock 反向同频率的信号。则在 MEM 阶段，CPUclock 上升沿时流水线寄存器将信号传递到 MEM 阶段，半个周期后，MEMclock 上升沿出现，数据单元访问数据 RAM。在 IF 阶段，CPUclock 上升沿时，PC 被更新，半个周期后 MEMclock 上升沿出现，指令寄存器读取指令 ROM。这样就保证了原来流水线 CPU 需要 4 个节拍排序的操作在流水线中也能够按顺序进行。

- **时钟信号生成：**

  在数据模块中，生成 Datamem 的时钟信号时，采用了与流水线中相同的 assign 逻辑来赋值。Assign 是一个组合逻辑，产生信号号时会有一定的时延，这个特性，在 Modelsim 的模拟中不会表现出来，在实际的硬件上会表现出来，并导致错误。在单周期中，由于没有流水线寄存器，所以信号量不会存在干扰。而在流水线中，Memclock 与 CPUclock 之间有一定的相位差，因此，在有时延的情况下，可能导致 Datamem 的时钟上升沿到来时，信号已经发生了改变。**在任何情况下，都不应该使用 assign 来生成时钟信号。**

## 3. 感想与反思

在本次实验中，我实践了单独设计流水线 CPU 的全过程，设计与实现由于有老师的讲解以及 PPT 十分顺利，但是在测试过程中遇到了许多问题，依靠着 ModelSim 解决了大部分的问题。不过最令我印象深刻的就是 Datamem 的 clock 的问题，由于这个问题在 modelsim 中没有出现而在实际环境中会出现，花费了大量时间排差。整个过程中验证与测试使得我对 Modelsim 的使用更加熟练，也真正充分了解了硬件开发问题的复杂性与多样性。