《计算机组成》课程实验之五级流水 CPU 模块设计仿真

要求说明与设计指导

202005

本课程实验教学目的，是要求同学们利用 Verilog 硬件描述语言、基于大规模可编程逻辑阵列器件 FPGA，进行计算机 CPU 的逻辑功能设计，包括 RISC 风格的单周期 CPU 设计、五级流水 CPU 设计、以及在其上的输入输出部件扩展和功能扩展。

以下为第三部分五级流水线 CPU 模块设计的实验目的、实验要求、实验步骤、设计指导、仿真要求、附录。

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、实验要求

1. 完成五级流水线CPU核心模块的设计。
2. 完成对五级流水线CPU的仿真，仿真测试程序应该具有与实验一提供的标准测试程序代码相同的功能。对两种CPU实现核心处理功能的过程和设计处理上的区别作对比分析。
3. 完成流水线CPU的IO模块仿真，对两种CPU实现相同IO功能的过程和设计处理上的区别作对比分析。

三、实验步骤

1. 基于实验一或者实验二的模块代码，根据个人情况选择其一，采用 Verilog语言在 quartus II 中实现具有 20 条 MIPS 指令的 5 段流水 CPU设计。如果基于实验二的已添加了IO模块的结果开始流水线CPU的设计，更便于下面第4-6步骤的快速完成。当然也可以基于实验一的结果开始。
2. 修正实验一提供的CPU执行程序标准测试代码，以适应流水线设计的CPU结构，完成仿真测试，获得和单周期CPU测试仿真一致的代码执行结果。
3. 对比实现相同功能的实验一和实验三的仿真结果的异同，理解流水线实现方法及流水线数据冒险、控制冒险的处理方法。
4. 利用实验二自己设计添加的 I/O 端口，修正实验二阶段自己编写的IO读写测试程序代码，通过 lw、sw 指令，在自己设计的5级流水线 CPU 上，模拟实现对外部输入开关或按键的状态输入，并将判别或处理结果，模拟利用 7 段 LED 数码管显示出来。
5. 对比实验二和实验三完成相同IO功能时候的差异，理解流水线冒险概念和引起停顿的处理方法。
6. 例如，将一路 5bit 二进制输入与另一路 5bit 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。(4-6具体任务形式不做严格规定，同学可自由创意)。

四、设计指导

本部分实验不再给详细的示例程序，仅给出顶层模块代码示例以及设计思路指导。

1. 实验设计原理参考课程教学相关内容及有关实验补充材料，本文档最后附上了摘取的关键点原理示意结构图。
2. 实验设计的顶层主要代码模块，分别是 5 个流水段的 module、以及五级流水段之间的 4 个流水线寄存器 module，外加 1 个程序计数器 module（也可以当作为最前面一级流水段的输入流水线寄存器），共计 10 个 module。
3. 每一个 module 的功能，参考课程 5 级流水线的设计原理，即可确定并进行设计。
4. 实验中的顶层文件结构可参考如下所示代码进行设计：(仅用于参考方法，不限定同学自己的设计风格和实现方法)

顶层模块代码示例：

```
module pipelined_computer (resetn, clock, mem_clock, opc, oinst, oins, oealu, omalu, owalu, onpc, in_port0, in_port1, out_port0, out_port1, out_port2, out_port3);
    //定义顶层模块 pipelined_computer，作为工程文件的顶层入口，如图 1-1 建立工程时指定。
    Input     resetn, clock;
    output    mem_clock;
    assign     mem_clock = ~clock;
  //定义整个计算机 module 和外界交互的输入信号，包括复位信号 resetn、时钟信号 clock、
  //以及一个和 clock 同频率但反相的 mem_clock 信号。mem_clock 用于指令同步 ROM 和数据同步 RAM 使用，其波形需要有别于实验一。
 //这些信号可以用作仿真验证时的输出观察信号。
    input    [5:0] in_port0, in_port1;
    output   [31:0]  out_port0,  out_port1,  out_port2,  out_port3;  /*  output  [6:0] out_port0,out_port1,out_port2,out_port3;*/

    wire [31:0] real_out_port0,real_out_port1,real_out_port2,real_out_port3;
    wire [31:0] real_in_port0 = {26'b00000000000000000000000000,in_port0};
    wire [31:0] real_in_port1 = {26'b00000000000000000000000000,in_port1};

    assign out_port0 = real_out_port0[31:0];//assign out_port0 = real_out_port0[6:0];
    assign out_port1 = real_out_port1[31:0];//assign out_port0 = real_out_port1[6:0];
    assign out_port2 = real_out_port2[31:0];//assign out_port0 = real_out_port2[6:0];
    assign out_port3 = real_out_port3[31:0];//assign out_port0 = real_out_port3[6:0];
     //IO 口的定义，宽度可根据自己设计选择。
      wire [31:0] pc,ealu,malu,walu;
      output [31:0] opc,oealu,omalu,owalu; // for watch
      assign opc = pc;
      assign oealu = ealu;
      assign omalu = malu ;
```

```
    assign owalu = walu ;

    output [31:0] onpc,oins,oinst; // for watch
    assign   onpc=npc;
    assign   oins=ins;
    assign   oinst=inst;
```
//模块用于仿真输出的观察信号。缺省为 wire 型。为了便于观察内部关键信号，将其接到输出管脚。不输出也一样，只是仿真时候要从内部信号里去寻找。

```
  wire     [31:0] bpc,jpc,pc4,npc,ins,inst;
```
//模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。IF 取指令阶段。
```
  wire     [31:0] dpc4,da,db,dimm,dsa;
```
//模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。ID 指令译码阶段。
```
  wire     [31:0] epc4,ea,eb,eimm,esa;
```
//模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。EXE 指令运算阶段。
```
  wire     [31:0] mb,mmo;
```
//模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。MEM 访问数据阶段。
```
  wire     [31:0] wmo,wdi;
```
//模块间互联传递数据或控制信息的信号线,均为 32 位宽信号。WB 回写寄存器阶段。
```
  wire     [4:0] ern0,ern,drn,mrn,wrn;
```
//模块间互联，通过流水线寄存器传递结果寄存器号的信号线，寄存器号（32 个）为 5bit。
```
  wire     [4:0] drs,drt,ers,ert;
```
// 模块间互联，通过流水线寄存器传递 rs、rt 寄存器号的信号线，寄存器号（32 个）为 5bit。
```
  wire     [3:0] daluc,ealuc;
```
//ID 阶段向 EXE 阶段通过流水线寄存器传递的 aluc 控制信号，4bit。
```
  wire     [1:0] pcsource;
```
//CU 模块向 IF 阶段模块传递的 PC 选择信号，2bit。
```
  wire           wpcir;
```
// CU 模块发出的控制流水线停顿的控制信号，使 PC 和 IF/ID 流水线寄存器保持不变。
```
  wire           dwreg,dm2reg,dwmem,daluimm,dshift,djal;   //id stage
```
// ID 阶段产生，需往后续流水级传播的信号。
```
  wire           ewreg,em2reg,ewmem,ealuimm,eshift,ejal;   //exe stage
```
//来自于 ID/EXE 流水线寄存器，EXE 阶段使用，或需要往后续流水级传播的信号。
```
  wire           mwreg,mm2reg,mwmem;   //mem stage
```
//来自于 EXE/MEM 流水线寄存器，MEM 阶段使用，或需要往后续流水级传播的信号。
```
  wire           wwreg,wm2reg;   //wb stage
```
//来自于 MEM/WB 流水线寄存器，WB 阶段使用的信号。
```
  wire           ezero,mzero;
```
//模块间互联，通过流水线寄存器传递的 zero 信号线
```
  wire           ebubble,dbubble;
```
//模块间互联，通过流水线寄存器传递的流水线冒险处理 bubble 控制信号线
```
  pipepc     prog_cnt(npc,wpcir,clock,resetn,pc);
```
//程序计数器模块，是最前面一级 IF 流水段的输入。
```
  pipeif     if_stage(pcsource,pc,bpc,da,jpc,npc,pc4,ins,mem_clock);     // IF stage
```

```
//IF 取指令模块，注意其中包含的指令同步 ROM 存储器的同步信号，
//即输入给该模块的 mem_clock 信号，模块内定义为 rom_clk。// 注意 mem_clock。
//实验中可采用系统 clock 的反相信号作为 mem_clock（亦即 rom_clock），
//即留给信号半个节拍的传输时间。
  pipeir     inst_reg(pc4,ins,wpcir,clock,resetn,dpc4,inst);   // IF/ID 流水线寄存器
   //IF/ID 流水线寄存器模块，起承接 IF 阶段和 ID 阶段的流水任务。
//在 clock 上升沿时，将 IF 阶段需传递给 ID 阶段的信息，锁存在 IF/ID 流水线寄存器
//中，并呈现在 ID 阶段。
  pipeid     id_stage(mwreg,mrn,ern,ewreg,em2reg,mm2reg,dpc4,inst,ins,
                      wrn,wdi,ealu,malu,mmo,wwreg,mem_clock,resetn,
                      bpc,jpc,pcsource,wpcir,dwreg,dm2reg,dwmem,daluc,
                      daluimm,da,db,dimm,dsa,drn,dshift,djal,mzero,
                      drs,drt/*,npc*/,ebubble,dbubble);   // ID stage
//ID 指令译码模块。注意其中包含控制器 CU、寄存器堆、及多个多路器等。
//其中的寄存器堆，会在系统 clock 的下沿进行寄存器写入，也就是给信号从 WB 阶段
//传输过来留有半个 clock 的延迟时间，亦即确保信号稳定。
//该阶段 CU 产生的、要传播到流水线后级的信号较多。
  pipedereg de_reg(dbubble, drs, drt, dwreg, dm2reg, dwmem,
                      daluc, daluimm, da, db, dimm, dsa, drn, dshift, djal,
                      dpc4, clock, resetn, ebubble, ers, ert, ewreg, em2reg,
                      ewmem, ealuc, ealuimm, ea, eb, eimm, esa, ern0, eshift, ejal, epc4);
                      // ID/EXE 流水线寄存器
//ID/EXE 流水线寄存器模块，起承接 ID 阶段和 EXE 阶段的流水任务。
//在 clock 上升沿时，将 ID 阶段需传递给 EXE 阶段的信息，锁存在 ID/EXE 流水线
//寄存器中，并呈现在 EXE 阶段。
  pipeexe    exe_stage (ealuc, ealuimm, ea, eb, eimm, esa, eshift, ern0,
                      epc4, ejal, ern, ealu, ezero, ert, wrn, wdi, malu, wwreg);   // EXE stage
//EXE 运算模块。其中包含 ALU 及多个多路器等。
  pipeemreg em_reg(ewreg, em2reg,ewmem,ealu,eb,ern,ezero,clock,resetn,mwreg,
                      mm2reg,mwmem,malu,mb,mrn,mzero); // EXE/MEM 流水线寄存器
//EXE/MEM 流水线寄存器模块，起承接 EXE 阶段和 MEM 阶段的流水任务。
//在 clock 上升沿时，将 EXE 阶段需传递给 MEM 阶段的信息，锁存在 EXE/MEM
//流水线寄存器中，并呈现在 MEM 阶段。
  pipemem mem_stage (mwmem,malu,mb,clock,mem_clock,mmo,resetn,
                        real_in_port0,real_in_port1,real_out_port0,real_out_port1,
                        real_out_port2,real_out_port3);   //      MEM stage
//MEM 数据存取模块。其中包含对数据同步 RAM 的读写访问。// 注意 mem_clock。
//输入给该同步 RAM 的 mem_clock 信号，模块内定义为 ram_clk。
//实验中可采用系统 clock 的反相信号作为 mem_clock 信号（亦即 ram_clk），
//即留给信号半个节拍的传输时间，然后在 mem_clock 上沿时，读输出、或写输入。
  pipemwreg mw_reg(mwreg,mm2reg,mmo,malu,mrn,clock,resetn,
                      wwreg,wm2reg,wmo,walu,wrn);   //   MEM/WB 流水线寄存器
//MEM/WB 流水线寄存器模块，起承接 MEM 阶段和 WB 阶段的流水任务。
//在 clock 上升沿时，将 MEM 阶段需传递给 WB 阶段的信息，锁存在 MEM/WB
//流水线寄存器中，并呈现在 WB 阶段。
    mux2x32 wb_stage(walu,wmo,wm2reg,wdi);   //     WB stage
//WB 写回阶段模块。事实上，从设计原理图上可以看出，该阶段的逻辑功能部件只
```

```
//包含一个多路器，所以可以仅用一个多路器的实例即可实现该部分。
//当然，如果专门写一个完整的模块也是很好的。
endmodule
```

5. 这样，以上的 10 个模块（流水段模块+各段之间的流水线寄存器模块）在计算机系统 clock 的同步协调触发下，就会将指令在 5 段流水线中顺序流水执行。
6. 实验三给同学提供设计顶层代码，实验一或者二的大部分代码可修改后使用，设计完成后，尽快掌握和精通流水线的设计原理和实现方法。并能触类旁通地体会和理解对其它算法逻辑进行流水处理的设计和实现方法。
7. 实验代码的工程文件 pipelined_computer.qpf 所包含的相关源代码文件组织结构如图1截图所示，这只是一种实现方式示例，供参考。

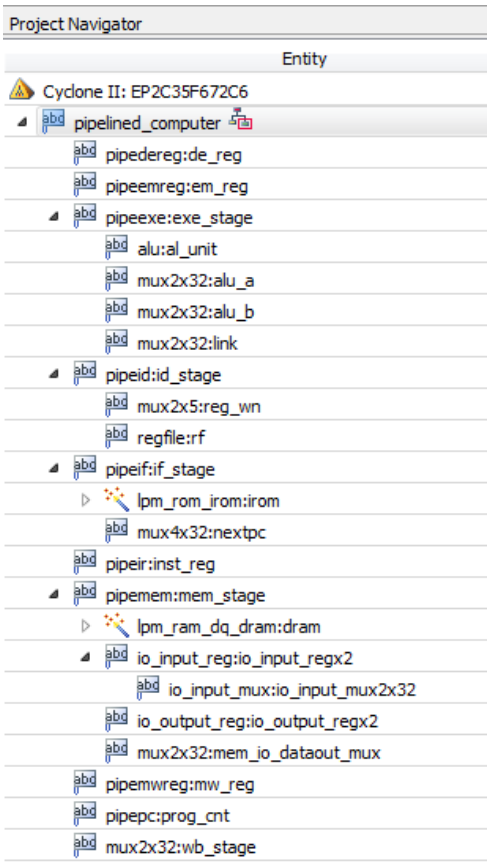


图 1 流水线 CPU 实验代码的工程文件示例

五、仿真要求

图 2 为仿真测试输入文件 pipelined_computer_test_wave_01.vwf 的输入波形。为观察直观，其它的输出观测信号设置为“uninitialized”状态。

实验一的标准测试程序汇编代码 **sc_code_sccpu.txt** 如下。可以用提供给同学们的 SE2.0 编译调试小工具生成用于数据和指令存储器初始化的.mif 文件。实验一我们已经为同学提供了生成好的.mif 文件。可通过 SE2.0 对汇编代码进行修改和编译，生成新的测试代码，适用于流水线 CPU 的执行。或者生成自定义的如 IO 扩展调试用的 CPU 程序代码的 mif 文件，以协助硬件调试。

基于实验一标准测试程序修正重新编译后，执行测试输入波形激励文件

pipelined_computer_test_wave_01.vwf，对流水线 CPU 仿真，输出如图 3 所示，说明流水线 CPU 模块设计完成，功能正确，可以完成等同于实验一的单周期 CPU 标准测试程序执行完成的功能。

之后可以选做自行添加对 IO 口操作的仿真，并与实验二的仿真结果做对比。

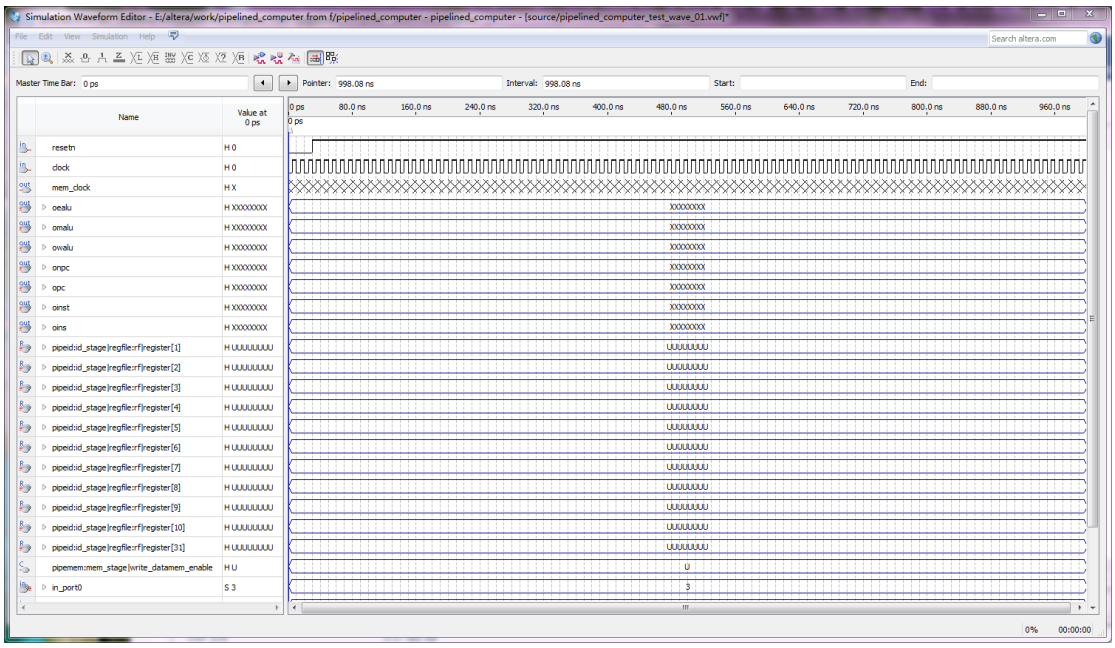


图 2 仿真测试输入 pipelined_computer_test_wave_01.vwf 的输入波形

下面列出实验一的标准测试程序汇编代码 **sc_code_sccpu.txt**，其功能为从内存 50h、54h、58h、5ch 处，逐次读入数据并四次调用子程序段 sum 进行求和，其结果 258h 存入 R9 寄存器。之后进行一系列立即数、逻辑操作、移位操作测试指令执行情况，程序最终停止于死循环语句（finish:j finish）之处。

```
main:   lui $1, 0                  # address of data[0] %
        ori $4, $1, 80             # address of data[0] %
        addi $5, $0, 4             # counter %
call:   jal sum                    # call function %
        sw $2, 0($4)               # store result %
        lw $9, 0($4)               # check sw %
        sub $8, $9, $4             # sub: $8 <- $9 - $4 %
        addi $5, $0, 3             # counter %
loop2:  addi $5, $5, -1            # counter - 1 %
        ori $8, $5, 0xffff         # zero-extend: 0000ffff %
        xori $8, $8, 0x5555        # zero-extend: 0000aaaa %
        addi $9, $0, -1            # sign-extend: ffffffff %
        andi $10, $9, 0xffff       # zero-extend: 0000ffff %
        or $6, $10, $9             # or: ffffffff %
        xor $8, $10, $9            # xor: ffff0000 %
        and $7, $10, $6            # and: 0000ffff %
        beq $5, $0, shift          # if $5 = 0, goto shift %
```

```
        j loop2                 # jump loop2 %
shift:  addi $5, $0, -1         # $5 = ffffffff %
        sll $8, $5, 15          # <<15 = ffff8000 %
        sll $8, $8, 16          # <<16 = 80000000 %
        sra $8, $8, 16          # >>16 = ffff8000 (arith) %
        srl $8, $8, 15          # >>15 = 0001ffff (logic) %
finish: j finish                # dead loop %
sum:    add $8, $0, $0          # sum %
loop:   lw $9, 0($4)            # load data %
        addi $4, $4, 4          # address + 4 %
        add $8, $8, $9          # sum %
        addi $5, $5, -1         # counter - 1 %
        bne $5, $0, loop        # finish? %
        sll $2, $8, 0           # move result to $v0 %
        jr $ra                  # return %
```

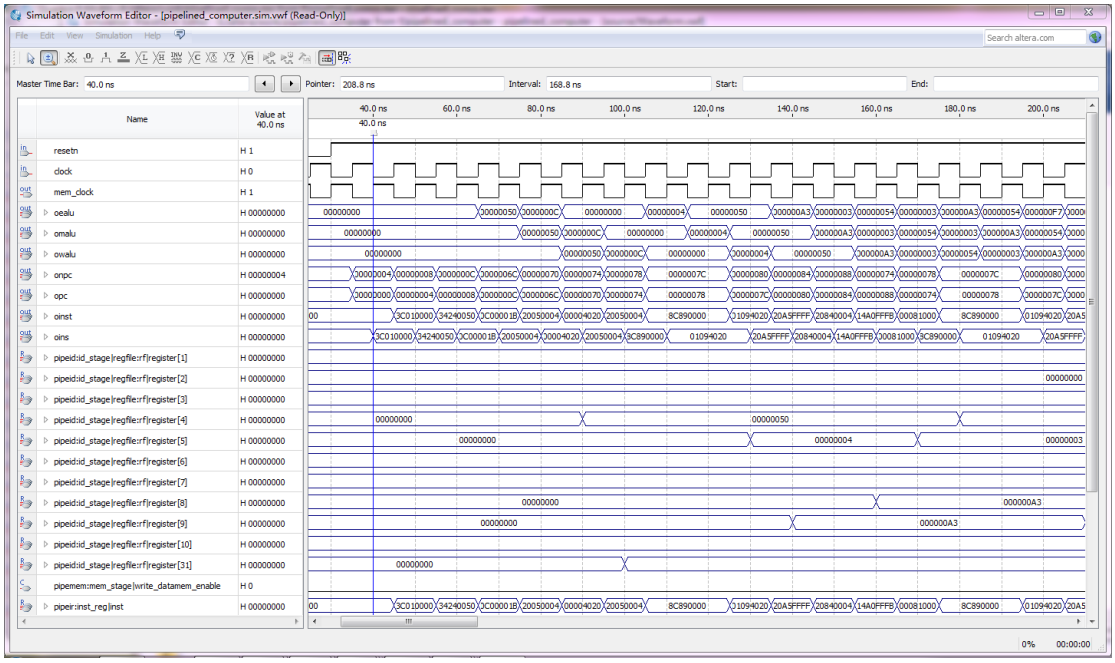


图 3（a） 流水线 CPU 的仿真运行输出-几级 ALU 信号逐级传递

从图 3（a）可见，reset 变高之后，第一个 clock 上升沿 35ns 处，程序指令计数器 pc（opc 信号）从 0 开始，在 40ns 处 clock 下降沿取指令，oins 信号表示取指令 IF 模块从指令存储器读出指令，在下一个时钟上升沿 40ns 时给出第一条指令码 3c010000，传到后一级指令解码 ID 模块。从第一个指令开始，在后续每个 clock 上升沿，每个时钟节拍进行一次指令处理。每条指令要经过流水线 5 个节拍后出结果。例如 50ns 处的第 2 条指令 34240050，其操作 ori $4, $1, 80 要将输入端口 0 的地址 50h 存入 R4 寄存器，R4 在 4 个 clock 后的 90ns 时改变为 50h。其间几级 ALU 信号逐级顺序传递过程也可通过波形图清晰展现。

从图 3（b）（c）可见，程序执行过程中各个寄存器数据变化情况，每条指令的流水执行过程。伴随着执行代码里面的转移指令、lw 指令执行，可见需要通过插入 bubble 打断流水线，获得正确的执行结果。参见图 3（b）110ns、180ns、250ns、320ns、440ns 等处。插入

bubble 的方式则可以是通过软件层面加入 nop 指令，或者在 CPU 硬件层面，添加对相关跳转指令的处理，自动停止一次 PC 指针更新。

通过图 3（c）与单周期 CPU 的相同功能程序仿真结果对比，理解功能相同但结构不同的 CPU 的执行过程。

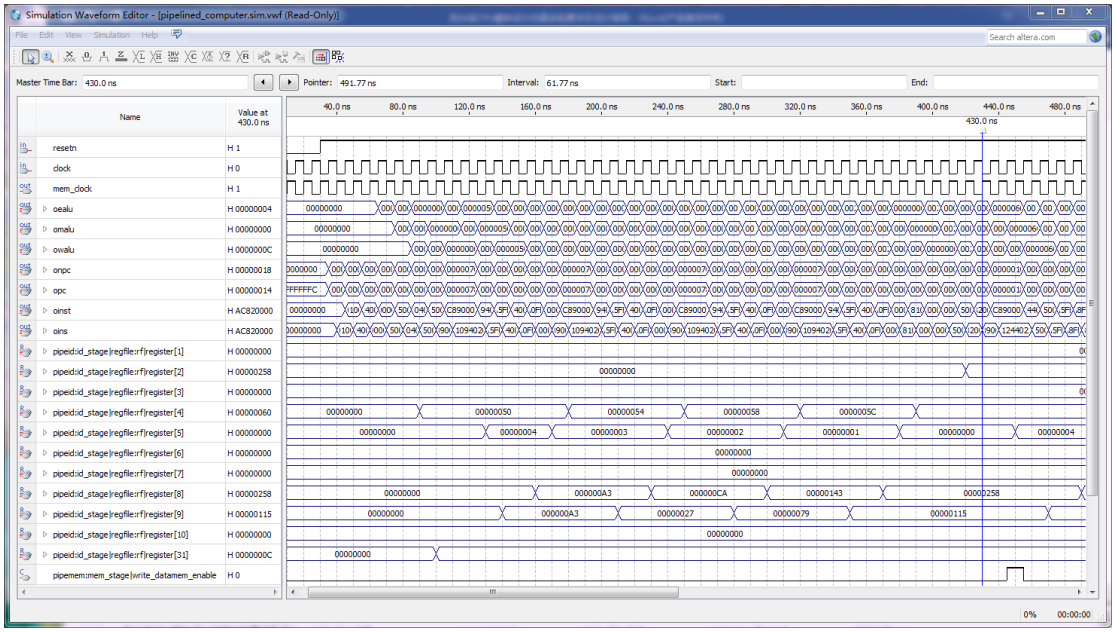


图 3（b） 流水线 CPU 的仿真运行输出-前半部分

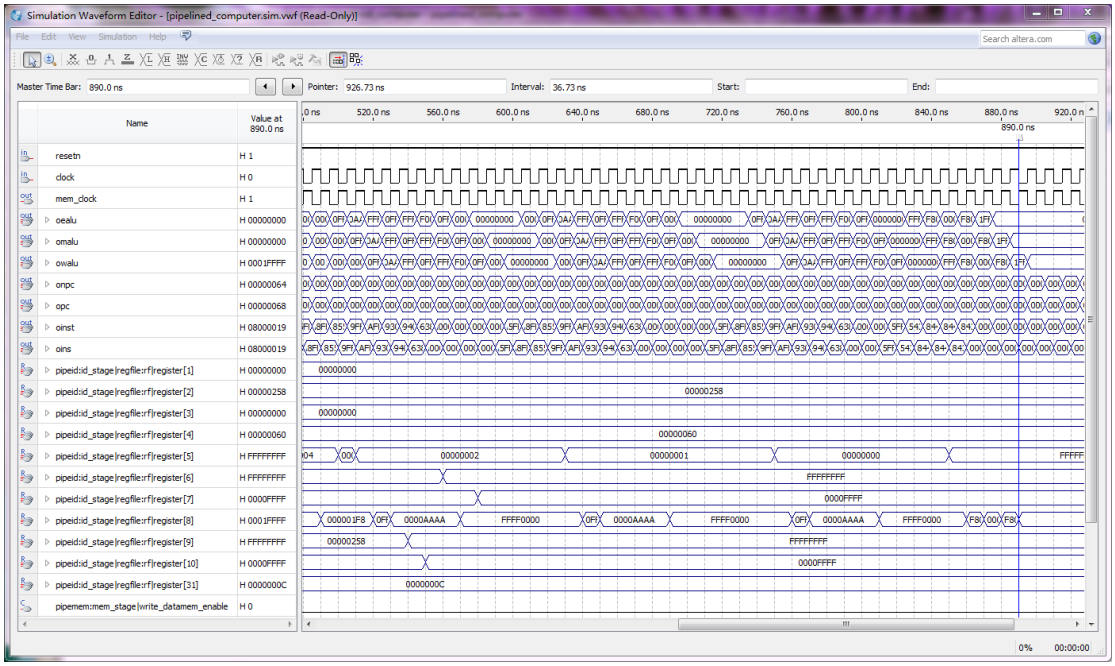


图 3（c） 流水线 CPU 的仿真运行输出-后半部分

如果用单周期 CPU 测试代码 sc_instmem.mif 直接运行，会得到什么样的结果呢？我们看到的波形如图 4。会发现 R8 最后不断循环，没有按照期望的执行结果，停在如图 3（c）里所示的 0001FFFFH 这个值。下面分析这个错误的仿真结果为什么会出现。

放大看，如图 4（b），这段循环是单周期测试 sc_instmem.mif 代码段第 10 行判断在满足 R5=0 条件下该跳转去第 12 行执行，但实际返回到前面执行第 8 行去了。跟在 720ns 之

后的两条指令 08000008、2005ffff（第 11,12 行指令）虽然已经被取出来，但是因为 720ns 的跳转而没有被执行，待之后第三条指令就接着执行跳转后的第 8 行 20a5ffff，R5 被-1，再经过三个周期流水后，R5 在 780ns 从 00000000 变为 FFFFFFFF。

造成错误的原因是跳转指令 beq $5, $0, shift【图中 4（b）中 oinst 信号在 720ns 时候 oinst=10a00001】引起的流水线填充问题，汇编代码没有做 bubble 处理。当我们对单周期测试代码在 beq 后面插入 nop 指令，并分析找到其他几处跳转指令以及因数据读取处理相关而易引起冲突的指令代码也插入 nop 一并优化后，这个问题得以解决。得到了如图 3（c）的和单周期 cpu 运行相同的程序执行结果，最后 R2=00000258 H。

图 4（c）中 800ns 处等同图 4（b）720ns 处指令，在以下的 sc_instmem.mif 代码 10 行的 beq、11 行的 j 后各添加一个 nop 指令，对应图 4（c）810ns 处，oinst 信号里插入了一个 bubble，即全 0 的指令，错误得以修正。修正后跳转到第 12 行 shift: addi $5, $0, -1 继续执行。830ns 时执行 000543c0，sll $8, $5, 15 指令，经过流水后 860ns 输出到 R8，R8 得到正确的值 FFFF8000H。

单周期测试 sc_instmem.mif 代码摘录：

```
 8 : 20a5ffff; % (20) loop2:     addi $5, $5, -1              # counter - 1 %
 9 : 34a8ffff; % (24)            ori $8, $5, 0xffff           # zero-extend: 0000ffff %
 A : 39085555; % (28)            xori $8, $8, 0x5555          # zero-extend: 0000aaaa %
 B : 2009ffff; % (2c)            addi $9, $0, -1              # sign-extend: ffffffff %
 C : 312affff; % (30)            andi $10, $9, 0xffff         # zero-extend: 0000ffff %
 D : 01493025; % (34)            or $6, $10, $9               # or: ffffffff %
 E : 01494026; % (38)            xor $8, $10, $9              # xor: ffff0000 %
 F : 01463824; % (3c)            and $7, $10, $6              # and: 0000ffff %
10 : 10a00001; % (40)            beq $5, $0, shift            # if $5 = 0, goto shift %
11 : 08000008; % (44)            j loop2                      # jump loop2 %
12 : 2005ffff; % (48)     shift:  addi $5, $0, -1              # $5 = ffffffff %
13 : 000543c0; % (4c)            sll $8, $5, 15               # <<15 = ffff8000 %
```

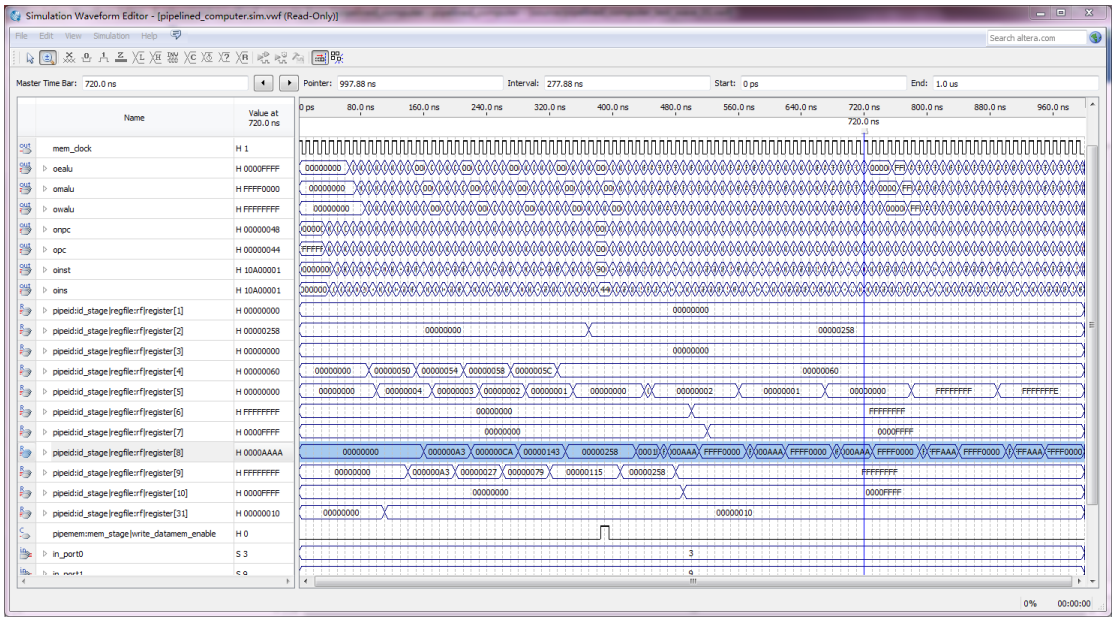


图 4（a）　流水线 CPU 的仿真运行输出-一个错误的仿真结果例子

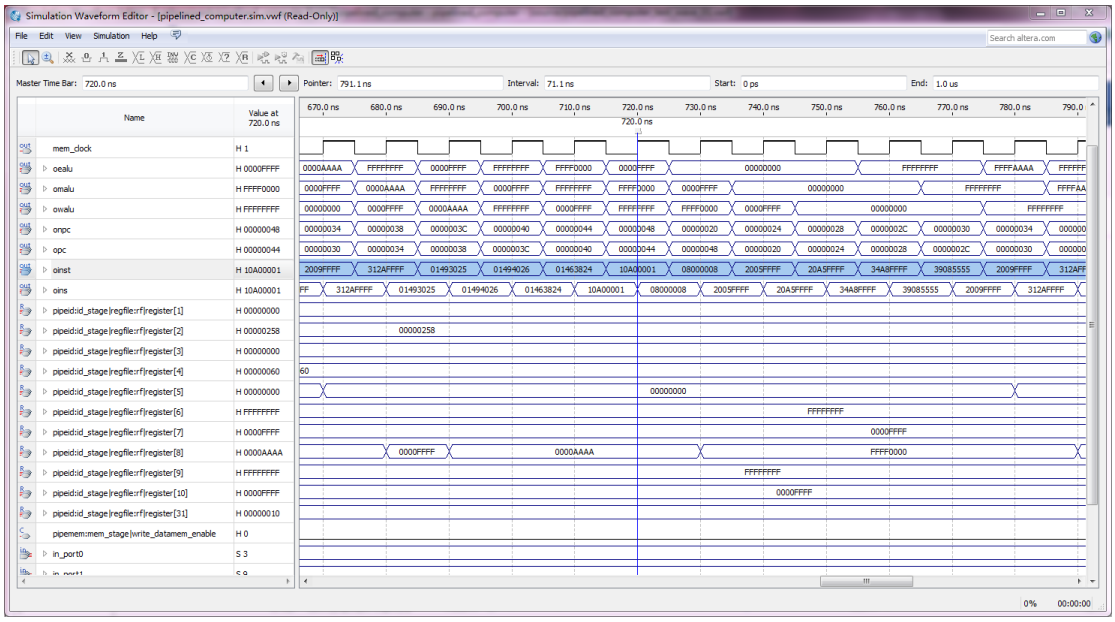


图 4（b） 流水线 CPU 的仿真运行输出-一个错误的仿真结果例子出错点

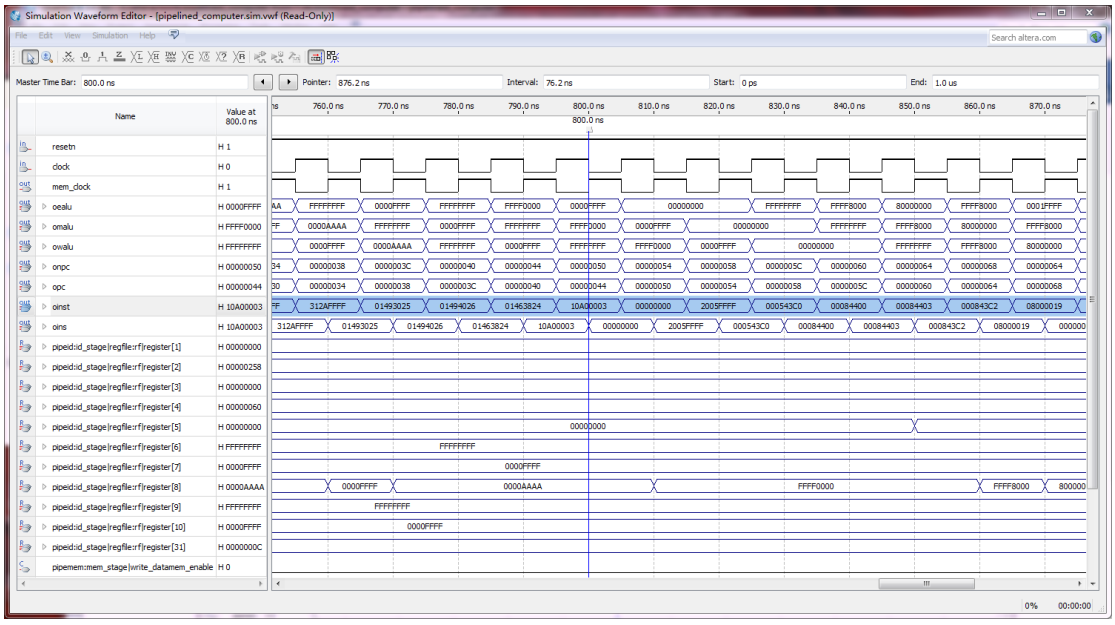


图 4（c） 流水线 CPU 的仿真运行输出-修正错误后的仿真结果出错点对比

图 4 (d) 所示为执行实验二的 IO 测试程序代码的输出结果。该段程序的.mif 文件如下：

```
DEPTH = 32;                 % 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 : 20010080;               % (00) main:    addi $1, $0, 128 # outport0, inport0                    %
```

```
1 : 20020084;          % (04)          addi $2, $0, 132 # outport1, inport1                  %
2 : 20030088;          % (08)          addi $3, $0, 136 # outport2                             %
3 : 8c240000;          % (0c) loop:    lw     $4, 0($1)     # input inport0 to $4             %
4 : 8c450000;          % (10)          lw     $5, 0($2)     # input inport1 to $5             %
5 : 00853020;          % (14)          add    $6, $4, $5    # add inport0 with inport1 to $6 %
6 : ac240000;          % (18)          sw     $4, 0($1)     # output inport0 to outport0      %
7 : ac450000;          % (1c)          sw     $5, 0($2)     # output inport1 to outport1      %
8 : ac660000;          % (20)          sw     $6, 0($3)     # output result to outport2       %
9 : 08000003;          % (24)          j loop               # repeat                          %
END ;
```

从图 4（d）可见，第一条写 80H 到 R1 的指令从 40ns 开始执行，经过流水第五个 clock 后 R1 改变，110ns 时候 inport0 的值 1 被读入 R4，120ns 时候 inport1 的值 2 被读入 R5，130ns 没有动作，140ns 时才将求和结果输出到 R6。这正是 100ns 时候流水线 CPU 为 lw 指令处理硬件上插入了一个 bubble 引起的。(如果硬件上不做处理，要得到正确结果就需要在第 4 行指令后添加 nop，产生一个 bubble。) 160ns 时 R6 里的求和结果 3 被输出到 outport2 端口。在 140ns 时执行跳转指令，循环执行第 3-9 行的 loop 代码段。

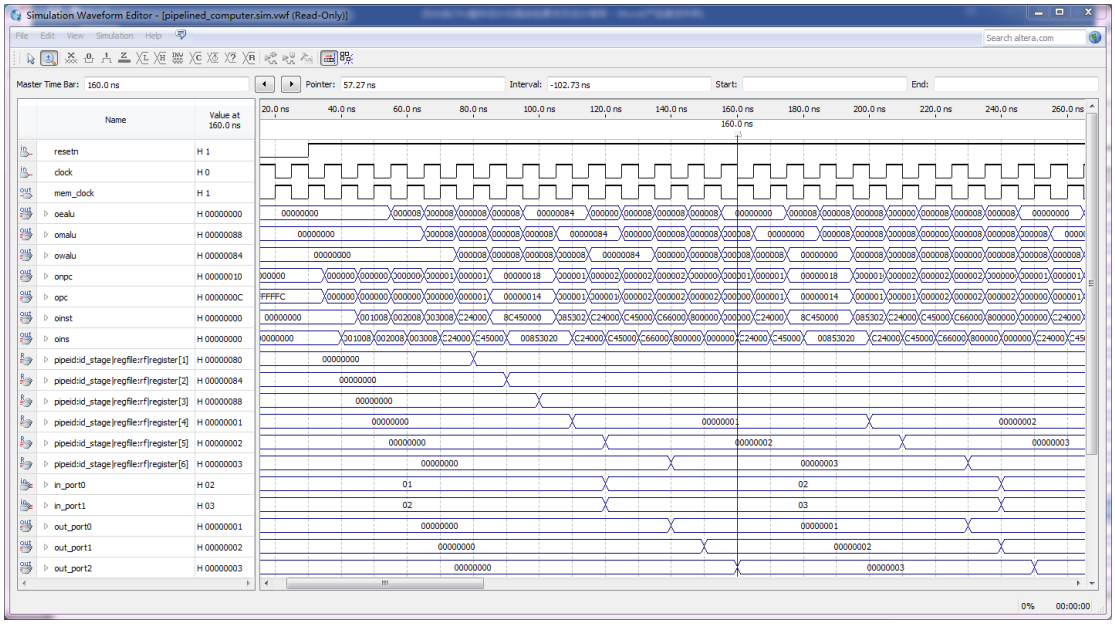
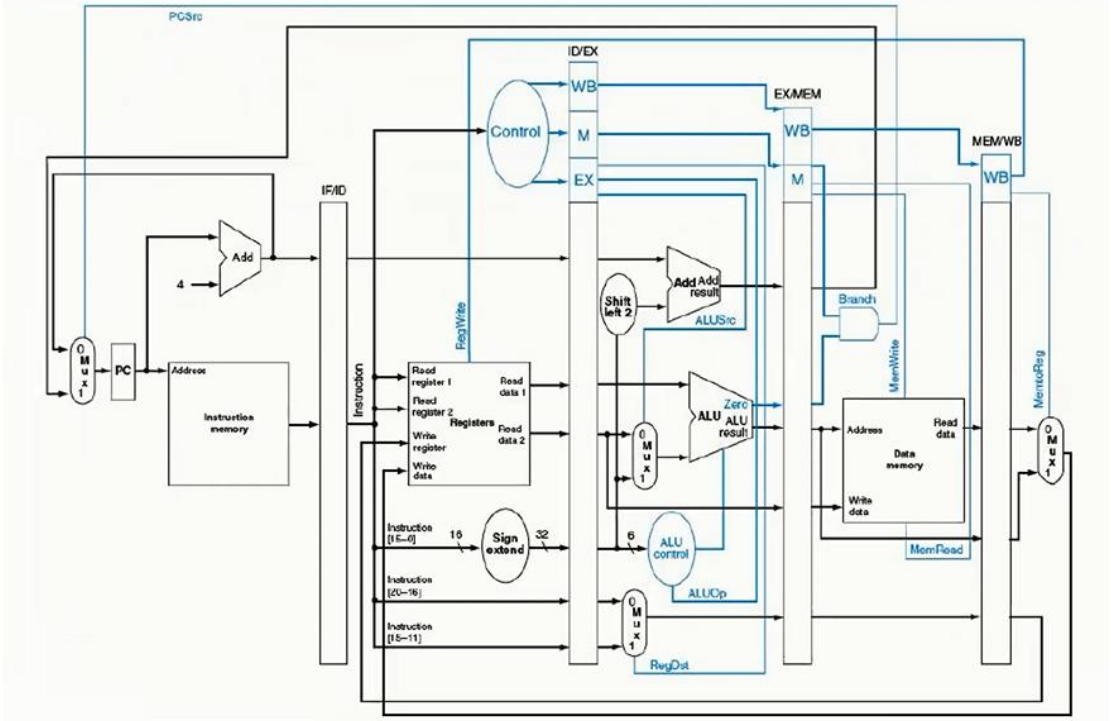


图 4（d）执行实验二的 IO 测试程序代码的输出结果细节

六、附录

摘取自课堂讲义（Chap.4 The Processor 之 4.5 An Overview of Pipelining）的关键点原理示意结构图如下，供参考。更详细的说明请参考课本相关内容及讲义。

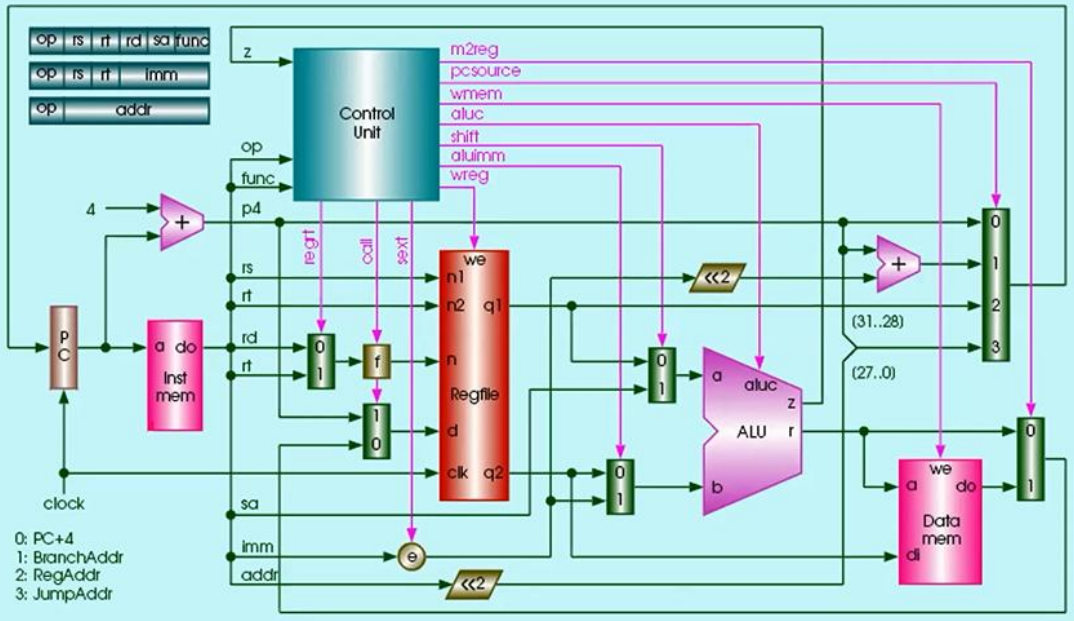
含有控制信号的流水线数据通路原理简图



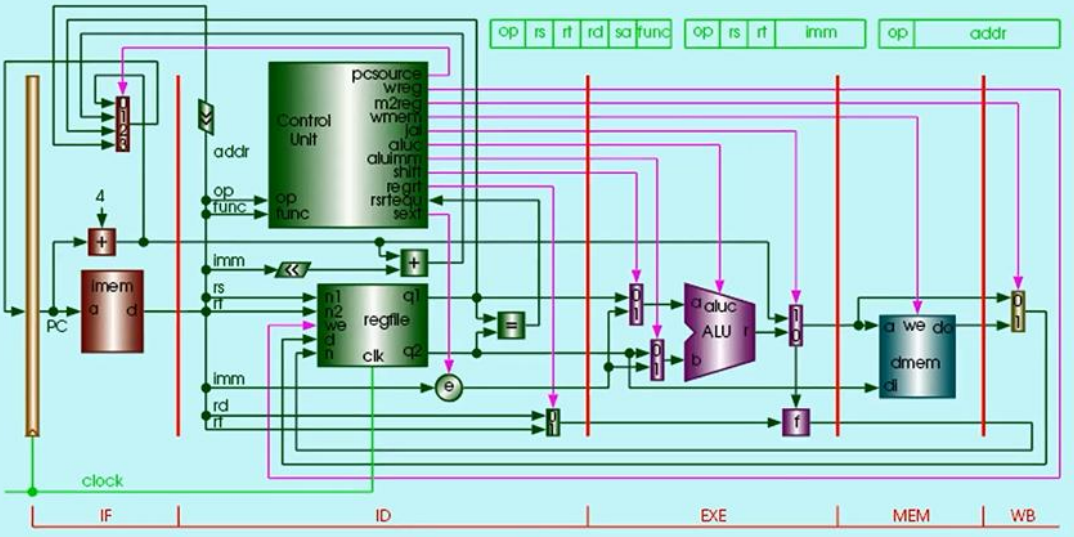
控制信号在各流水段中的分布（请参考前图6-22）

指令	EX段				MEM段			WB段	
	RegDst	ALUOp1	ALUOp2	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
R型	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

単周期CPU

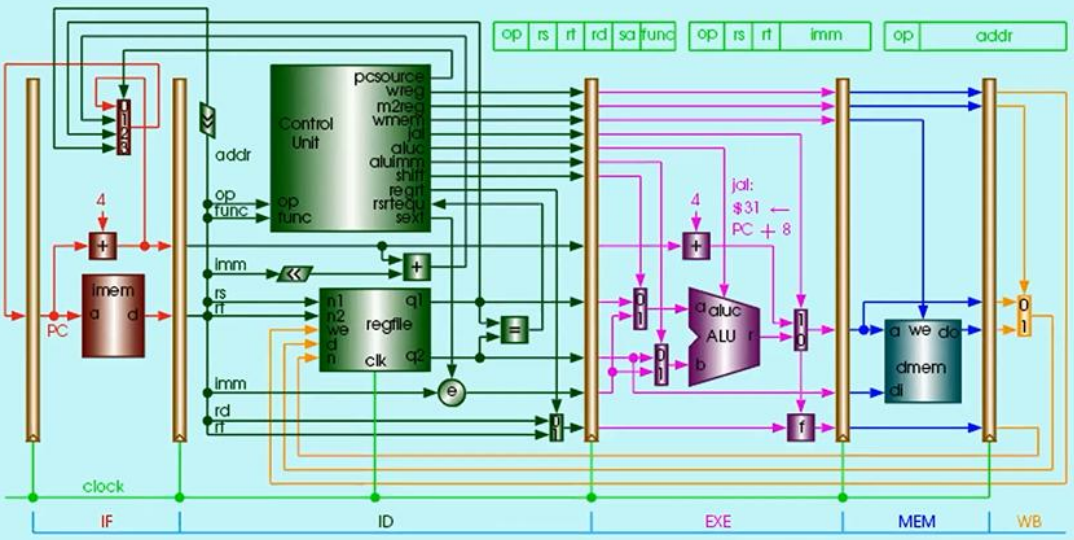


単周期CPU



分割成級 (Stage)

5级流水线CPU



級與級之間插入流水綫寄存器