

五段流水线 CPU 设计报告

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。

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 I/O 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。

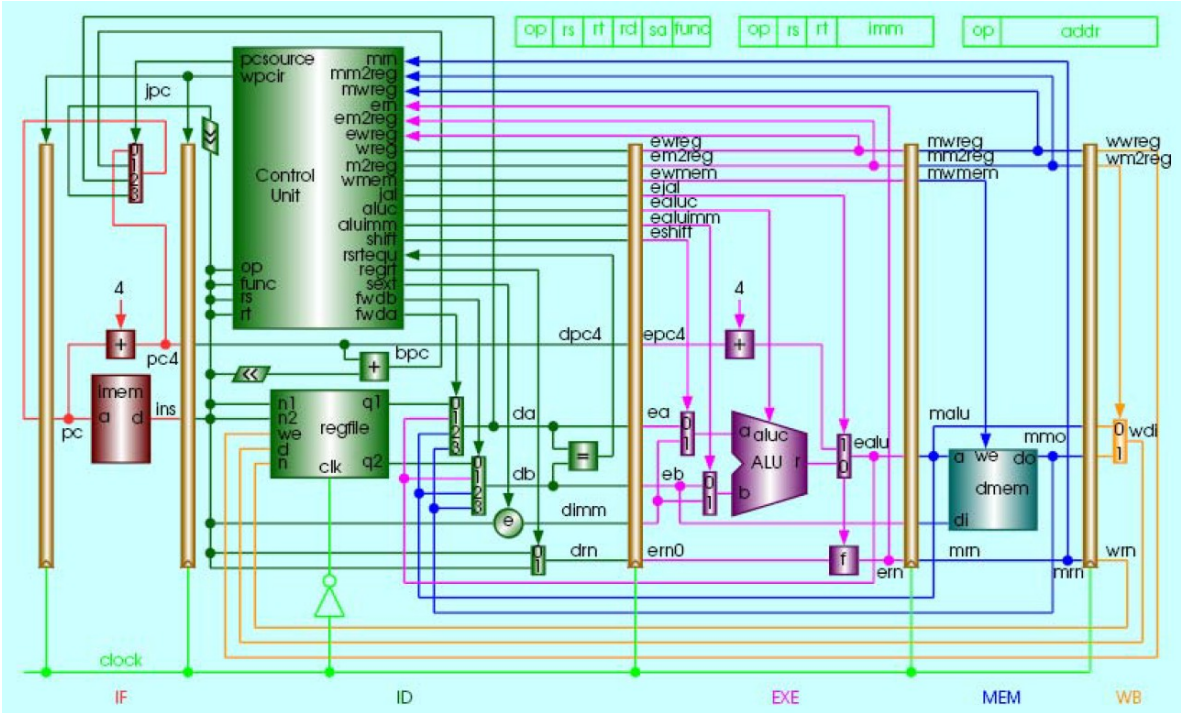
实验步骤及原理

本次实验可以分为以下几个步骤：初步理解流水线、设计流水线寄存器、设计流水段、解决数据冒险、解决控制冒险、指令无效化处理。

初步理解流水线

由于本次实验采用的是随实验指导书给出的顶层设计代码，因此初步的对流水线的构建基于这份顶层设计代码展开。

首先需要解决的问题是理解顶层代码的变量名含义。这里我在理解时参照了参考材料 2 这本书，以及上课时课件中的一张图。



在理解了变量名的含义后，我对变量名进行了改写，使之更加容易理解。同时我根据顶层模块中的端口信号分配方式推断出了每一个流水段或者流水线寄存器的输入、输出信号，完成了每一个模块端口列表的编写。

设计流水线寄存器

流水线寄存器的功能较容易设计：

- 它们都只包含一个 `always` 代码块用于对时钟沿输入或者重置信号做出响应。
- 输入是上一级需要流入下一级的信号，输出则由锁存了这些信号的寄存器单元提供。
- 当遇到时钟上升沿时，寄存器单元所存输入信号。
- 当重置信号有效时，将所有寄存器单元重置为 0。

以 MEM/WB 流水线寄存器为例，功能代码如下：

```
1  always @ (posedge clock or negedge resetn)
2  begin
3      if (resetn == 0) begin
4          WB_wreg <= 0;
5          WB_m2reg <= 0;
6          WB_mem_out <= 0;
7          WB_alu <= 0;
8          WB_write_reg_number <= 0;
9      end else begin
10         WB_wreg <= MEM_wreg;
11         WB_m2reg <= MEM_m2reg;
12         WB_mem_out <= MEM_mem_out;
13         WB_alu <= MEM_alu;
14         WB_write_reg_number <= MEM_write_reg_number;
15     end
16 end
```

稍微特殊的是 PC 和 IF/ID 流水线寄存器。这两个寄存器都需要有一个额外的输入信号 `wpcir`，用于处理 1w 发生数据冒险、导致流水线必须停顿一个时钟周期的问题。其为高有效，有效时的效果是：

- PC 和 IF/ID 寄存器停止更新（除非是被重置）。
- 在 ID 段插入一条 NOP 指令，或者做一些等价的操作，如通过修改控制信号使 ID 段指令无效化。








效果之二会放在“解决控制冒险”一节讨论。对于效果之一，只需要在 `always` 块中加入一个判断条件即可。以 PC 为例，功能代码如下：

```
1  always @ (negedge resetn or posedge clock)
2  begin
3      if (resetn == 0)
4          pc <= -4;
5      else if (wpcir == 0)
6          pc <= next_pc;
7  end
```

流水线寄存器的模块设计文件均以 `pipe_reg` 开头，后面的两个流水段名称标识该寄存器位于哪两个段之间。

设计流水段

这部分的重点在于：将原有单周期 CPU 的各个功能模块“拆解”，放到合适的流水线段当中去，并且做适当的调整，以保证各个流水线段之间相对独立。在我的设计中，架构如图所示。

 Cyclone II: EP2C70F672C6
▼  sc_computer 
 pipe_reg_EXE_MEM:EXE_MEM_reg
▼  pipe_stage_EXE:EXE_stage
 alu:al_unit
 mux2x32:alu_a
 mux2x32:alu_b
 mux2x32:link
 pipe_reg_ID_EXE:ID_EXE_reg
▼  pipe_stage_ID:ID_stage
 sc_cu:control_unit
 mux4x32:fwd_mux1
 mux4x32:fwd_mux2
 pipe_fwd_controller:pfc
 regfile:register_file
 mux2x5:write_reg_number_mux
 pipe_reg_IF_ID:IF_ID_reg
▼  pipe_stage_IF:IF_stage
>  lpm_rom_irom:irom
 mux4x32:next_pc_mux
 pipe_reg_MEM_WB:MEM_WB_reg
▼  pipe_stage_MEM:MEM_stage
>  lpm_ram_dq_dram:dram
 mux2x32:io_data_mux
>  io_input:io_input_reg
 io_output:io_output_reg
 mux2x32:WB_stage
 pipe_reg_PC:prog_cnt

在下面的描述中并不会给出具体的代码，因为大部分代码都可以直接沿用自单周期 CPU。

流水段的模块设计文件均以 `pipe_stage` 开头，后面的流水段名称标识该文件实现的是哪一个流水段。

IF 段的设计

IF 段进行的是取指令操作，所承担的功能包括将指令从指令存储器中取出，以及计算下一条指令的地址。这两个功能分别使用了一个 ROM 模块和一个四路多路器模块实现。多路器模块根据 `pcsource` 控制信号决定下一条指令的地址。

值得注意的是，由于流水线的特殊性质，下一条指令的计算需要在 ID 段完成，这也就导致了控制冒险的发生。

ID 段的设计

ID 段可以算是整个流水线设计中最为复杂的一个模块。它不仅承担了译码的职责，还需要将流水线后半部分可能用到的所有信号传递给 ID/EXE 寄存器。同时，后续还将要在这一段解决数据冒险和控制冒险。

就译码而言，可以沿用单周期 CPU 的 `sc_cu` 模块。而由于产生 `pcsource` 这一控制信号，我们需要在这一段就知道 `beq`、`bne` 两条指令所用到两个寄存器上值的比较结果。因此，将寄存器堆也放在这一段，且采用组合逻辑读寄存器堆、时序逻辑写寄存器堆的方式。

除了产生控制信号外，在 ID 段还需要计算出跳转地址，以传递给 IF 段使用，同时还需要将立即数、移位量计算出来。

和之前一样，对冒险的处理会在后面几节介绍。

EXE 段的设计

EXE 段的主要工作是完成计算，这一部分可以沿用单周期 CPU 的 ALU 模块。同时还需要额外的几个多路器以根据控制信号选择出正确的操作数。

EXE 段的另一个工作是处理 `jal` 指令。对于 `jal` 指令而言，在 ID 段译码出的写目标寄存器号可能不是 31，因此需要在 EXE 段将其设定为 31。这也是我们设置 `jal` 这一控制信号的原因。

MEM 段的设计

MEM 段基本承接自单周期 CPU 的 `sc_datamem` 模块。这里为了可拓展性，沿用的是实验二的带 I/O 端口的 `sc_datamem` 模块。

在这一段中需要注意写使能信号的处理以及写时钟的设置。由于已经留给了信号半个时钟的传输时间，因此可以直接将 `wmem` 作为写使能信号。

WB 段的设计

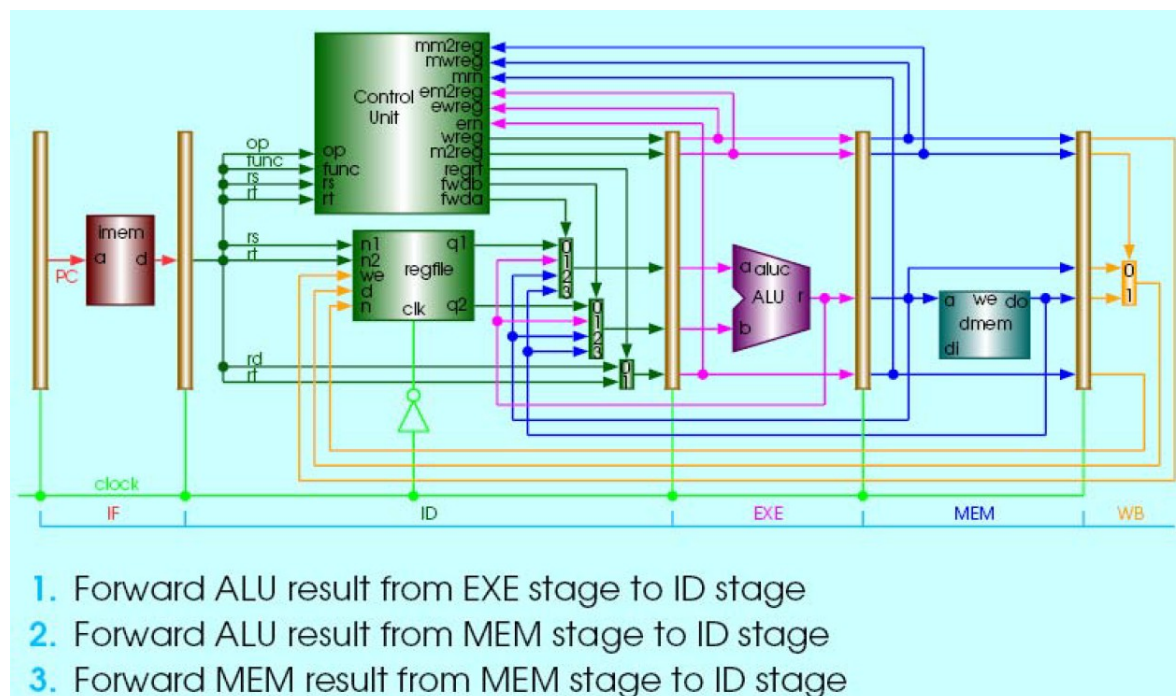
WB 段已经在顶层文件中给出，它的功能就是将 ALU 的运算结果或者从数据存储器中读出的结果写入寄存器堆。

解决数据冒险

现在来解决数据冒险问题。数据冒险一方面要处理直通，另一方面要处理无法直通时的停顿。

直通处理

在处理这一部分时，我参照了课件所给出的提示，如图所示。



可以看出，直通有三个来源：在 EXE 段指令的运算结果、在 MEM 段指令的运算结果、在 MEM 段指令从数据存储器读到的结果。加上原有的从寄存器堆读到的值，总共有四个来源。

因此，我设计了一个直通处理器 `pipe_fwd_controller`，放在 ID 段用于处理直通。该处理器产生两个控制信号 `fwd_q1_sel` 和 `fwd_q2_sel`，以让四路多路器确定 ID 段输出的 `q1`、`q2` 两个运算数的来源。

- 当 `fwd_q1_sel` 为 0 时，选择原始输入（即寄存器堆读到的值）。
- 当 `fwd_q1_sel` 为 1 时，选择在 EXE 段指令的运算结果。
- 当 `fwd_q1_sel` 为 2 时，选择在 MEM 段指令的运算结果。
- 当 `fwd_q1_sel` 为 3 时，选择在 MEM 段指令从数据存储器读到的结果。

`fwd_q2_sel` 类似。

该直通处理器的核心功能代码如下。

```
1  always @ (*)
2  begin
3      fwd_q1_sel = 2'b00;
4      if (EXE_wreg && (EXE_m2reg == 0) &&
5          (EXE_write_reg_number != 0) && (EXE_write_reg_number == ID_rs))
6          fwd_q1_sel = 2'b01;
7      else if (MEM_wreg &&
8          (MEM_write_reg_number != 0) && (MEM_write_reg_number == ID_rs))
9          fwd_q1_sel = {1'b1, MEM_m2reg};
10
11     fwd_q2_sel = 2'b00;
12     if (EXE_wreg && (EXE_m2reg == 0) &&
13         (EXE_write_reg_number != 0) && (EXE_write_reg_number == ID_rt))
14         fwd_q2_sel = 2'b01;
15     else if (MEM_wreg &&
16         (MEM_write_reg_number != 0) && (MEM_write_reg_number == ID_rt))
17         fwd_q2_sel = {1'b1, MEM_m2reg};
18 end
```

直通处理器的设计文件为 `pipe_fwd_controller.v`。

停顿处理

正如课上学到的一样，直通无法解决所有问题。当 `1w` 指令要写入的寄存器，在下一条指令就要被读取时，必须将流水线停顿一个周期以保证不出现错误。

我们希望尽早地发现这个问题。而最早可以在 ID 段发现这一问题，即通过译码发现 ID 段的指令要读 EXE 段指令的写目标寄存器，且 EXE 段指令恰为 `1w`。因此将这部分停顿处理也放在 ID 段进行。

我们使用 `wpcir` 作为标识信号。其为高时表明出现了这样的情况。它的判断标准就如上面叙述的那样：

- EXE 段的指令要读数据存储器，且要写寄存器。
- ID 段的指令要读寄存器，且读的寄存器与 EXE 段要写的一致。

两条条件满足时设定 `wpcir` 为高。这部分的代码如下所示。


```

1 wire rs_read_reg, rt_read_reg;
2 assign rs_read_reg = i_add | i_sub | i_and | i_or | i_xor | i_jr | i_addi |
  i_andi | i_ori | i_xori | i_lw | i_sw | i_beq | i_bne;
3 // 为 1: 本条指令读取 rs 寄存器
4 assign rt_read_reg = i_add | i_sub | i_and | i_or | i_xor | i_sll | i_srl |
  i_sra | i_sw | i_beq | i_bne;
5 // 为 1: 本条指令读取 rt 寄存器
6 assign wpcir = EXE_wreg & EXE_m2reg & (EXE_write_reg_number != 0) &
  ((rs_read_reg & (EXE_write_reg_number == ID_rs)) | (rt_read_reg &
  (EXE_write_reg_number == ID_rt)));
7 // 具有这种特征的都需要停下。
8 // 目前只有 lw 导致的停顿。后面可能会有别的

```

`wpcir` 有效时不仅要停止 PC 和 IF/ID 运转，同时也要使 ID 段指令无效化，也就是让它在流水线上继续运行时不会产生副作用。下面我们就讨论如何实现这一点。

解决控制冒险

在这种流水线设定下，控制冒险只会在发生跳转时，即 `beq`、`bne`、`j`、`jr`、`jal` 几条指令运行时出现。

这里我们通过无效化跳转指令后的一条指令来解决控制冒险。和数据冒险时类似，为了减少成本，将是否发生跳转判定放在 ID 段。

这里没有采用延迟槽，因此 `jal` 指令出现时，返回的指令将是原 PC + 4。

我们用 `bubble` 表示这几条指令是否出现。`bubble` 的计算非常简单：如果 `pcsource` 不为 0，那么这几条指令就出现了。如下面代码所示。

```

1 assign ID_bubble = |pcsource;

```

为了稳定性考虑，将 `bubble` 信号保留至 ID/EXE 寄存器中再进行指令无效化的处理。

指令无效化

停顿了 PC 和 IF/ID 寄存器，还需要将 ID 段的指令无效化，防止其产生副作用。

用 `nostall` 表示是否需要将 ID 段的指令无效化。其为高时表示不需要。它的产生如下。

```

1 wire nostall;
2 assign nostall = ~(wpcir | EXE_bubble);

```

怎么进行无效化处理呢？一种方法是把所有的控制信号都和 `nostall` 做按位与，这样一旦 `nostall` 为低就不会有任何有效的控制信号产生，从而避免了副作用。还有一种成本更低的方法是按照上面的方法，只处理 `wmem`、`wreg`、`jal` 三种信号，也可以达到相同的效果。

这里的设计采用的是上述第二种方法，详细见 `SC_CU.v`。

波形仿真验证

使用实验一的测试数据

将实验一的指令存储器、数据存储器复制过来后，运行波形仿真。得到的波形图如图所示。

