武汉大学计算机学院
本科生课程实验报告

RISC-V CPU 的流水线实现

专 业 名 称 ： 计算机科学与技术

课 程 名 称 ： 计算机组成与设计课程设计

指 导 教 师 ： 龚奕利 副教授

By LTH

二○二三年四月

摘　　要

本实验的目的是使用 Verilog 语言设计一个支持 RV32I 中大部分指令的 RISC-V 流水线 CPU，使用 Modelsim 软件与示例汇编程序对该 CPU 进行仿真验证，并使用 Vivado 软件下载到 N4 FPGA 板上进行综合、实现和运行。

在此过程中熟悉上述语言、软件的使用；加深对计算机组成与设计课程所教授知识（特别是 CPU 中各模块的工作原理与相互联系）的理解；学习 EDA（Electronic Design Automation）技术；了解 SOC 系统。

关键词: RISC-V；流水线；CPU 设计

目　　录

1 概述 **1**

1.1 基本步骤 1

1.2 实现的指令 1

2 单周期 CPU 设计 **2**

2.1 控制器设计 2

2.2 数据通路设计 2

3 流水线设计 **4**

3.1 五阶段流水线的划分 4

3.2 前递单元设计 4

3.2.1 EX 数据冒险 4

3.2.2 MEM 数据冒险与加载-使用数据冒险 5

3.2.3 不同类型的数据冒险同时发生的情况 6

3.3 冒险检测单元设计 6

3.3.1 停顿的实现 6

3.3.2 控制冒险的处理 7

3.4 简化的动态分支预测（extra） 8

4 使用 FPGA 板进行测试 **9**

4.1 硬件描述 9

4.2 从仿真代码到 FPGA 代码 9

4.3 FPGA 板测试 9

5 实验中遇到的一些问题 **11**

5.1 模块输入输出端口对齐——重视编译时的警告 11

5.2 充分使用 Verilog 模块所提供的封装性 11

5.3 关于 PC 寄存器 11

结论 **12**

1 概述

1.1 基本步骤

本实验的大致步骤如下：

1. 初步实现一个单周期 CPU，使其能够支持 RV32I 中大部分指令。
2. 在单周期 CPU 的基础上划分阶段，将其初步改为一个流水线 CPU。
3. 对于流水线 CPU 可能出现的各种冒险（Hazard）情况，额外引入处理模块。
4. 将完成的流水线 CPU 下载到 FPGA 板上进行测试。

1.2 实现的指令

- 用于初始化寄存器的指令：lui, auipc
- 运算指令：add, addi, sub, xor, xori, or, ori, and, andi, srl, srli, sra, srai, sll, slli
- 比较指令：slt, sltu, slti, sltiu
- 访存指令：sb, sh, sw, lb, lh, lw, lbu, lhu
- 跳转指令：jal, jalr
- 条件分支指令：beq, bne, blt, bltu, bge, bgeu

2 单周期 CPU 设计

2.1 控制器设计

控制器的功能为按照当前指令，生成控制其它各部件的控制信号，具体如下表所示：

表 2.1 控制信号设计

信号名	位宽	描述
immctrl	5	控制立即数符号扩展
itype	1	是否为 i 型指令
jump	1	是否为跳转指令
jalr	1	是否为 jalr 指令
bunsigned	1	条件分支运算数 是否为无符号数
pcsrc	1	下一个 pc 的来源
aluctrl	4	alu 控制信号
alusrca	2	alu 源操作数 a 的来源
alusrcb	1	alu 源操作数 b 的来源
memwrite	1	内存写信号
memtoreg	1	写回信号
regwrite	1	寄存器写信号
branch	4	条件分支信号
swhb	2	内存写的控制 信号（位宽）
lwhbu	3	内存读的控制信号 （位宽与有无符号）

2.2 数据通路设计

单周期 CPU 在一个时钟周期内完成一条指令的执行，在时钟上升沿信号到达时，首先按照 PC 寄存器中储存的地址，从 I-MEM 中读入指令，完成解码后将各个字段传入控制器，控制器传出各个控制信号。而后各个部件按照控制信号进行

相应操作，完成指令。而后根据指令类型确定下一条指令的地址是现地址 +4 还是跳转地址，并更新 PC 寄存器。

寄存器堆（Registers）按照解码后的字段从对应寄存器中读出数据；再按照 regwrite 信号决定是否对寄存器进行写操作，同时写寄存器的数据是通过一个数据选择器按照 memtoreg 信号选择得到的。

在数据通路中，使用两个数据选择器依照 alusrca、alusrcb 选择输入 ALU 的操作数，ALU 按照 aluctrl 信号对两个操作数进行不同的运算，并输出结果。

对于数据内存（D-MEM）模块，数据通路将读内存地址、写内存地址以及写数据传至数据内存，数据内存按控制器生成的信号（memwrite）决定是否执行写操作。

3 流水线设计

3.1 五阶段流水线的划分

为了加速程序的运行，提高并行程度从而提升 CPU 效率，将一条指令的执行划分为以下五个阶段：

- IF：取指令、更新 PC
- ID：指令解码、读寄存器、立即数符号扩展
- EX：ALU 运算
- MEM：内存（D-MEM）操作
- WB：写回

需要注意的是，本次实验中所实现的 CPU 在 **EX 阶段**计算得到分支、跳转指令的目标地址。同时假设寄存器堆与数据内存在前半个时钟周期写、后半个时钟周期读，从而**通过硬件避免了结构冒险**。

3.2 前递单元设计

3.2.1 EX 数据冒险

考虑如下 RISC-V 指令序列：

```
add x1, x1, x2
add x1, x1, x3
```

此时发生了 EX 数据冒险，第二条指令在进入 EX 阶段之前，需要前一条指令在 EX 阶段的运行结果。即需要将 EX/MEM 流水线寄存器的数据前递至 ID/EX 流水线寄存器中。该前递的条件为：前一条指令为寄存器写指令；前一条指令的目标寄存器与后一条指令的源寄存器相同；前一条指令的目标寄存器不为 x0（x0 寄存器不可被修改），可用如下代码实现：

```
    if(EXMEM_regwrite && EXMEM_rd!=5'b00000 && EXMEM_rd==IDEX_rs1)
    begin rdata1_forwardingconsidered <= EXMEM_data; end
    if(EXMEM_regwrite && EXMEM_rd!=5'b00000 && EXMEM_rd==IDEX_rs2)
    begin rdata2_forwardingconsidered <= EXMEM_data; end
```

3.2.2 MEM 数据冒险与加载-使用数据冒险

考虑如下两组 RISC-V 指令序列：

add x1, x1, x2

addi x2, x2, 1

add x1, x1, x3

lw x1, 0(x0)

add x3, x1, x2

在第一种情况（MEM 数据冒险）中，触发了冒险的指令（第 3 条指令）在进入 EX 阶段之前需要第一条指令的结果，此时第一条指令位于 WB 阶段前，结果已在一个周期前运算完成。因而只需要将 MEM/WB 流水线寄存器中的数据前递至 ID/EX 流水线寄存器中即可。

而第二种情况更为复杂，被称为加载-使用冒险（load-use hazard）。第二条指令在进入 EX 阶段前需要上一条**加载指令**的结果，而上一条指令还未进入 MEM 阶段（即加载指令还未从内存中得到数据）。在这种情况下，不得不使加载指令之后的指令**停顿**（即将流水线中加载指令后的所有指令重复地在一个流水线阶段执行一次）一个时钟周期（并且**清空** EX/MEM 流水线寄存器中的内容，以清空后一条指令在 EX 阶段得到的错误的结果）。在停顿后，即转为第一种情况的 MEM 数据冒险，正常前递即可。在本节中，仅描述前递单元的实现，暂时略过停顿与清空的实现（具体见 3.3 节）。

总而言之，以上两种情况在前递单元中进行的操作是相同的，即将 MEM/WB 流水线寄存器中的数据前递至 ID/EX 流水线寄存器中。这种前递可用如下代码实现：

```
    if(MEMWB_regwrite && MEMWB_rd!=5'b00000 && MEMWB_rd==IDEX_rs1)
    begin rdata1_forwardingconsidered <= MEMWBwdata; end
    if(MEMWB_regwrite && MEMWB_rd!=5'b00000 && MEMWB_rd==IDEX_rs2)
    begin rdata2_forwardingconsidered <= MEMWBwdata; end
```

3.2.3 不同类型的数据冒险同时发生的情况

通常情况下，不同类型的数据冒险同时发生不会产生额外的问题，如第三条指令的两个源寄存器分别是上两条指令要写入的目标寄存器，但也有例外，考虑如下 RISC-V 指令序列：

```
add x1, x1, x2
add x1, x1, x3
add x1, x1, x4
```

可见第二条指令出发了 EX 数据冒险，不产生额外问题。但第三条指令所读取的 x1 寄存器的值应该是第二条指令运行的结果，而实际上第三条指令同时满足 EX 数据冒险与 MEM 数据冒险的条件。教材上的解决方案为：由于应选择更“近”的结果使用，故当同时满足 EX 数据冒险与 MEM 数据冒险的条件时，应视为 EX 数据冒险，所以需要在 MEM 数据冒险的条件中加入“不产生 EX 数据冒险”。但事实上，在 Verilog 实现中，**可简单地使用有顺序的 if-else if 语句实现，而无需额外加入判断条件**：

if *EX hazard conditions* **then**
    forward EX/MEM;
else if *MEM hazard conditions* **then**
    forward MEM/WB;
else forward nothing;

3.3 冒险检测单元设计

3.3.1 停顿的实现

出于希望前递模块仅涉及数据的传输，将所有异常的控制转移都由冒险检测单元实现的考虑，将停顿放到此处实现。在数据冒险中，只有加载使用冒险这一特殊情况需要将 CPU 流水线中的一部分停顿一个时钟周期，而这种停顿的实现事实上仅仅是**禁止 PC 寄存器、IF/ID、ID/EX 流水线寄存器被写入**，并**清空 EX/MEM 流水线寄存器将要被写入的错误数据**，从而使前几个流水线级重复地执行一次上一周期地操作，从而使未停留在停顿地流水线中的加载指令能够在额外的一周期时间内将下一条指令所需的数据从内存中加载出来。

因而，冒险检测单元仅需输出一个一位的停顿信号与一个一位的 EX/MEM

清空信号即可实现加载-使用冒险所需的停顿。其判断条件也十分简单：

```
    if(EXMEM_memtoreg && (EXMEM_rd==IDEX_rs1 || EXMEM_rd==IDEX_rs2))
        begin
        stall <= 1'b1;
        EXMEM_flush <= 1'b1;
        end
```

3.3.2 控制冒险的处理

由于 CPU 在 EX 阶段计算出分支、跳转指令的目标地址，此时后两条指令已在 CPU 的前两个阶段执行。因而，若确实要跳转时，需要清空不应被执行的指令。此时使用 flush 信号对相应流水线寄存器进行清空，并将 PC 寄存器赋值为跳转目标地址。

当 EX 阶段的指令为跳转指令（jal、jalr）时，由于必然会发生跳转，因而直接将前两阶段的流水线寄存器的 flush 信号置为 1，PC 寄存器置为目标跳转地址即可：

```
if(IDEX_jump == 1'b1) begin
    IFID_flush <= 1'b1;
    IDEX_flush <= 1'b1;
    case(IDEX_jalr)
        1'b0: nextpc <= jalpc; // jal
        1'b1: nextpc <= jalrpc; // jalr
    endcase
end
```

而当 EX 阶段指令为分支指令，则需要通过 ALU 的计算结果判断是否跳转，再进行操作，具体逻辑如下：

```
if(IDEX_branch != 4'b0000) begin
    if((IDEX_branch==4'b1000 && IDEX_zero)||(IDEX_branch==4'b0100 &&
        ~IDEX_lt)||(IDEX_branch==4'b0010 && IDEX_lt)||(IDEX_branch==4'
        b0001 && ~IDEX_zero))begin
        nextpc <= branchpc;
        IFID_flush <= 1'b1;
        IDEX_flush <= 1'b1;
    end
    else begin
        nextpc <= IF_pcplus4;
        IFID_flush <= 1'b0;
        IDEX_flush <= 1'b0;
    end
end
```

3.4 简化的动态分支预测（extra）

上述 CPU 设计在处理分支指令时，始终假设分支不发生。为减少预测错误带来的的开销，引入一个**简化的**全局动态分支预测策略。如下实现了一个 2 位（4 状态）的动态分支预测器，其状态转移图如下：

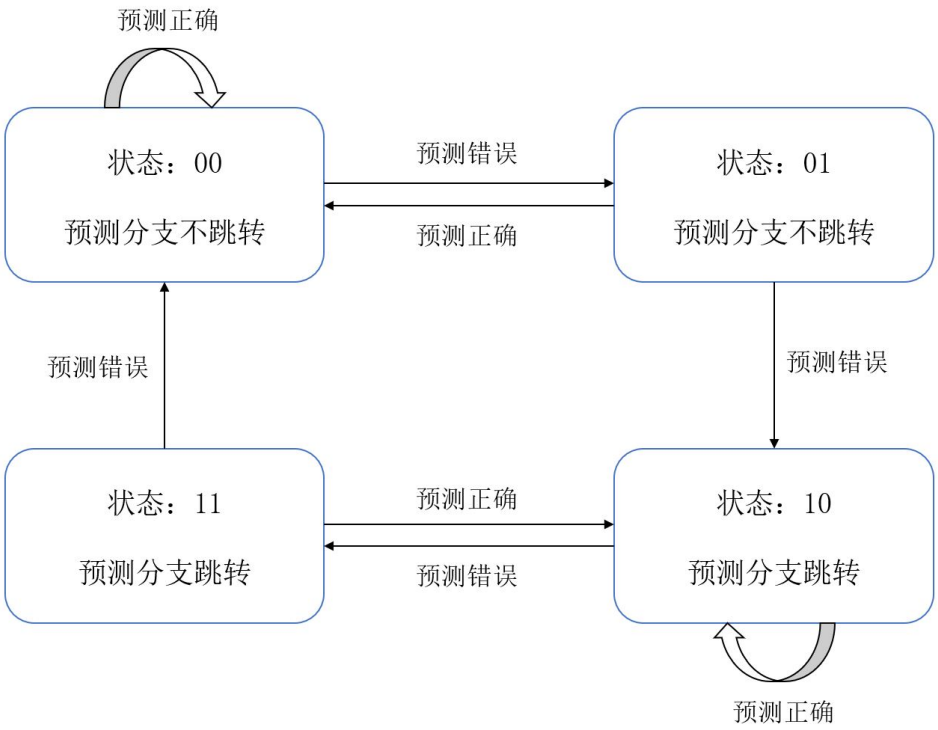

图 3.1　状态转移图

动态分支预测器在处理分支指令的 PC 寄存器更新时，按照该两位二进制的状态信号预测是否顺序执行。而该状态信号的更新由冒险检测单元控制。由于状态信号暗含了当前分支预测的决策，每当检测控制冒险的发生时，同时检查状态信号，并按照状态转移表更新该状态信号，从而实现动态分支预测。

需要注意的是，**上述动态预测策略并不完整**。在现实中，程序中的每个分支指令都对应一个独立的预测状态，并需要一个数组存储这些状态，而非仅仅存在在一个全局状态。

4 使用 FPGA 板进行测试

4.1 硬件描述

本次实验所使用的测试硬件为 Nexys 4 DDR，其搭载了 Xilinx® Artix™-7 FPGA 芯片, 并集成了 USB、以太网以及其它端口。

4.2 从仿真代码到 FPGA 代码

在 Vivado 中，使用 IP 核代替 I-MEM，所以需要将源代码中的 I-MEM 模块删除，将用于测试的汇编程序代码编译为 coe 文件，用以构建 IP 核从而代替指令内存。

在 CPU 实现中增加寄存器编号（Reg_sel）与数据 (Reg_data) 的管线以便在 FPGA 板的数码管上显示（用于验证 CPU 实现的正确性）；而后将 FPGA 测试模块（fpga_top.v）中调用的 CPU 模块名改为实现的 CPU 的顶层模块名，再仔细对照各个部件中所使用的管线名称，修改 FPGA 测试模块，使其一一对应。

检查 CPU 实现中的代码是否满足 FPGA 板的如下要求：

- 禁止出现 initial 语句
- 禁止出现 casex、casez
- 禁止使用 # 表达电路延迟
- 时钟信号 clock 只允许出现在 always @(posedge clock) 语句中

4.3 FPGA 板测试

使用 Vivado 软件导入约束文件，约束文件中描述了顶层电路的输入输出与 FPGA 芯片的 I/O 引脚之间的绑定关系。

在对上述修改后的代码进行综合（Synthesis）、实现（Implementation），并生成位流（Bitstream）并下载到 FPGA 板上后，即可以在 FPGA 板上进行测试。

约束文件中对 FPGA 板上的按键开关定义如下：

表 4.1　按键开关定义

按键开关编号							功能描述
sw15	sw5	sw4	sw3	sw2	sw1	sw0	
0	X	X	X	X	X	X	CPU 全速运行
1	X	X	X	X	X	X	CPU 慢速运行
X	0	0	0	0	0	1	指令编号（PC»2）
X	0	0	0	0	1	0	指令地址（PC）
X	0	0	0	0	1	1	指令
X	0	0	0	1	0	0	访问存储器或外设的地址
X	0	0	0	1	0	1	写入存储器或外设的数据
X	0	0	0	1	1	0	数据存储器读出的数据
X	0	0	0	1	1	1	数据存储器的访问地址
X	1	寄存器编号					对应寄存器的值
X	0	X	1	X	X	X	0xFFFFFFFF
X	0	1	X	X	X	X	0xFFFFFFFF

5 实验中遇到的一些问题

5.1 模块输入输出端口对齐——重视编译时的警告

由于数据通路、控制器等模块的输入输出端口较多，在修改时容易使端口错位。但好在相邻端口的位宽有时是不同的，故而在使用 ModelSim 编译时会报警告（Warning）。同时，端口错位引发的错误很难排查，往往会导致模拟或运行时出现令人疑惑的未定态信号，所以必须要重视编译时的警告。值得一提的是，由于 ModelSim 有时信息窗口较小，警告可能会被其它信息遮盖，故在编译时要注意具体是否有警告并查看。

5.2 充分使用 Verilog 模块所提供的封装性

从模拟电路到数字电路再到如今的 CPU 芯片，封装性是非常重要的特性，可以让设计者仅关注底层模块所提供的端口，忽略底层具体的实现。在本次实验中，模块提供了封装性，充分使用这一特点可以减少很多不必要的工作量以及错误。尤其是最后在将原代码改为能在 FPGA 板上运行的代码时，若没有充分利用封装性，将一部分功能在它应在的模块之外实现，就会需要很多额外的工作。因而，应当充分利用模块提供的封装性提升效率。

5.3 关于 PC 寄存器

我认为本次实验较为麻烦的部分之一是 PC 寄存器的更新，相较于其它一些指令导致的计算错误，PC 寄存器设置错误会导致程序的执行顺序被打乱。而跳转指令以及一些冒险的情况都会涉及对 PC 寄存器的写。因而，要仔细地处理更新 PC 寄存器的逻辑，并擅于使用 ModelSim 进行模拟调试，修正错误。

结　论

本次实验所完成的流水线化 RISC-V CPU 实现能够通过在线测试平台（https://cslabcg.whu.edu.cn/）提供的全部测试集（包括对指令运行正确性、数据冒险、控制冒险等的测试）、并能在 FPGA 板上正确运行示例程序。

在此次实验中，我熟悉了 Verilog 语言、ModelSim、Vivado、RARS 等软件的使用，学习了 LaTeX 语言、Overleaf 平台的使用，并第一次在一个项目中频繁地使用 git 工具、平台，同时对上一学期计算机组成与设计课中 CPU 设计部分有了更加深刻的认知。