流水线处理器实验报告

王艺坤 20CS

32 位流水线 MIPS 指令集 CPU，使用 SystemVerilog 编写。[1]

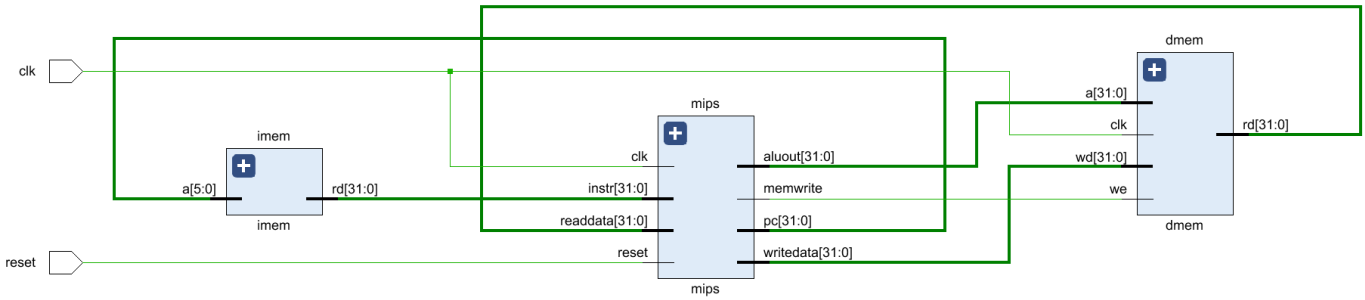
1. 指令集架构

同多周期处理器：

```
add     $rd, $rs, $rt               # [rd] = [rs] + [rt]
sub     $rd, $rs, $rt               # [rd] = [rs] - [rt]
and     $rd, $rs, $rt               # [rd] = [rs] & [rt]
or      $rd, $rs, $rt               # [rd] = [rs] | [rt]
slt     $rd, $rs, $rt               # [rd] = [rs] < [rt] ? 1 : 0
addi    $rt, $rs, imm               # [rt] = [rs] + SignImm
andi    $rt, $rs, imm               # [rt] = [rs] & ZeroImm
ori     $rt, $rs, imm               # [rt] = [rs] | ZeroImm
slti    $rt, $rs, imm               # [rt] = [rs] < SignImm ? 1 : 0
lw      $rt, imm($rs)               # [rt] = [Address]
sw      $rt, imm($rs)               # [Address] = [rt]
j       label                       # PC = JTA
beq     $rs, $rt, label             # if ([rs] == [rt]) PC = BTA
bne     $rs, $rt, label             # if ([rs] != [rt]) PC = BTA
nop                                 # No operation
```

2. 实现

2.0 总览



如图所示为流水线 MIPS CPU 的整体概览，与单/多周期 MIPS CPU 是一样的，区别在于 CPU 核心 mips 的实现。以下将仅介绍与单周期不同的部分。

2.1 mips

本流水线 CPU 的实现中，将 datapath 按照流水线的 5 个阶段划分为了 5 个模块（Fetch, Decode, Execute, Memory, Writeback），并增加了一个用于处理冲突的冲突单元（Hazard Unit）。其中各模块的作用如下[1]：

- Fetch：取指令阶段，从指令存储器中读取指令
- Decode：译码阶段，从寄存器文件中读取源操作数，并对指令译码以产生控制信号

- Execute：执行阶段，使用 ALU 执行计算
- Memory：存储器阶段，读写数据存储器
- Writeback：写回阶段，按需将结果写回到寄存器文件
- Hazard Unit：冲突单元，用于发现及处理数据冲突和控制冲突

方便起见，将 Fetch 阶段和 Decode 阶段之间的流水线寄存器命名为 decode_reg，并置于 Fetch 模块中，其余流水线寄存器同理。

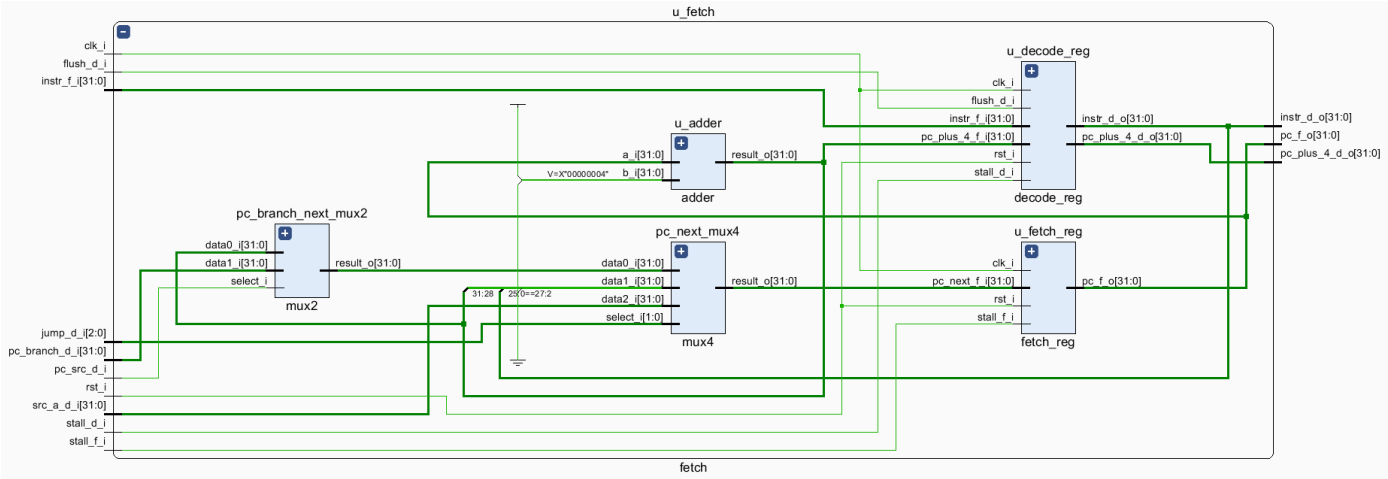
具体模块功能分析将在下文阐述。这里 mips 的作用就是将这些模块连接起来，其中相同名称的端口即连通。除此以外特殊的几条连线如下所示：

- mips 读端口 reset, instr, readdata 分别与 rst, instr_f, read_data_m 连通
- mips 写端口 aluout, memwrite 分别与 alu_out_m, mem_write_m 连通

了解了以上连接规则后，展示 mips 的完整总览图就不那么必要。比起大而繁乱的连线总览图，直接看代码甚至都更为直观。

代码详见 `code/Pipeline/mips.sv`

2.2 fetch

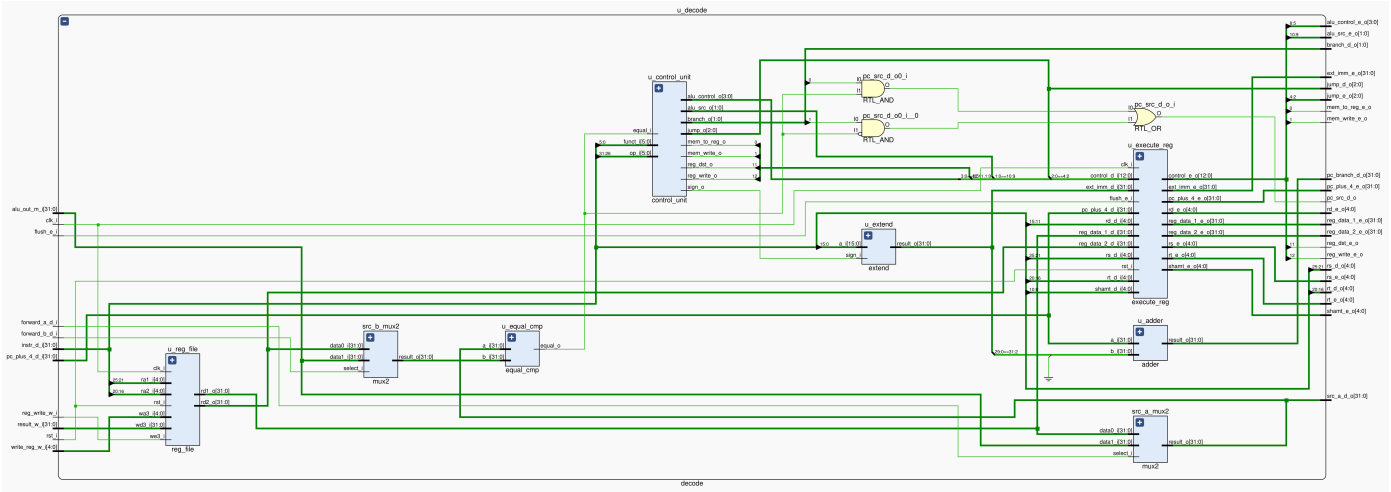


Fetch 阶段，通过 pc_f 输出指令地址 `pc` 到 imem，通过 instr_f 从 imem 读入指令 `instr`，存储到流水线寄存器 decode_reg 中，在下一个时钟上升沿到达时从 instr_d 输出。

此外，本阶段还需要完成 PC 的更新。`pc_next`（新的 PC 值）的选择逻辑同单周期实验报告第 2.8 节所述，这里就不再赘述了。需要注意的是 Fetch 阶段需要用到一些 Decode 阶段的数据，也就是上一条指令计算得到的相对寻址地址 `pc_branch_d`、用于指令 jr 跳转的地址 `src_a_d`、指令解析得到的 `pc_src_d`，`jump_d` 信号，用来确定 pc_next 的值。

在需要解决冲突的情况下，通过 `stall_f`，`stall_d`，`flush_d` 信号决定是否保持（stall）或清空（flush）对应流水线寄存器保存的数据，其中 `stall_f` 为 `1` 时保持当前 PC 值不更新，`stall_d` 为 `1` 时保持当前 decode_reg 的数据不更新，`flush_d` 为 `1` 时清空 decode_reg 的数据。具体这些信号在何时为何值，将在 hazard_unit 章节详细阐述，下同。

2.3 decode



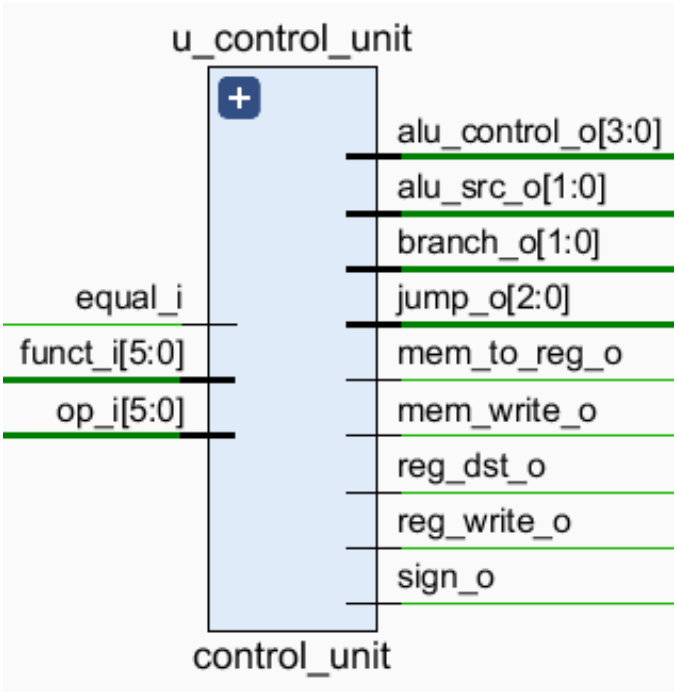
Decode 阶段，读入指令 `instr_d`，由控制单元 control_unit 解析，决定各个控制信号。此外，本阶段还需要完成相对寻址地址 `pc_branch_d` 的计算，然后交给下一条指令的 Fetch 阶段。

作为**静态分支预测**，本阶段新增了比较器 equal_cmp，用来比较从寄存器中读出的两个数 `src_a`，`src_b` 是否相等，其作用是将指令 beq, bne 的比较过程提前到 Decode 阶段，提前得到 `pc_src` 信号，从而提高效率。这里需要用到 Memory 阶段的数据 `alu_out_m` 以应对数据冒险，`src_a`，`src_b` 取值的选择由 `forward_a_d`，`forward_b_d` 信号控制。

在实现中，将寄存器文件 reg_file 放在了 decode 模块里，因此 Writeback 阶段的寄存器写入操作也将在这里完成。所以这里需要用到一些 Writeback 阶段的数据，也就是 `reg_write_w` 信号、目标寄存器 `write_reg_w`、写入数据 `result_w`。

在需要解决冲突的情况下，通过 `stall_e`，`flush_e` 信号决定是否保持或清空 execute_reg 保存的数据。

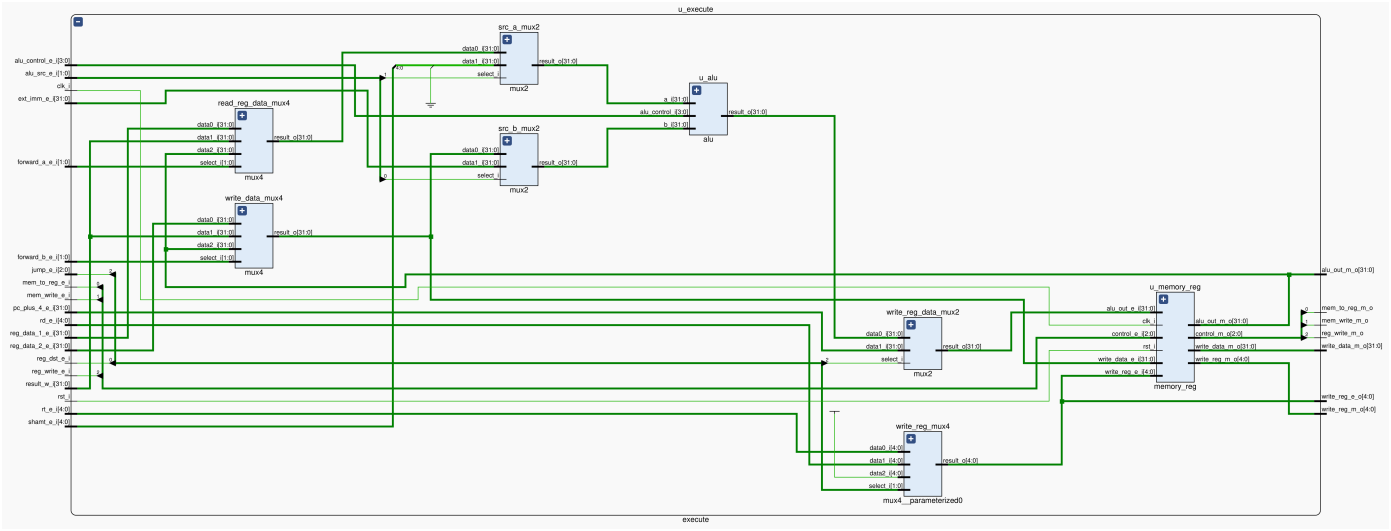
2.3 control_unit



流水线版本中，control_unit 新增了一个控制信号 `sign`，用于控制对立即数进行符号扩展还是无符号扩展，解决了新增测试样例 i-type 中遇到的一些问题。此外为了调试方便，本实现中将控制信号中的无关项 `x` 都改成了 `0`，其余同单周期版本。主译码器中，新增信号的真值表如下：

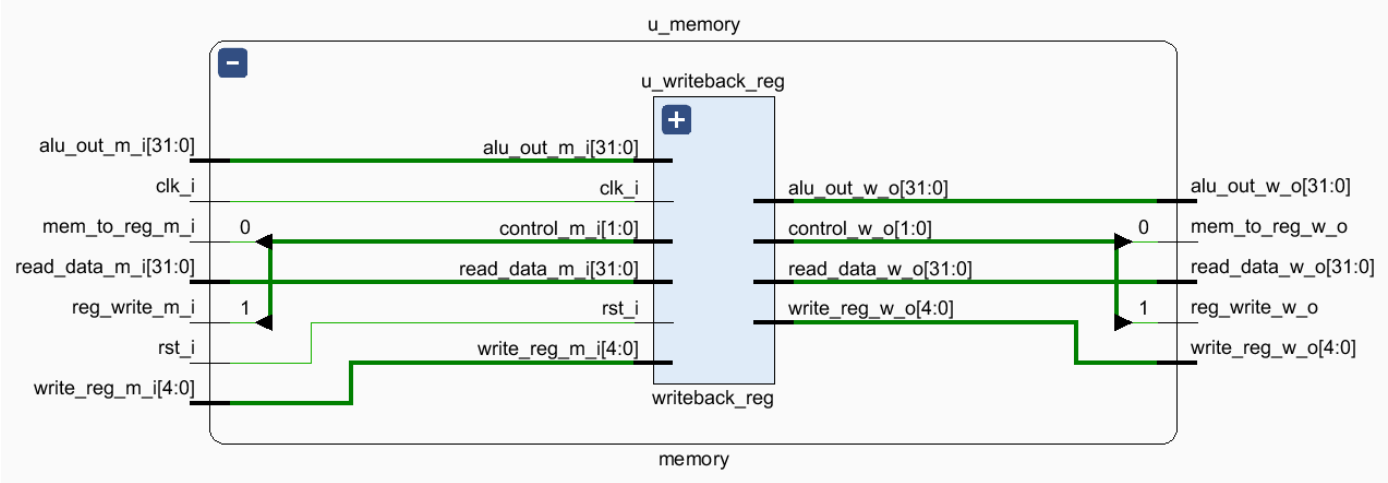
指令	opcode	sign
R-type	000000	0
addi	001000	1
andi	001100	0
ori	001101	0
slti	001010	1
lw	100011	1
sw	101011	1
j	000010	0
jal	000011	1
jr	001000	0
beq	000100	1
bne	000101	1

2.4 execute



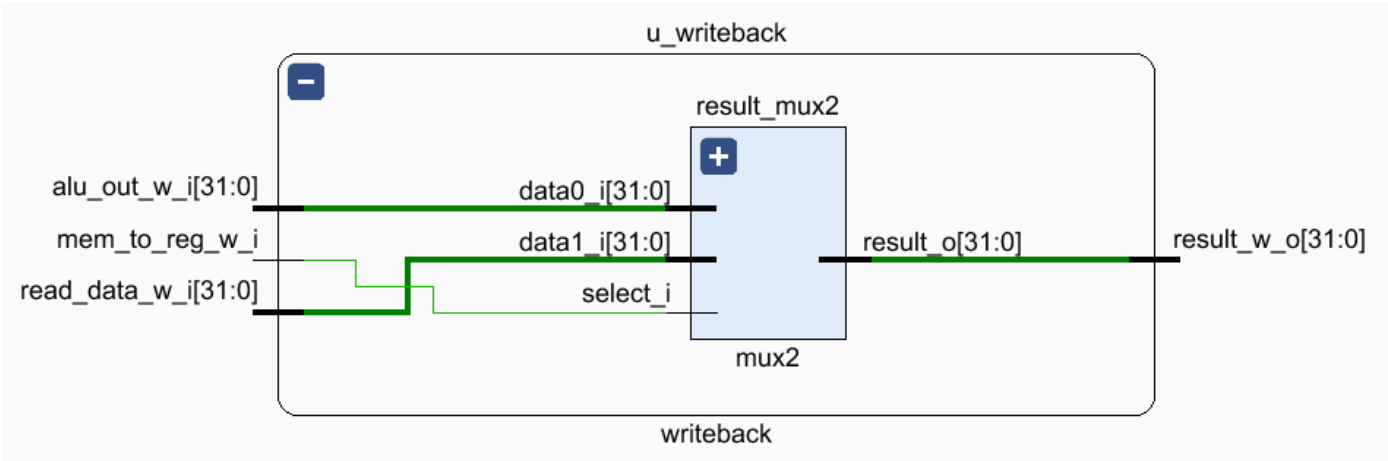
Execute 阶段，对操作数 `src_a`, `src_b` 使用 ALU 执行计算。在单周期版本的基础上，新增了两个 mux4 用于转发逻辑。这里需要用到 Memory 阶段的数据 `alu_out_m` 和 Writeback 阶段的数据 `result_w` 以应对数据冒险，`read_reg_data`, `write_data`（分别为通常情况下的 `src_a`, `src_b`）取值的选择由 `forward_a_e`, `forward_b_e` 信号控制。

2.5 memory



Memory 阶段，当 `mem_write` 为 `1` 时，在 dmem 的目标地址 `alu_out` 存储需要写入的数据 `write_data`。不过实际上这件事情并不是在 memory 模块内完成的，因为 dmem 在 mips 外面。因此实现中是在 execute 模块通过 mem_write_m, alu_out_m, write_data_m 将 `mem_write`, `alu_out`, `write_data` 直接输出到 dmem，在下一个时钟上升沿到达时（即 Memory 阶段）写入 dmem。memory 模块内则是通过 read_data_m 从 dmem 读入数据 `read_data`。

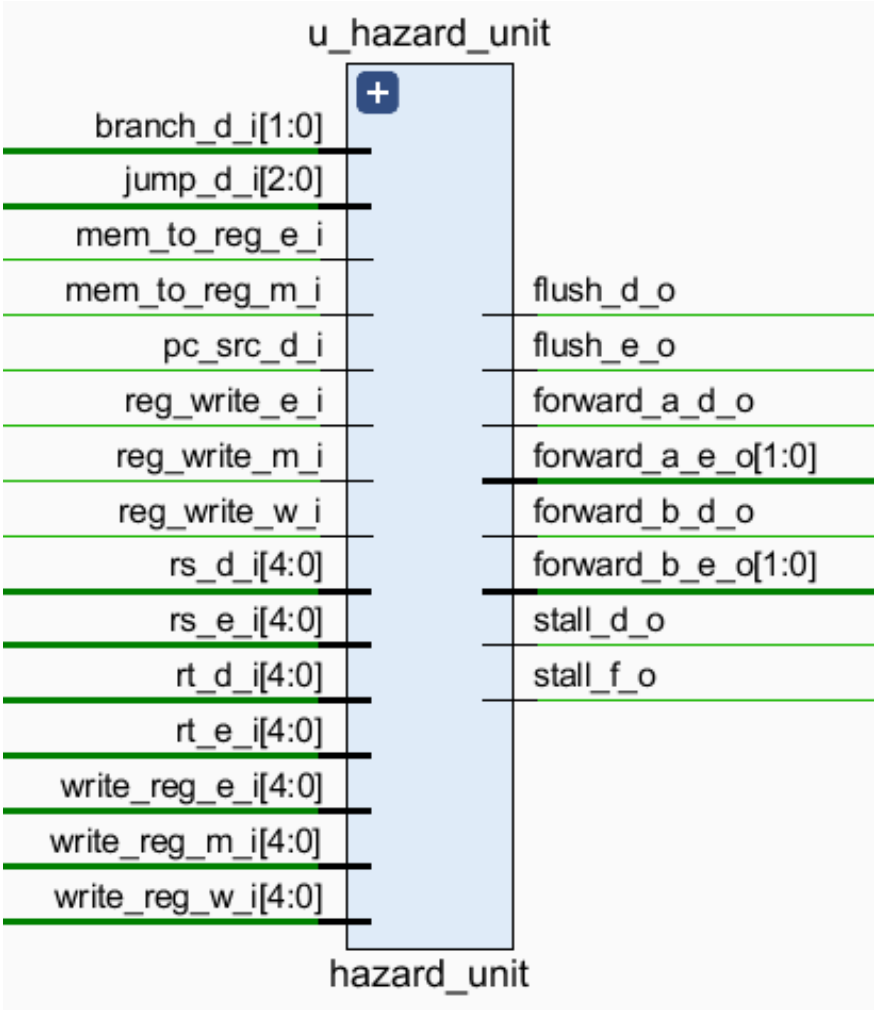
2.6 writeback



Writeback 阶段，由 `mem_to_reg` 信号控制 result_mux2 选择写入 reg_file 的数据为 `alu_out` 还是 `read_data`。

写入逻辑放在了 decode 模块，参见第 2.3 节。

2.7 hazard_unit



冲突单元根据传入的各阶段寄存器和控制信号，检查是否存在数据冲突或控制冲突，并输出相应的控制信号（forward, stall, flush）以处理冲突。

2.7.1 数据冲突

当一条指令依赖于另一条指令的结果，而此结果还未写入寄存器文件时，将发生写后读（Read After Write, RAW）数据冲突。解决 RAW 冲突的方法：如果此时这个结果已经在某个阶段被计算出来，那么可以使用重定向（forwarding）将数据转发过来；否则需要阻塞（stall）流水线直到结果可用。需注意，`$0` 寄存器硬连接为 `0`，因此源寄存器为 `$0` 时不需要进行转发或阻塞。以下将阐述这两种方法的具体实现。

2.7.1.1 使用重定向解决冲突

当 Execute 阶段的源寄存器 `$rs` 或 `$rt` 与 Memory 阶段或 Writeback 阶段（即前两条指令）的写入目标寄存器 `write_reg` 相同，且其 `reg_write` 信号为 `1` 时（即需要写入目标寄存器），重定向对应的 `src_a` 或 `src_b`。以 `$rs` 的情况为例（`$rt` 同理），重定向逻辑如下：

```
if (rs_e_i && rs_e_i == write_reg_m_i && reg_write_m_i) begin
  forward_a_e_o = 2'b10;
end else if (rs_e_i && rs_e_i == write_reg_w_i && reg_write_w_i) begin
  forward_a_e_o = 2'b01;
end else begin
  forward_a_e_o = 2'b00;
end
```

其中当 `forward_a_e` 为 `10` 时，Memory 阶段转发的数据是 `alu_out_m`；当 `forward_a_e` 为 `01` 时，Writeback 阶段转发的数据是 `result_w`。

需注意这里 Memory 阶段的优先级高于 Writeback 阶段，因为 Memory 阶段的指令后执行，包含的数据更新。

2.7.1.2 使用阻塞解决冲突

对于指令 lw，因为它有两个周期的延迟，意味着其他指令至少要到两个周期后才能使用它的结果。如果指令 lw 后紧接着一个使用其结果的指令，则使用重定向无法解决这种冲突，此时需要阻塞流水线。现实中，编译器会针对这种情况做一定的优化，通过调整指令顺序，在发生数据冲突的两条指令间插入一条无关指令，从而避免这种冲突。

当 Execute 阶段正在执行的指令是 lw（此时 `mem_to_reg` 信号为 `1`），且 Decode 阶段的任一源操作数 `$rs` 或 `$rt` 与 Execute 阶段的目的寄存器 `$rt` 相同，阻塞 Decode 阶段直到源操作数准备好。阻塞逻辑如下：

```
assign lw_stall = (rs_d_i == rt_e_i || rt_d_i == rt_e_i) && mem_to_reg_e_i;

assign flush_e_o = lw_stall;
assign stall_d_o = flush_e_o;
assign stall_f_o = stall_d_o;
```

这里阻塞 Decode 阶段的同时也要阻塞 Fetch 阶段，并且刷新（flush）Execute 阶段，产生气泡。

2.7.2 控制冲突

在取下一条指令时还不能确定指令地址时，将发生控制冲突，即处理器不知道应该取哪条指令。解决控制冲突的方法：预测下一条指令地址，如果预测错误则刷新流水线。目前的实现中使用的是静态分支预测，事实上动态分支预测可以获得更高的性能。

然而，静态分支预测将导致新的 RAW 冲突，因此需要再次使用第 2.7.1 节解决数据冲突时的两种方法。

2.7.2.1 使用重定向解决冲突

如果指令的结果在 Writeback 阶段，则它将在前半周期写入寄存器，而在后半周期进行读操作，此时不会产生冲突。如果指令的结果在 Memory 阶段，则可以将它重定向回 Decode 阶段的 equal_cmp。类似第 2.7.1.1 节，以 `$rs` 的情况为例（`$rt` 同理），重定向逻辑如下：

```
assign forward_a_d_o = rs_d_i && rs_d_i == write_reg_m_i && reg_write_m_i;
```

2.7.2.2 使用阻塞解决冲突

如果指令的结果在 Execute 阶段，或者指令 lw 的结果在 Memory 阶段，则需要阻塞流水线。类似第 2.7.1.2 节，阻塞逻辑如下：

```
assign branch_stall = (branch_d_i || jump_d_i[1])
    && (reg_write_e_i && (rs_d_i == write_reg_e_i || rt_d_i == write_reg_e_i)
    || mem_to_reg_m_i && (rs_d_i == write_reg_m_i || rt_d_i == write_reg_m_i));

assign stall_d_o = lw_stall || branch_stall;
assign flush_e_o = stall_d_o;
assign stall_f_o = stall_d_o;
```

其中，`jump_d[1]` 信号为 `1` 时表示当前指令为 jr。

2.7.2.3 清除无效数据

当发生跳转时，需要清除跳转指令之后多读的一条无效指令，即刷新 Decode 阶段，产生气泡。刷新逻辑如下：

```
assign flush_d_o = pc_src_d_i || jump_d_i;
```

3. 测试

3.1 benchtest

和多周期处理器的测试一样，代码详见 `ext-seq.sv`，有关测试的更多细节在多周期报告中，这里无必要再赘述。

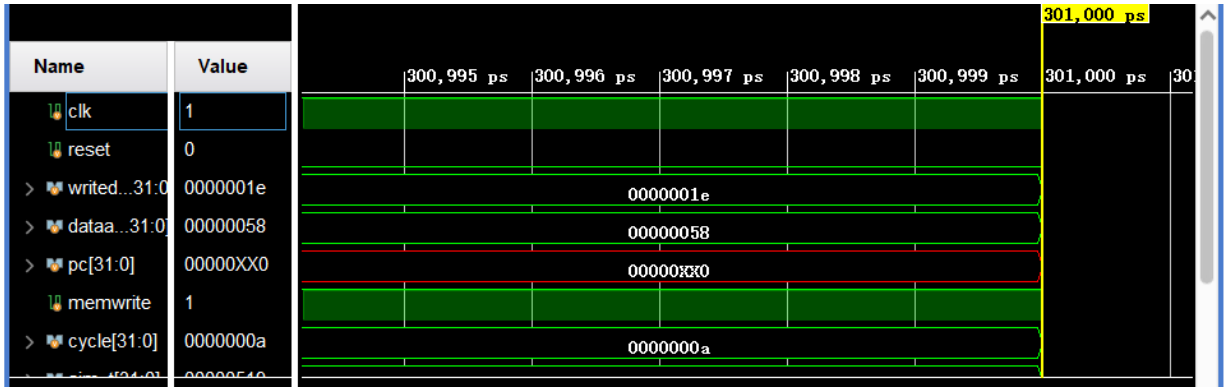
run simulation 测试结果：

```
epoch           29



epoch           30
Succeeded
relaunch_sim: Time (s): cpu = 00:00:02 ; elapsed = 00:00:13 . Memory (MB): peak = 820.777 ; gain = 0.000
```

vivado 仿真波形：



3.2 测试环境

- OS: windows11
- vivado 2018.3

4.Contributor

- Ekon Wang - Fudan University

5. License

6. Reference

---

1. https://en.wikipedia.org/wiki/SystemVerilog ↩ ↩