MIPS 流水线处理器实验报告

张作柏

17300240035

2019 年 5 月 31 日

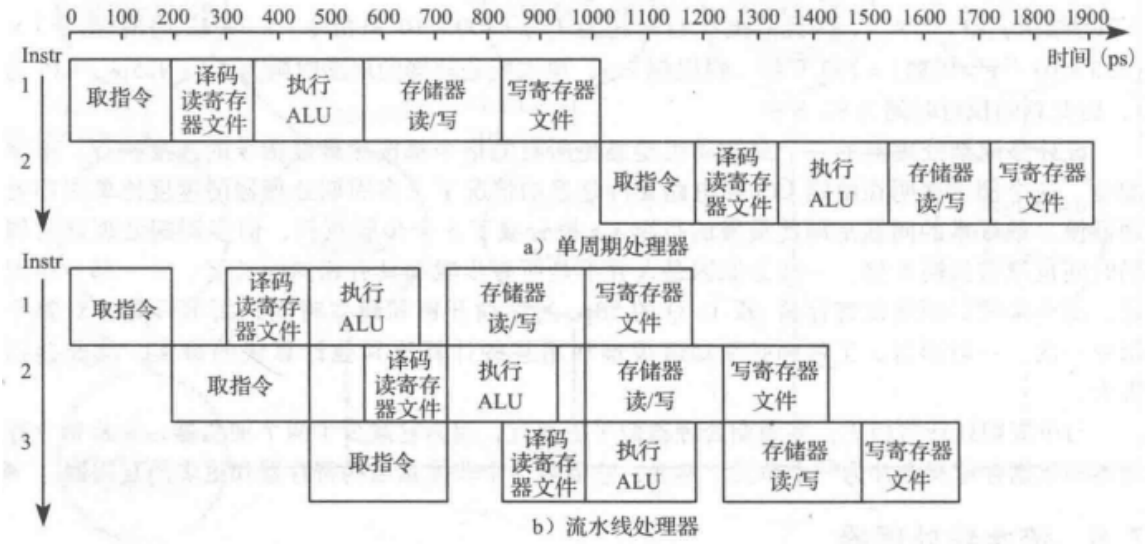
目录

1 流水线处理器简介

流水线技术是提高数字系统吞吐量的有效手段。通过将单周期处理器分解成 5 个流水线阶段来构成流水线处理器。因此，可以在流水线中同时执行 5 条指令，时钟频率几乎可以提高 5 倍。

流水线被划分为五个阶段，每个阶段完成一个操作：取指令 (Fetch)，译码 (Decode)，执行 (Execute)，存储器 (Memory) 和写回 (Writeback)。

- **取指**：处理器从指令存储器中读取指令。
- **译码**：处理器从寄存器文件中读取源操作数并对指令译码以便产生控制信号。
- **执行**：处理器使用 ALU 执行计算。
- **存储器**：处理器读或写数据存储器。
- **写回**：处理器将结果协会到寄存器文件。



上图对比了单周期处理器与流水线处理器的时序图[1]。可以看出，流水线的吞吐量明显高于单周期，也说明了流水线处理器的高效性。

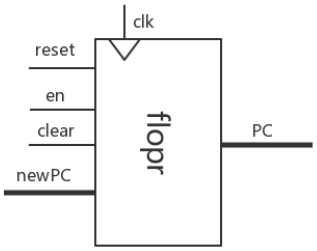
流水线系统中的核心问题是冲突 (Hazard)，即当后一条指令需要前一条指令的计算结果，而前一条指令还没有执行完时就会发生冲突。在这里，我们可以使用重定向、阻塞和刷新三种处理方法。

[1]图源于教材 256 页

2 部件分析

多周期处理器中大部分器件都与单周期相似，在此只列出新添加或发生改动的部件，其余请参考我的单周期报告。

2.1 触发器 flopr

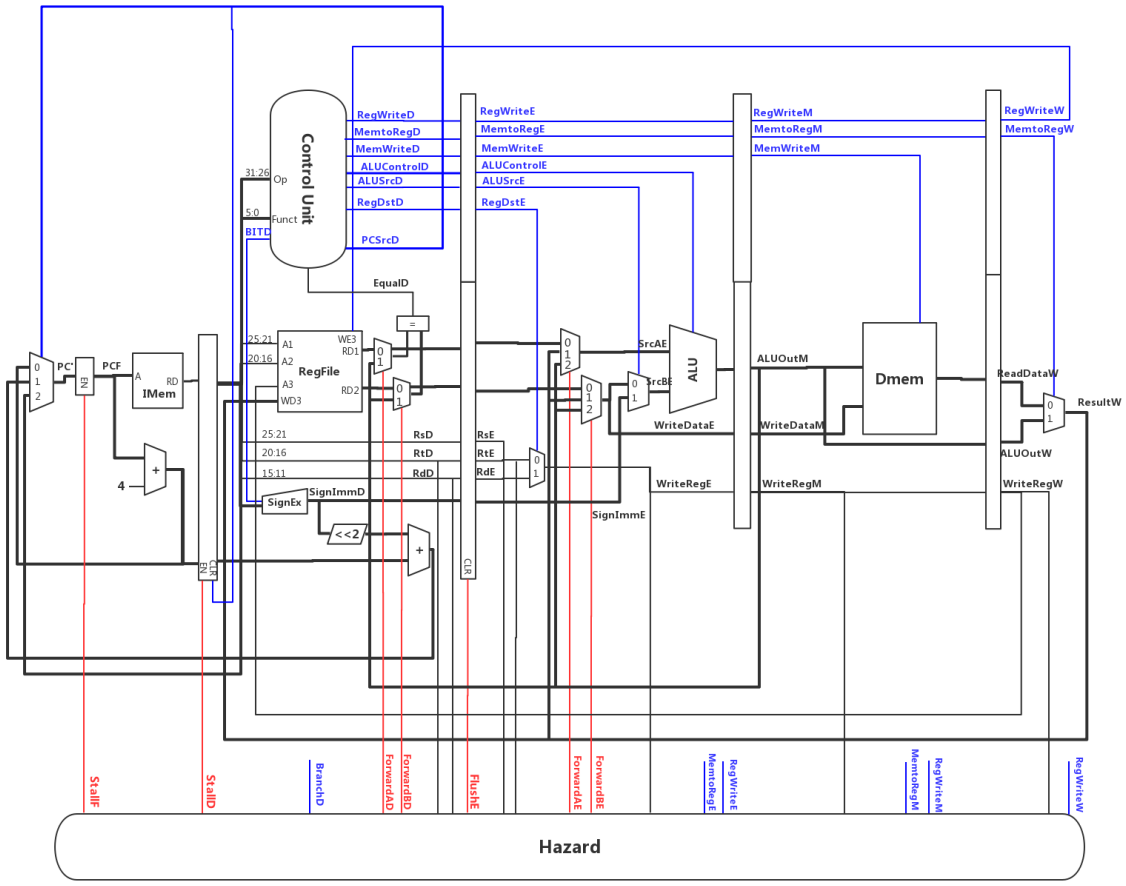


流水线中，除了本来 datapath 中的寄存器，我们还需要添加两两阶段之间的状态寄存器。而为了支持状态的阻塞和刷新操作，我们需要给寄存器加入同步清零功能。所以，寄存器共有异步清零 reset、使能 en 和同步清零 clear 三个控制口。

2.2 比较器 eqcmp

为了提前预测 beq 和 bne 是否跳转，我们需要在 decode 阶段预判两个操作数是否相等，所以需要添加一个比较器。A 与 B 都是 32 位数字，当 A==B 时，eq=1，当 A!=B 时，eq=0。

2.3 数据通路 Datapath



功能说明：流水线数据通路与单周期相似，在单周期的结构上加入了状态寄存器，下面简单介绍一下结构。

- 左侧 PC 多路器，受 PCSrcD 和 Jump 信号控制，选择相应的 PC 值：信号为 0 时，选择下一条指令地址 PC+4；信号为 1 时，选择 D 阶段计算出的相对寻址地址；信号为 2 时，选择指令中给出的绝对地址拼接 PC 高四位组成的地址。
- 左侧 PC 寄存器，由 StallF 控制是否写入 PC 值，若 StallF 为 1，则令 F 阶段暂停，不写入 PC 值。
- 左侧 Imem 指令存储器，接受 PC 值作为输入地址，读出的指令传递给状态寄存器。
- 左侧加法器计算 PC+4 的值，作为预测值传回 PC 多路器。
- Decode 阶段寄存器受 StallD 和 PCSrcD(Jump) 信号控制，分别控制其是否暂停和刷新，暂停时保持之前的状态，不写入新状态；刷新时，将当前阶段的状态清零。
- 寄存器文件、位扩展、移位器、加法器作用都与单周期相似，在此略去。
- Decode 阶段增加了比较器，来比较从 RegFile 中取出的两个数是否相等，用于 bne 和 beq 指令的提前预测结果。

- RegFile 取出的两个数增加了转发逻辑，以应对数据冒险，将 Memory 阶段的计算值转发回来，以便用于相等判断。
- Execute 阶段状态寄存器中，需要接受解码出的控制信号、两个操作数、三个操作寄存器和位扩展后的立即数，由 flushE 信号控制是否刷新寄存器。
- ALUSrcA 和 SrcB 的参数直接由 Execute 的寄存器传入，加入转发逻辑，将 Memory 阶段与 Writeback 阶段计算出的信号传回，受信号 ForwardAE 与 ForwardBE 控制。
- ALUSrcB 的参数与单周期相似，可以是立即数，也可以是寄存器取出的值。
- Memroy 阶段状态寄存器中，存储 ALU 的计算结果，从寄存器中读取的需要写入内存的值，以及之后要进行写入的寄存器，还有几个控制信号。
- Memory 阶段与 Writeback 阶段功能结构基本与单周期相似，多出来的是加入了一些转发逻辑。

2.4 控制模块 Controller

2.4.1 控制信号说明

控制信号有以下几种：

- **RegWrite:** 控制是否写寄存器，会传递到 Writeback 阶段后作为 RegFile 的写使能。
- **MemtoReg:** 是否是从 Memory 取数后写回到 RegFile 的指令，传递到 Writeback 阶段，选择返回的 ResultW。
- **MemWrite:** 是否需要写内存，传递到 Memory 阶段后，作为 Dmem 的写使能信号。
- **ALUControl:** 选择 ALU 的运算，传递到 Execute 阶段后，接入 ALU。
- **ALUSrc:** 选择 ALUSrcB 是寄存器值还是立即数，传递到 Execute 阶段。
- **RegDst:** 选择写入的寄存器，用于区分 R 型指令与 I 型指令，传递到 Execute 阶段选择完寄存器后，便可丢掉。
- **Branch:** 判断指令是否为 beq，用于 PCSrc 的计算。
- **BNE:** 判断指令是否为 bne，用于 PCSrc 的计算。
- **Jump:** 判断指令是否为 j，用于 PC 来源的选择。
- **PCSrc:** 用于选择 PC 的来源，判断分支指令是否执行。若 eqcmp 结果为 1 且 beq 等于 1，或 eqcmp 结果为 0 且 bne 等于 1，则发生跳转，PCSrc=1。
- **BIT:** 判断位扩展时是采用符号扩展还是零扩展。

2.4.2 实现细节

控制模块的实现与单周期完全相同，可以直接 copy 过来。

2.5 冲突处理模块 Hazard

本节主要介绍 Hazard 模块的信号逻辑设置，具体冒险的处理将在下一节中详细讨论。

Hazard 模块接受控制信号、寄存器信息等等，返回控制信号，以控制整个流水线的转发逻辑、暂停和刷新逻辑，以此来应对冒险。输出的控制信号在图中已用红线标出，分为三种：Stall、Flush 和 Forward 逻辑，分别对应暂停、刷新和转发，以下将介绍每种逻辑的控制。

- Decode 阶段的转发信号 ForwardAD 与 ForwardBD 主要用于选择比较器的参数，因为 Decode 阶段中读的 RegFile 可能在之前的指令中已经被修改，但还未来得及写入寄存器中，所以这里需要引入转发逻辑，将 Memory 阶段的 ALU 结果转发回来。注意这里不能对 Execute 阶段的值转发，因为不一定来得及计算出来；同时，不需要对 Writeback 阶段的值进行转发，因为该值会在下降沿写入 RegFile，在该时钟周期结束前，两个操作数的值就已变成正确的值了。信号的逻辑为，如果 RegWriteM 信号值为 1，且写入的寄存器 WriteRegM 与相应的寄存器相同的话，则设置 Forward 信号为 1。

- Execute 阶段的转发信号 ForwardAE 与 ForwardBE 主要用于选择 ALU 的参数。这两个参数是从寄存器文件中读出来的，而有可能该值已被前两条指令修改，但是在读的时候还未写入。这里的转发需要同时转发 Memroy 和 Writeback 阶段的值，所以信号的逻辑应为先判断 RegWriteM 是否为 1 和是否写入相应寄存器，再判断 RegWriteW 是否为 1 和是否写入相应寄存器。注意 M 的优先级高于 W，因为 M 指令是后执行的。

- 暂停和刷新的控制是一致的，即 StallF、StallD 与 FlushE 的控制逻辑相同。第一种情况是 lw 指令需要从内存中取值写入寄存器，因为在 Memory 阶段结束时才能取出值，所以无法进行转发。所以当 Execute 阶段的指令用到了在 Memory 阶段的 lw 指令的值时，需要插入一条 nop 指令，对应让 Execute 阶段刷新，Fetch 和 Decode 阶段暂停。第二种情况是分支指令预测时，相应的寄存器值无法转发，必须要暂停一个周期，等寄存器值更新后再去判断。控制逻辑为，若控制信号 BNE 或 Branch 为 1，且 Execute 阶段需要写入寄存器或 Memory 阶段为 lw 指令，并且写入寄存器与读的寄存器相同时，该信号为 1。

- Decode 阶段寄存器的刷新信号由 PCSrcD 与 Jump 信号控制，当发现需要跳转时，我们之前取出的指令就需要作废了，所以刷新 Decode 寄存器避免该指令传递下去。

3 冲突处理与额外指令分析

3.1 额外指令分析

因为时间就比较紧张，所以我在流水线阶段没有加入太多的指令。实际上，加入指令的逻辑并不复杂，且比较无聊，所以自己也就没有太大兴致继续加指令了。

3.1.1 移位指令 sll

移位指令的加入主要是为了配合我的测试程序使用，有了移位指令，简单的乘法操作就可以实现了。加入的逻辑并不复杂，与多周期相同，直接把移位数传入 ALU，并用 ALUControl 信号控制 ALU 运算即可。

3.2 冲突处理

冲突的分析与处理是流水线设计的难点。在流水线系统中，多条指令同时执行。当一条指令依赖于还没有结束的另一条指令的结果时，将发生冲突。冲突可分为*数据冲突*和*控制冲突*。当一条指令试图读取前一条指令还未写回的寄存器时将发生数据冲突。在取指令时还未确定下一条指令应取的地址时将发生控制冲突。

3.2.1 利用转发 (Forward) 解决冲突

当 Execute 阶段中的指令有一个与 Memory 阶段或 Writeback 阶段中的目的寄存器相匹配的源寄存器时，需要进行转发。一个典型的例子如下：

```
0x0 : add $s0, $s2, $s3
0x4 : and $t0, $s0, $s1
0x8 : or  $t1, $s4, $s0
0xc : sub $t2, $s0, $s5
```

在 or 指令中源寄存器 $s0 的值来源于 add 指令的计算结果，但是当 or 指令位于 Decode 阶段时，add 指令还在 Memory 阶段，所以寄存器的值还没有被写入。所以当 or 指令进入 Execute 阶段后，需要通过转发逻辑将 add 指令的计算结果从 Writeback 转发过来。对于 Memory 阶段的数值也需要进行相似的转发操作。

3.2.2 利用阻塞 (Stall) 解决冲突

利用转发可以解决写后读 (RAW) 的冲突，但是 lw 指令知道存储器阶段后才能完成读数据，因此它的结果不能重定向到下一步指令的执行阶段。我们采取的解决方法是阻塞流水线，将操作挂起直至数据有效，等价于在 Execute 的阶段插入一条 nop 指令，同时使前面的指令暂停。一个典型的例子是：

```
0x0 : lw  $s0, 40($0)
0x4 : and $t0, $s0, $s1
0x8 : or  $t1, $s4, $s0
0xc : sub $t2, $s0, $s5
```

这里 and 指令用到了 lw 指令中写入 $s0 中的值，然而 and 在 Decode 阶段 lw 还没进行写入，and 在 Execute 阶段时 lw 会在时钟周期结束时才能取出值，所以无法进行转发。这时的处理方法是让 and 指令等一个周期，在 Execute 中插入一条 nop 指令，同时让 Decode 阶段与 Fetch 阶段的指令暂停。

3.2.3 解决控制冲突

beq 指令将产生控制冲突，因为在取下一条指令时分支是否发生还未确定，所以流水线处理器不知道取哪条指令。这里我们在 Decode 阶段加入比较器来预测分支跳转是否会发生。

糟糕的一条是，提前确定分支的硬件会产生新的 RAW 数据冲突。特别是，如果分支指令的一个源操作数由前一条指令计算得到且还没有写入寄存器文件，分支指令将从寄存器文件中读取错误的操作数值。我们可以通过前面使用的转发技术来解决数据冲突 (如果可以进行转发)，或者使用阻塞的方式，等到数据写入后再进行预测。

3.2.4 一个糟糕的问题

我在执行 quick_multiply.in 这个样例时，发现了课本中提供的代码的错误。参考下面这个例子：

```
0x10 : andi $t2, $t1, 1     | 312a0001
0x14 : beq $t2, $0, target  | 100a0001
0x18 : add $s0, $s0, $t0    | 02088020
0x1c : target:              |
```

当 beq 指令进行预测时，其使用的操作数 $t2，在上一条指令中被修改，所以需要暂停，等待 andi 指令到 Memory 阶段进行转发。若此时 $t2 的值恰好为 0，那么就会出现 FlushD(需要跳转，故刚刚读出的指令不向下传递) 与 StallD(暂停，等待 andi 指令计算完) 同时为 1 的情况，此时，我们应该选择刷新还是暂停呢？课本中直接将两个信号同时传入，因为在寄存器的设计中刷新的优先级较高，所以会将该指令刷新掉。与此同时，Fetch 阶段暂停了，Execute 阶段刷新了，所以这条 beq 指令就凭空消失了，当真的需要执行跳转分支时，也不会执行，所以会出现错误。

正确的处理方式是在传入 flushD 信号时，保证二者不会同时为 1，即传入以 stallD & flushD 信号传入。这样当 stallD 为 1 时，flushD 就永远不会为 1。那么，为什么我们不直接在寄存器中设置写使能的优先级高于清零呢？因为如果写使能优先级始终较高的话，对于一些写使能恒为 1 的寄存器，就无法执行刷新操作了。当然，也可以为它单独设计一个寄存器。不过相比之下，强制让信号互斥是更好的处理方案。

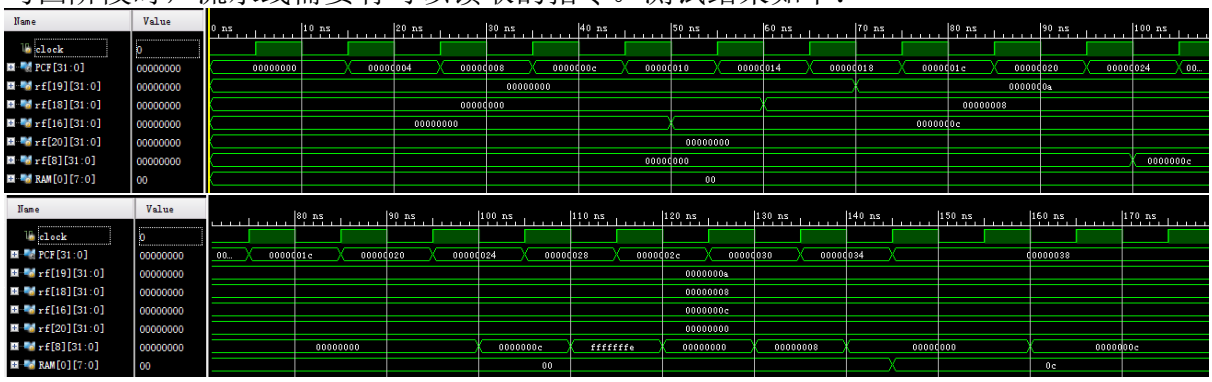
4 测试样例与结果

与单周期相同的测试样例均已通过，这里只列举几个关键的和新加入的测试样例。

4.1 all.in

```
 0x0 : addi $s0, $0, 12      | 2010000c
 0x4 : andi $s2, $s0, -8     | 3212fff8
 0x8 : ori $s3, $s1, 10      | 3633000a
 0xc : slti $s4, $s2, 5      | 2a540005
0x10 : nop                   | 00000000
0x14 : add $t0, $s0, $s1     | 02114020
0x18 : sub $t0, $s2, $s3     | 02534022
0x1c : and $t0, $s3, $s1     | 02714024
0x20 : or $t0, $s1, $s2      | 02324025
0x24 : slt $t0, $s0, $s2     | 0212402a
0x28 : sw $s0, 0($0)         | ac100000
0x2c : lw $t0, 0($0)         | 8c080000
0x30 : nop                   | 00000000
0x34 : nop                   |
```

注意，这里需要在所有指令结束后添加 nop 指令，因为当最后一条指令在存储器和写回阶段时，流水线需要有可以读取的指令。测试结果如下：



上图中依次给出了寄存器 $s3, $s2, $s0, $s4 和 $t0，以及内存地址 0 处的值。每条指令的结果依次应为：

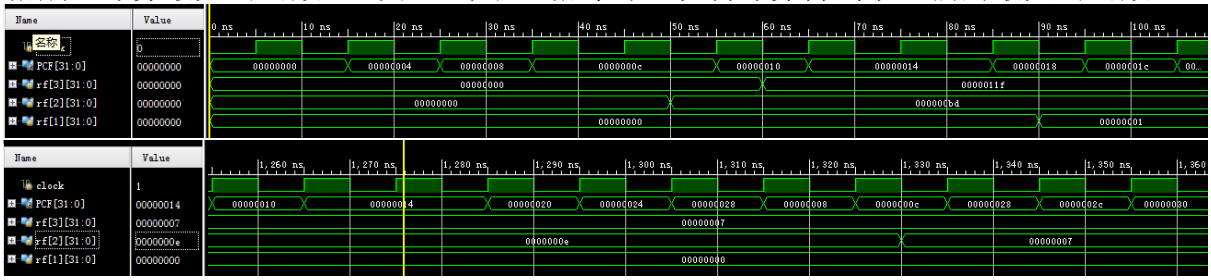
$s0 = 12 → $s2 = 8 → $s3 = 10 → $s4 = 0 → nop → $t0 = 12 → $t0 = -2 → $t0 = 0 → $t0 = 8 → $t0 = 0 → 0($0) = 12 → $t0 = 12

4.2 gcd.in

```
 0x0 : addi $v0,$0,189       | 200200bd
 0x4 : addi $v1,$0,287       | 2003011f
 0x8 : main:                 |
 0x8 : beq $v0,$v1,end       | 10620007
```

```
 0xc : slt $at,$v0,$v1       | 0043082a
0x10 : beq $at,$0,run        | 10010003
0x14 : add $at,$v0,$0        | 00400820
0x18 : add $v0,$v1,$0        | 00601020
0x1c : add $v1,$at,$0        | 00201820
0x20 : run:                  |
0x20 : sub $v0,$v0,$v1       | 00431022
0x24 : j main                | 08000002
0x28 : end:                  |
0x28 : add $t3, $0, $0       | 00005820
```

这个测试样例是从黄文皓同学那里借来的，通过循环计算两个数的最大公约数。开始时，我们把需要计算的两个数字 189 和 287 分别赋值给 $v0 和 $v1。之后，通过辗转相减法计算最大公约数。当 $v0 与 $v1 相等时，说明计算得到了它们的最大公约数。



这里的两幅图分别展示了程序开始与结束时的计算结果。从第一幅图可以看到程序正常的执行循环，后一幅图则说明程序成功地计算出两数的最大公约数为 7。

4.3 quick_multiply.in

```
 0x0 : addi $t0, $0, 99      | 20080063
 0x4 : addi $t1, $0, 37      | 20090025
 0x8 : addi $s0, $0, 0       | 20100000
 0xc :                       |
 0xc : while:                |
 0xc : beq $t1, $0, done     | 10090006
0x10 : andi $t2, $t1, 1      | 312a0001
0x14 : beq $t2, $0, target   | 100a0001
0x18 : add $s0, $s0, $t0     | 02088020
0x1c : target:               |
0x1c : sll $t0, $t0, 1       | 00084040
0x20 : srl $t1, $t1, 1       | 00094842
0x24 : j while               | 08000003
0x28 :                       |
0x28 : done:                 |
0x28 : add $t3, $0, $0       | 00005820
```

这个代码曾在单周期处理器中演示过，用于计算两数的乘积，方法类似快速幂。首先将 99 与 37 写入 $t0 和 $t1 寄存器，然后进入循环，每次根据 $t1 的末位是否为 1，来

决定是否向 $s0 进行累加，然后每次 $t0 乘 2，$t1 右移一位。最终，计算结果存在 $s0 中，应为 0xe4f。这个测试样例测试了跳转、移位等指令，测试结果如下：

图中依次展示的是寄存器 $s0,$t2,$t1,$t0 的值，可以看到程序开始时，寄存器 $t0 与 $t1 分别被赋值为 0x63 与 0x25，分别对应 99 与 37。循环过程中，$t0 每次乘 2，$t1 每次除以 2，最终 $s0 变为 0xe4f，结果正确，说明程序正确执行。

5 注意事项

5.1 遇到的问题

1. 寄存器文件要设置为下降沿写入。
2. 不要迷信书上的代码，书上的代码也会有错。
3. 使用未定义的变量，Verilog 是不会报错的，但是在仿真的时候会出现奇怪的数值，所以写完代码后一定要仔细检查变量名与连线。

6 申 A 理由

- 实现了所有的基本指令
- 首先发现了书中代码的 bug，并帮助其他人解决了这个问题
- 自己动手画图来描绘流水线的结构 (见 Datapath)