MIPS 多周期处理器实验报告

张作柏

17300240035

2019 年 5 月 5 日

目录

1 多周期处理器简介

单周期处理器虽然实现简单，但仍存在着许多问题。本节中，我们先简要讨论单周期处理器中的问题，借此引出多周期处理器，并介绍多周期处理器的主要特征。

1.1 单周期处理器的缺点

单周期处理器的主要缺点有三：

1. **效率低**：在单周期中，所有指令的执行时间都是一个 CPU 周期，这就意味着 CPU 周期的设置必须要保证所有指令都能在一个周期内执行完毕。而不同的指令执行效率是相差极大的，比如与访存相关的 lw、sw 指令所需要的时间远大于普通的 R 型运算指令。然而在实际系统中，用时较短的指令占到多数，为照顾耗时较长的指令而调高 CPU 周期是得不偿失的。

2. **资源浪费**：在单周期电路中，我们需要使用多个加法器，ALU 和用于 PC 逻辑的多个加法器。那么，能否通过一个 ALU 就实现这些加法器的功能呢？虽然这样的节省在一个 CPU 上减少不了多少成本，但是当 CPU 进行批量生产时，则能省下较大的开支。

3. **内存设计不合理**：单周期处理器中，为了方便指令的执行，数据存储器与指令存储器是分离的。然而，在现实系统中往往二者是结合的。大多数计算机有一个单独的大容量存储器来存储指令和数据，并且支持读和写操作。

1.2 多周期处理器的特征

多周期处理器通过将每个指令的执行过程分解为多个较短的步骤来解决这些问题。其主要特征为：

1. **状态机**：多周期将每条指令的执行分为多个阶段，每个阶段对应状态机中的一个状态。在这里使用状态机模型的好处是：1) 合并不同指令执行中的公共状态 2) 模式化状态转移的过程。因为不同指令的状态执行数不同，因此耗时也不同，用时较短的指令数量较少，可以大大减少其耗时。

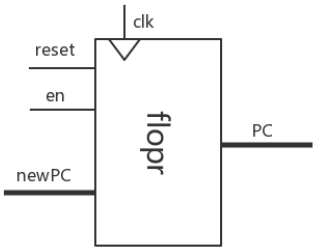
2. **资源共享**：整个处理器只需要一个 ALU，在多周期中，ALU 利用闲置时间，充当原来多余加法器的角色，进行 PC 逻辑的计算。

3. **控制逻辑复杂**：因为引入了状态机模型，所以也就要考虑相应的状态转移，同时需要的控制逻辑也会变得更加复杂，带来更大的编程复杂度。同时，因为多周期的控制逻辑更加复杂，所以 CPU 也会在此花费更多的时间，最终导致多周期处理器的性能相对于单周期并不会带来非常大的提升。

2 部件分析

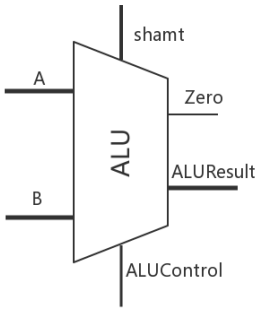
多周期处理器中大部分器件都与单周期相似，在此只列出新添加或发生改动的部件，其余请参考我的单周期报告。

2.1 触发器 flopr



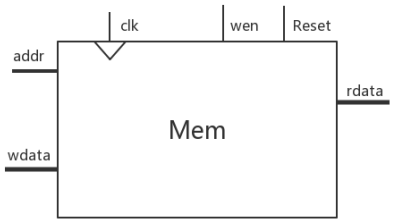
多周期中的触发器增加了一个使能端口 en，当时钟上升沿到达，若 en=1，则将 newPC 写入 PC，否则 PC 保持原值。增加这一端口的原因是，多周期中并不是每个上升沿都要进行写操作，有的寄存器需要保持原来存储的值不做变化。

2.2 算数逻辑单元 ALU



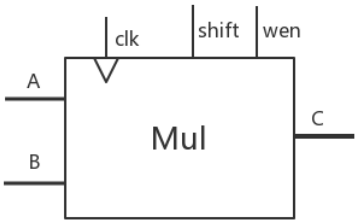
按照原来的设计，其实单周期与多周期的 ALU 应该是相同的。然而，我现在在 ALU 上多加入了一个 shamt 口用来接收移位的数据，这样做的好处是减少了控制信号，同时也降低了控制的复杂度，随之而来的问题就是 ALU 的成本会变大。实际上，只有在移位操作时才会使用 shamt，那么完全可以把 shamt 的数据通过 A 或 B 口传入，只不过会增加一些控制的难度罢了。

2.3 存储器 Mem



存储器最大的不同是将指令存储器 Imem 与数据存储器 Dmem 合并起来，以模拟真实情景。mem 内部的构造与单周期的 Dmem 相同，只不过使用 0 99 的地址空间来存储指令，之后的地址空间才存储数据罢了。

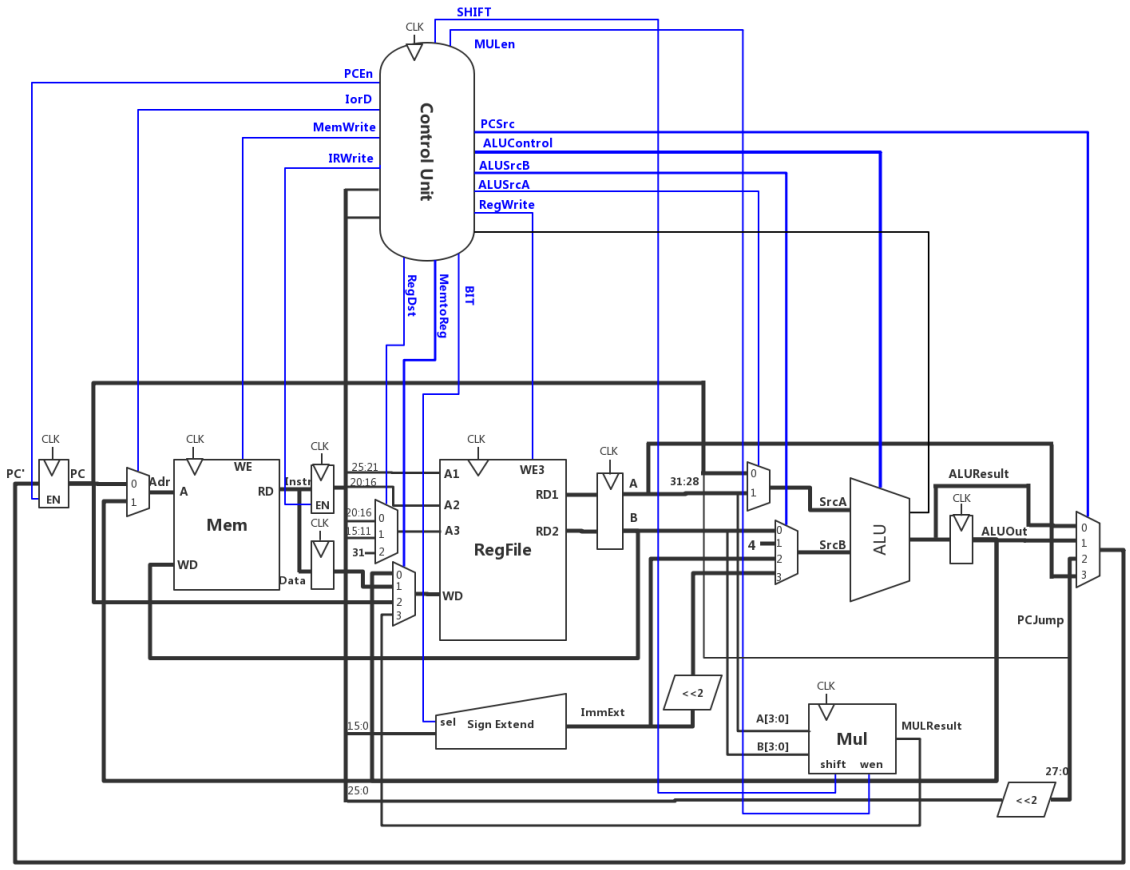
2.4 乘法器 mul



乘法器是新加入的部件，其功能是执行 A 与 B 的乘法，并将结果输出到 C 中，其中 A 与 B 都是四位二进制数，C 是八位二进制数。在后面乘法器的原理分析中，我们可以发现可以很容易的将四位乘法拓展到 32 位乘法，在此为节省演示时间，只涉及了四位乘法的乘法器。

乘法器原理就是在计算机组成课上讲的移位乘法，利用一个 8 位寄存器存储当前结果，每次做加法写入高位后，将整个寄存器右移一位，四次移位后，即为答案。在实现的时候，我使用了 shift 和 wen 两个接口来接受控制器传来的移位信号与写使能信号：当 wen 信号为 1 时，将 B 的值写入结果寄存器低位，并将高位置为 0；否则当 shift 信号为 1，执行加法并左移操作；否则结果寄存器保持原值。这里我为乘法器模块单独设计了一个加法器，而实际上这里共用 ALU 是更划算的，然而控制逻辑有点复杂，所以我就偷了个懒。。。

2.5 数据通路 Datapath



功能说明：因为与单周期相比很多部件都没有改变，所以这里只说明变化的部分。

- 左侧 PC 寄存器，添加了 PCEn 信号，控制寄存器写入逻辑，只在需要更新 PC 值的状态才将 PCEn 设置为 1。
- 左侧二路选择器，因为 Dmem 与 Imem 合二为一了，所以要通过 IorD 信号选择读取指令还是读取数据。
- 左侧 Instr 寄存器，只有当需要读指令时，才将 IRWrite 信号设为 1，其余状态则需要存储原先的指令。
- 中间 A、B 两个寄存器用于存从 RegFile 中读出来的值，因为这些值可能会延迟一个状态使用，所以需要用寄存器储存。
- 中间 SrcA 是 ALU 的 A 参数，由 ALUSrcA 信号控制：当 ALUSrcA 为 0 时，选择 PC 值作为运算对象 (常用作计算下一指令的 PC 值)；当 ALUSrcB 为 1 时，选择 A 寄存器的值作为运算对象，用于 R 型和部分 I 型指令。
- 中间 SrcB 是 ALU 的 B 参数，由 ALUSrcB 信号控制：当 ALUSrcB 为 00 时，选择 B 寄存器的值作为运算对象，用于 R 型指令；当 ALUSrcB 为 01 时，选择 4 作为

运算对象，用于计算下一条指令的 PC；当 ALUSrcB 为 10 时，选择移位前的立即数作为运算对象，用于 I 型指令；当 ALUSrcB 为 11 时，选择移位后的立即数作为运算对象，用于计算 jal、bne、beq 指令跳转的目标地址。

- 右侧 ALUOut 寄存器用于存储 ALU 运算的结果，以供下一周期使用。
- 右侧四路选择器用于选择 PC 的值，由 PCSrc 信号控制，共四种选择：ALUResult(下一指令 PC)，ALUOut(jal、bne、beq 指令的跳转地址)、PCJump(j 指令的跳转地址)、A(jr 指令的跳转地址)。

2.6 控制模块 Controller

2.6.1 控制信号说明

以下简单解释一下我使用的控制信号。

- PCEn 用于控制是否写入 PC，由 PCWrite、Branch、Branch1 和 Zero 信号控制：

```
assign PCEn = (Branch & Zero) | (BranchBNE & ~Zero) | PCWrite;
```

- IorD 用于控制读指令还是数据
- MemWrite 用于控制是否写入内存
- IRWrite 用于控制是否将从 Mem 中读出的数据写入 Instr 寄存器
- RegDst 用于控制写入寄存器的选择
- MemtoReg 用于控制写入数据的来源
- BIT 用于控制移位是符号扩展还是 0 扩展
- SHIFT 和 MULen 分别作为乘法器的移位信号和写使能，将在乘法指令的实现细节中讨论
- PCSrc 用于选择 PC 值的来源
- ALUControl 用于控制 ALU 的运算
- ALUSrcA、ALUSrcB 用于选择 ALU 的 A、B 参数
- RegWrite 用于控制 RegFile 的写使能

2.6.2 实现细节

在单周期中，我们可以直接根据指令的类型来调整控制信号；而在多周期中，我们则需要根据当前所处的状态来进行信号赋值：

```
reg [21:0] controls;
    assign {SHIFT, MULen, IorD, memwrite, IRWrite, RegDst, MemtoReg,
   PCWrite, branch, branchBNE, ALUSrcA, RegWrite, BIT, ALUSrcB, PCSrc,
   aluop} = controls;
    always @(*)
      case (state)
        Fetch         : controls <= 22'b0000100001000000100000;
        Decode        : controls <= 22'b0000000000000001100000;
       ......
        default: controls <= 22'bx;
      endcase
```

比单周期更复杂的是，多周期中增加了状态机的设置，也就需要我们书写状态机的转移代码。为了表示方便，我们可以先用易于理解的常量表示状态和指令：

```
parameter Fetch = 0;
parameter Decode = 1;
......
parameter LW = 6'b100011;
parameter SW = 6'b101011;
......
```

之后，根据当前状态和指令的值，用 case 语句选择下一状态：

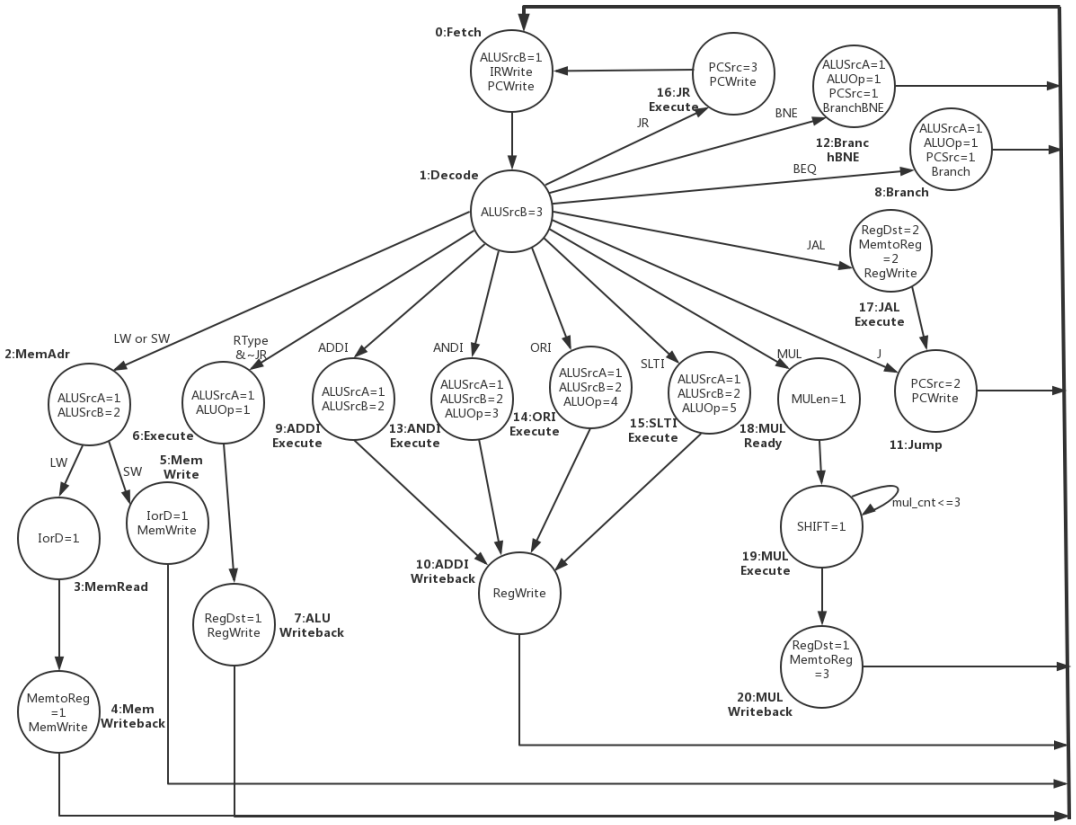
```
always @(*)
      case (state)
        Fetch: nextstate = Decode;
        Decode: case (op)
                  ......
                endcase
        MemAdr: case (op)
                  ......
                endcase
        MemRead: nextstate = MemWriteback;
        MemWriteback: nextstate = Fetch;
        ......
      default: nextstate = 5'bx;
     endcase
```

最后，在每个时钟上升沿到来时，用 nextstate 更新 state：

```
always @(posedge clk or posedge Reset)
    if (Reset) state <= 0;
    else state = nextstate;
```

3 有限状态机与指令分析

3.1 状态机图与状态分析



图中未标出的控制变量均默认为 0。

3.2 额外指令分析

常规指令的状态设计在教材中已经分析的相当透彻了，因此我在这里只列出我增加的指令的状态设计思路。

3.2.1 移位指令 sll

注意到，这里没有为移位指令单独设计状态，而是将其作为普通的 R-Type 指令来处理。因为我改造了 ALU，把移位数传到了 ALU 中，这样就 ALU 就可以直接根据 funct 进行移位操作了。过程与其它 R-Type 指令完全一样，无需增加新的状态。

3.2.2 函数调用指令 jal

我为 jal 指令新增加了一个状态 17：JALExecute，在这个状态中我们要将返回地址写入寄存器 $ra 中。那么，

之后，我们可以像 jump 一样，根据指令中的立即数计算目标地址，选择 PCSrc 为 2，更新 PC。

3.2.3 跳转寄存器指令 jr

跳转寄存器指令也比较好实现，只需要为 PC 的来源增加一个选项为寄存器值即可。然后增加一个状态 JRExecute，选择 PCSrc=3，并设置 PCWrite 为 1。

3.2.4 乘法指令 mul

乘法指令分三个步骤：

1. **MULReady**：将操作数写入乘法器寄存器，设置 MULen 为 1
2. **MULExecute**：执行加法并右移，执行四次，设置 SHIFT 为 1。为了控制移位的次数，我们引入 mul_cnt 变量，每当进入 MULExecute 状态时，该变量清零，之后每次时钟上升沿到来便加一，直到累加到三后，设置转移状态为 MULWriteback。
3. **MULWriteback**：同 R 型指令一样选择 RegDst=1，以指令中指定的寄存器作为目的寄存器，同时设置 MemtoReg=3，选择写入的数字来源为乘法器结果。

乘法位数的扩展：因为乘法操作的周期数与乘法的位数有关，所以我们只实现了四位的乘法器，不过这种乘法器可以很容易的扩展到 32 位：

1. 改变乘法器部件内部设置，将寄存器长度调整为 32 和 64
2. 设置 mul_cnt 的上限为 32

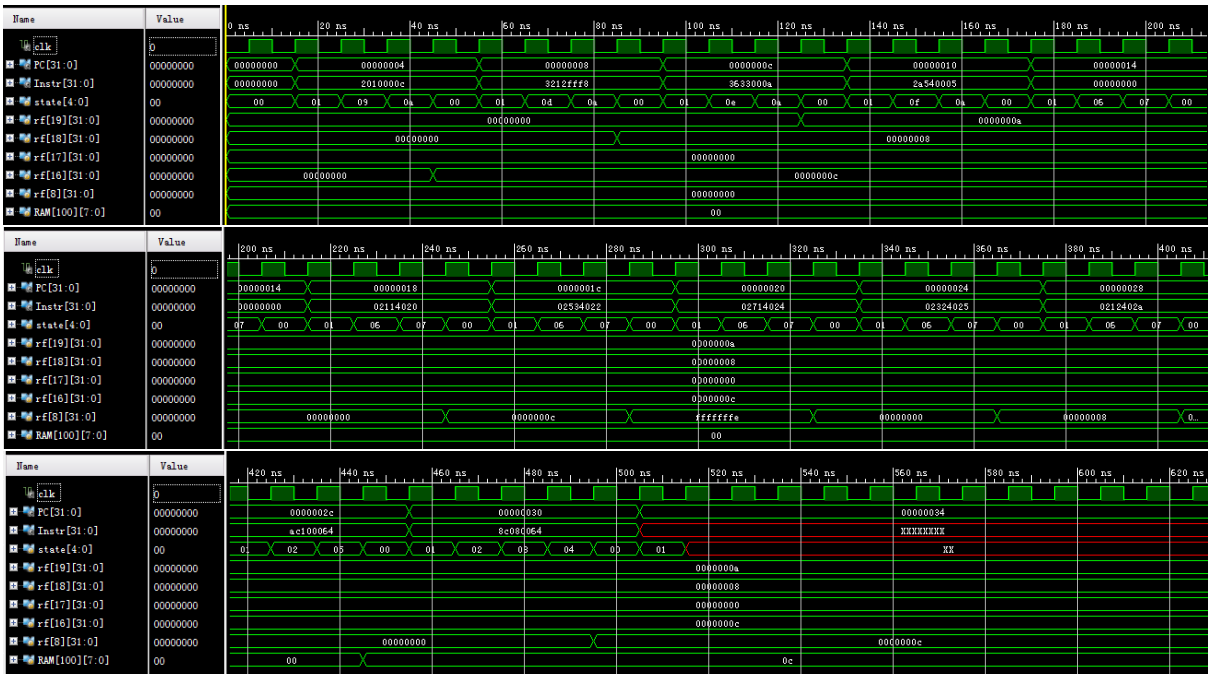
这样就完成了乘法的一般化过程。

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, 100($0)       | ac100064
0x2c : lw $t0, 100($0)       | 8c080064
```

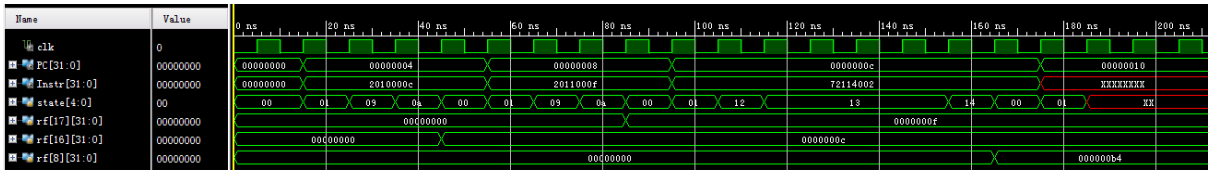


这里显示了 PC、state、instr 的值，并可以看到相关变量的运算结果。可以发现，多周期的执行时间明显变长了，而且一条指令会包含多个状态的切换过程。容易验证，所有的运算结果都是正确的。

4.2 mul.in

```
 0x0 : addi $s0, $0, 12      | 2010000c
 0x4 : addi $s1, $0, 15      | 2011000f
```

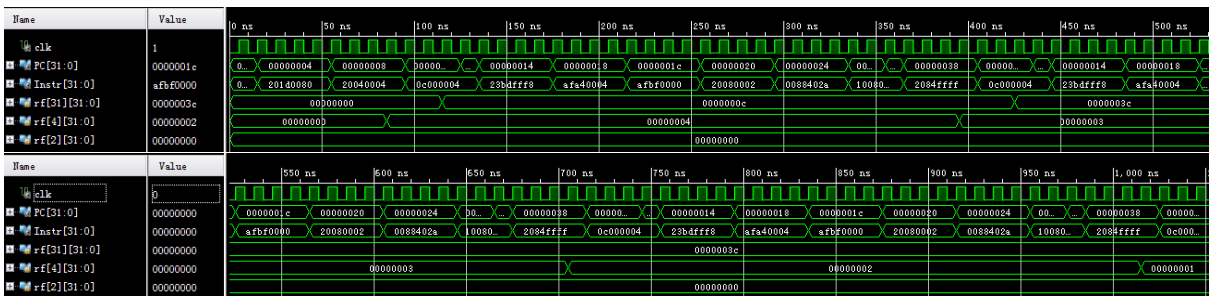
```
 0x8 : mul $t0, $s0, $s1     | 72114002
```

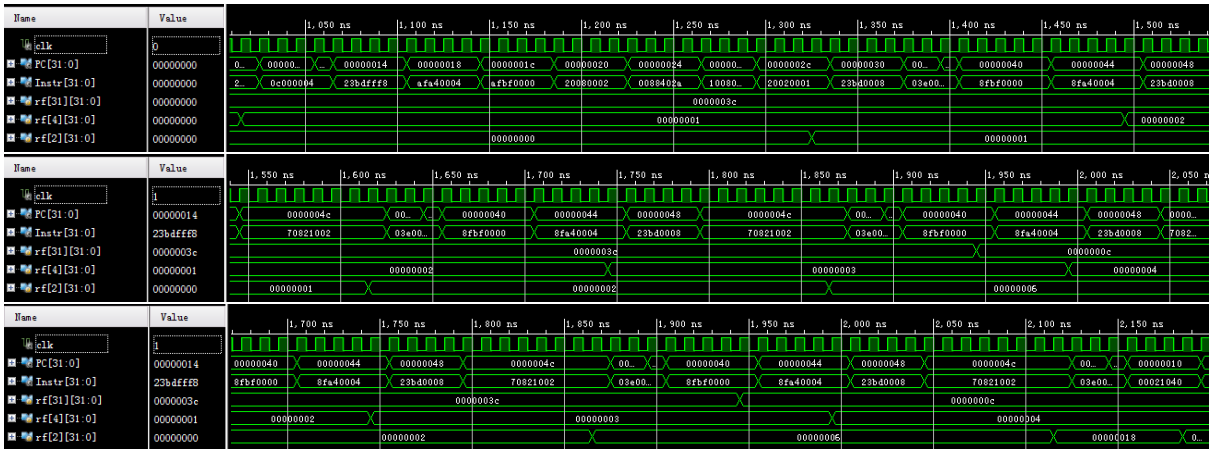


这一样例简单的演示了乘法的执行过程，首先将 s0 和 s1 分别赋值为 12 和 15，然后在用 mul 运算执行乘法，并把结果写入 t0 中。可以看到在执行的过程中，状态 13 的时间是尤其长的，这是因为需要执行四次移位操作。可以看到最后寄存器 t0 的值为 b4=180=12*15，是正确的运算结果。

4.3 real_factorial.in

```
 0x0 : addi $sp, $0, 128     | 201d0080
 0x4 : addi $a0, $0, 4       | 20040004
 0x8 : jal factorial         | 0c000004
 0xc : sll $v0, $v0, 1       | 00021040
0x10 :                       |
0x10 : factorial:            |
0x10 : addi $sp, $sp, -8     | 23bdfff8
0x14 : sw $a0, 4($sp)        | afa40004
0x18 : sw $ra, 0($sp)        | afbf0000
0x1c : addi $t0, $0, 2       | 20080002
0x20 : slt $t0, $a0, $t0     | 0088402a
0x24 : beq $t0, $0 ,else     | 10080003
0x28 : addi $v0, $0, 1       | 20020001
0x2c : addi $sp, $sp, 8      | 23bd0008
0x30 : jr $ra                | 03e00008
0x34 : else:                 |
0x34 : addi $a0, $a0, -1     | 2084ffff
0x38 : jal factorial         | 0c000004
0x3c : lw $ra, 0($sp)        | 8fbf0000
0x40 : lw $a0, 4($sp)        | 8fa40004
0x44 : addi $sp, $sp, 8      | 23bd0008
0x48 : mul $v0, $a0, $v0     | 70821002
0x4c : jr $ra                | 03e00008
```





因为这幅图耗时略长，所以我把比例调的比较小，以便展示全程。这个样例是递归计算阶乘的小程序，与单周期不同，这次我们实现了乘法器，所以计算的是真正的阶乘。可以看到，ra 用于存储返回地址，始终存储的是 0xc 与 0x3c 这两个返回地址；a0 用于传参，从 4 逐渐递减至 1，又累增至 4；v0 则用于存储计算结果，初始时一直进行递归调用，所以 v0 始终保持为 0，而后每次 v0 会与 a0 相乘，最终得到结果 0x18=24=4*3*2*1。另外，可以从 PC 值看出，程序是正确执行了的。

5 注意事项

5.1 遇到的问题

想了一下，好像没啥问题。。。

5.2 显示模块

这次的显示模块和上次一模一样，不需要做任何改动。不过为了方便演示，我加入了一个暂停功能：

```
wire clk, clk0, clk1, clk2, clk3;
clkdiv CLK(CLK100MHZ, clk2, clk1, clk0);
MUX2 #(1) selclk3(sel, clk0, clk1, clk3);
MUX2 #(1) selclk(stop, clk3, 0, clk);
```

以 sel 信号作为调整时钟快慢的信号，以 stop 信号来控制是否暂停。当 stop 为 1 时，直接将 0 作为时钟信号传入即可。

6 申 A 理由

- 实现了所有的基本指令
- 增加了移位指令 sll,srl,sra

- 增加了与函数调用相关的跳转指令 jal,jr
- 增加了乘法指令 mul
- 为乘法指令和函数调用指令添加了相应的测试样例