MIPS多周期实验报告

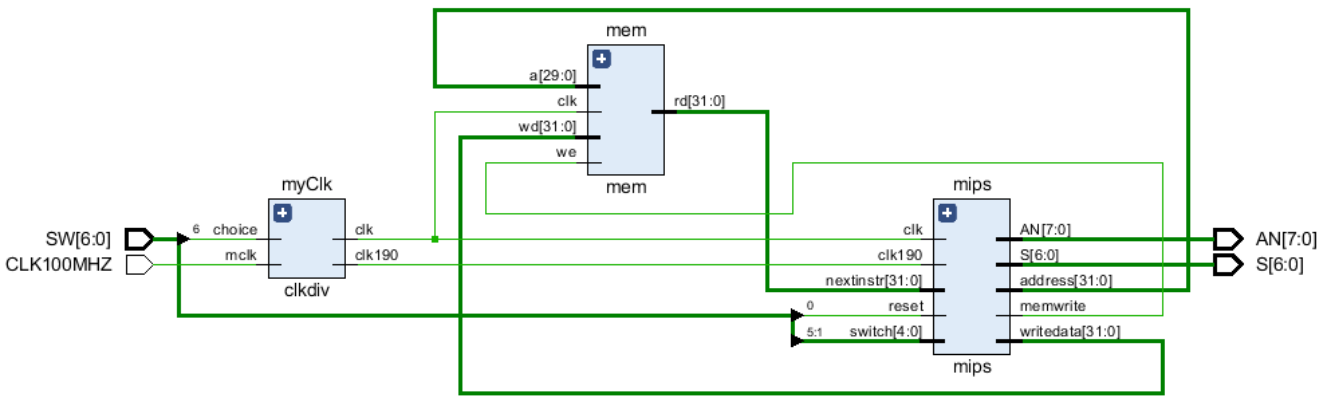
罗翔
17307130191

MIPS多周期实验报告
- 各部件详解
  - Control Unit
  - ALU
  - Reg / IReg
  - Mux
  - Mem
- 添加指令及模拟测试
  - 基本功能测试
  - sra, sll, srl
  - jal, jr
  - mul
- 申A理由
- 参考文献

本次CPU大部分延续上一次单周期的设计思路，除Control Unit做了较大改动外，其他部分内容因为与上次单周期高度重复所以在报告中省略，包括：与单周期相同的部件，展示细节。

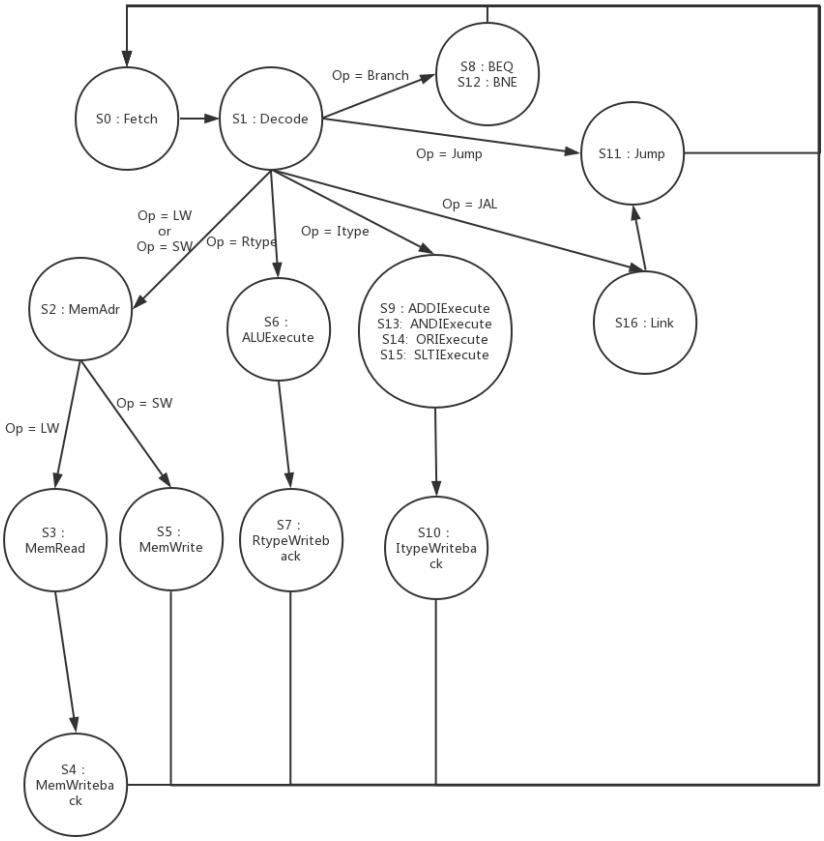
各部件详解

本次MIPS多周期的总体框架如下：



Control Unit

本次多周期的难点主要是有限状态机的设定，本实验中我设计的有限状态机图如下



多周期的核心就在于将原来单周期内完成的指令拆成多个周期完成，以使得多周期中每个周期的时间小于单周期中的时间，因此多周期在一些jump, branch指令较多的进程中能取得比单周期更高的效率(单周期中为保证所有给定指令都能在单个周期内做完，一般以时间最长的指令所需时间作为周期如store指令，它在多周期中可以分为 5 个阶段：Fetch, Decode, MemAdr, MemRead, MemWriteback；而jump指令只需要3个阶段：Fetch，Decode，Jump，多周期只需保证每个阶段(周期)的时间相同，在单周期中必须等待的时间或阶段(浪费CPU资源)在多周期中则可以通过有限状态机的设计来避免)

每个阶段对应的控制信号如下：

阶段	控制信号
Fetch	IorD = 0, AluSrcA = 0, ALUSrcB = 01, ALUOp = 000, PCSrc = 0, IRWrite, PCWrite, exp = 1
Decode	ALUSrcA = 0, ALUSrcB = 11, ALUOp = 000, exp = 1
MemAdr	ALUSrcA = 1, ALUSrcB = 10, ALUOp = 000, exp = 1
MemRead	IorD = 1
MemWriteback	RegDst = 0, MemtoReg = 1, RegWrite
MemWrite	IorD = 1, MemWrite
ALUExecute	ALUSrcA = 1, ALUSrcB = 00, ALUOp = 010, exp = 1
RtypeWriteback	RegDst = 1, MemtoReg = 0, RegWrite
BEQ	ALUSrcA = 1, ALUSrcB = 00, ALUOp = 001,PCSrc = 01, Branch = 01, exp = 1
ADDIExecute	ALUSrcA = 1, ALUSrcB = 00, ALUOp = 000, exp = 1
ItypeWriteback	RegDst = 0, MemtoReg = 0, RegWrite
Jump	PCSrc = 10, PCWrite
BNE	ALUSrcA = 1, ALUSrcB = 00, ALUOp = 001,PCSrc = 01, Branch = 10, exp = 1
ANDIExecute	ALUSrcA = 1, ALUSrcB = 00, ALUOp = 011, exp = 0
ORIExecute	ALUSrcA = 1, ALUSrcB = 00, ALUOp = 100, exp = 0
SLTIExecute	ALUSrcA = 1, ALUSrcB = 00, ALUOp = 101, exp = 1
Link	Jal = 1

ALU

本次我参考计算机组成与系统[1]中的相关内容在ALU中加入Booth乘法，算法如下：

当计算$[X*Y]_{补}$时：

- 乘数最低位增加一位辅助位y_{-1} = 0.
- 判断y_iy_{i-1}的值，决定是 +X、-X，+0.
  - 若$y_iy_{i-1}=01$，则为+X
  - 若$y_iy_{i-1}=10$，则为-X
  - 若y_iy_{i-1}=00 或 11，则为0
- 每次加减后，算术右移一位，得到部分积.
- 重复第(2)和第(3)步n次，得到$[X*Y]_{补}$.

具体代码如下：

```
reg [64:0] c7;
always@(*)
begin
    c7 = 0;
    c7[32:0] = {b,1'b0};
    for(i = 0; i < 32; i = i + 1)
    begin
        case(c7[1:0])
        2'b00, 2'b11:  c7 = {c7[64], c7[64:1]};
        2'b01: begin c7[64:33] = c7[64:33] + a; c7 = {c7[64], c7[64:1]}; end
        2'b10: begin c7[64:33] = c7[64:33] + a_n_1; c7 = {c7[64],c7[64:1]}; end
        endcase
    end
    ...
end
```

Reg / IReg

因为多周期存在时序逻辑，如当前处理的Instruction只有在每条指令的Fetch阶段才能更改，所以需要寄存器IReg来保持Instruction.

```
always@(posedge clk)
    if(en) instr <= rd;
    else instr <= instr;
```

而其他的如从Memory，Regfile中读出的值也需要在下一个时钟上升沿才允许写入寄存器中进而流向下一阶段.

```
always@(posedge clk)
    Dataout <= Datain;
```

Mux

这里需要说明的使用多路选择器的组件有Adr, AluSrcA, AluSrcB, PCSrc.

Adr输出下一个从Memory中读出数据的地址，因为在多周期中我们将指令存储器与数据存储器复用，所以需要指明下一个读出的是指令还是数据，通过IorD确定，如果是指令则令IorD = 0，给出下一条指令的地址PC，如果是数据则令IorD = 1，给出数据的地址Aluout——在MemAdr由ALU算出。

AluSrcA在多周期中有三个作用，在Fetch阶段读取指令时令AluSrcA = 0，计算出PC + 4并作为下一条指令的地址；在Decode阶段译码时令AluSrcA = 0、AluSrcB = 11，将预测的branch指令的跳转地址计算出；在Execute阶段令AluSrcA = 1，将从Regfile中读出的第一个寄存器的值传入ALU进行相应计算。

AluSrcB在多周期中有四个作用，在Fetch阶段令AluSrcB = 01，此时向ALU中传入4；在Decode阶段令AluSrcB = 11，此时向ALU传入branch指令中的PC跳转的差值；在Execute和BEQ/BNE等指令中，令AluSrcB = 00，将Regfile中读出的第二个寄存器的值传入ALU中；在MemAdr中令AluSrcB = 10，将指令中的经符号扩展的立即数传入ALU中作为基地址的偏移量，计算出lw，sw指令在Memory中的对应地址。

PCSrc在多周期中有三个作用，在Fetch阶段中令PCSrc = 00，将PC + 4传入PC寄存器中；在BEQ/BNE中令PCSrc = 01，将对应的目的地址传入PC寄存器；在Jump中令PCSrc = 10，将对应的跳转地址传入PC寄存器中。

Mem

多周期中将指令存储器和数据存储器复用，所以在指令编写时就需要注意存取数据时不要将指令内容覆盖，向Memory中写入数据是时序操作且需要Memwrite = 1，从Memory中读出数据则是组合逻辑，只需要确定指令地址即可。

添加指令及模拟测试

基本功能测试

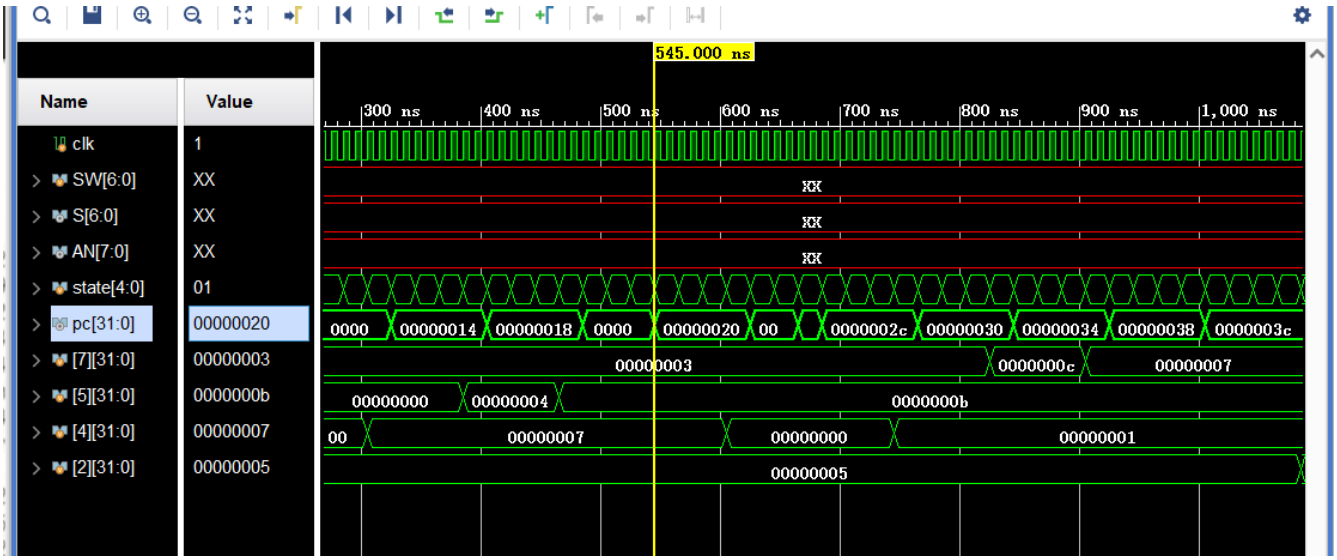
首先测试基本功能，使用数字设计和计算机体系结构[2]书上的随机测试，指令如下

```
#         Assembly          Description              Address    Machine
main:     addi $2, $0, 5    # initialize $2 = 5       0          20020005
          addi $3, $0, 12   # initialize $3 = 12      4          2003000c
          addi $7, $3, -9   # initialize $7 = 3       8          2067fff7
          or   $4, $7, $2   # $4 = (3 OR 5) = 7       c          00e22025
          and  $5, $3, $4   # $5 = (12 AND 7) = 4     10         00642824
          add  $5, $5, $4   # $5 = 4 + 7 = 11         14         00a42820
          beq  $5, $7, end  # shouldn't be taken      18         10a7000a
          slt  $4, $3, $4   # $4 = 12 < 7 = 0         1c         0064202a
          beq  $4, $0, around # should be taken       20         10800001
          addi $5, $0, 0    # shouldn't happen        24         20050000
around:   slt  $4, $7, $2   # $4 = 3 < 5 = 1          28         00e2202a
          add  $7, $4, $5   # $7 = 1 + 11 = 12        2c         00853820
          sub  $7, $7, $2   # $7 = 12 - 5 = 7         30         00e23822
          sw   $7, 68($3)   # [80] = 7                34         ac670044
          lw   $2, 80($0)   # $2 = [80] = 7           38         8c020050
          j    end          # should be taken         3c         08000011
          addi $2, $0, 1    # shouldn't happen        40         20020001
end:      sw   $2, 84($0)   # write mem[84] = 7       44         ac020054
```

这些指令都是测试多周期的基本功能的，选出几个变化较多的寄存器。

$4: 0x7 - 0x0 - 0x1

$7: 0x3 - 0xc - 0x7



sra, sll, srl

```
  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:                |
```

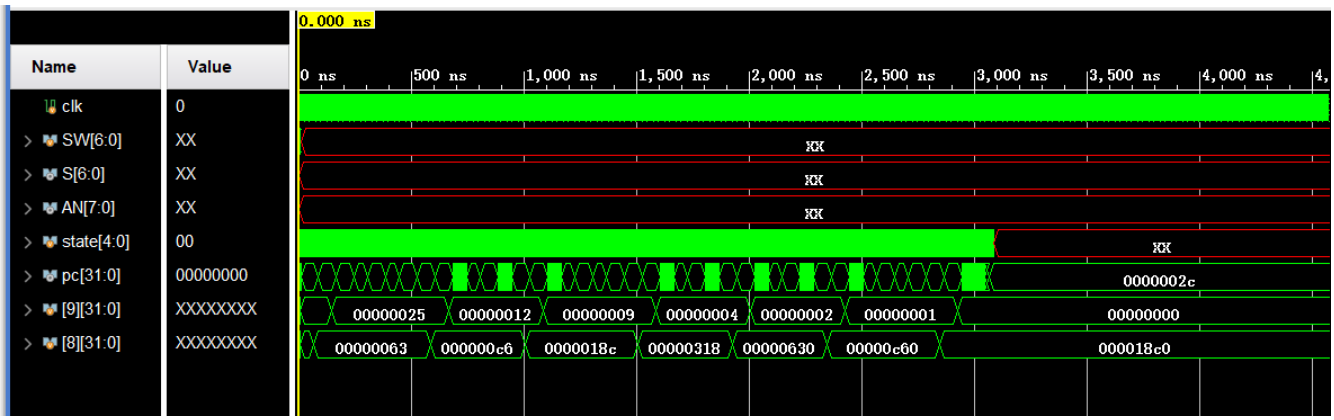
这些指令都属于Rtype类指令，故只需要修改Aludec、ALU即可。在Aludec中为这些指令设置相应的alucontrol，ALU中需要传入对应的移位数shamt。因为shamt在Decode阶段就读取出来了，但要到Execute阶段才会使用，所以可以加一个寄存器延迟一个周期。其中逻辑左移和逻辑右移可以通过移位操作实现，算术右移可通过如下方法实现：

```
assign c6 = ({{31{b[31]}}, 1'b0} << (~shamt) )| (b >> shamt);
```

样例中$t0逻辑左移，$t1逻辑右移直至$t1 = 0.

$t0: 0x63 - 0xc6 - 0x18c - 0x318 - 0x630 - 0xc60 - 0x18c0

$t1: 0x25 - 0x12 - 0x9 - 0x4 - 0x2 - 0x1 - 0x0



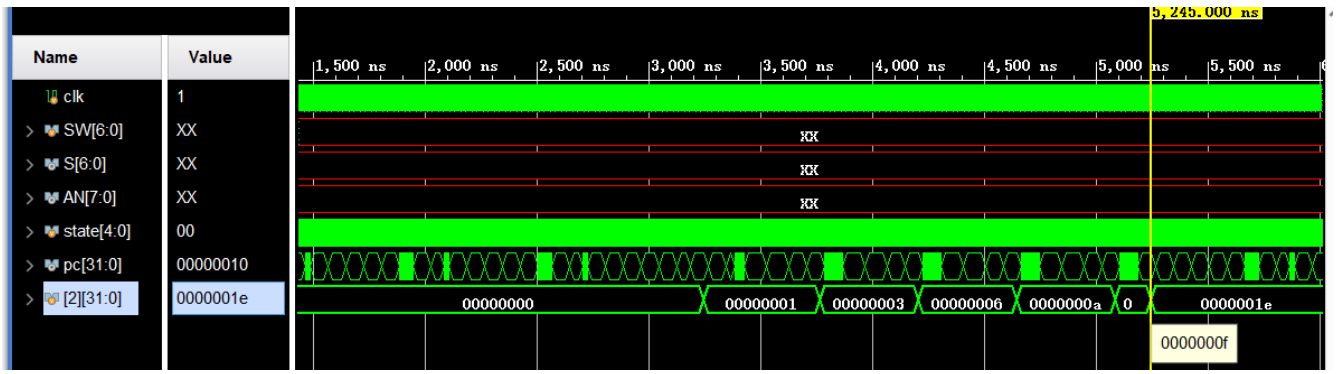
jal, jr

```
 0x0 : addi $sp, $0, 128    | 201d0080
 0x4 : addi $a0, $0, 5      | 20040005
 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 : add $v0, $a0, $v0    | 00821020
0x4c : jr $ra               | 03e00008
```

jal是jump and link指令，需要先将PC + 4的值存入$ra中，然后jump到指定地址，jr是将寄存器中的值作为地址跳转，一般jr与jal联合使用可以用来模拟简单的栈操作。在实现中，我将jal分为四个阶段，在第三阶段Link中，需要将PC + 4存入$ra中，因为此时已经完成Fetch阶段，PC寄存器中的值，已经更新为PC + 4，此时只需要令新加入的信号Jal = 1，就可以将PC存入到$ra中，然后跳转。jr只需将Regfile中读出的第一个寄存器的值作为下一个PC的地址传入PC寄存器即可。因为jr是Rtype类指令，所以我在Aludec里加入了一个新的ret信号传入datapath中，这里我又加入了一个二路选择器，当ret = 1时PC值被置为Regfile中读出的第一个寄存器的值。

本测试样例是在模拟栈操作过程，每次将$a0存入4($sp)中，将$ra存入0($sp)中，当$a0 < 2时跳转到factorial，从栈中取出相应的$a0进行累加操作，最终$vo为0xf.

$v0: 0x1 - 0x3 - 0x6 - 0xa - 0xf

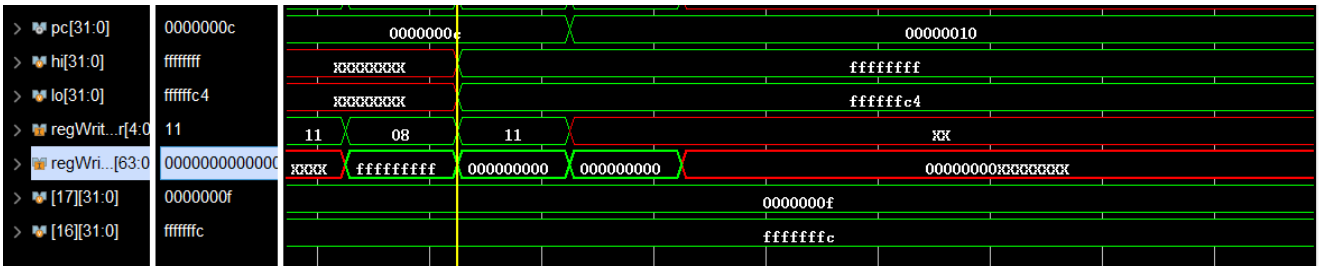


mul

乘法的具体实现原理以在ALU中详细讲述，现对此进行测试

```
0x0 : addi $s0, $0, 12      | 2010fffc
0x4 : addi $s1, $0, 15      | 2011000f
0x8 : mul $s0, $s1          | 02110018
```

这里$-4 * 15 = -60$(即0xfffffffc * 0xf = 0xffffffc4)，且因为乘法的结果是64位，需要专门保存在{hi, lo}寄存器中，结果如下



申A理由

本次多周期实验中我额外添加了sra, sll, srl, jal, jr, mul等指令以支持移位操作，模拟简单的栈的过程以及乘法运算，并且设计了有效的有限状态机以实现完备性。

参考文献

1. 戴维·莫尼·哈里斯，莎拉 L. 哈里斯. 数字设计和计算机体系结构（原书第 2 版）
2. 袁春风. 计算机组成与系统结构
3. 除随机测试外，其余所有测试指令皆参考张作柏同学GitHub上的相关测试指令https://github.com/Oxer11/MIPS/tree/master/Assembler/example

---

1. 袁春风. 计算机组成与系统结构↩

2. 戴维·莫尼·哈里斯，莎拉 L. 哈里斯. 数字设计和计算机体系结构（原书第 2 版）↩