The SME309 Project Introduction

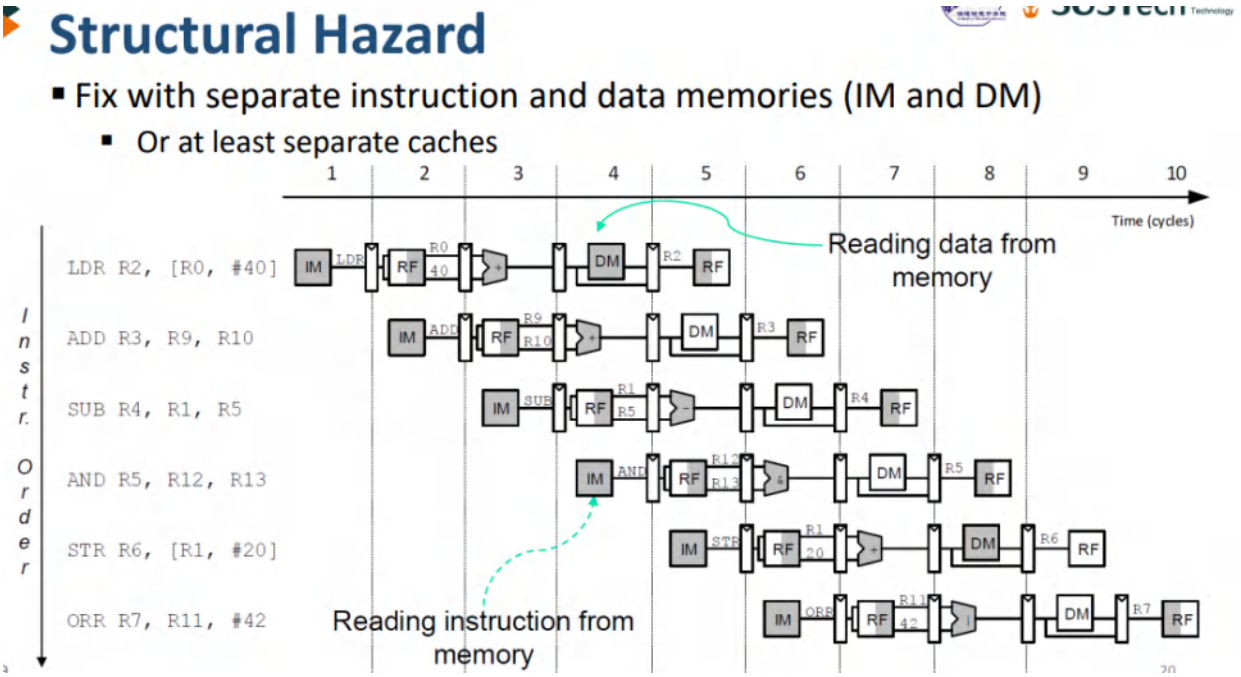
贡献比:每人25%

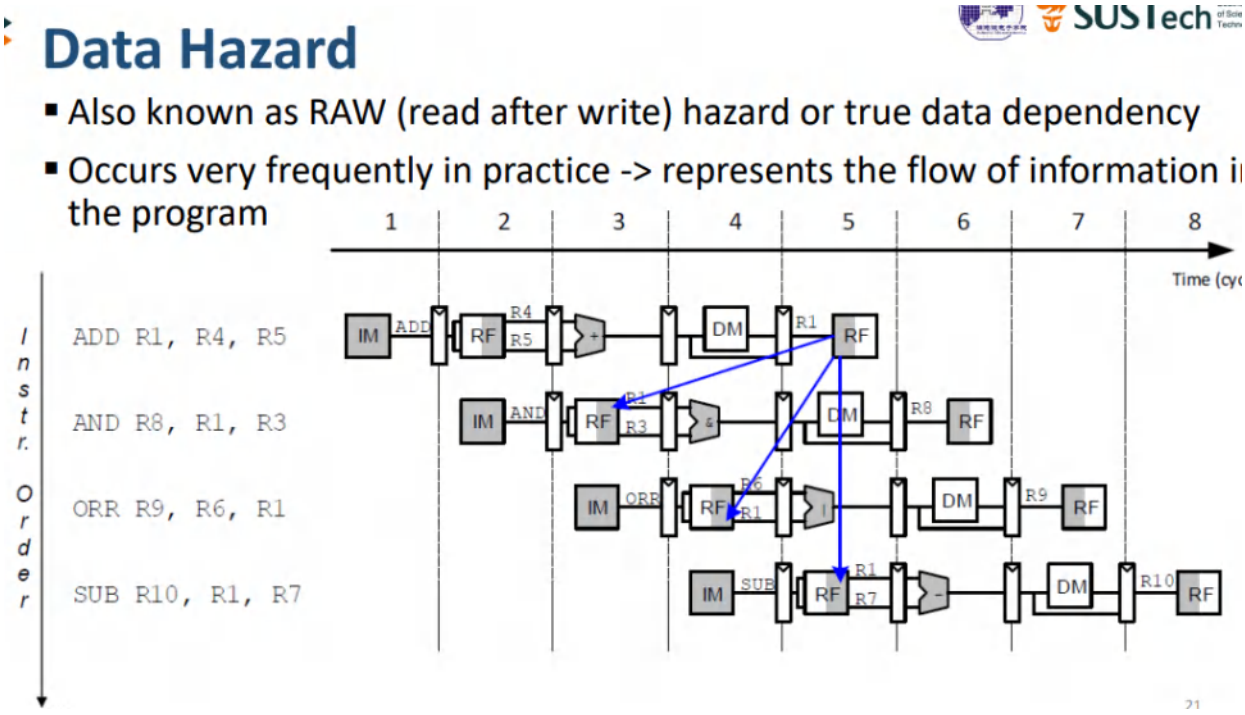
12110206 王启帆 12111031 郑振涛 12111008屠耘诚 12111605田伟彤

> 本次项目的文件夹中的RTL文件为Q1-Q5的verilog代码文件，RISCV32I文件夹为Q6的答案，bonus文件夹中为实现的超标量处理器。其中所有的仿真Instr在各个Wrapper中，在每个上面标着仿真的题号。

Hazards：

任何一本讲解CPU的流水线设计的教科书，都会提到流水线设计需要解决的三大冒险，分别是**结构冒险**（Structural Hazard）、**数据冒险**（Data Hazard）以及**控制冒险**（Control Hazard）。



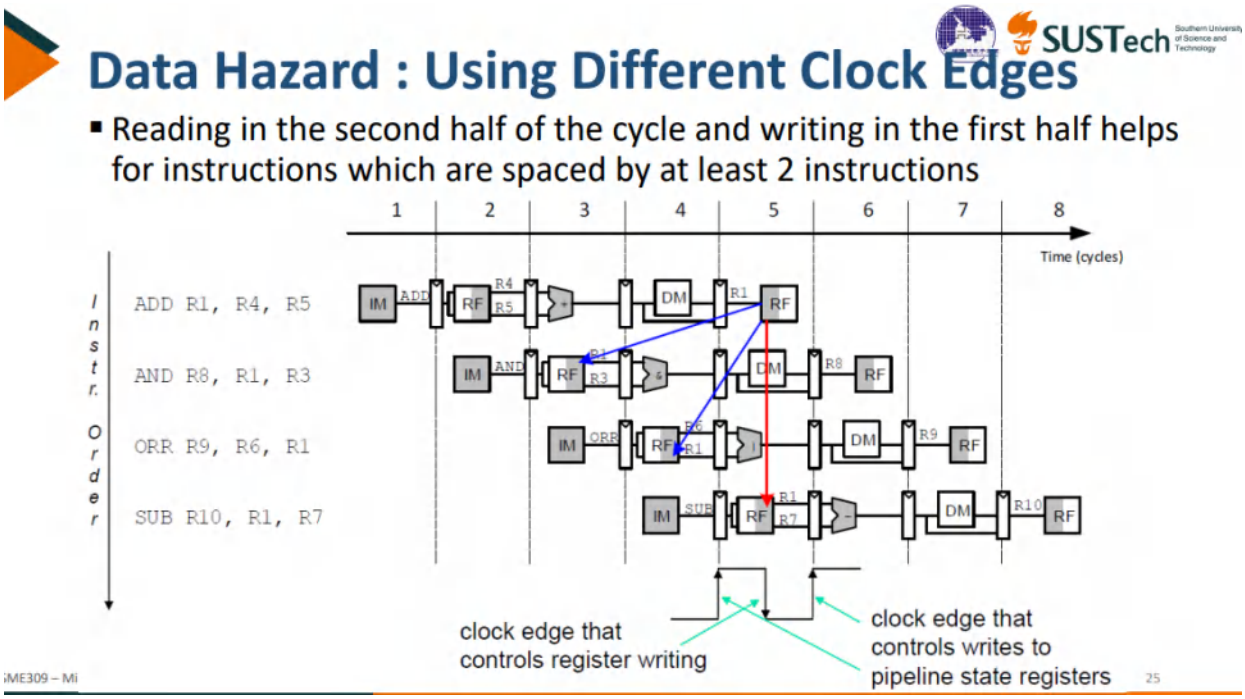


Data Hazard：当指令依赖于前一个指令的结果，而该指令结果尚未计算时，就会发生数据危害。每当两个不同的指令使用相同的存储时。该位置必须显示为按顺序执行。

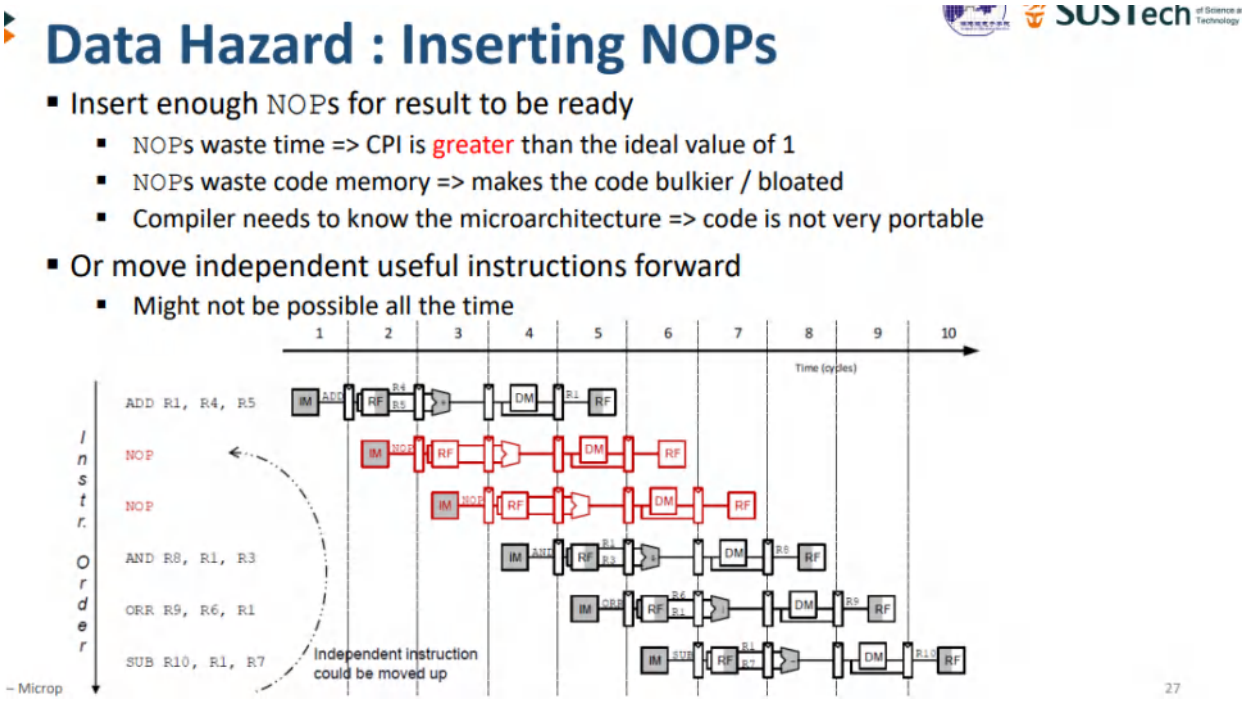
Structure Hazard: 结构冒险的本质是硬件层面资源的竞争。CPU 在同一个时钟周期，同时在运行两条计算机指令的不同阶段。但是这两个不同的阶段，可能会用到同样的硬件电路。

The resolution of this harzard:

1. **Using Different Clock Edges**: Reading in the second half of the cycle and writing in the first half helps for instructions which are
   spaced by at least 2 instructions.

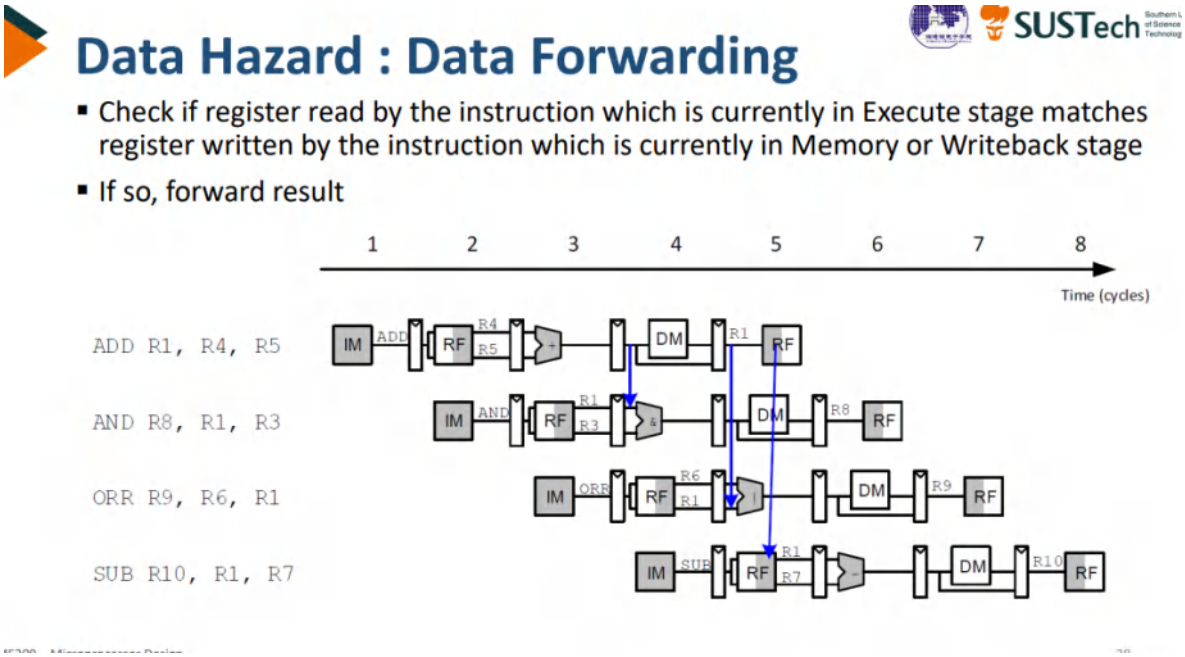



```
2. **Stalling (inserting NOPs)** :
```



In some cases, forwarding still won't solve the harzard and we'll need to use stalling. For example, after a data is load from memory, and executed in the next instruction. We need to insert NOPs to solve the harzard

3. **Data Forwarding**



Memory-to-Execute or Writeback-to-Execute copy

Source register for instruction in Execute stage matches destination register for instruction in Memory stage or Writeback stage.

Memory-Memory copy :

Check if the register used in Memory stage by the STR instruction matches register written by LDR in Writeback stage.

Control Hazard: 尝试在评估条件和计算新的 PC 目标地址之前对程序控制流做出决策

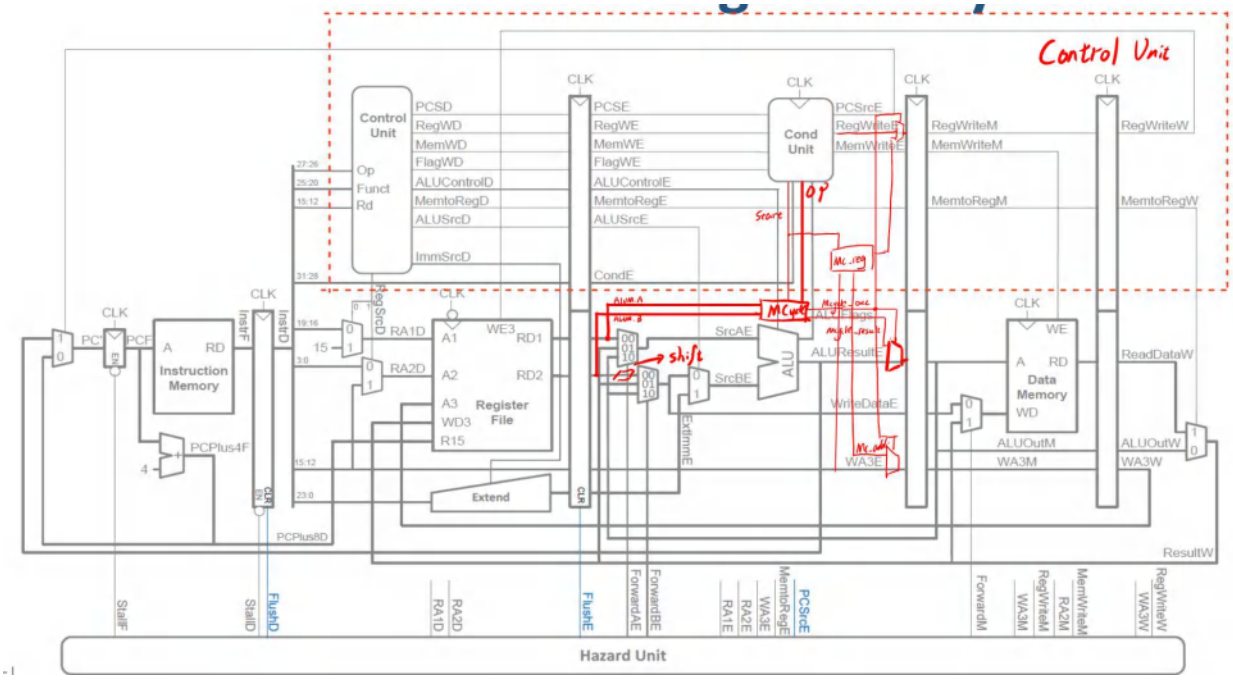
Q1

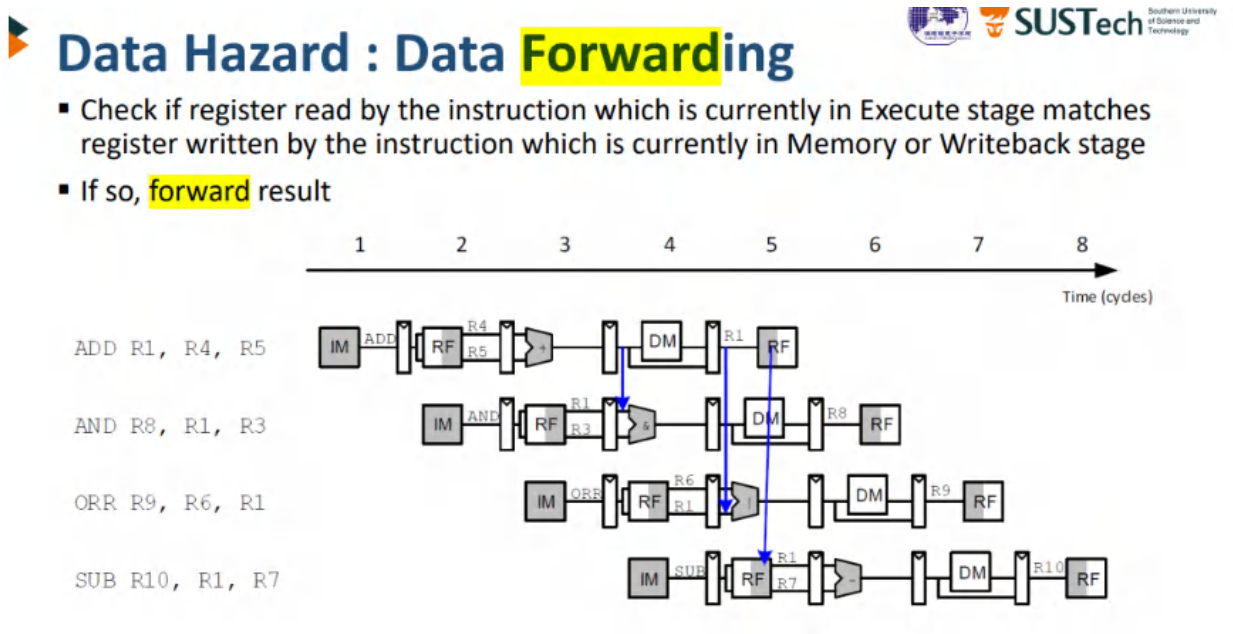
第一问是让我们完成一个处理器的流水线结果，并且用Forwarding，Stall，Flush来处理各类出现的Hazard。

Hazard 分类（Data hazard, Control hazard）
下面我们分别讲一下我们的解决策略，由于这部分上课讲的也比较清晰，所以我们不做过多的解释。

对于Forwarding的解决策略，检查当前处于执行阶段的指令是否读取与当前处于存储或写回阶段的指令写入的寄存器匹配。

将结果从存在于任何后续（在代码中较早）流水线状态寄存器中的最早阶段/点（在代码中较新）转发到需要它的功能单元（例如，ALU）的那个周期。





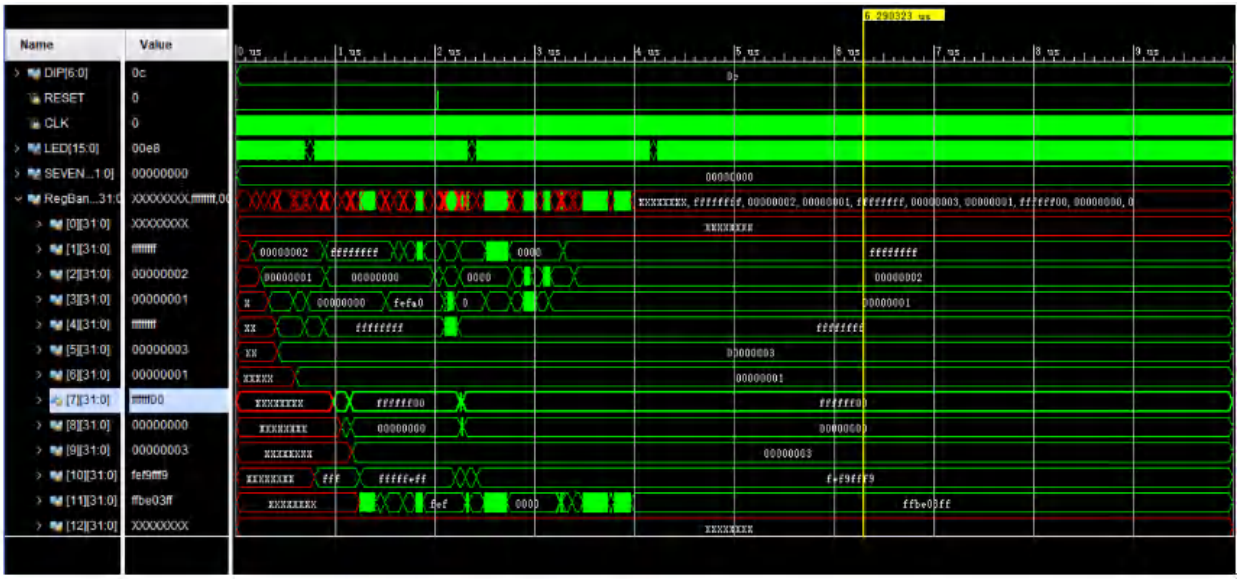
以下是我们不太一样的地方：

（1）我们的寄存器模块分为两部分，一部分是data的寄存器，一部分是condition的寄存器。主要原因是希望在之前的代码基础上不作过多的改动，之前代码Decoder和CondLogic都在Control Unit里面，所以这边我们在Control Unit里面添加寄存器，状态的寄存器包括了D2E，E2M，M2W。并在某个信号该输出的地方输出，不会将其它多余的wire信号放在我们的顶层中，简化了代码，增加了可读性。

（2）由于新的图不包括shift和Mcycle模块，所以我们这边要添加Shift和Mcycle。我们把这两个值添加到如下图所示的地方，shift添加到E层,主要原因是Mcycle使用的是不移位的Register值。所以我们要保留原本的操作数留到E层，一个直接给MCycle，一个给ALU部分。

（3）对于Mcycle，我们将Mcycel和之前一样，放到和ALU同级别的地方，可以这样认为，使用Mcycle的时候就不能用ALU，当Mcycle运行的时候，将Busy信号输出，把所有的寄存器模块阻塞（第一问）。其它对于更详细的Mcycle流程，就不在Q1地方详细描述了，我们会将这部分移动至Q2的时候介绍它的工作流程.

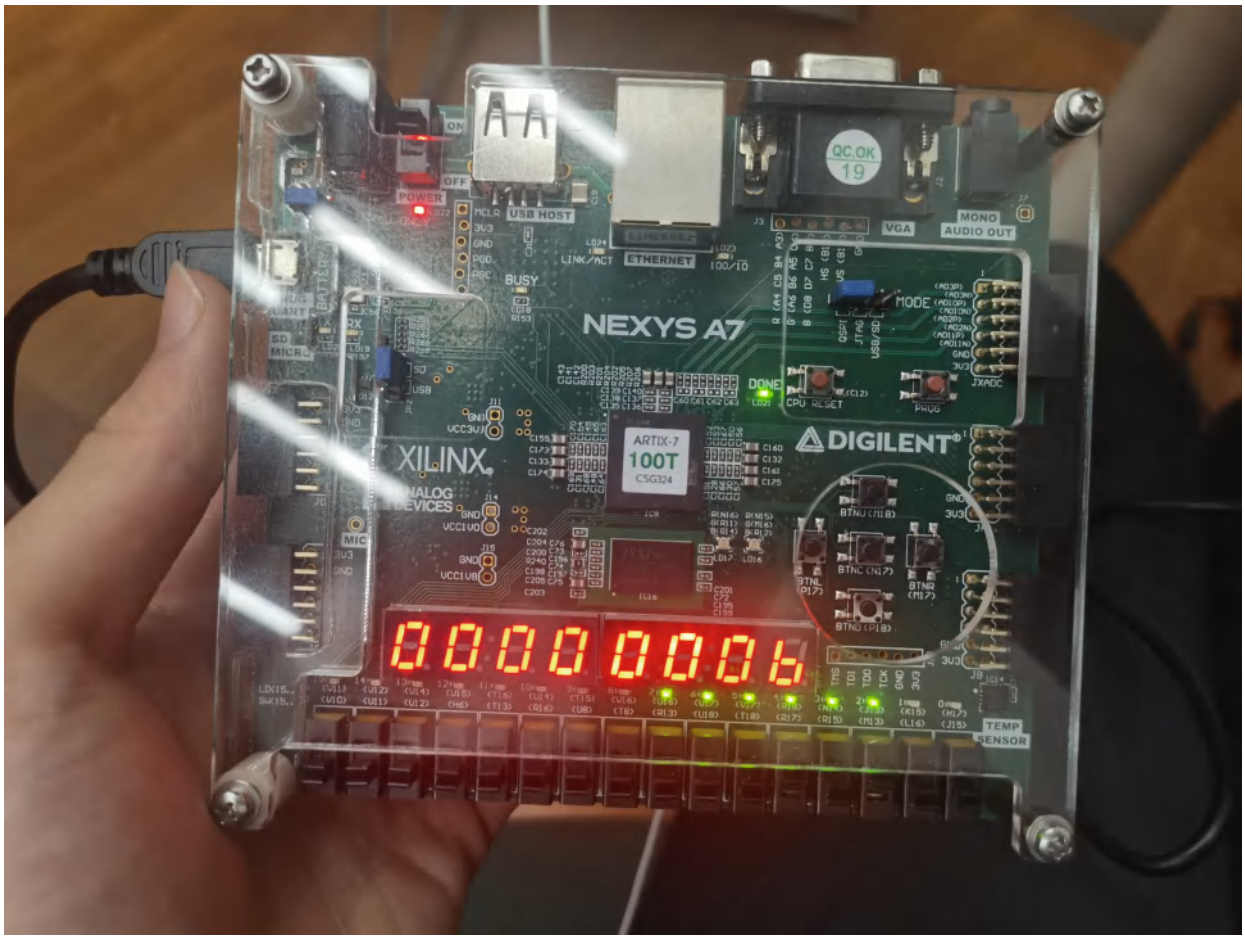
我们利用Lab2的testbench进行了充分的仿真，仿真波形图如下（regbank）：

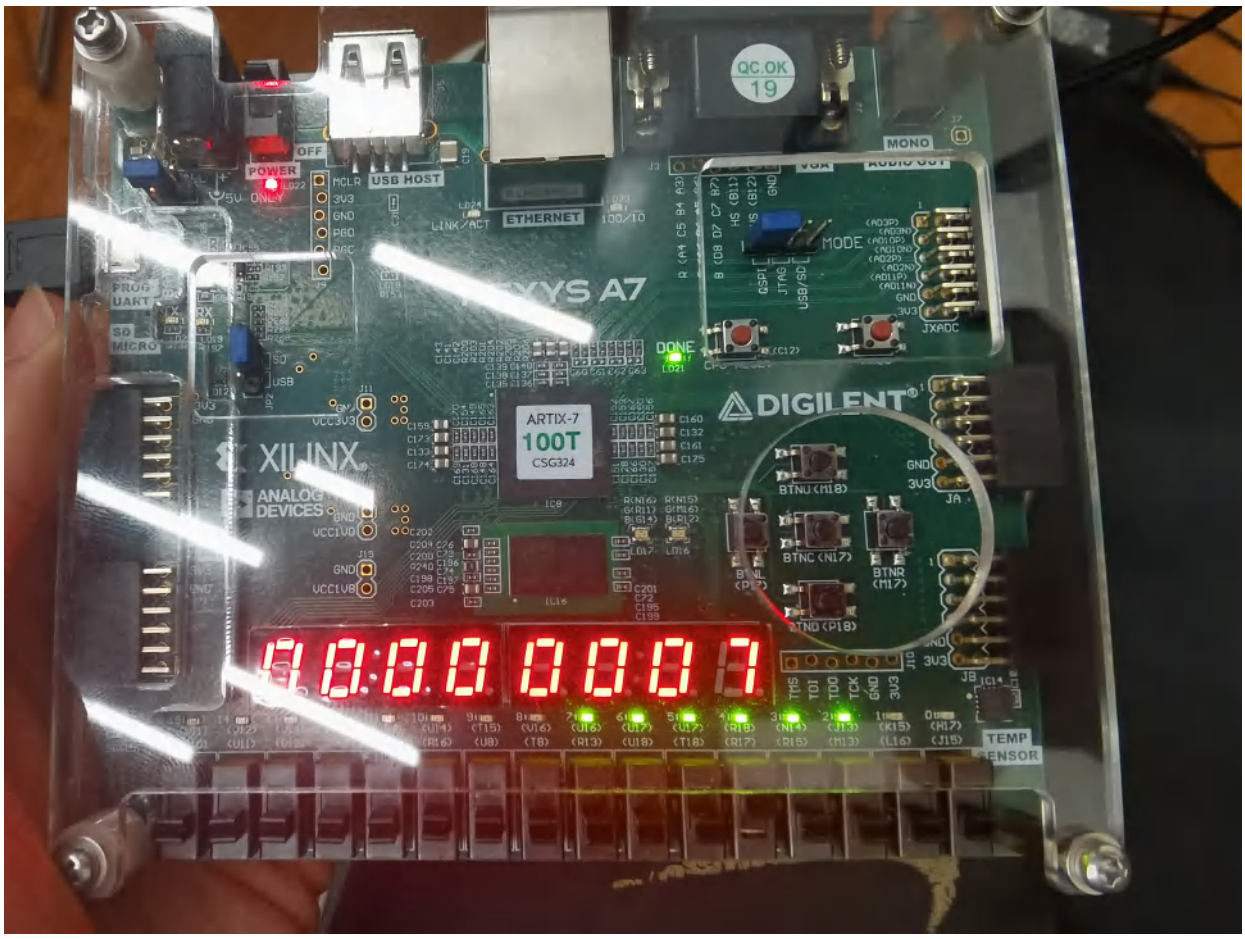


汇编语言如下：

```
    AREA    MYCODE, CODE, READONLY, ALIGN=9
      ENTRY

; ------- <code memory (ROM mapped to Instruction Memory) begins>

    LDR R1, constant1;
    LDR R2, constant2;
    LDR R3, addr1;
    LDR R4, addr2;
    ADD R5, R1, R2; R5 = a1 + a2;

    STR R5, [R3,#4];
    ADD R3, R3, #8;
    LDR R5,[R3,#-4]; R5 = a1 + a2;

    SUB R6, R1, R2;  R6 = a1 - a2;
    STR R6, [R4,#-4];
    SUB R4, R4, #8;
    LDR R6,[R4,#4];  R6 = a1 - a2;

    AND R3, R1, #0x00000000; R3 = 0
    LDR R10, constant3;
    ORR R4, R1, R10; R4 = 0xFFFFFFFF;
    ORR R1, R1, R4; R1 = 0xFFFFFFFF;
    AND R2, R2, R3; R2 = 0;

    AND R7,R1,R1,LSL #16; R7 = 0xFFFF0000;
    AND R8,R1,R1,LSR #16; R8 = 0x0000FFFF;
    AND R7,R1,R7,ASR #8; R7 = 0xFFFFFF00;
    AND R8,R1,R8,ASR #16; R8 = 0;
```

下面是我们的上板结果：





Q2 Non-stalling CPU for multi-cycle instructions.

在Non-stalling的处理中，我们可以通过在MCycle运行过程中不出现Hazard开始，一步步的来思考我们需要做什么。

1、首先第一步就是，我们需要记录MCycle计算的数据输出到哪里去，这里就要保存下Addr的值；

2、同时当数据计算出来的时候，我们暂停掉当前即将输出E的数据（并保存），并将这个数据替换成MCycle的输出值。

3、最后对于Condition Unit模块输出值，PCSrc = 0, RegWrite = 1&Exution(是否运行)，MemWrite = 0，这几个值都要在MCycle输出的周期同时输出。

搞清楚不出现Hazard的情况，出现Hazard的情况就比较好做了。

1、判断Hazard用我们第一步存下的MCycle的目标地址来和RA1和RA2来进行判断或者下一个指令是否要启动MCycle，如果相同，将会对F，D stage暂停，对E stage清除（相当于插入Nop），这一步相当于暂停前面F，D，E stage的运行，并让M和W周期的值不受影响（因为他们没有问题）。直到MCycle完成计算。

在考虑完需要做什么之后，下一步我们进行分析，怎么实现：

对于第一条，我们需要构造一个reg寄存器，来存输出地址的值。

对于第二条，我们构建了一个MCycle_out的值，MCycle_out = Busy_reg & ~i_Busy; Busy_reg是保存上一个周期是否是Busy，这个代码其实就是为了得到Busy的下降沿周期，在该周期我们进行暂停并输出真实结果。这个结果将会连接到Stall F, D, E，暂停这三个Stage的信号。

对于第三条，我们再次利用MCycle_out这个值，当用Mcycle_out拉高时，这三个信号分别置为0，1，0。在实际设计的过程中，我们不需要考虑是否运行的情况，因为当该指令不运行的时候Start信号不会来搞，整个MCycle模块都不会运行。

对于Hazard的情况，我们新创立MCycle_Stall = MCycle_Stall = ((MCycle_addr=RA1) || (MCycle_addr=RA2) || StartD) && Busy;

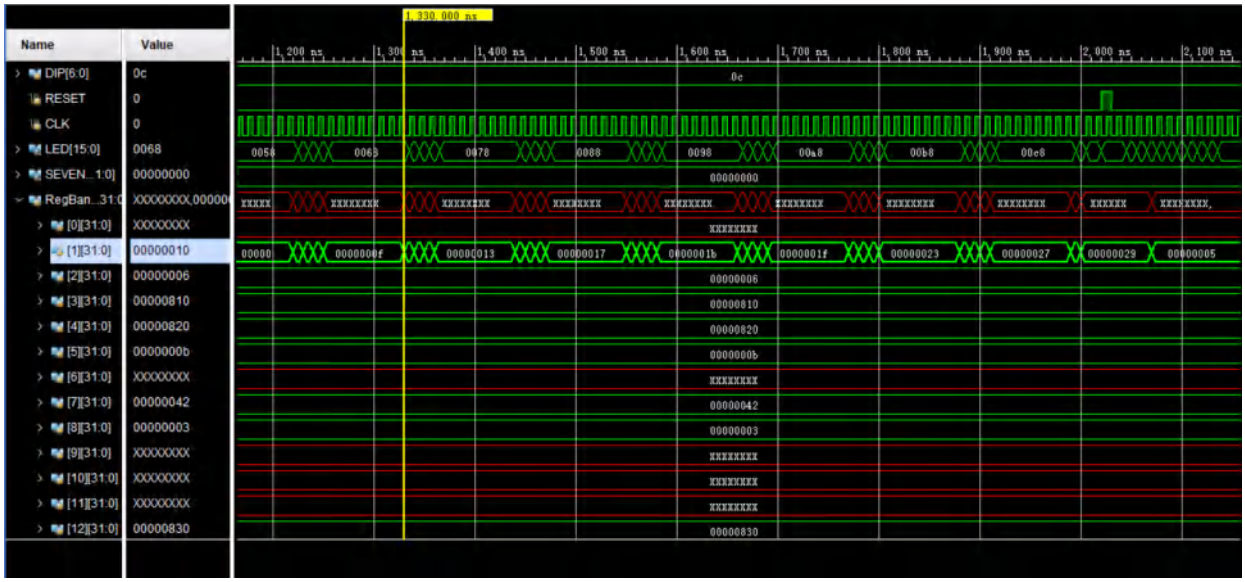
在最后的仿真过程中，我们发现当两个MCycle指令接连到达的时候，会出现MCycle出现Bug。为了解决两个指令连续进入的bug，bug产生的原因是：

```
// state machine
always @(posedge CLK or posedge RESET) begin
  if(RESET)
    state <= IDLE;
  else
    state <= n_state;       You, 上周 • FPU be included
end

always @(*) begin
  case(state)
    IDLE: begin
      if(Start & ~signal_reg) begin
        n_state = COMPUTING;
        Busy = 1'b1;
      end
      else  begin
        n_state = IDLE;
        Busy = 1'b0;
      end
    end

    COMPUTING:  begin
      if(~done) begin
        n_state = COMPUTING;
        Busy = 1'b1 ;
      end
      else begin
        n_state = IDLE;
        Busy = 1'b0;
      end
    end
  endcase
end
```

每次乘法指令执行完，n_state会变到IDLE状态；State在下一个上升沿的时候会被赋值IDLE，同时经过组合逻辑n_state变为COMPUTING。在下图这个部分会直接进入从computing并不执行赋初值的命令，所以我们在上图代码中加入一个暂停信号signal_reg,当计算完成之后会有一个周期的赋初值，避免这个bug。

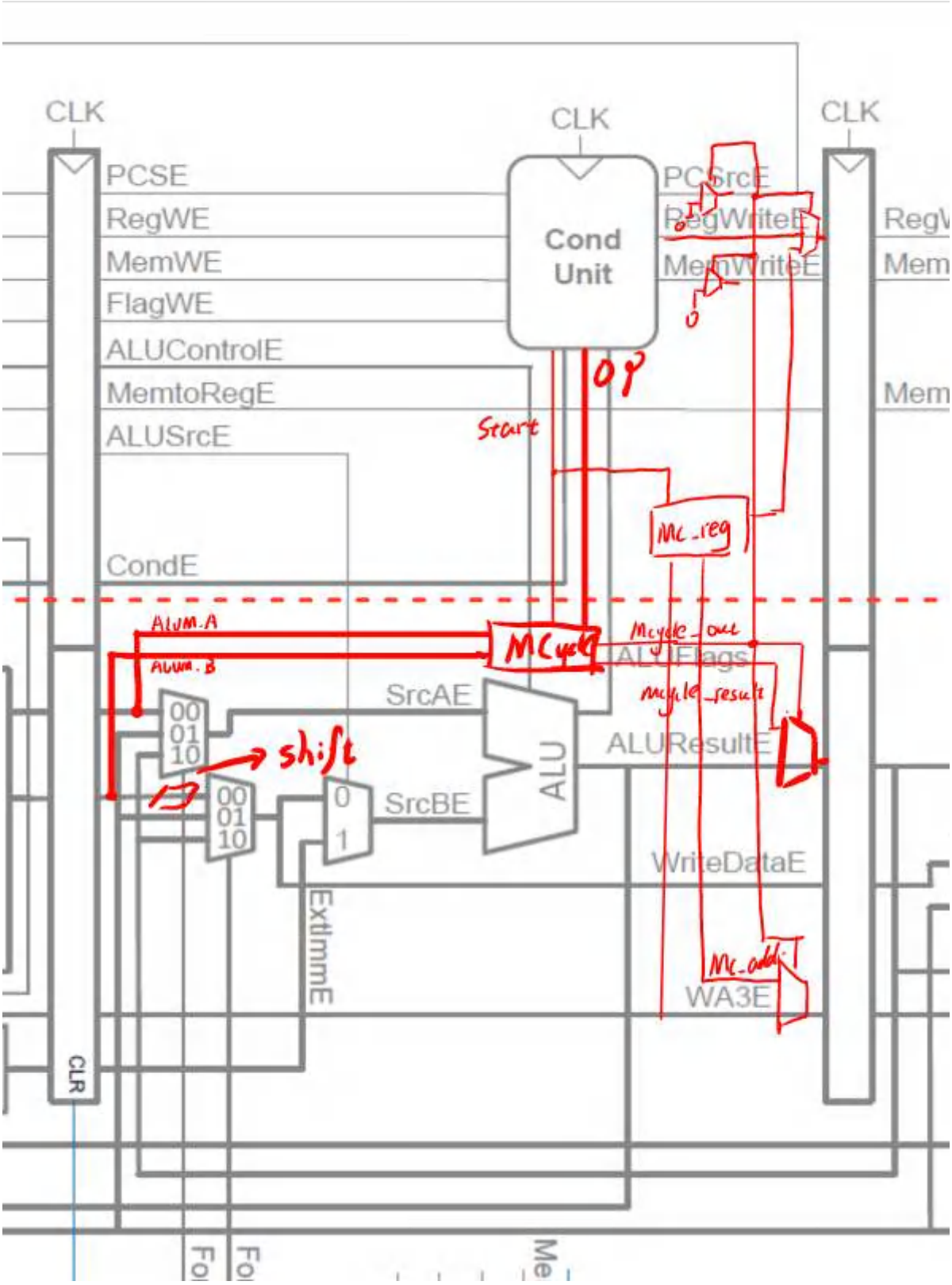
```
// state: IDLE
else if(state == IDLE) begin
  if(n_state == COMPUTING) begin
    count <= 0 ;
    shifted_op1 <= {{(WIDTH-1){1'b0}},Operand1};
    op2 <= Operand2;
    done <= 0;
    reg_op <= MCycleOp;
    // sign_extend <= 1'b0;
  end
  // else IDLE->IDLE: registers unchanged
end
// state: COMPUTINGq
else if(n_state == COMPUTING)
begin
  case (reg_op)
    2'd0: begin
      if(count == WIDTH-1) begin // last cycle
          done <= 1'b1;
          count <= 0;
        end else begin
          done <= 1'b0;
          count <= count + 1;
        end
        if(shifted_op1[0])  begin
          shifted_op1 <= {op2, {(WIDTH-1){1'b0}}} + shifted_op1[2*WIDTH-1:1];
        end
        else  begin
          shifted_op1 <= {1'b0, shifted_op1[2*WIDTH-1:1]};
        end
        // else temp_sum unchanged
```

经过上述步骤Q2完成，下面是我们的tb波形验证：

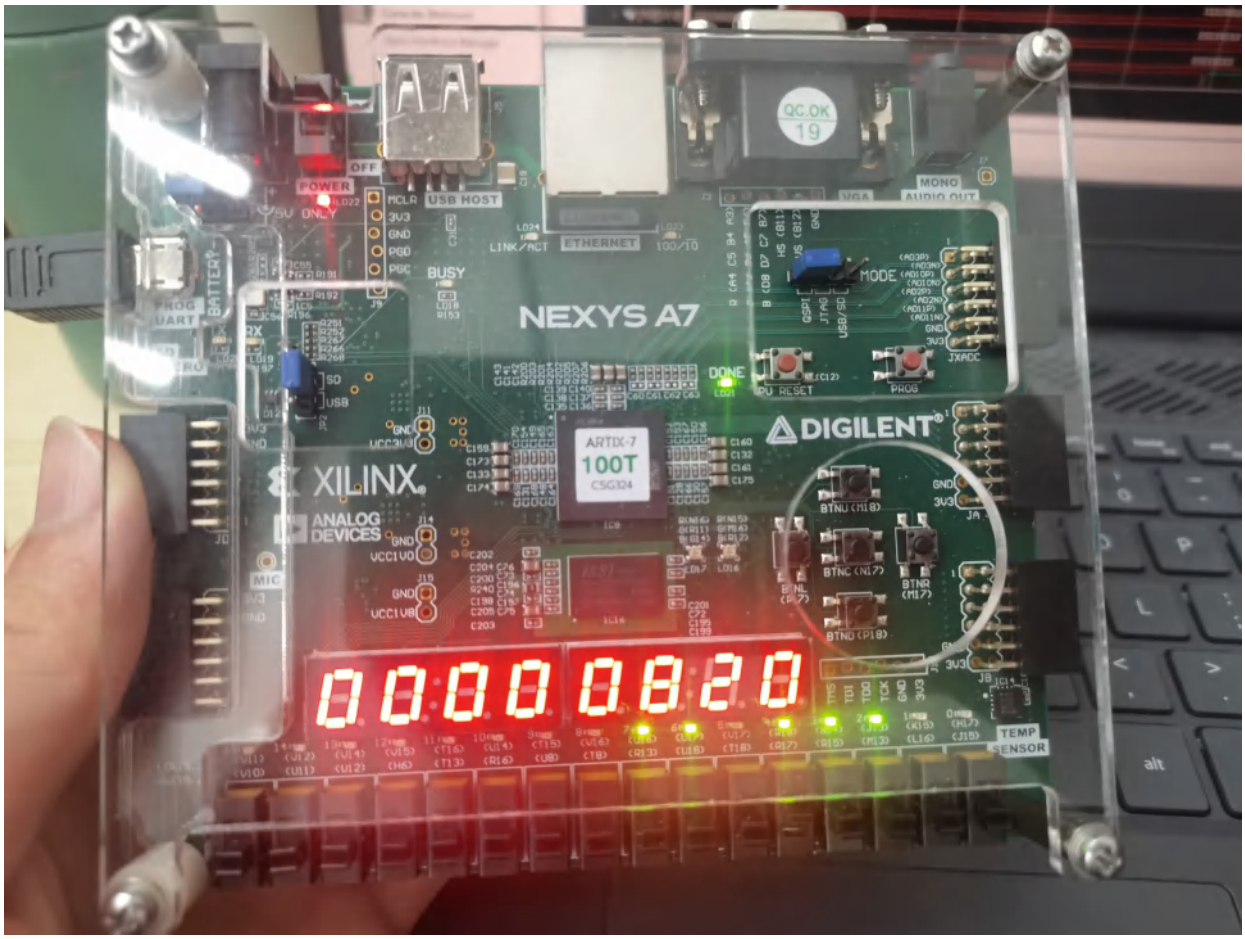


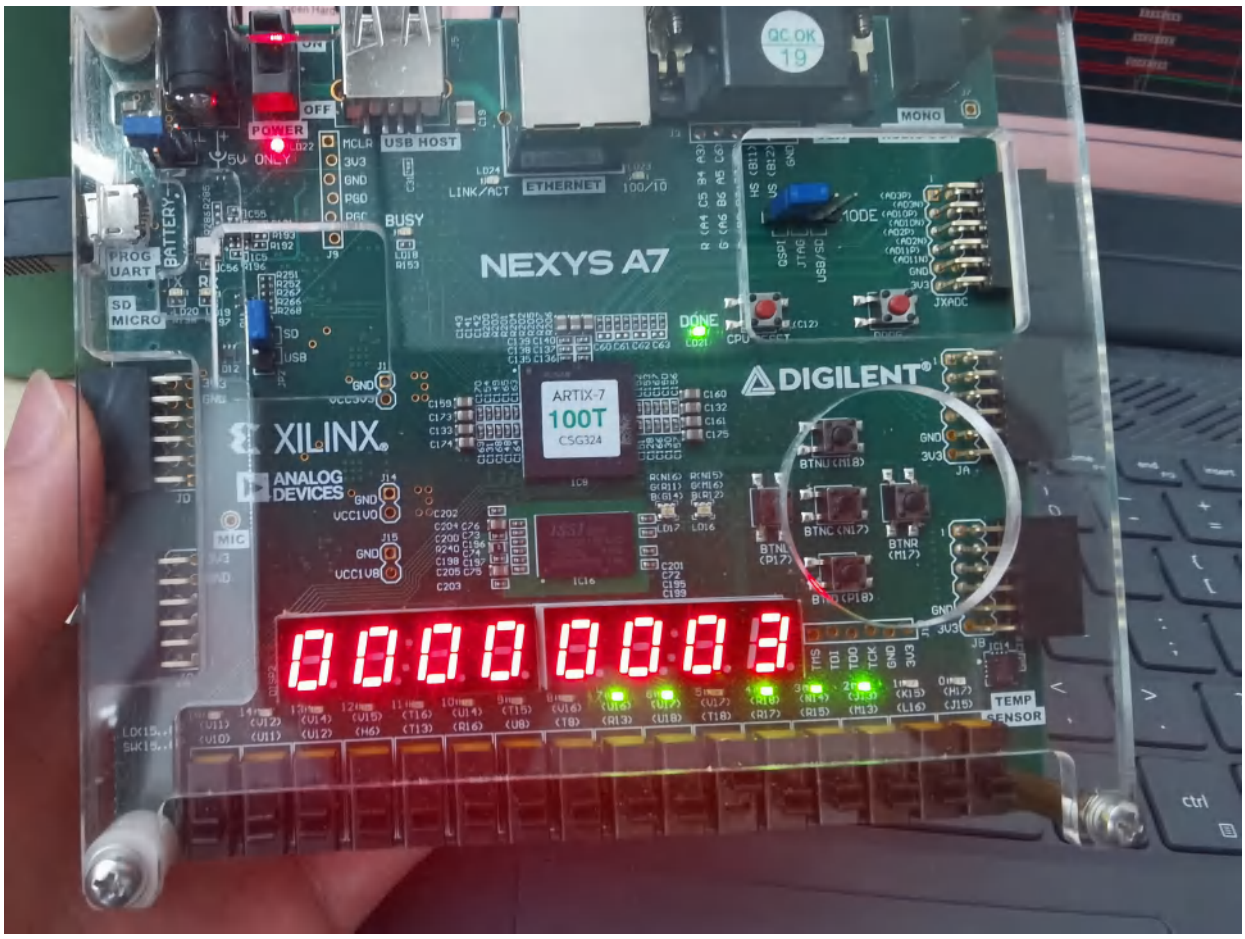
下面是我们的tb的汇编语言：

```
    AREA    MYCODE, CODE, READONLY, ALIGN=9
      ENTRY

; ------- <code memory (ROM mapped to Instruction Memory) begins>

    LDR R1, constant1; R1=5
    LDR R2, constant2; R2=6
    LDR R3, addr1; 810
    LDR R4, addr2; 820
    LDR R12,addr3; 830
    ADD R5, R1, R2; R5 = a1 + a2;

    MUL R7,R5,R2;R7=66
    LDR R8,constant3; R8=3
    LDR R3,number0;R3=0
    MULEQ R7,R1,R8; not execute,R7=66
    ADDS R3,R3,#0; SET Z FLAG = 1
    MULEQ R10,R1,R8; R10=15;
    ADDS R10,R10,R7; R10 =66+15=81,flags are 0
    ADD R1,R1,#1; R1 =82
    ADD R1,R1,#1; R1 =83
    ADD R1,R1,#1; R1 =84
    ADD R1,R1,#1; R1 =85
    ADD R1,R1,#1; R1 =82
    ADD R1,R1,#1; R1 =82
    ADD R1,R1,#1; R1 =82
    ADD R1,R1,#1; R1 =82
    ADD R1,R1,#1; R1 =82
    ADD R1,R1,#1; R1 =82
    ADD R1,R1,#1; R1 =82
    ADD R1,R1,#1; R1 =82
    ADD R1,R1,#1; R1 =82
    ADD R1,R1,#1; R1 =82
    ADD R1,R1,#1; R1 =82
```


CLK
CLK
CLK
PCSE
RegWE
MemWE
FlagWE
ALUControlE
MemtoRegE
ALUSrcE
CondE
Cond Unit
PCSrcE
RegWriteE
MemWriteE
op
start
Mc_reg
ALUA
ALUB
MCycle
Mcycle_out
ALUFlags
Mcycle_result
SrcAE
00
01
10
shift
ALU
ALUResultE
SrcBE
0
1
WriteDataE
ExtImmE
Mc_addr
WA3E
CLR

上板测试如下：


NEXYS A7
DIGILENT
XILINX
ANALOG DEVICES
ARTIX-7
100T
CSG324
00000820


NEXYS A7
DIGILENT
XILINX
ANALOG DEVICES
ARTIX-7
100T
CSG324
00000003

Q3 Expand the ARM processor to support all the 16 Data Processing Instructions.

在写第三题的时候，我们详细读了Arm的开发手册：

2. Data-processing Instructions

According to the ARM Architecture Reference, we can get the Opcode and actions of 16 data-processing instructions.

Opcode	Mnemonic	Operation	Action
0000	AND	Logical AND	Rd := Rn AND shifter_operand
0001	EOR	Logical Exclusive OR	Rd := Rn EOR shifter_operand
0010	SUB	Subtract	Rd := Rn - shifter_operand
0011	RSB	Reverse Subtract	Rd := shifter_operand - Rn
0100	ADD	Add	Rd := Rn + shifter_operand
0101	ADC	Add with Carry	Rd := Rn + shifter_operand + Carry Flag
0110	SBC	Subtract with Carry	Rd := Rn - shifter_operand - NOT(Carry Flag)
0111	RSC	Reverse Subtract with Carry	Rd := shifter_operand - Rn - NOT(Carry Flag)
1000	TST	Test	Update flags after Rn AND shifter_operand
1001	TEQ	Test Equivalence	Update flags after Rn EOR shifter_operand
1010	CMP	Compare	Update flags after Rn - shifter_operand
1011	CMN	Compare Negated	Update flags after Rn + shifter_operand
1100	ORR	Logical (inclusive) OR	Rd := Rn OR shifter_operand
1101	MOV	Move	Rd := shifter_operand (no first operand)
1110	BIC	Bit Clear	Rd := Rn AND NOT(shifter_operand)
1111	MVN	Move Not	Rd := NOT shifter_operand (no first operand)

在对ARM处理器进行功能扩展的研究中，我们首先详细阅读了ARM的开发手册。在ARM处理器的16种数据处理功能中，我们已经完成了6种。现在我们将描述如何扩展剩余10种功能：

ADC，BIC，EOR，MOV，MVN，RSB，RSC，SBC，TEQ，TST

需求分析

ADC：主要是增加了Carry信号的判断；

BIC：和AND很像，一个是保留置为1的位置，一个是清除置为1的位置，都可以用AND；

EOR：异或，是之前没有实现的操作类型，需要增加ALU输入的Control位宽，并增加异或单元。

MOV: 传送，可以选择将加或减的第二位置零，也可以用新的方法；

MVN：同样是传送，MOV取反即可；

RSB：反向减法，B-A，在A和B进加法器的取反前反转一次；

RSC：反向减法加Carry信号 -！Carry；

SBC：借位减法， -！Carry；

TEQ：EOR加上Nowrite

TST：AND加上Nowrite

实现策略

1、对于SBC、ADC和RSC我们往ALU中引入了Carry信号，并且用Carry_use来判断是否需要使用Carry信号

wire Carry_ALU = Carry_pre & Carry_use;

用这个方法，无论是否使用Carry在加法器里面都用同一个信号Carry_ALU，保持了一个加法器。

2、对于BIC、MVN加入Reverse_B信号（从decoder中判断），Re_src = Reverse_B ? ~ Src_B : Src_B BIC和AND用同一种方法，在输入进方法前判断是否用反转。同理MVN和MOV也共用一种方法。

3、对于EOR增加异或符号 3'b100：ALUResult = Src_A ^ Src_B。在增加异或的同时，考虑到我们要用到三位ALUControl。但我们只用了5种操作符，有三种操作符的空缺。这种情况下，我们考虑到尽量减少代码的修改和少增加选择器，所以我们为MOV和MVN单独设置了一种方法 3'b101：ALUResult = Re_src。

4、对于RSB和RSC，我们增加了一个Reverse_src的信号，从decoder中判断，这个信号在输入进ALU前对SrcA和SrcB交换位置。

5、在decoder中对TEQ和TST加入nowrite判断

NZCV赋值考虑

In either case, the new condition code flags (after the instruction has been executed) usually mean:

N Is set to bit 31 of the result of the instruction. If this result is regarded as a two's complement signed integer, then N = 1 if the result is negative and N = 0 if it is positive or zero.

Z Is set to 1 if the result of the instruction is zero (which often indicates an *equal* result from a comparison), and to 0 otherwise.

C Is set in one of four ways:

- For an addition, including the comparison instruction `CMN`, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
- For a subtraction, including the comparison instruction `CMP`, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
- For non-addition/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.
- For other non-addition/subtractions, C is normally left unchanged (but see the individual instruction descriptions for any special cases).

V Is set in one of two ways:

- For an addition or subtraction, V is set to 1 if signed overflow occurred, regarding the operands and result as two's complement signed integers.
- For non-addition/subtractions, V is normally left unchanged (but see the individual instruction descriptions for any special cases).

在NZCV赋值的时候，我们并没有按照严格的手册来进行。对于C的赋值，除了加和减及相关指令外，当用到shift数值时，C为移出去的最后一个值，这个我们沿袭前面几次Lab的要求（并询问过学长这个要求），认为没有必要实现，就没有变化。但对其它部分，我们均完成了NZCV的考虑。

基于这些操作后，我们成功做出了DP指令的16种指令的拓展。

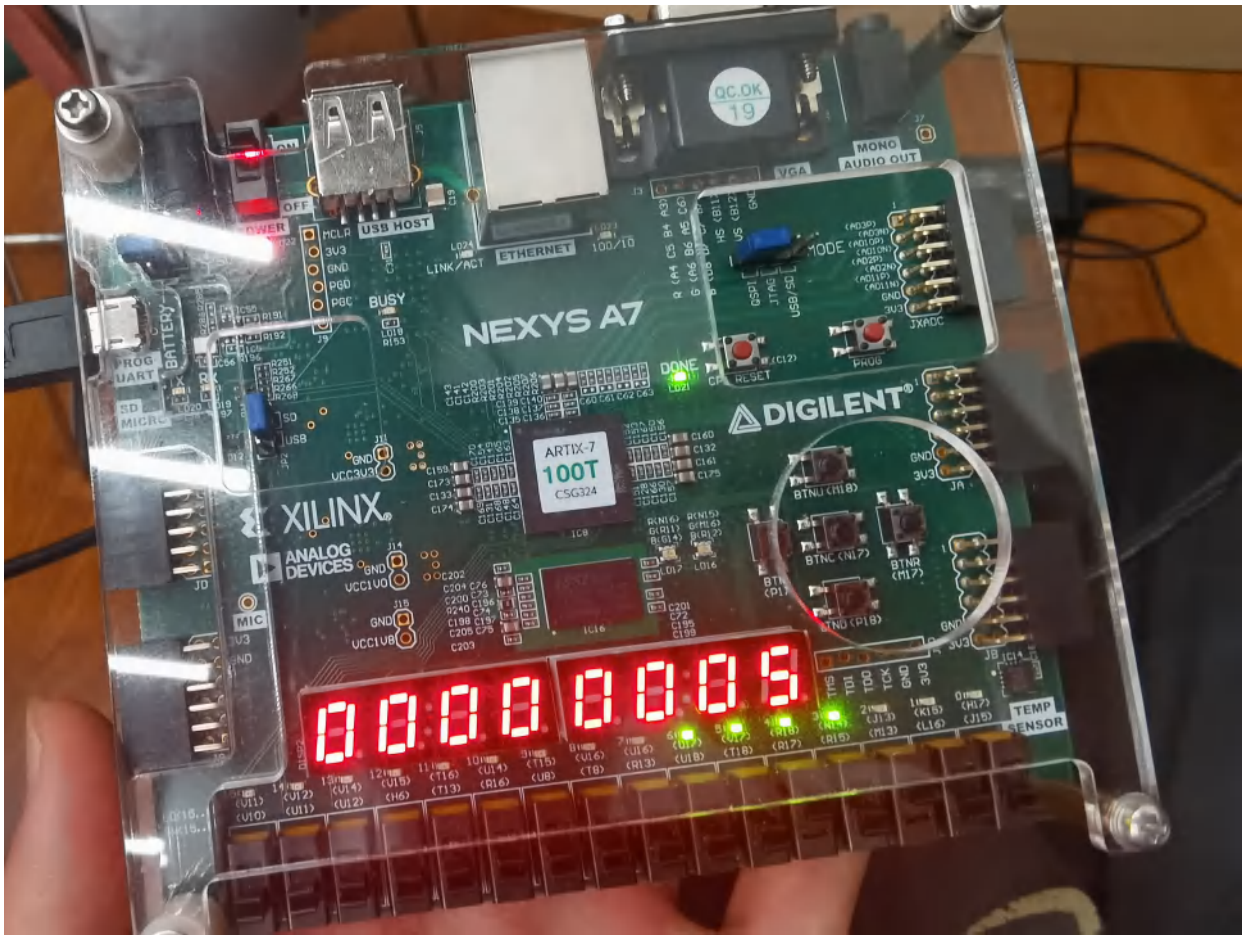
下面是tb和对应的汇编语言：

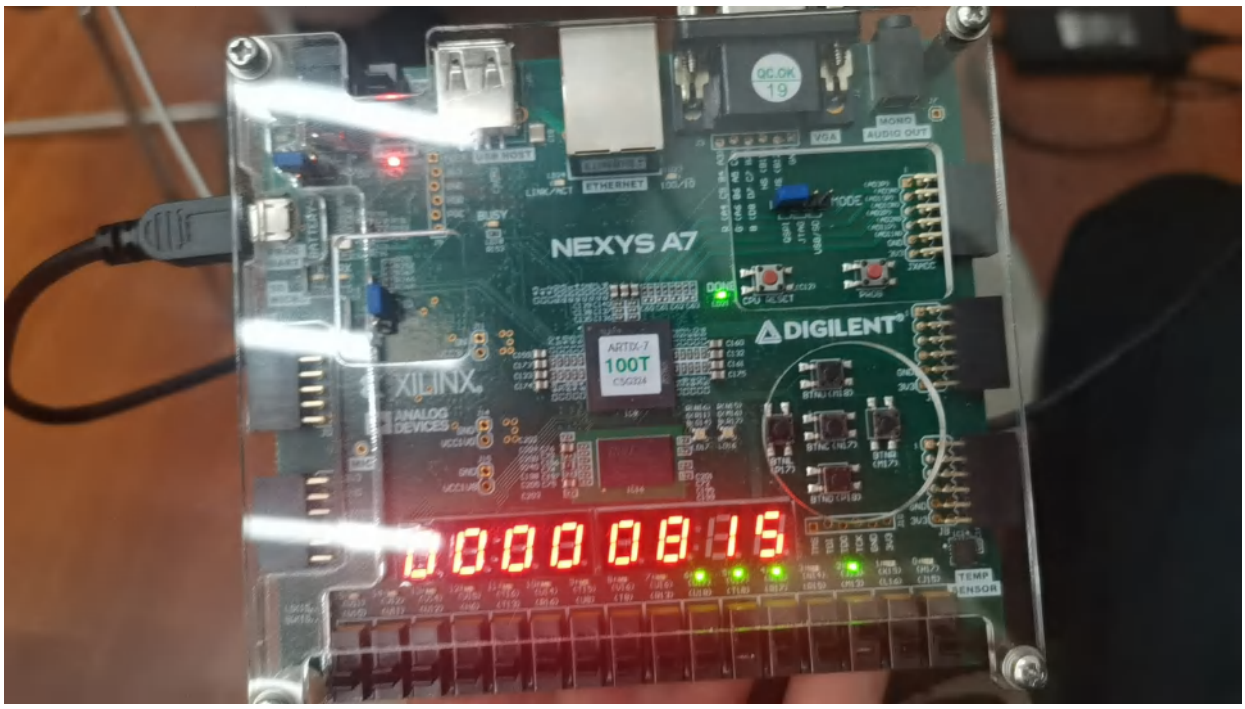
tb

汇编语言

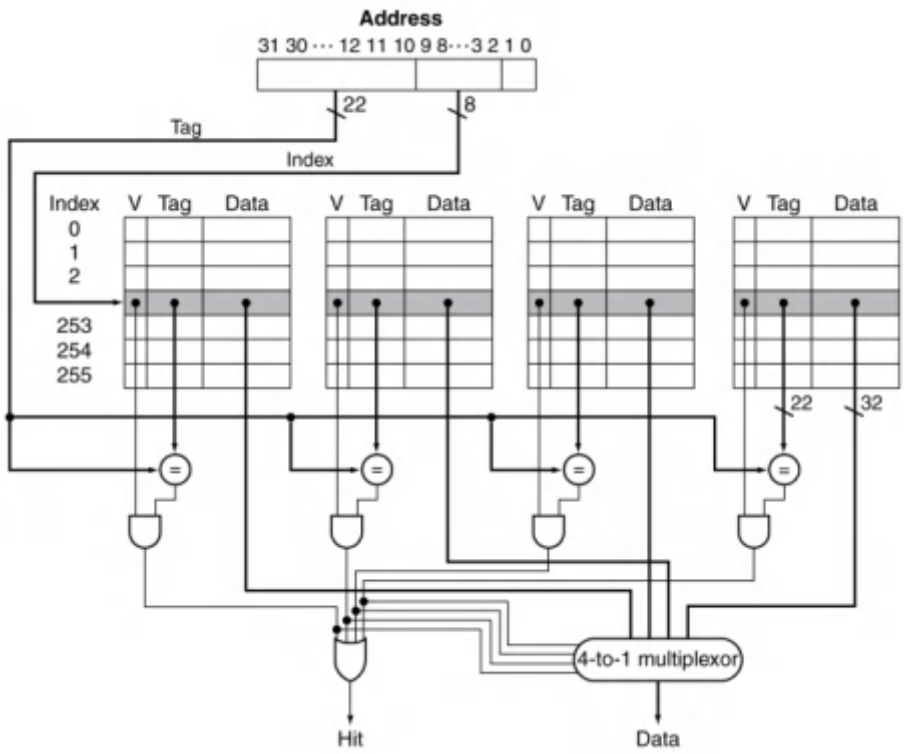
```
    AREA    MYCODE, CODE, READONLY, ALIGN=9
    ENTRY

; ------- <code memory (ROM mapped to Instruction Memory) begins>

    LDR R1, constant1; R1=5
    LDR R2, constant2; R2=6
    LDR R3, addr1; 810
    LDR R4, addr2; 820
    LDR R12,addr3; 830
    ADD R5, R1, R2; R5 = a1 + a2;

    AND R5, R1, #0;
    EOR R6, R1, R3;
    SUB R7, R4, R3;
    RSB R8, R7, R1;
    ADD R5, R7, R8;
    SUB R6, R1, #6;
    ADD R6, R6, R6;
    ADC R6, R1, R2;
    SUB R6, R1, #6;
    SBC R7, R2, R1;
    SUB R9, R1, R2;
    RSC R7, R2, R1;
    TST R1, #0;
    ADD R10, R1, R2;
    TEQ R10, R10;
    CMP R5, R6;
    RSB R8, R5, #0;
    CMN R5, R8;
    ORR R9, R3, R1;
    MOV R12, R4;
    BIC R13, R4, R3;
    MVN R10, R2;

halt
```

上板测试如下：





Q4 A 4-way set associative cache between memory and ARM CPU.

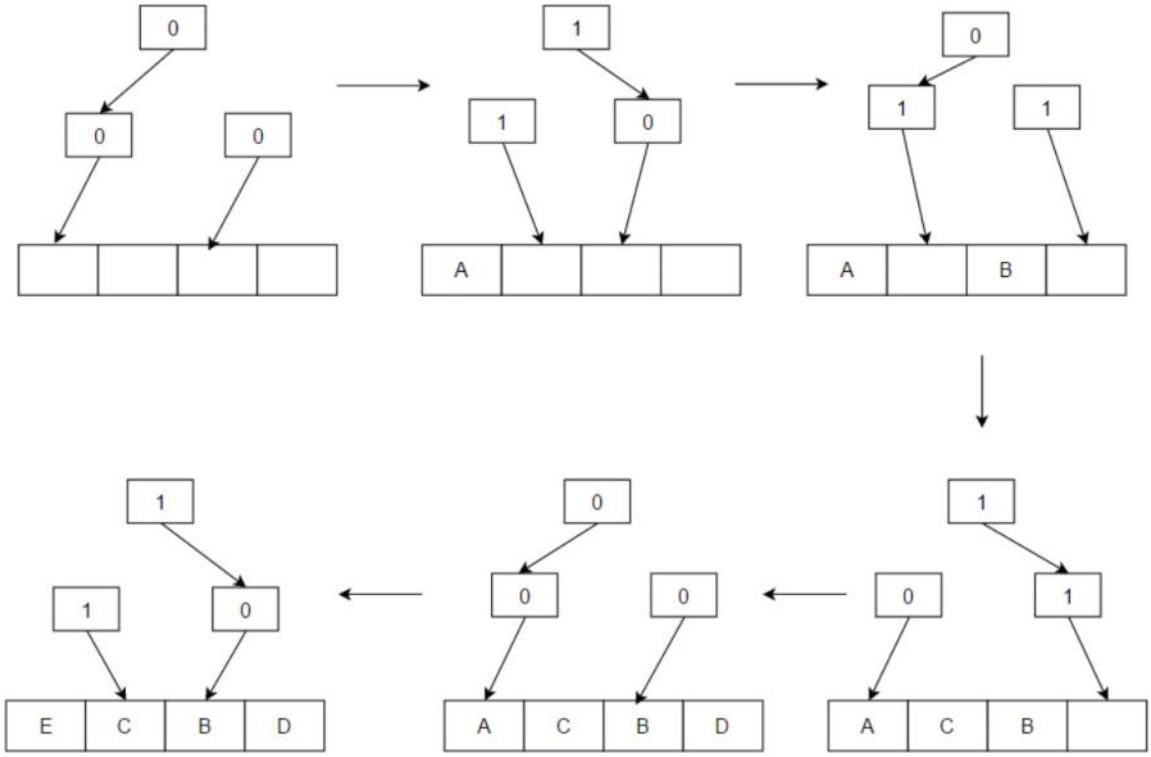


缓存结构设计

根据上图所示的结果，我们构建出了基本的cache结构。我们分别用四个1024大小的reg矩阵来构建，大小分别是1、1、22、32，代表的内容分别是valid，dirty，tag和data。之所以用1024个矩阵主要原因是我们可以用index的前两位来区分set0-4，降低了我们写if-else的个数（虽然if-else和0-4表示综合出来资源应该相差不大），增加了代码的可读性。将56位的数据分开也是相同的原因，valid, dirty, tag, data，四个数据操作起来相对独立，所以可以用不同的信号来表示，这样不论是写代码的时候操作和判断比较方便，debug的时候也比较清晰。

LRU算法替换

在替换策略部分，我们选择了Pseudo-LRU的算法，也叫伪LRU。因为实际的LRU在实现上比较困难，且会消耗相当大的资源（涉及排序）。所以为了降低资源的使用，并且达到类似LRU的算法，我们引入了伪LRU算法，伪LRU实现如下图所示。



总而言之这个方法就是用一个树状结构来选择该去的位置，每次选择某一位的时候，上面的值和下面指向的值就要进行翻转。下一次指向的就是翻转后的值。如上图所示，达到一种实现LRU的结果。但是，这个方法也只是近似的LRU，在部分情况下，该方法所替换的不是最久未使用的单元。综上伪LRU方法的替换效果应该略差于LRU，好于随机替换，并且它的资源远小于LRU。尤其是在我们实现的情况下，有256组记录数据，用LRU绝对是一种非常奢侈的资源消耗。

缓存操作逻辑

除了替换策略外，正常情况下，cache会优先选择hit和empty的位置，当cache满的时候才会选择Pseudo-LRU。同时我们模拟了一个memory 5个周期penalty，当memory给到memory_ready信号的时候，才让cache赋值。当处于!Hit状态时，将整个处理器暂停。

之后考虑的就是 write-allocate 和 write-back，write allocate就是write miss时，修改memory的值，并且赋给Cache；我们这边属于特殊情况，一个block当中只有一个data。所以这边的write allocate就是修改memory值和Cache即可，不用从Memory中调取data。

write-back是，当set中的值被覆盖时（write or read），如果block中的值是dirty的，就将block中的dirty值给赋到memory中。

所以对于这种情况，我们分为两步进行分析：

一步是Cache内部赋值，需要考虑三种情况：

1、Hit并且是Write的情况，直接写入数据。

2、!Hit并且是Write的情况，修改Tag, data和Dirty，valid。

3、!Hit并且是Read的情况，修改Tag, data和Dirty，valid。

后两种情况都需要等到Memory的ready信号才能进行，保证数据传输成功。

实现代码如下

```
always @(posedge clk) begin
        if(!Hit & !MemWrite & memory_ready) begin
            Data_Block_valid[{BLK_NUM, Addr_index}] <= 1'b1;
            Data_Block_Tag[{BLK_NUM, Addr_index}] <= Addr_tag;
            Data_Block_Data[{BLK_NUM, Addr_index}] <= ReadData;
            Data_Block_Dirty[{BLK_NUM, Addr_index}] <= 1'd0;
        end
        else if(Hit & MemWrite)begin
            Data_Block_valid[{BLK_NUM, Addr_index}] <= 1'b1;
            Data_Block_Data[{BLK_NUM, Addr_index}] <= WriteData;
            Data_Block_Dirty[{BLK_NUM, Addr_index}] <= 1'd1;
        end
        else if(!Hit & MemWrite & memory_ready)    begin
            Data_Block_valid[{BLK_NUM, Addr_index}] <= 1'b1;
            Data_Block_Tag[{BLK_NUM, Addr_index}] <= Addr_tag;
            Data_Block_Data[{BLK_NUM, Addr_index}] <= WriteData;
            Data_Block_Dirty[{BLK_NUM, Addr_index}] <= 1'd1;
        end
    end
```

第二步是对于Cache的输出情况:

1、Write-back的实现：当没有hit的情况，并通过我们的选择set模块得到的set非空并dirty，则将这个数据传输到Memory当中。

2、对于处理器需要的数据，直接将Data输出。

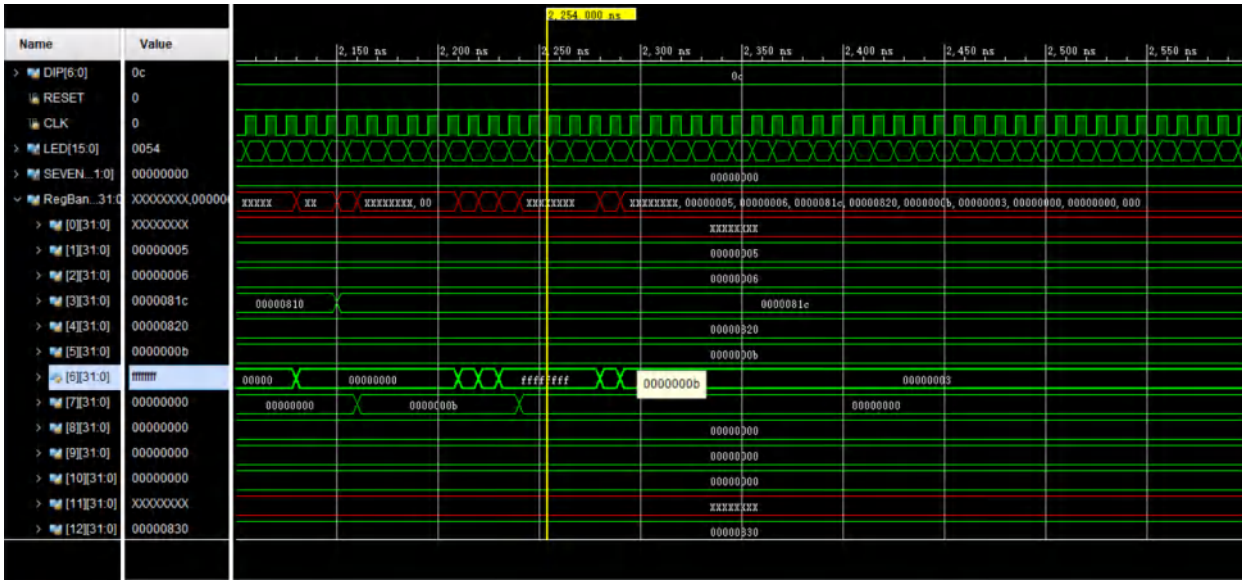
实现代码如下

```
assign Data = Tag0_equal ? Data_Block_Data[{2'd0, Addr_index}] :
                  Tag1_equal ? Data_Block_Data[{2'd1, Addr_index}] :
                  Tag2_equal ? Data_Block_Data[{2'd2, Addr_index}] :
                  Tag3_equal ? Data_Block_Data[{2'd3, Addr_index}] : 32'dz;

assign MemWrite2Memory = (!Hit & Data_Block_valid[{BLK_NUM, Addr_index}] &
Data_Block_Dirty[{BLK_NUM, Addr_index}]);
assign Data2Memory = Data_Block_Data[{BLK_NUM, Addr_index}];
assign MissAddr = {Data_Block_Tag[{BLK_NUM, Addr_index}], Addr_index, 2'b00};
```

经过上述步骤，我们就可以完成对Cache的构建，一下是对Cache的tb和汇编文件

在仿真的过程中，我们发现了一个问题，在miss的情况下，我们会对整个处理器做暂停处理，但是MCycle模块独立于整个处理器，所以它仍然在运行。所以如果它完成计算的一周期是在miss的周期当中，MCycle输出的值无法进入M stage中。在这里我们做了一个特殊的处理。对于MCycle做一个类似buffer的输出，当Cache miss的情况下，如果输出数据，则将数据存储，当Miss消失的时候再输出。如此一来，我们既保留了Non-stalling的特性，也使他正常输出。

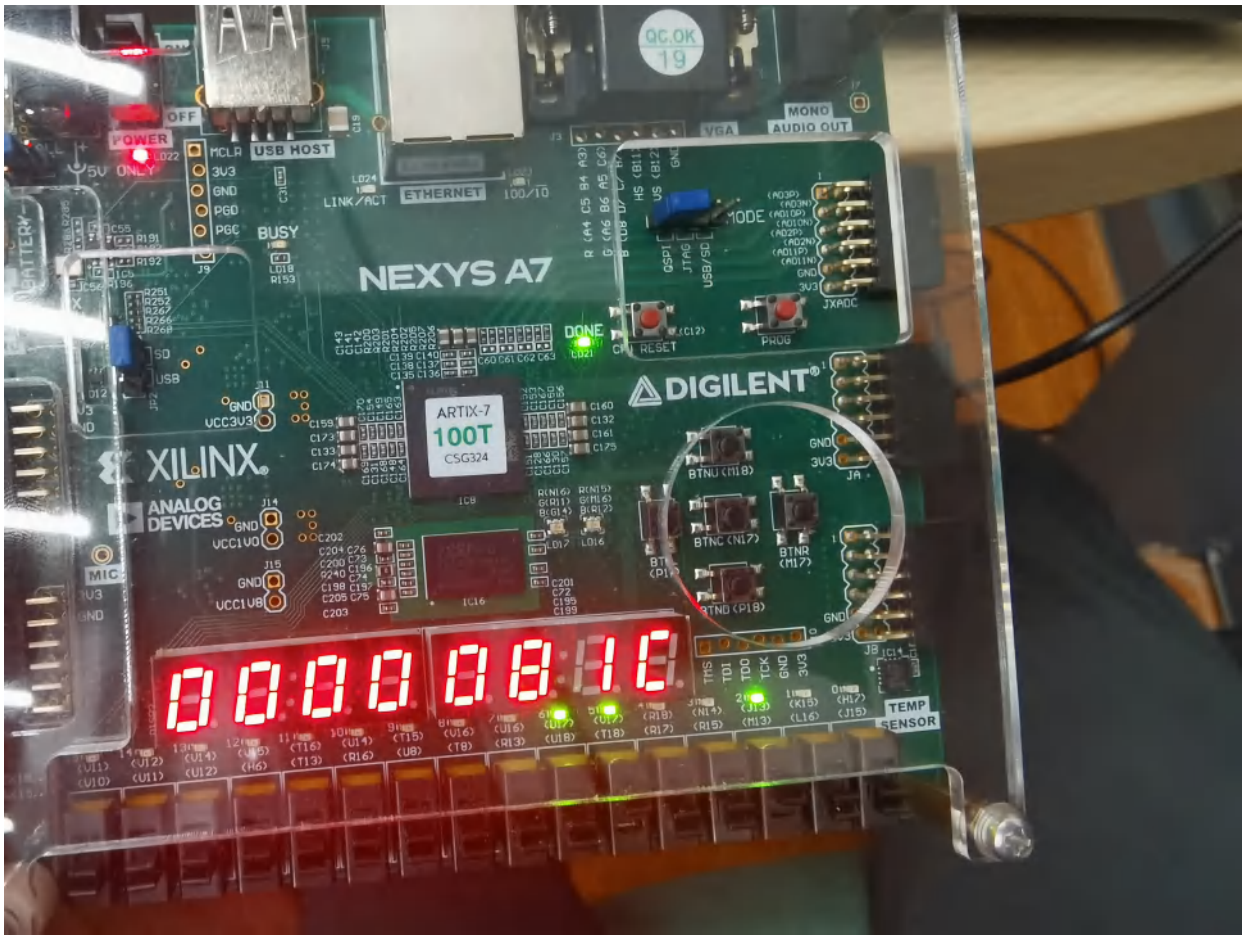
tb

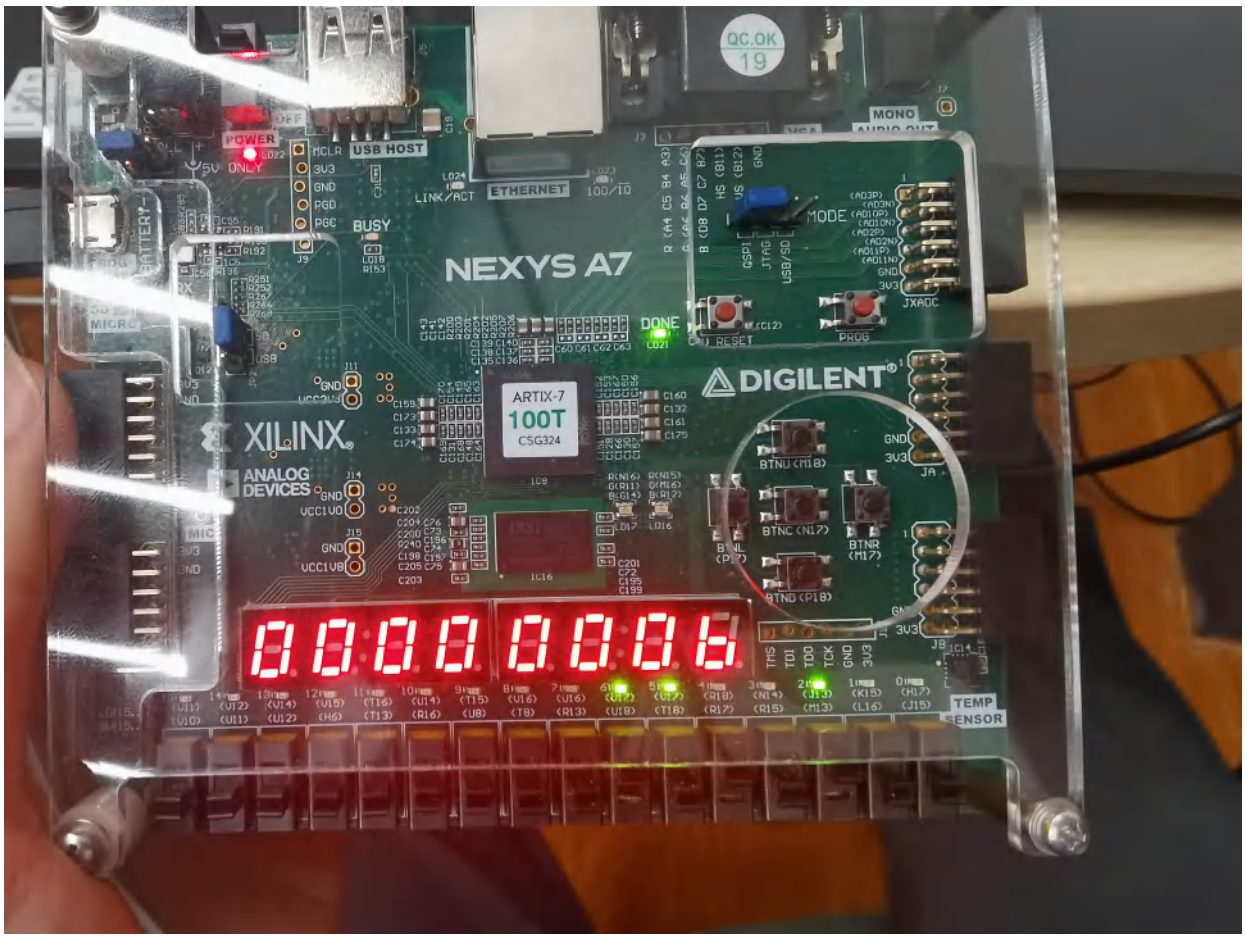


汇编文件

```
    AREA    MYCODE, CODE, READONLY, ALIGN=9
      ENTRY

; ------- <code memory (ROM mapped to Instruction Memory) begins>

    LDR R1, constant1; R1=5
    LDR R2, constant2; R2=6
    LDR R3, addr1; 810
    LDR R4, addr2; 820
    LDR R12,addr3; 830
    ADD R5, R1, R2; R5 = a1 + a2;
    LDR R6, number0;

    STR R5, [R3,#-4];
    ADD R3, R3, #12;
    LDR R7, [R3, #-16];

    STR R1, [R6];
    STR R2, [R6, #1024];
    STR R3, [R6, #2048];
    STR R4, [R6, #3072];
    ADD R6, R6, #3072;
    ADD R6, R6, #1024;
    NOP;
    NOP;
    ;STR R5, [R6];
    ;STR R12, [R6, #1024];

    SUB R6, R6, #3072;
    SUB R6, R6, #1024;
    LDR R7, [R6];
    LDR R8, [R6, #1024];
    LDR R9, [R6, #2048];
    LDR R10, [R6, #3072];
    ADD R6, R6, #3072;
    ADD R6, R6, #1024;
```

上板测试如图：





Q5 Floating-Point Unit

FADD

加法处理过程中存在以下两种情况

1） 输入数据异常，内容包括输入正负无穷，NaN,非规格数。

处理方式：

非规格数：当输入数据为非规格数时，FPU模块参照Intel的SSE2指令集直接将非规格化浮点数四舍五入至0进行正常处理。

NaN：当一输入数据为NaN时，输出结果也为NaN。

无穷：当一输入数据为正无穷时，若另一数据不为NaN和负无穷时，输出结果为正无穷，负无穷类似处理。

信号判定逻辑见下，两个输入处理相同，故此处仅列一个。

```
 localparam EXP_WIDTH = 8;

 localparam FRAC_WIDTH = 24;

 localparam EXP_MAX = 2 ** EXP_WIDTH - 1;

wire [EXP_WIDTH-1:0] exp;

wire [FRAC_WIDTH-2:0] frac;

wire inf_sig,nan_sig;

wire zero_sig; // we take denorm number as zero

assign {sign,exp,frac} = Operand1; //数据分割

assign inf_sig = (exp==EXP_MAX)&&(frac1=={(FRAC_WIDTH-1){1'b0}}); //

assign nan_sig = (exp==EXP_MAX)&&(frac1!={(FRAC_WIDTH-1){1'b0}});

assign zero_sig = (exp == 0);
```

inf_sig以及nan_sig判定方式与上课介绍相同，故不赘述，由于我们将非规格数同样当作0处理，因此0信号判定只需指数位等于0。

输出控制见下：

```
always @(posedge CLK) begin

  if(nan_sig1_add ||nan_sig2_add || ( inf_sig1_add && inf_sig2_add && sign1_add !=sign2_add )  )

   fl_out_add <= {1'b0, {EXP_WIDTH{1'b1}}, 1'b1, {(FRAC_WIDTH-2){1'b0}}}; //  frac != 0 NaN

  else if(  (inf_sig1_add && !sign1_add ) || (inf_sig2_add && !sign2_add ) )

   fl_out_add <= {1'b0, {EXP_WIDTH{1'b1}}, {(FRAC_WIDTH-1){1'b0}}}; // pos inf

  else if(  (inf_sig1_add && sign1_add ) || (inf_sig2_add && sign2_add ) )

   fl_out_add <= {1'b1, {EXP_WIDTH{1'b1}}, {(FRAC_WIDTH-1){1'b0}}}; // neg inf
```

```
  else if(zero_judge_add)
   fl_out_add <= 32'b0; // zero
  else
   fl_out_add <= {add_num_out[FRAC_WIDTH + 1],exp_o_add,shift_data[FRAC_WIDTH : 2]};
end
```

由于NaN与inf区别为小数部分不为全0，在FPU中我们将小数部分最高位设置为1，其余为0以区分NaN和inf。

前三个条件与上述介绍相同，第五个条件为正常输出情况，第四个条件将在后面介绍。

2. 正常输入

当两个正常数据输入时，我们将两个数的指数位进行比较，将指数位小的数据进行右移以对齐定点数，随后将数据从原码转为补码。代码处理如下（同样，只列举一个）：

```
wire [EXP_WIDTH -1:0] exp_b = exp1> exp2 ? exp1 : exp2;
wire [EXP_WIDTH - 1:0] shift_num1 = exp_b - exp1; // must >= 0
wire [FRAC_WIDTH - 1 : 0] shift_data1= {1'b1,frac1} >>shift_num1;
reg signed [FRAC_WIDTH + 1 : 0] fix_num1 ;
wire [FRAC_WIDTH : 0] shift_data1_comp = ~shift_data1 + 1;
always @(posedge CLK) begin
 if(zero_sig1)
  fix_num1 <= 0;
 else if(sign1)
  fix_num1 <= { sign1,shift_data1_comp};
 else
  fix_num1 <= { sign1,1'b0,shift_data1};
end
```

由于两个数据相加，可能产生进位信号，为防止数据溢出，我们将定点位宽增加一位。输出数据为补码形式，所以我们进行上述类似的补码转原码操作。

两个定点数相加后，输出结果的指数位可能发生改变，取决于原码非符号位中最高的1的位置，为找到1的位置，处理代码如下：

```
wire [31:0] a = add_num_out[FRAC_WIDTH : 0]; // needs zero padding
wire [4:0] one_location;
wire [1:0] find_one1;
```

```
wire [3:0] find_one2;

wire [7:0] find_one3;

wire [15:0] find_one4;

assign one_location[4] = (|a[31:16]);

assign find_one4 = one_location[4] ? a[31:16] : a[15:0];

assign one_location[3] = (|find_one4[15:8]);

assign find_one3 = one_location[3] ? find_one4[15:8] : find_one4[7:0];

assign one_location[2] = (|find_one3[7:4]);

assign find_one2 = one_location[2] ? find_one3[7:4] : find_one3[3:0];

assign one_location[1] = (|find_one2[3:2]);

assign find_one1 = one_location[1] ? find_one2[3:2] : find_one2[1:0];

assign one_location[0] = find_one1[1];
```

第一行补全无意义，在高位添0。处理过程为二分，将数据中高位的一半进行一个按位或操作，若为1则说明高位中存在1，否则不存在。后接一个MUX来选择接下来在高位或者低位进行寻找，判定依据即为按位或信号，以0x00b10为例，高16位为0，按位或结果为0，数据选择0b10,高8位为b，按位或结果为1，数据选择0b,高4位为0，按位或结果为0，数据选择b,高2位为01，按位或结果为1，选择01，最后one_location[0]=0。One_location结果为=5'b01010(10)，与00b10中最高位1在10一致。然而当数据为24'h00001和24'h00000时，one_location均为0，前者实际数值不为0，后者实际数值为0，因此我们需要添加新的判定信号，条件如下：

```
wire zero_judge_add = (one_location == 1'b0) && (find_one1[0] == 1'b0) ? 1'b1 : 1'b0;
```

``

当location = 0且最低位也为0是，zero_judge_add 置为1，且在上述输出MUX中进行选择。

若onelocation正常，需要将定点数数据移位节选（将最高位的1移除，移位位宽与one_location相关），设为输出数据的小数部分，同样指数部分取决与输入时较大的指数以及one_location。代码如下：

```
wire [EXP_WIDTH - 1:0] exp_o_add = exp_b_delay + one_location - 23; // when exp b == 127 expo = 128 there has a accum overflow

wire [EXP_WIDTH - 1:0] shift_num = 25 - one_location;

wire [FRAC_WIDTH : 0] shift_data = add_num_out << shift_num; // shift one to highest width
```

FMUL

乘法处理过程中存在以下两种情况

1） 输入数据异常，内容包括输入正负无穷，NaN,非规格数。NaN,非规格数处理方式与加法相同，故不赘述。当一输入数据为无穷时，若另一数据不为NaN，输出结果为无穷，符号由两输入数据异或结果决定。

```
always @(posedge CLK) begin
 if(nan_sig1_mul ||nan_sig2_mul  )
  fl_out_mul <= {1'b0, {EXP_WIDTH{1'b1}}, 1'b1, {(FRAC_WIDTH-2){1'b0}}}; //  NaN
 else if(  inf_sig1_mul || inf_sig2_mul || inf_judge)
  fl_out_mul <= {sign_mul_delay, {EXP_WIDTH{1'b1}}, {(FRAC_WIDTH-1){1'b0}}}; // inf
 else
  fl_out_mul <= {sign_mul_delay, exp_o_mul  , mantissa_mul };
end
```

inf_judge信号在在后面解释。

2） 正常输入

输入时将小数部分前补，成为完整的定点数，同时判定输入数据是否为非规格数，若为非规格数则将定点数设为0。后将两定点数输入乘法模块（非先前的定点乘法），在25周期后得到48位积。由于两数均为1.x大小数据，输出结果介于1.0-4.0间，指数位存在两种情况，取决于最高位是否为1，因此在决定输出指数以及小数部分时，将通过一个mux，mux的控制信号为积的最高位。代码如下：

```
wire [EXP_WIDTH : 0] exp_mul = exp1_delay + exp2_delay - 127; // supposed to limited in range 0 - 255, a delay is needed  if highest bit is 1 means it's bigger than 255 or smaller than 0 we take it as nan
wire [EXP_WIDTH : 0] exp_mul_p = exp_mul + 1;
wire inf_judge = add_one_sig ? exp_mul_p[EXP_WIDTH] : exp_mul[EXP_WIDTH];
wire [FRAC_WIDTH -2 :0] mantissa_mul = add_one_sig? mul_num_out[FRAC_WIDTH * 2 -2 -: (FRAC_WIDTH -1) ] : mul_num_out[FRAC_WIDTH * 2 -3 -: (FRAC_WIDTH -1)  ];
wire [EXP_WIDTH -1 :0] exp_o_mul = add_one_sig? exp_mul_p[EXP_WIDTH -1 :0] : exp_mul[EXP_WIDTH -1 :0];
```

由于两数的积过大可能单精度浮点数无法表示，因此将输出指数位宽扩大一位，作为控制输出无穷的信号inf_judge，当最高位为1时，即超过正常表示范围，我们将输出结果设为inf。

由于FPU是多周期指令，数据存在延时，此处不特介绍数据延时逻辑。加法总延时为3，乘法总延时为26。

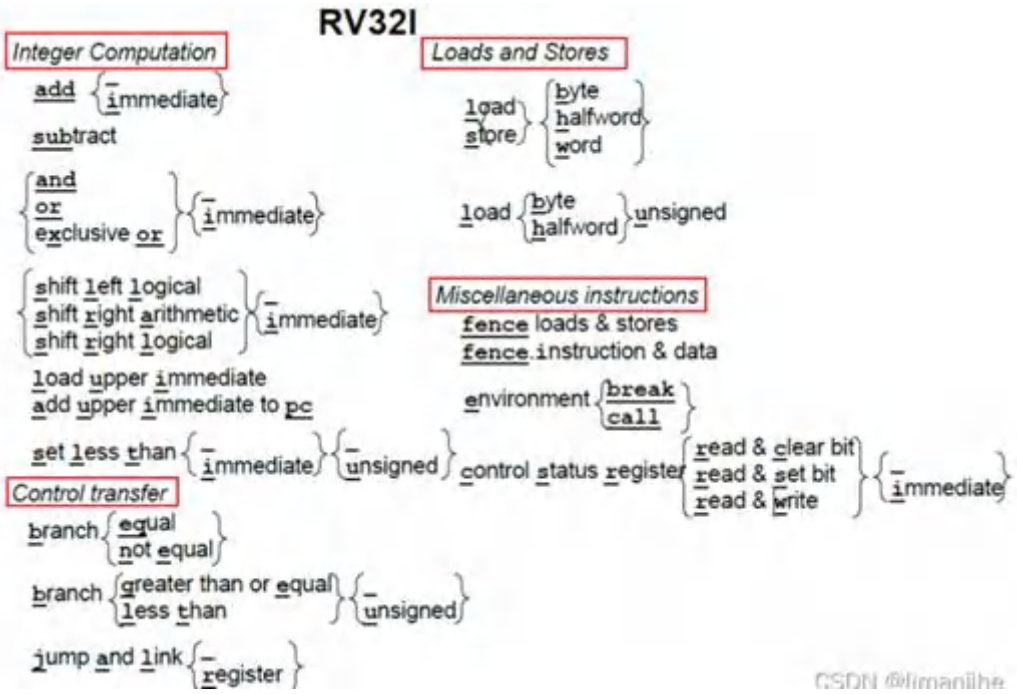
算法以外

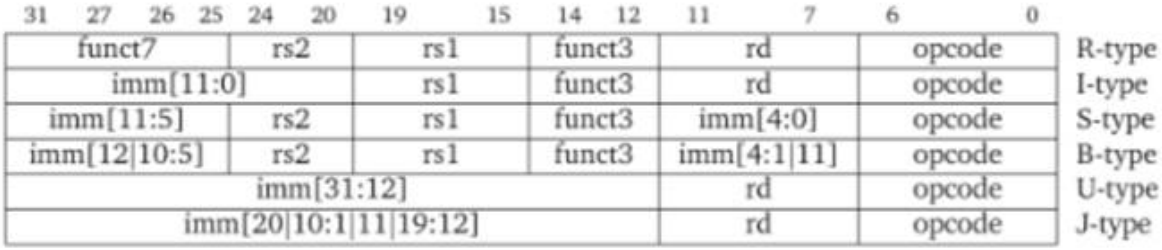
由于浮点运算与定点乘法除法均为多周期运算指令，为控制信号方便，我们将两模块合并，整合为一个大的多周期运算模块，且keil编译器无法编译FADD指令，我们自设为与整数除法相近的指令，整数除法SBZ = 0000，浮点加SBZ = 0001，浮点乘 SBZ = 0010。

结果验证

以下为测试的汇编指令以及期望结果以及波形图(测试指令由于keil无法编译故不上传，wrapper中指令常量数据及波形图截图将上传)，以上特殊结果处理以及正常计算均有涉及，经对比，计算结果无误。

```
LDR R1, constant1; R1 = +inf

LDR R2, constant2; R2 = 1.111 2^70

LDR R3, constant3; R3 = - 1.111 2^70

LDR R4, constant4; R4 = - 1.1111111...111

LDR R5, constant5; R5= - 1.111000...001 2^70

FADD R6, R2, R3; = 0 0000_0000

FADD R7, R2, R2; = 1.111 2^71 6370_0000

FADD R8, R2, R5; = -1 2^48

FMUL R9, R1, R3; -inf ff80_0000

FMUL R10, R2, R2; inf 7f80_0000

FADD R11, R2, R9; -inf ff80_0000

FADD R12, R10, R9; NaN 7fc0_0000

FMUL R13 ,R3 ,R4;636f_ffff

FMUL R14, R12, R2; NAN 7fc0_0000

constant1

    DCD 0x7f80_0000;

constant2

    DCD 0x62f0_0000;

constant3

    DCD 0xe2f0_0000;

constant4

  DCD 0xbfff_ffff;

constant5

    DCD 0xe2f0_0001;
```

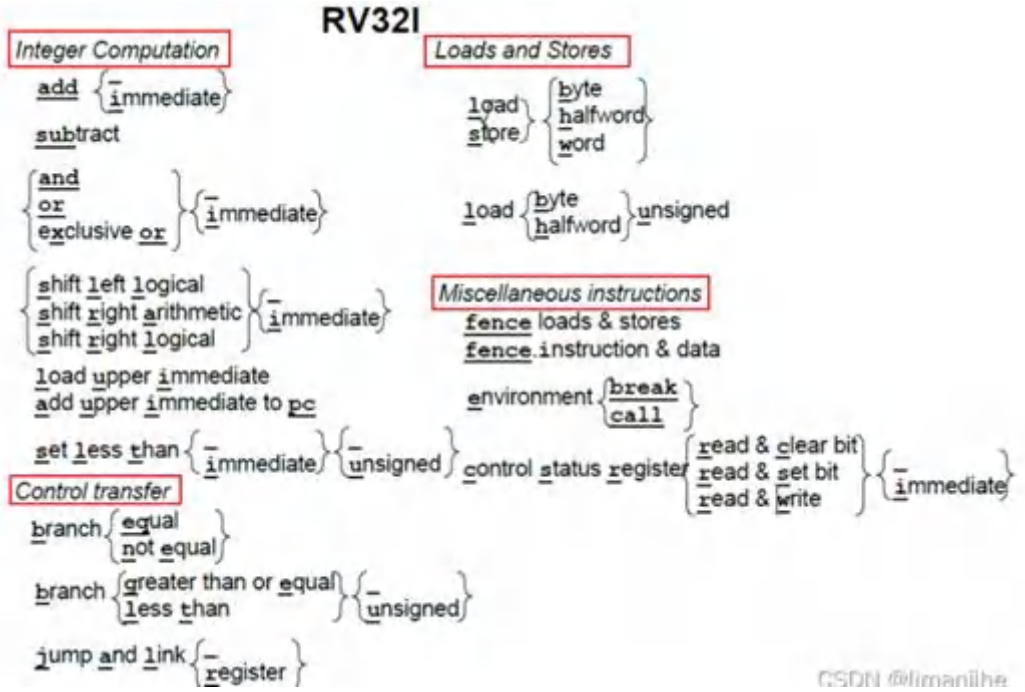




以上为寄存器R9 R10数据，与期望结果波形图均相同。

Q6 RiscV32I

在RISCV32I中，我们要完成如下这些指令，下面我们首先介绍一下RISCV32I的基本样貌



RV32I 基本指令格式如下图图所示，RV32I 的基本指令格式只有 4 种，分别是寄存器类型（R-TYPE）、短立即数类型（I-TYPE）、内存存储类型（S-TYPE）、高位立即数类型（U-TYPE）。

为了方便跳转指令，RV32I 还包含两种衍生格式 B-TYPE（Branch，条件跳转）与 J-TYPE（Jump，无条件跳转）。B-TYPE 衍生于 S-TYPE，B-TYPE 除了立即数的位排列与 S-TYPE 不一样外，其他的格式都与 S-TYPE 一样。J-TYPE 也是通过类似的方式衍生于 U-TYPE。用这种方式衍生新格式的目的是便于硬件产生目标地址。

上面这些格式，除 R-TYPE 外，其他的格式都需要把最高位（第 31 位）做符号扩展，以产生一个 32 位的立即数，作为指令的操作数。

上图所示的这些指令格式非常规整，其操作码、源寄存器和目标寄存器总是位于相同的位置上，简化了指令解码器的设计。总的六种指令为：

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

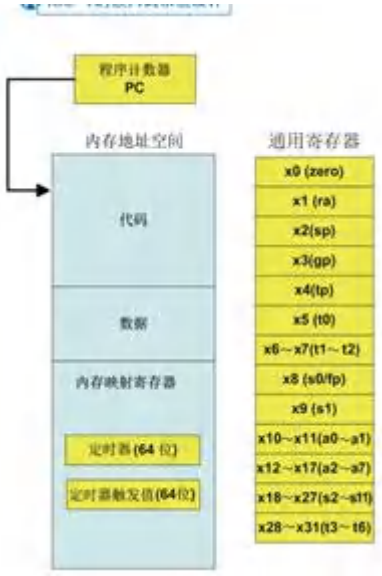
其中RISCV用Opcode来区分各类的方法，用funct3加之以funct7来特定区分不同类站之间的区别

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

在RISCV32I当中有几个与ARM不一样的地方：

1、寄存器有32个，其中R0表示0



2、在计算方法中，将shift加入了计算中，没有移位加的操作了。

3、Extend module 需要比以前更复杂，因为立即数不在同一个地方，甚至顺序都是乱的。

4、在立即数的拓展中，有最高位扩展（有符号），也有直接添加0扩展（无符号好）。

5、在立即数操作中，有截取立即数进行操作的。

6、Store和load指令更多，且需要考虑load和store大小。

7、branch指令中要比较从RF中得到的值，并让PC和Imm相加。

8、Jal指令中，既要写入PC+4，也要执行PC+Imm。

9、Cond判断被取消

等.....

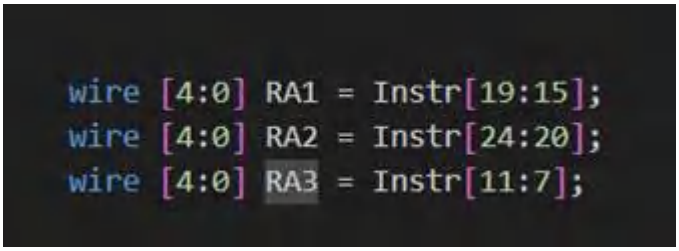
针对以上各类情况：

我们需要做如下的改变：

1、首先将RF改成32位，并将X0改成0的连线，如下图所示

```
    );

    // declare RegBank
    reg [31:0] RegBank[1:31];

    integer i;
    initial begin
        for(i = 1; i < 32; i = i + 1)  begin
            RegBank[i] = 32'd0;
        end
    end
    assign RD1 = (A1 == 0) ? 0: RegBank[A1];
    assign RD2 = (A2 == 0) ? 0: RegBank[A2];

    always@(posedge CLK)    begin
        if(WE3) begin
            RegBank[A3] <= WD3;
        end
    end

endmodule
```

2、在核的外部重新排线，R1, R2, R3不再需要经过选择器，直接接入到固定的Instr位置。



```
wire [4:0] RA1 = Instr[19:15];
wire [4:0] RA2 = Instr[24:20];
wire [4:0] RA3 = Instr[11:7];
```

3、因为PC不再存在R15，所以直接将PC连到ALU的外侧，和RD1一起进行选择，选择信号ALUSrc_A由Decoder产生。

4、扩展ALU的指令，将ALUcontrol扩展到12种方法，分别对应R-type的十种方法和U-type的两种方法，删除Shift模块。

5、对于Condition被取消，我们也不再设立Cond Logic，对于Branch中的几种判断，专门新写一个state_compare模块进行是否运行的判断，并将Condex接入Decoder里面，近似之前的Cond Logic的用法。

```
module State_compare
(
    input [31:0] Instr,

    input [31:0] RS1,
    input [31:0] RS2,

    output Condex
);

    wire [6:0] opcode = Instr[6:0];
    wire funct3 = Instr[14:12];

    wire whether_B = (opcode != 7'b1100011) ? 1'b1: 1'b0;

    reg cond;
    always @(*) begin
        if(funct3 == 3'd0) begin
            cond = (RS1==RS2);
        end
        if(funct3 == 3'd1) begin
            cond = (RS1!=RS2);
        end
        if(funct3 == 3'd4) begin
            cond = (RS1 < RS2);
        end
        if(funct3 == 3'd5) begin
            cond = (RS1 >= RS2);
        end
        if(funct3 == 3'd6) begin
            cond = (RS1 < RS2);
        end
        if(funct3 == 3'd7) begin
            cond = (RS1 >= RS2);
        end
    end
    assign Condex = whether_B | cond;
```

6、对于立即数的扩展，我们在Extend种进行判断。并对是最高位还是0填充，我们增加了ImmSrc一位位宽，用以判断用最高位还是0填充。代码如下：

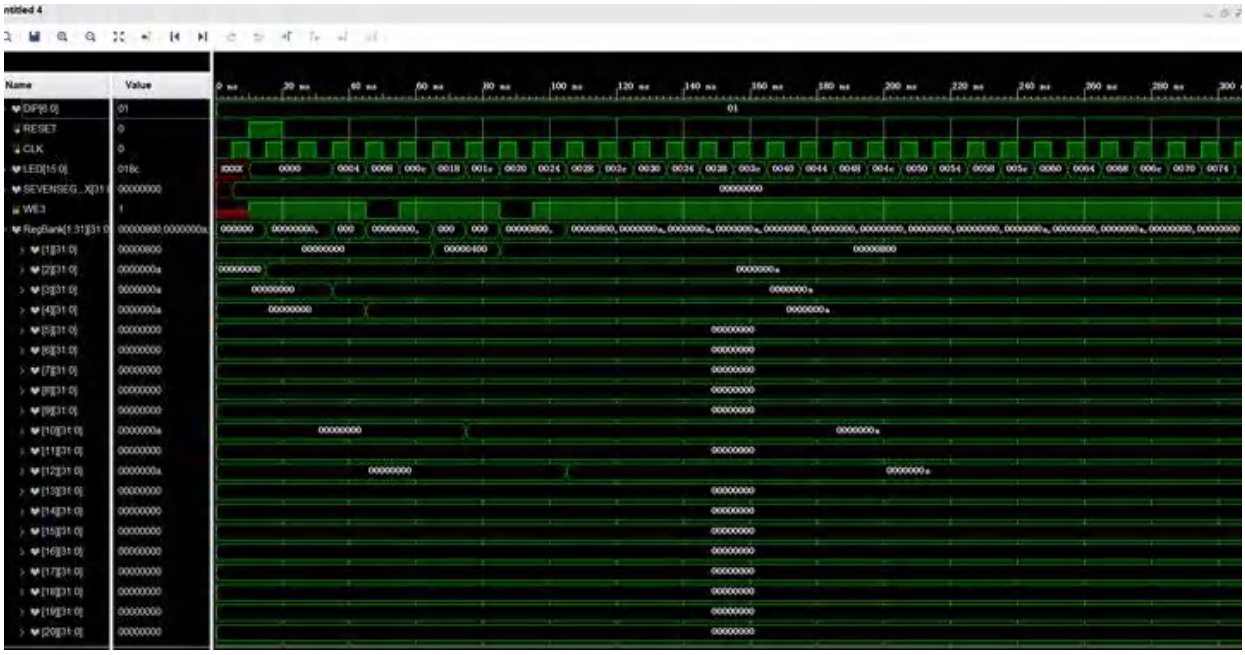
```
    // ImmSrc 00: DP    01: LDR/STR 10: B
    always @(*) begin
        case(ImmSrc[2:0])
            `I_type:    ExtImm = {{(20){InstrImm[31]&(!ImmSrc[3])}},InstrImm[31:20]};
            `S_type:    ExtImm = {{(20){InstrImm[31]}},InstrImm[31:25],InstrImm[11:7]};
            `B_type:    ExtImm = {{(19){InstrImm[31]&(!ImmSrc[3])}}, InstrImm[31], InstrImm[7], InstrImm[30:25], InstrImm[11:8], 1'b0};
            `U_type:    ExtImm = {InstrImm[31:12], 12'b0};
            `J_type:    ExtImm = {{(11){InstrImm[31]}},InstrImm[31],InstrImm[19:12], InstrImm[20],InstrImm[30:21],1'b0};
            default:    ExtImm = 32'B0;
        endcase
    end
```

7、对于Jal指令，我们将RF中的PC+4引出，用以信号控制（PC4）。PC4由decoder产生，当PC4拉高的时候，通过一个选择器进入RF中。

8、对于立即数操作，有截取立即数进行操作的，我们新创一个信号Imm，当它拉高的时候，截取Src_B的前五位进行操作。

9、我们重新设计了Decoder的结构，增加了许多新的信号，除了上文已经提到的，我们还增加了PCsrc_direct和PCsrc做或操作判断是否处理器是否需要跳转，因为RISCV32I中有强制跳转指令。所以我们新创立了该信号。

10、还有许多其他在报告中未提及的小修改处，最终我们完成了RISCV32I的设计。下面是仿真出的波形图和汇编代码



```
addi x2, x1, 10
xor x3, x2, x1
add x4, x5, x3
beq x2, x3, 12
addi x2, x1, 10
addi x2, x1, 10
sub x3, x4, x2
xori x10, x10, 10
xori x1, x1, 820
sw x10, 8(x1)
lw x12, 8(x1)
```

汇编代码包括了寄存器运算、立即数运算、跳转、存取四大类的运算，算出的结果均正确，并通过上版测试，当拨键为四号键的时候，数码管是0000000A。

Bonus1 Superscalar

根据PC值，处理器向ins_mem顺序读取两条指令，判定两条指令是否可以同时执行，若可以执行，则PC自增8，若两条指令无法同时执行，我们只执行第一条，第二个处理器不进行指令操作，（实际上处理器执行AND R1 ,R1指令），PC自增4。

以下为本组多核处理器考虑到的可能出现的冒险现象导致无法同时执行两条指令的情况。

1. Data hazard

a) RAW，当第二条指令读取的寄存器值被第一条指令改写，两者无法同时进行，DP指令可以通过forwarding解决，双核处理器的forwarding与单核略有不同，后面详细介绍，LDR指令需要stall（此处不过多介绍，前面已涉及），第二条指令不执行。

b) WAW，当两条指令同时写入一个寄存器，此时第二条指令此周期不执行。

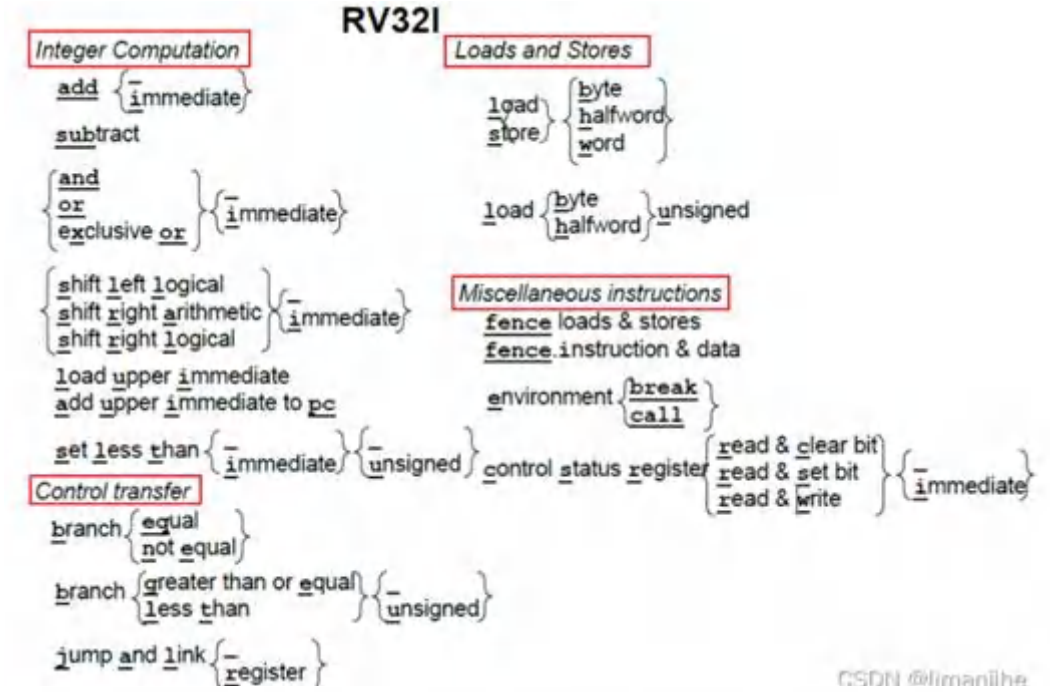
2. Control hazard

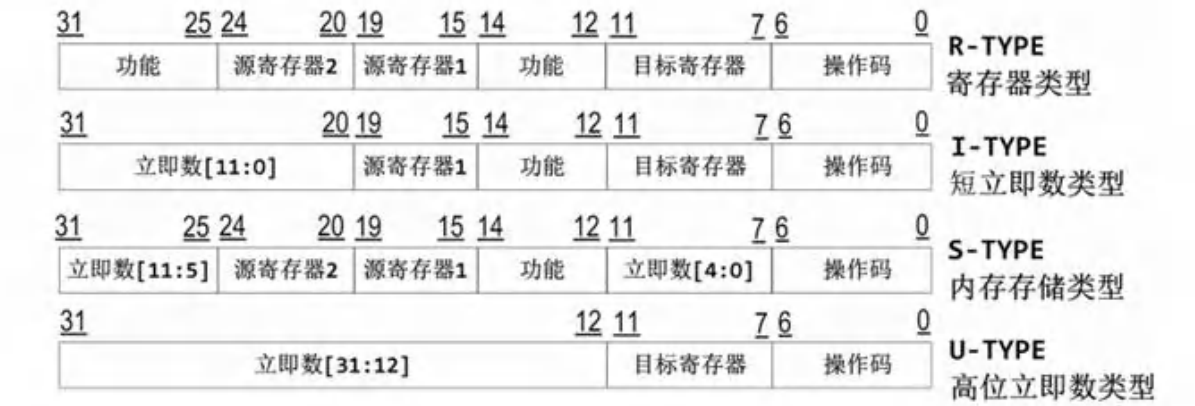
当存在branch指令在第一条时，第二条指令未必执行，导致寄存值可能出错，因此当第一条指令为branch指令时，第二条不执行。

```
always @(*) begin
  if( op1 == 2'b10)
   sup_sig = 1'b0;
  else if(op1 == 2'b00 && S1 == 1'b1 && cond2[3:1] !=3'b111 ) //
   sup_sig = 1'b0;
  else if(op2 == 2'b00 && ( Rd1 == Rm2 || (Rd1 == Rs2 && r2or1)  ))
   sup_sig = 1'b0;
  else if(Rd1 == Rd2)
   sup_sig = 1'b0;
  else
   sup_sig = 1'b1;
 end
 assign Instr2_out = sup_sig ? Instr2 : 32'hE001_1001;
```

Data_forwarding

双核处理器data_forwarding需要考虑的forwarding信号为两个，因此所有存在data forwarding的数据MUX的输入信号变多，控制信号位宽变大，由于改动地方过多，此处仅列部分。





RegisterFile共用，由于需要对于一个RF进行读写，RF输入输出端口数量变为原来的两倍，由于在先前控制了WAW情况，并不需要考虑写入冲突现象，代码修改较简单，故不特意展示。

以下为测试的汇编指令以及期望结果以及波形图。

```
LDR R1, constant1; R1 =9
```

``

```
 ADD R2, R1, R1;    ; R2 = R1 + R1 = 12
```

``

```
 SUB R3, R1, R2;    ; R3 = R1 - R2 = ffff_fff7(-9)
 ADD R4, R1, R2;    ; R4 = R1 + R3 = 1b
```

``

```
 SUB R5, R2, R4;    ; R5 = R2 - R4 = ffff_ffdc(-36)
 ADD R6, R3, R4;    ; R6 = R3 + R4 = 12
```

``

```
SUB R7, R6, R1;   ; R7 = R6 - R1 = 09
```

``

```
ADD R8, R4, R7;   ; R8 = R4 + R7 = 24
```

`constant1`

```
  DCD 0x0000_0009;
```

将每周期执行指令用空格区分开，1，2指令间存在数据依赖，2，3间存在，因此1，2均独立运行，3，4不存在数据依赖可以并行，5，6可并行操作且使用了双数据的data_forwarding，证明有效。7，8间存在数据依赖，因此无法同时执行。从下面波型图中可以看到，数据写入顺序为R1,R2,R3R4(同时)，R5R6(同时),R7,R8符合上面的分析，且结果同分析一致。