数逻实验流水线处理器大作业实验报告

郭东琦 2023010477 无 31

一、实验目的

（1）根据数逻理论课学习到的知识以及春季数逻实验课学习到的 verilog 知识，自行设计出一个五级流水线结构的 MIPS 处理器，并使用 forward、stall 等方法消除数据冒险与控制冒险。

（2）该处理器最终需要完成稀疏矩阵乘法任务，并将最终结果的矩阵的各个元素以 16 进制的形式逐个显示在数码管上，并且需要采用软件译码的形式。

（3）分析自己设计出的五级流水线结构的处理器的性能。

二、设计方案

我按照理论课的设计，将五级流水线分为以下五个阶段：

（1）IF 阶段：取指令

（2）ID&RF 阶段：解码指令+读取寄存器堆+判断是否需要 jump

（3）EX 阶段：进行 ALU 计算+判断是否分支跳转

（4）MEM 阶段：读写存储器

（5）WB 阶段：将数据写回到寄存器堆中

为了实现以上阶段，我首先构造了 **IF.v、ID.v、EX.v、MEM.v、WB.v 五个模块**

此外，为了解决冒险问题，我使用 forward 数据转发来解决 RAW（read after write）数据冒险问题；使用 forward+stall 来解决 load-use 数据冒险问题；对于分支指令在 EX 阶段判断，在分支发生时刻取消 ID 和 IF 阶段的两条指令；对于 J 类指令在 ID 阶段判断，并取消 IF 阶段指令。以此来解决冒险问题。为了对何时转发数据以及何时 flush/keep 级间寄存器进行控制，我构造了 **forward_unit.v 和 hazard.v 两个模块**

除了以上主要模块以外，还有：

DataMemory.v（数据存储器模块，用来存储初始化数据以及最终计算结果）

ALU.v（计算模块，根据不同的控制信号对输入进行不同运算）

control.v （在 ID 阶段使用，用来根据 32bit 的指令编码解析出不同的控制信号）

ALUcontrol.v （在 ID 阶段使用，根据指令解析出 ALU 的控制信号）

registerfile.v （寄存器堆模块，包含 32 个 32bit 寄存器单元）

instructionMemory.v （指令存储器模块，用来存储需要执行的所有指令）

MYCPU.v （整合各个模块，对各模块的输入输出进行连接）

top.v （顶层文件）

通过控制信号的传递，可以将以上模块连接起来。

指令与部分控制信号对应关系如下：

	opcode	functcode	pcsrc[1:0]	regwrite	regdst[1:0]	memread	memwrite	memtoreg[1:0]	alusrc2	alusrc1	extop	luop
lw	**23**	**-**	**00**	**1**	**00**	**1**	**0**	**01**	**0**	**1**	**1**	**0**

sw	**2b**	**-**	**00**	**0**	**x**	**0**	**1**	**x**	**0**	**1**	**1**	**0**
lui	**0f**	**-**	**00**	**1**	**00**	**0**	**0**	**00**	**0**	**1**	**x**	**1**
add	**0**	**20**	**00**	**1**	**01**	**0**	**0**	**00**	**0**	**0**	**x**	**x**
addu	**0**	**21**	**00**	**1**	**01**	**0**	**0**	**00**	**0**	**0**	**x**	**x**
sub	**0**	**22**	**00**	**1**	**01**	**0**	**0**	**00**	**0**	**0**	**x**	**x**
subu	**0**	**23**	**00**	**1**	**01**	**0**	**0**	**00**	**0**	**0**	**x**	**x**
addi	**08**	**-**	**00**	**1**	**00**	**0**	**0**	**00**	**0**	**1**	**1**	**0**
addiu	**09**	**-**	**00**	**1**	**00**	**0**	**0**	**00**	**0**	**1**	**1**	**0**
ori	**0d**	**-**	**00**	**1**	**01**	**0**	**0**	**00**	**0**	**0**	**x**	**x**
and	**0**	**24**	**00**	**1**	**01**	**0**	**0**	**00**	**0**	**0**	**x**	**x**
or	**0**	**25**	**00**	**1**	**01**	**0**	**0**	**00**	**0**	**0**	**x**	**x**
xor	**0**	**26**	**00**	**1**	**01**	**0**	**0**	**00**	**0**	**0**	**x**	**x**
nor	**0**	**27**	**00**	**1**	**01**	**0**	**0**	**00**	**0**	**0**	**x**	**x**
andi	**0c**	**-**	**00**	**1**	**00**	**0**	**0**	**00**	**0**	**1**	**1**	**0**
sll	**0**	**00**	**00**	**1**	**01**	**0**	**0**	**00**	**1**	**0**	**x**	**x**
srl	**0**	**02**	**00**	**1**	**01**	**0**	**0**	**00**	**1**	**0**	**x**	**x**
sra	**0**	**03**	**00**	**1**	**01**	**0**	**0**	**00**	**1**	**0**	**x**	**x**
slt	**0**	**2a**	**00**	**1**	**01**	**0**	**0**	**00**	**0**	**0**	**x**	**x**
sltu	**0**	**2b**	**00**	**1**	**01**	**0**	**0**	**00**	**0**	**0**	**x**	**x**
slti	**0a**	**-**	**00**	**1**	**00**	**0**	**0**	**00**	**0**	**1**	**1**	**0**
sltiu	**0b**	**-**	**00**	**1**	**00**	**0**	**0**	**00**	**0**	**1**	**1**	**0**
beq	**04**	**-**	**00**	**0**	**x**	**0**	**0**	**x**	**0**	**0**	**1**	**0**
bne	**05**	**-**	**00**	**0**	**x**	**0**	**0**	**x**	**0**	**0**	**1**	**0**
blez	**06**	**-**	**00**	**0**	**X**	**0**	**0**	**x**	**0**	**0**	**1**	**0**
bgtz	**07**	**-**	**00**	**0**	**X**	**0**	**0**	**X**	**0**	**0**	**1**	**0**
bltz	**01**	**-**	**00**	**0**	**x**	**0**	**0**	**X**	**0**	**0**	**1**	**0**
j	**02**	**-**	**11**	**0**	**x**	**0**	**0**	**X**	**x**	**x**	**x**	**X**
jal	**03**	**-**	**11**	**1**	**10**	**0**	**0**	**10**	**x**	**x**	**x**	**X**
jr	**0**	**08**	**10**	**0**	**x**	**0**	**0**	**x**	**x**	**x**	**x**	**X**
jalr	**0**	**09**	**10**	**1**	**10**	**0**	**0**	**10**	**x**	**x**	**x**	**0**

以下我将主要介绍一些在单周期处理器中没有，在五级流水线处理器中才加入的模块。

以下框图通过 VScode 中的 DIDE 插件实现。

（一）IF 模块

这个模块负责从指令存储器中取出指令，并根据输入的各种控制信号的值，判断下一条指令从哪个地址取出（jump、branch、PC+4）

图 1　IF 模块框图

名称	方向	位宽	描述
clk	input	——	时钟
reset	input	——	重置
dojump	input	——	要不要跳转
pc_keep	input	——	flush时候 keep PC
dobranch	input	——	zero&&isbranch,应该在EX阶段产生并送入
jumpaddress	input	[31:0]	如果jump，jump的地址
branchaddress	input	[31:0]	如果分支跳转，跳转的地址
next_condition	input	[1:0]	whether to flush
if_id_pc4in	input	[31:0]	此时IF/ID寄存器存的值，如果flush，则可能把这两个再输出
if_id_instrin	input	[31:0]	此时IF/ID寄存器存的值，如果flush，则可能把这两个再输出
if_id_pc4out	output	[31:0]	所取的指令的地址+4的结果
if_id_instrout	output	[31:0]	所取的指令

图 2　IF 模块各信号的含义

（二）ID 模块

这个模块负责解码指令，根据传入的 32bit 指令，解析出各类控制信号的值。同时还会判断是否为 J 型指令，并将是否跳转以及跳转地址传给 IF 模块。

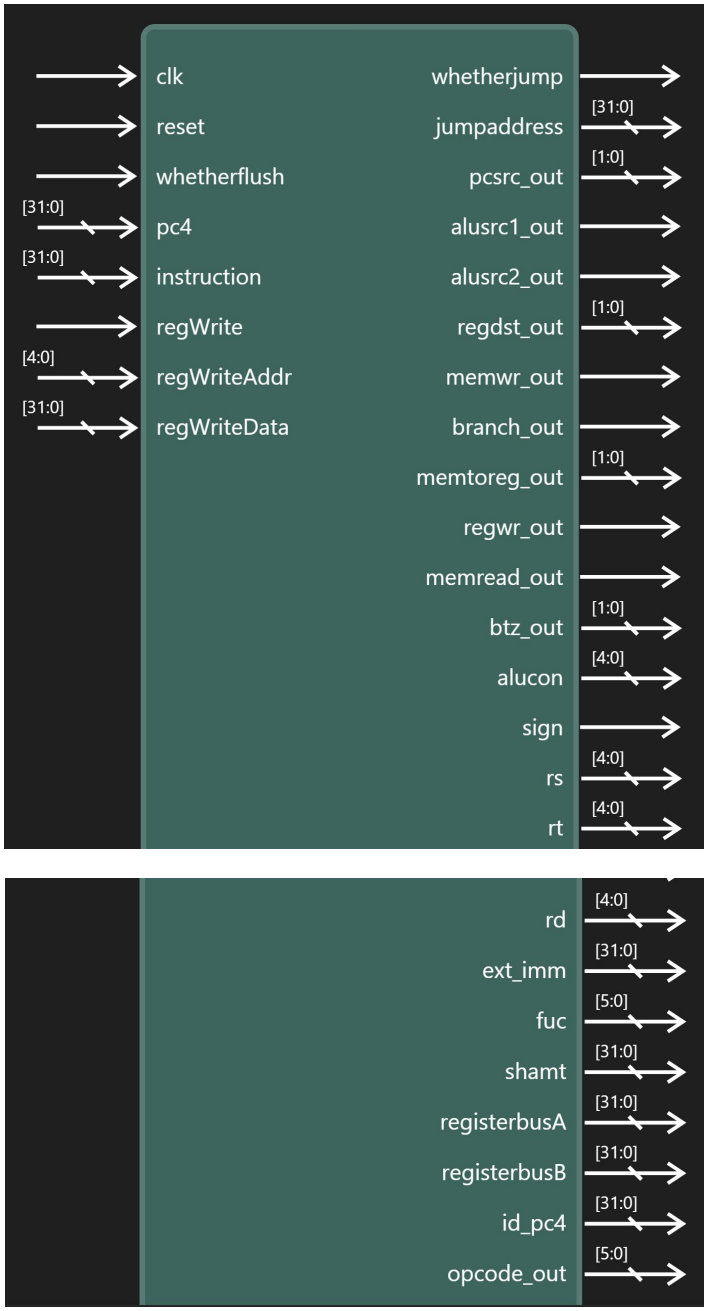


图 3 ID 模块系统框图

名称	方向	位宽	描述
clk	input	——	
reset	input	——	
whetherflush	input	——	
pc4	input	[31:0]	IF/ID 级间寄存器数据 IF 传来的指令地址
instruction	input	[31:0]	IF 传来的指令

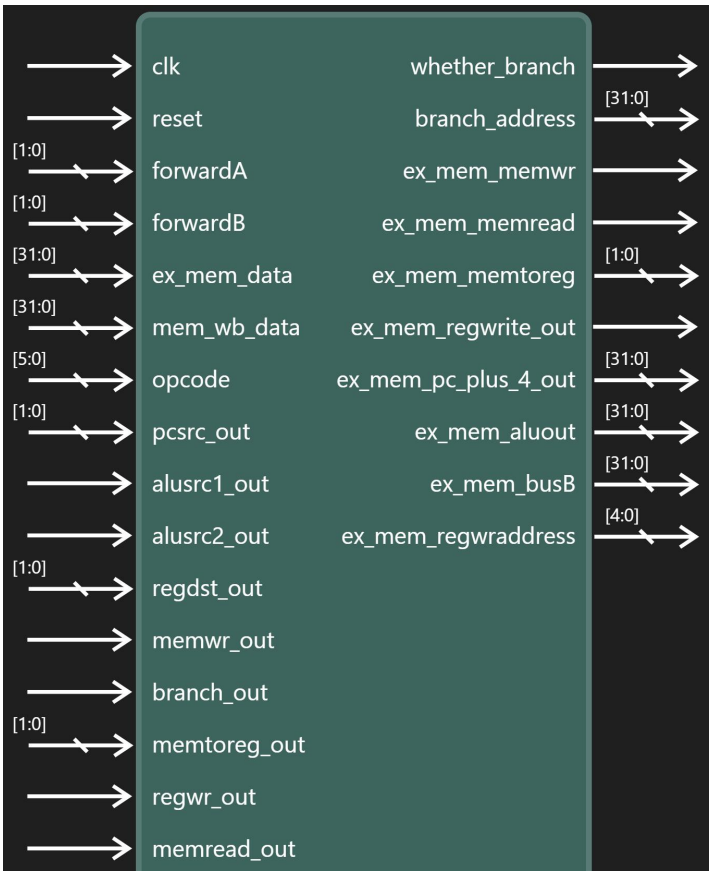
regWrite	input	——	可能有数据需要写回到寄存器堆中 WB 带来的控制信号
regWriteAddr	input	[4:0]	WB 带来的写入地址
regWriteData	input	[31:0]	WB 带来的写入数据
whetherjump	output	——	是否 jump 是否 jump
jumpaddress	output	[31:0]	如果 jump，jump 的地址
pcsrc_out	output	[1:0]	控制信号 可能来自 PC+4 00 PC+4 再偏移 01 $rs（jr） 10, 11 j/jal
alusrc1_out	output	——	output reg luop_out, // for lui 判断使用来自 busB 的数据 0 or 立即数 1
alusrc2_out	output	——	针对 srl,sll,sra,是否使用 shamet
regdst_out	output	[1:0]	可能要写到 $rt 00 $rd 01 $ra 10
memwr_out	output	——	是否要写入存储器
branch_out	output	——	是否分支跳转
memtoreg_out	output	[1:0]	可能要把 存储器中的值 01、ALU 计算的值 00、PC+4 的值（jal）写到寄存器堆中 10
regwr_out	output	——	是否写入寄存器堆
memread_out	output	——	used in hazard unit
btz_out	output	[1:0]	00 for bltz ;01 for bgtz; 10 for blez
alucon	output	[4:0]	output reg[3:0]aluop_out, ALU 控制信号
sign	output	——	ALU 控制信号，是否为有符号形式
rs	output	[4:0]	指令中包含的数据信息 rs 是什么
rt	output	[4:0]	rt 是什么
rd	output	[4:0]	rd 是什么
ext_imm	output	[31:0]	拓展后的常数
fuc	output	[5:0]	function code

shamt	output	[31:0]	移位
registerbusA	output	[31:0]	从 rs 那条通路读出的数据
registerbusB	output	[31:0]	从 rt 那条通路读出的数据
id_pc4	output	[31:0]	指令地址，继续往后传
opcode_out	output	[5:0]	opcode

表 1 ID 模块各信号的含义

（三）EX 模块

该模块需要使用 ALU 模块进行各类计算，并将计算结果传给后续模块。同时还需要判断是否进行分支跳转，并将分支跳转信号和分支跳转地址传给 IF 模块。



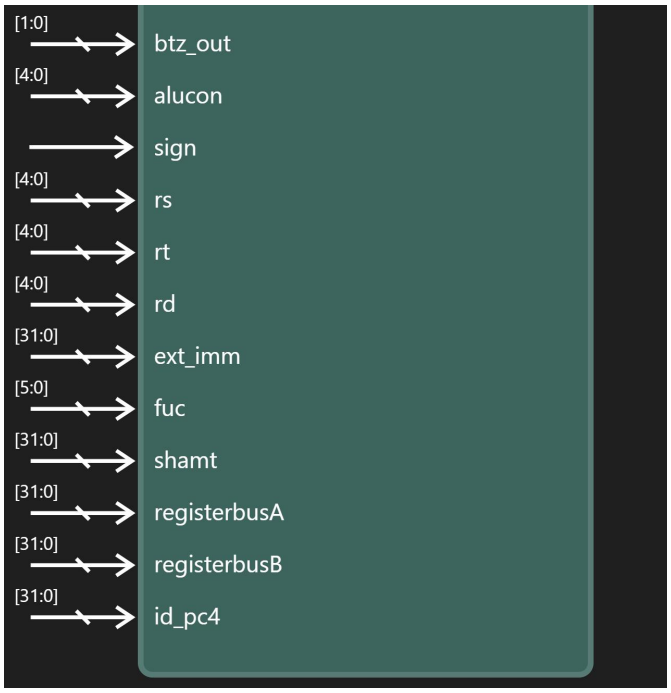


图 4 EX 模块的系统框图

名称	方向	位宽	描述
clk	input	——	
reset	input	——	
forwardA	input	[1:0]	数据旁路 forward_unit 给出的信号，判断使用哪个数据进行计算
forwardB	input	[1:0]	forward_unit 给出的信号，判断使用哪个数据进行计算
ex_mem_data	input	[31:0]	EX_MEM 寄存器传回的信号
mem_wb_data	input	[31:0]	MEM_WB 寄存器传回的信号
opcode	input	[5:0]	opcode
pcsrc_out	input	[1:0]	ID 写到 ID/EX 寄存器里的数据 以下为之前解析出的指令中的信息
alusrc1_out	input	——	
alusrc2_out	input	——	
regdst_out	input	[1:0]	
memwr_out	input	——	
branch_out	input	——	
memtoreg_out	input	[1:0]	
regwr_out	input	——	

memread_out	input	——	
btz_out	input	[1:0]	
alucon	input	[4:0]	
sign	input	——	
rs	input	[4:0]	
rt	input	[4:0]	
rd	input	[4:0]	
ext_imm	input	[31:0]	
fuc	input	[5:0]	
shamt	input	[31:0]	
registerbusA	input	[31:0]	
registerbusB	input	[31:0]	
id_pc4	input	[31:0]	
whether_branch	output	——	for branch 是否分支跳转
branch_address	output	[31:0]	如果分支跳转，跳转到的地址
ex_mem_memwr	output	——	是否写存储器
ex_mem_memread	output	——	是否读存储器
ex_mem_memtoreg	output	[1:0]	写入寄存器堆中的数据来自哪里
ex_mem_regwrite_out	output	——	是否写寄存器堆
ex_mem_pc_plus_4_out	output	[31:0]	指令地址，继续后传
ex_mem_aluout	output	[31:0]	ALU 计算结果
ex_mem_busB	output	[31:0]	rt 通路的数据，后续可能要写到存储器里，=
ex_mem_regwraddress	output	[4:0]	写回寄存器堆中的哪个寄存器

表 2 EX 模块各信号的含义

（四）MEM 模块

该模块中需要利用传入的各类控制信号实例化一个 DataMemory 模块，决定是否读取或写入数据存储器。此外，该模块需要输出数码管的控制信号 leds，实现数码管显示数字的效果。

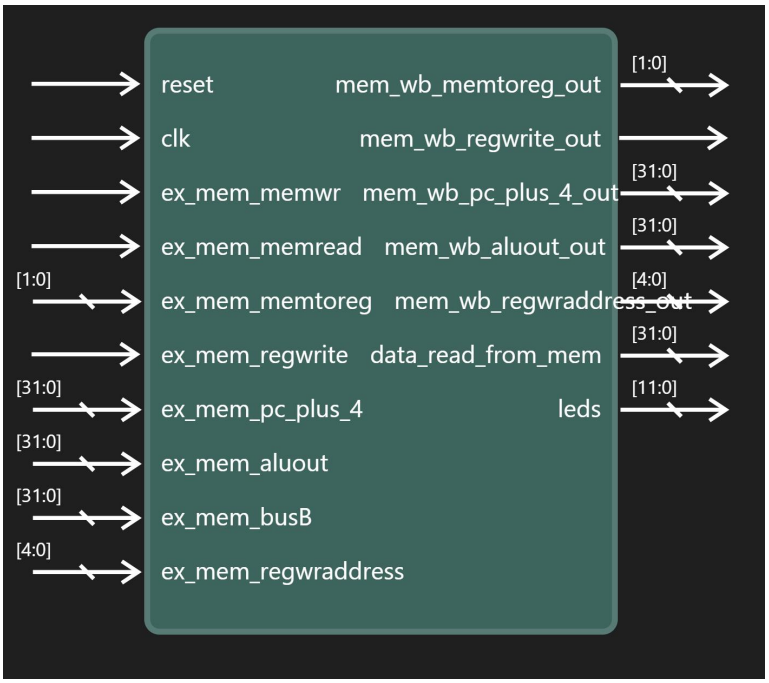


图 5 MEM 模块的系统框图

（五）WB 模块

该模块根据 memtoreg 控制信号的值，决定将哪个数据写回到寄存器堆中。该模块的功能较为简单。

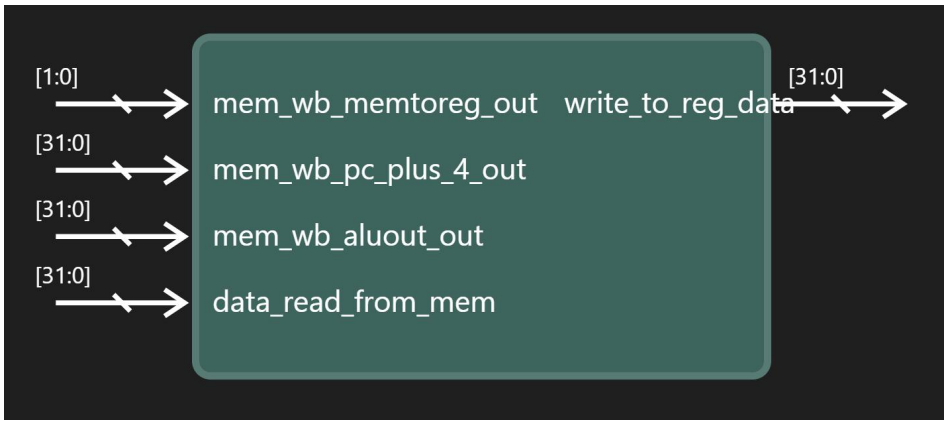


图 6 WB 模块的系统框图

（六）forward_unit 模块

该模块根据先前指令是否会写入寄存器堆、写入的寄存器堆与后续指令读取的寄存器堆是否一致等信息，决定后续指令 EX 阶段计算时使用的输入数据的来源，可能来源于寄存器堆、EX/MEM 寄存器、MEM/WB 寄存器。

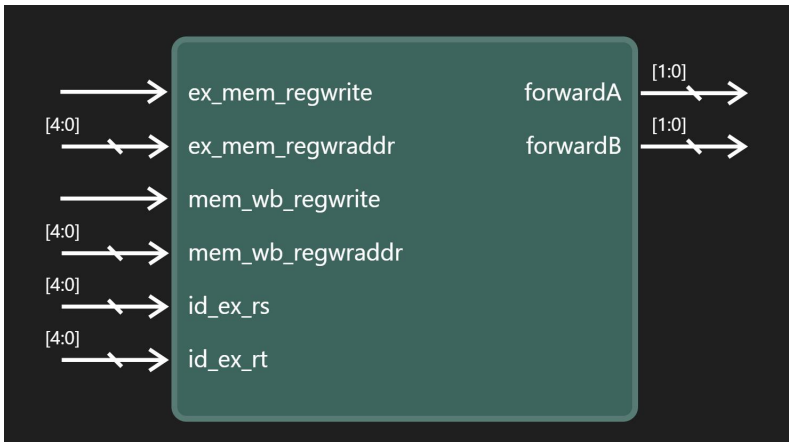


图 7 forward_unit 模块系统框图

（七）hazard 模块

该模块根据是否需要分支跳转、是否需要 jump、是否 load-use，来控制 PC、IF/ID 寄存器、ID/EX 寄存器的更新。

如果分支跳转，则 flush IF/ID and ID/EX，不 keep PC

如果有 load-use 冒险，则 keep PC，keep IF/ID，flush ID/EX

如果有 jump，则 flush IF/ID，不 flush ID/EX，不 keep PC

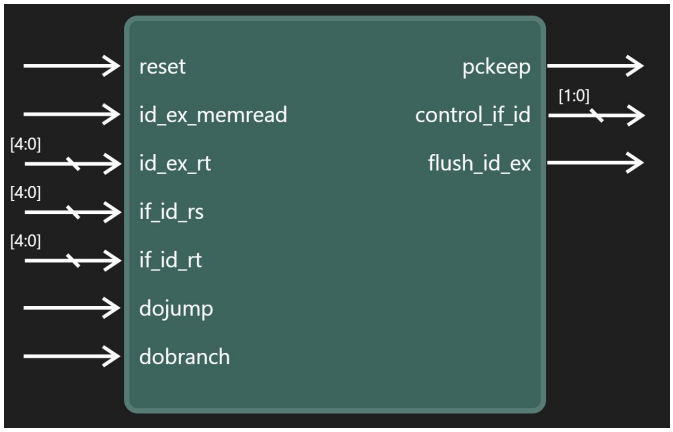


图 8 hazard 模块的系统框图

（八）control 模块

该模块是五级流水线处理器中十分重要的一个模块，负责根据输入的指令解码出各个控制信号的值，控制信号将不断向后续模块传递，控制后续模块执行的操作。

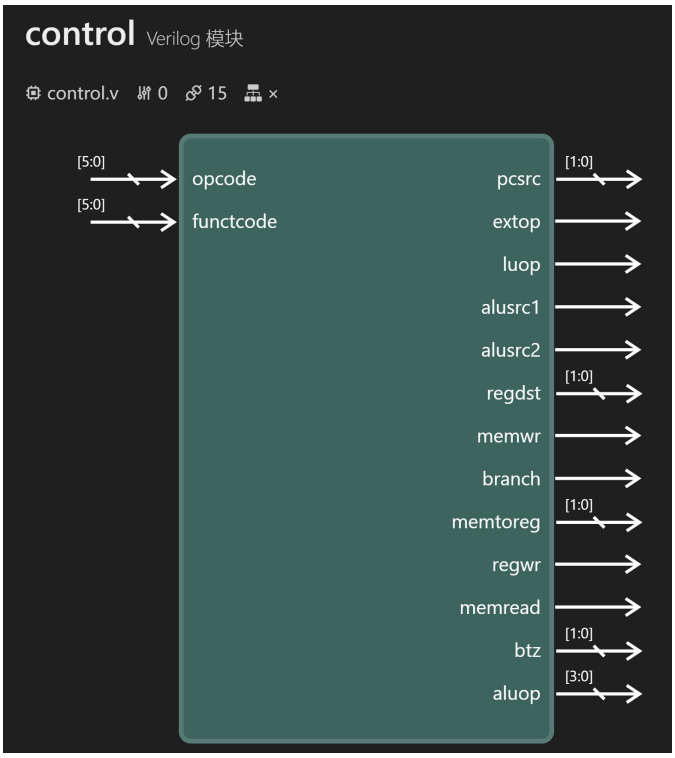


图 9　control 模块的系统框图

三、算法与指令（软件部分设计方案）

（一）计算稀疏矩阵部分代码

稀疏矩阵的计算部分基本采用的是数逻理论课期末大作业的代码，其中有少量修改。例如原本是从文件中利用 syscall 读取矩阵的数据，在处理器中则改为将矩阵数据预先初始化在 DataMemory 中（如图 10），之后从相应地址的 DataMemory 中直接读取矩阵数据，而后进行计算。

```
RAM_data[0]<=32'd3; //M   0000_0000
RAM_data[1]<=32'd4; //N
RAM_data[2]<=32'd5; //P
RAM_data[3]<=32'd4; //S
// value
RAM_data[4]<=32'd9;
RAM_data[5]<=32'd7;
RAM_data[6]<=32'd15;
RAM_data[7]<=32'd9;
//col
RAM_data[8]<=32'd2;
RAM_data[9]<=32'd1;
RAM_data[10]<=32'd0;
```

图 10

具体代码如下：

sparse_matmul:

```
    li $t0, 0
    lw $s0, 0($t0)    # $s0: m
    lw $s1, 4($t0)    # $s1: n
    lw $s2, 8($t0)    # $s2: p
    lw $s3, 12($t0) # $s3: s

    addi $s4, $t0, 16 # $s4: values

    sll $s5, $s3, 2
    add $s5, $s5, $s4 # $s5: col_indices

    sll $s6, $s3, 3
    add $s6, $s6, $s4 # $s6: row_ptr

    addi $s7, $s0, 1
    sll $s7, $s7, 2
    add $s7, $s7, $s6 # $s7: B

    # TODO
    move $t0,$zero
    mul $t1,$s0,$s2 #    m*p
    addi $t2,$s7,400
    #la $t2,C
initializeC:
sw $zero,0($t2)
addi $t2,$t2,4
addi $t0,$t0,1
bne $t0,$t1,initializeC
#  完成对 C 的初始化

move $t0,$zero # i
move $t1,$zero # j
move $t2,$zero # l


for_i:
lw $t3,0($s6) # start
lw $t4,4($s6) # end
move $t1,$t3 # j=start

for_j:
move $t5,$s5 #    col_indices
move $t6,$s4 #values
mul $t7,$t1,4 #4j 4j 4j
```

```
add $t5,$t5,$t7
add $t6,$t6,$t7
lw $t5,0($t5)# t5 for    kkkk
lw $t6,0($t6) # t6 for val val val
move $t2,$zero # l  重置 L 为 0
for_l:
addi $t7,$s7,400
#la $t7,C
mul $t8,$t0,$s2 # ip
add $t8,$t8,$t2 # ip+L
mul $t8,$t8,4 #4ip+4
add $t7,$t8,$t7
lw $t9,0($t7)    #t9 for c_val c_val


mul $t8,$t5,$s2 #kp
add $t8,$t8,$t2 # kp+L
mul $t8,$t8,4 #4kp+4
add $t8,$t8,$s7
lw $t8,0($t8) # t8 for B[kp+1]
mul $t8,$t8,$t6 # now t8 for val*B[kp+1]
add $t9,$t9,$t8 # new c_val
sw $t9,0($t7)
#完成循环内的操作  完成循环内的操作  完成循环内的操作

addi $t2,$t2,1
blt $t2,$s2,for_l
addi $t1,$t1,1
blt $t1,$t4,for_j
addi $t0,$t0,1
addi $s6,$s6,4 # s6+4, row_ptr[i->i+1]
blt $t0,$s0,for_i
```

（二）BCD 数码管控制代码

本次实验通过软件译码的形式控制 BCD 数码管显示数字。

稀疏矩阵计算完毕后，会将 C 矩阵的结果存储在 DataMemory[100]及之后的存储器单元中。此外，我已经先将数字 0~f 对应的段码（即 BCD 的 12 位控制信号）存储在了 DataMemory[36]至 DataMemory[51]中，因此之后要显示某个数字，只需要到对应地址查表获得其段码即可。

对 BCD 应显示的数字进行软件译码时，首先从对应位置读取一个 C 矩阵中的元素数字。该数字为 int 型，因此有 32bit，但四位数码管以十六进制形式显示数字的话，最多显示 16 位二进制数字。不过由于 C 矩阵的元素数值都不是非常大，因此取其低 16 位进行显示即可。

将低 16 位分为 4 个 4bit 数字，每个 4bit 数字对应一个十六进制数字。根据这一十六进

制数字是多少，可以直接从 DataMemory 对应地址读取数码管的控制信号，使得数码管显示这个数字。而后通过一个循环，使得该数字持续显示 1ms，之后显示另外几个十六进制数字，并且控制其他位置的数码管亮起。如此循环显示这 4 个十六进制数字，每个显示 1ms 而后循环，整体持续显示 1s。1s 后开始对 C 矩阵中的下一个元素进行译码与显示。

BCD 数码管控制代码的重要思路在于通过分段译码来确定每一位该显示什么数字、通过循环来控制数字的显示时间。

具体代码如下：

```
# 显示结果矩阵 C
    addi $t7, $s7, 400 # C 矩阵地址
    addi $t8, $zero, 0 # 计数器
    mul $t9, $s0, $s2 # 元素个数 (m*p)
     lui $t2,0x4000
      addiu $t2,$t2,0x0010




    display_loop:
    beq $t8, $t9, end_program
    lw $t6, 0($t7)      # 读取当前元素
     addi $t5, $zero, 2000 # 设置循环计数器为 250 (250*4ms=1000ms)
    jal displaylow16bit
    addi $t7,$t7,4
    addi $t8,$t8,1
    j display_loop


displaylow16bit:

 #andi $t4, $t6, 0xFFFF        # 取参数的低 16 位存入 s0
    #addiu $t5, $zero, 250       # 设置循环计数器为 250 (250*4ms=1000ms)
   # addi $t5, $zero, 2

    srl $t4, $t6, 12             # 右移 12 位获取最高 4 位
    andi $t4, $t4, 0xF           # 屏蔽高位，保留最低 4 位
    addi $t3, $zero, 16          # 设置数码管位置为 0（最左边）


    addi $t0,$t4,36
    sll $t0,$t0,2
    lw $t1,0($t0) # t1 for a-g
    sll $s1,$t3,7
    add $s2,$s1,$t1
```

```
    sw $s2,0($t2)
        # 返回调用者
    addiu $s3, $zero, 0x0940      # 设置计数器低 16 位 (0x0D40 = 12500)
delay_1ms_loop1:
    addiu $s3, $s3, -1            # 计数器减 1
    bne $s3, $zero, delay_1ms_loop1



   srl $t4, $t6, 8              # 右移 12 位获取最高 4 位
    andi $t4, $t4, 0xF            # 屏蔽高位，保留最低 4 位
    addi $t3, $zero, 8          # 设置数码管位置为 0（最左边）


    addi $t0,$t4,36
    sll $t0,$t0,2
    lw $t1,0($t0) # t1 for a-g
    sll $s1,$t3,7
    add $s2,$s1,$t1
    sw $s2,0($t2)
        # 返回调用者
    addiu $s3, $zero, 0x0940      # 设置计数器低 16 位 (0x0D40 = 12500)
delay_1ms_loop2:
    addiu $s3, $s3, -1            # 计数器减 1
    bne $s3, $zero, delay_1ms_loop2



   srl $t4, $t6, 4              # 右移 12 位获取最高 4 位
    andi $t4, $t4, 0xF            # 屏蔽高位，保留最低 4 位
    addi $t3, $zero, 4          # 设置数码管位置为 0（最左边）


    addi $t0,$t4,36
    sll $t0,$t0,2
    lw $t1,0($t0) # t1 for a-g
    sll $s1,$t3,7
    add $s2,$s1,$t1
    sw $s2,0($t2)
        # 返回调用者
    addiu $s3, $zero, 0x0940      # 设置计数器低 16 位 (0x0D40 = 12500)
delay_1ms_loop3:
    addiu $s3, $s3, -1            # 计数器减 1
```

```
        bne $s3, $zero, delay_1ms_loop3



        andi $t4, $t6, 0xF              # 屏蔽高位，保留最低 4 位
        addi $t3, $zero, 2          # 设置数码管位置为 0（最左边）


        addi $t0,$t4,36
        sll $t0,$t0,2
        lw $t1,0($t0) # t1 for a-g
        sll $s1,$t3,7
        add $s2,$s1,$t1
        sw $s2,0($t2)
              # 返回调用者
        addiu $s3, $zero, 0x0940       # 设置计数器低 16 位  (0x0D40 = 12500)
delay_1ms_loop4:
        addiu $s3, $s3, -1              # 计数器减 1
        bne $s3, $zero, delay_1ms_loop4


        addi $t5,$t5,-1
        beq $t5,$zero,jrra
        j displaylow16bit
        jrra:
        jr $ra
end_program:
```

最终将这些汇编代码转换为指令，写到 instructionMemory 中

```
8'd0: Instruction <= 32'h24020002;
8'd1: Instruction <= 32'h24080000;
8'd2: Instruction <= 32'h8d100000;
8'd3: Instruction <= 32'h8d110004;
8'd4: Instruction <= 32'h8d120008;
8'd5: Instruction <= 32'h8d13000c;
8'd6: Instruction <= 32'h21140010;
8'd7: Instruction <= 32'h0013a880;
8'd8: Instruction <= 32'h02b4a820;
8'd9: Instruction <= 32'h0013b0c0;
8'd10: Instruction <= 32'h02d4b020;
8'd11: Instruction <= 32'h22170001;
8'd12: Instruction <= 32'h0017b880;
8'd13: Instruction <= 32'h02f6b820;
8'd14: Instruction <= 32'h00004021;
8'd15: Instruction <= 32'h72124802;
```

四、关键代码及文件清单

（一）文件清单

文件名称	文件功能描述
top.v	总模块

MYCPU.v	对流水线处理器的各个功能模块进行连接组装
IF.v	取指令，并决定下一个指令从什么地址读取
instructionMemory.v	指令存储器
hazard.v	冒险检测，并控制各个级间寄存器的更新方式
ID.v	对指令进行解析，生成各类控制信号，并判断是否 jump
control.v	根据指令，生成各类控制信号
registerfile.v	寄存器堆，支持先写后读功能
ALUControl.v	生成 ALU 控制信号，告知 ALU 该进行何种操作
forward_unit.v	冒险检测，并进行数据转发
EX.v	对数据进行计算，将计算结果传给后续模块
ALU.v	计算模块
MEM.v	数据存储器模块，可对存储器进行读、写操作
datamemory.v	数据存储器模块
WB.v	写回模块，决定将何来源的数据写回到寄存器堆中
yueshu.xdc	约束文件，绑定板子的管脚
tb.v	仿真文件
change.asm	汇编文件，将该文件中的汇编代码转换为指令，令处理器执行这些指令

（二）关键代码

MYCPU 对 IF、ID、EX、MEM、WB 五个主要阶段以及 control、forward_unit、hazard 等重要模块进行连接与组装。之后再使用 IP 核生成 55MHz 的时钟信号驱动处理器运行。

（1）MYCPU.v

```
`timescale 1ns / 1ps

module MYCPU(
input clk,
input reset,
output [11:0]leds
    );

    wire dojump;
    wire dobranch;
    wire [31:0]jumpaddress;
    wire [31:0]branchaddress;
    wire [31:0]if_id_pc_plus_4;
    wire [31:0]if_id_instr;
```

```
wire [1:0]control_if_id;
wire flush_id_ex;
wire pckeep;
wire [5:0]id_ex_opcode;
wire id_ex_memread;
wire [4:0]id_ex_rt;

 wire mem_regWrite_out; // WB 带来的控制信号
 wire [4 : 0] mem_regWriteAddr_out; // WB 带来的写入地址
 wire [31 : 0] mem_regWriteData_out; // WB 带来的写入数据



    //output reg extop_out,
    //output reg luop_out, // for lui
    wire id_ex_alusrc1; //判断使用来自 busB 的数据  0 or  立即数  1
    wire id_ex_alusrc2; //  针对 srl,sll,sra,是否使用 shamet
    wire [1:0]id_ex_regdst;//  可能要写到  $rt 00 $rd 01 $ra 10
    wire id_ex_memwr;
    wire id_ex_branch;
    wire [1:0]id_ex_memtoreg;//  可能要把  存储器中的值 01、ALU 计算的值 00、PC+4
的值（jal）写到寄存器堆中 10
    wire id_ex_regwr;
    wire[1:0]id_ex_btz;
  wire[4:0] id_ex_alucon;
wire id_ex_sign;
//指令中包含的数据信息

wire [4:0]id_ex_rd;
wire[31:0]id_ex_ext_imm;
wire[5:0]id_ex_fuc;
wire[31:0]id_ex_shamt;
wire[31:0] id_ex_registerbusA;
wire[31:0] id_ex_registerbusB;
wire[31:0] id_ex_id_pc4;
wire [4:0]id_ex_rs;

 wire [1:0] id_ex_pcsrc_out; //??????????????????????????

  wire ex_mem_memwr;
    wire ex_mem_memread;
    wire [1:0] ex_mem_memtoreg;
    wire ex_mem_regwrite_out;
```

```
        wire [31 : 0] ex_mem_pc_plus_4_out;
        wire [31 : 0] ex_mem_aluout;
        wire [31 : 0] ex_mem_busB;
        wire [4 : 0] ex_mem_regwraddress;
    wire [1:0]forwardA;
    wire[1:0]forwardB;
    wire[1:0] mem_wb_memtoreg_out;
    wire[31 : 0] mem_wb_pc_plus_4_out;
    wire[31 : 0] mem_wb_aluout_out;
    wire[31:0] mem_wb_data_read_from_mem;
    wire[11:0]ledmid;



  IF IFreal(
  .clk(clk),
.reset(reset),
  .dojump(dojump),
  .pc_keep(pckeep), //flush 时候  keep PC
  .dobranch(dobranch),// zero&&isbranch,应该在 EX 阶段产生并送入
  .jumpaddress(jumpaddress),
  .branchaddress(branchaddress),
  .if_id_pc4in(if_id_pc_plus_4),
  .if_id_instrin(if_id_instr),//此时 IF/ID 寄存器存的值，如果 flush，则可能把这两个再输出
  .next_condition(control_if_id),// whether to flush
  .if_id_pc4out(if_id_pc_plus_4),
  .if_id_instrout(if_id_instr)
        );


  hazard hazardreal(
        .reset(reset),
        .id_ex_memread(id_ex_memread),
        .id_ex_rt(id_ex_rt),
        .if_id_rs(if_id_instr[25:21]),
        .if_id_rt(if_id_instr[20:16]),      // for load-use
        .dojump(dojump), // for j/jal/jr
        .dobranch(dobranch),// for beq bne
        .pckeep(pckeep),
        .control_if_id(control_if_id), // 00 go on      ; 01 flush    ;10 keep
        .flush_id_ex(flush_id_ex)
              );
```

```
ID IDreal(
    .clk(clk),
    .reset(reset),
    .whetherflush(flush_id_ex),
    //IF/ID 级间寄存器数据
    .pc4(if_id_pc_plus_4),
    .instruction(if_id_instr),
    //不需要传入 ID/EX 寄存器的数据，因为 ID/EX 要不然正常 go on，要不然 flush 清
零

    //可能有数据需要写回到寄存器堆中
    .regWrite(mem_regWrite_out), // WB 带来的控制信号
    .regWriteAddr(mem_regWriteAddr_out), // WB 带来的写入地址
    .regWriteData(mem_regWriteData_out), // WB 带来的写入数据

    //是否 jump
    .whetherjump(dojump),
    .jumpaddress(jumpaddress),
    //输出数据，包括一大堆控制信号+指令中包含的数据信息
    //控制信号
    .pcsrc_out(id_ex_pcsrc_out),// 可能来自 PC+4 00   PC+4 再偏移 01   $rs（jr） 10, 11
j/jal
    //output reg extop_out,
    //output reg luop_out, // for lui
    .alusrc1_out(id_ex_alusrc1), //判断使用来自 busB 的数据 0 or 立即数 1
    .alusrc2_out(id_ex_alusrc2), // 针对 srl,sll,sra,是否使用 shamet
    .regdst_out(id_ex_regdst),// 可能要写到 $rt 00 $rd 01 $ra 10
    . memwr_out(id_ex_memwr),
    .branch_out(id_ex_branch),
    .memtoreg_out(id_ex_memtoreg), // 可能要把 存储器中的值 01、ALU 计算的值 00、
PC+4 的值（jal）写到寄存器堆中 10
    .regwr_out(id_ex_regwr),
    .memread_out(id_ex_memread), // used in hazard unit
    .btz_out(id_ex_btz), // 00 for bltz ;01 for bgtz; 10 for blez
    //output   reg[3:0]aluop_out,
    .alucon(id_ex_alucon),
    .sign(id_ex_sign),
    //指令中包含的数据信息
    .rs(id_ex_rs),
    .rt(id_ex_rt),
     .rd(id_ex_rd),
```

```
        .ext_imm(id_ex_ext_imm),
        .fuc(id_ex_fuc),
        .shamt(id_ex_shamt),
        .registerbusA(id_ex_registerbusA),
        .registerbusB(id_ex_registerbusB),
        .id_pc4(id_ex_id_pc4),
        .opcode_out(id_ex_opcode)
                );



    forward_unit forward_unitreal(
                .ex_mem_regwrite(ex_mem_regwrite_out),
                .ex_mem_regwraddr(ex_mem_regwraddress),
               .mem_wb_regwrite(mem_regWrite_out),
                .mem_wb_regwraddr(mem_regWriteAddr_out),
                .id_ex_rs(id_ex_rs),
                .id_ex_rt(id_ex_rt),
                .forwardA(forwardA),
                .forwardB(forwardB)
                      );



    EX EXreal(
                .clk(clk),
                .reset(reset),
                // 数据旁路
                .forwardA(forwardA),
                .forwardB(forwardB),
                .ex_mem_data(ex_mem_aluout),
                .mem_wb_data(mem_regWriteData_out),

                .opcode(id_ex_opcode),


                // ID 写到 ID/EX 寄存器里的数据
                .pcsrc_out(id_ex_pcsrc_out),// 可能来自 PC+4 00  PC+4 再偏移 01  $rs（jr）
10, 11 j/jal
                .alusrc1_out(id_ex_alusrc1), //判断使用来自 busB 的数据 0 or 立即数 1
                .alusrc2_out(id_ex_alusrc2), // 针对 srl,sll,sra,是否使用 shamet
                .regdst_out(id_ex_regdst),// 可能要写到 $rt 00 $rd 01 $ra 10
                .memwr_out(id_ex_memwr),
                .branch_out(id_ex_branch),
```

```
        .memtoreg_out(id_ex_memtoreg), //  可能要把  存储器中的值 01、ALU 计算的值 00、PC+4 的值（jal）写到寄存器堆中 10
        .regwr_out(id_ex_regwr),
        .memread_out(id_ex_memread), // used in hazard unit
        .btz_out(id_ex_btz), // 00 for bltz ;01 for bgtz; 10 for blez
        .alucon(id_ex_alucon),
       .sign(id_ex_sign),
        //指令中包含的数据信息
        .rs(id_ex_rs),
        .rt(id_ex_rt),
        .rd(id_ex_rd),
        .ext_imm(id_ex_ext_imm),
        .fuc(id_ex_fuc),
        .shamt(id_ex_shamt),
        .registerbusA(id_ex_registerbusA),
        .registerbusB(id_ex_registerbusB),
        .id_pc4(id_ex_id_pc4),

        // for branch
        .whether_branch(dobranch),
        .branch_address(branchaddress),

        .ex_mem_memwr(ex_mem_memwr),
        . ex_mem_memread(ex_mem_memread),
        .ex_mem_memtoreg(ex_mem_memtoreg),
       .ex_mem_regwrite_out(ex_mem_regwrite_out),

        . ex_mem_pc_plus_4_out(ex_mem_pc_plus_4_out),
        .ex_mem_aluout(ex_mem_aluout),
        .ex_mem_busB(ex_mem_busB),
        .ex_mem_regwraddress(ex_mem_regwraddress)

            );


    MEM MEMreal(
            .reset(reset),
            .clk(clk),

            .ex_mem_memwr(ex_mem_memwr),
           . ex_mem_memread(ex_mem_memread),
           .ex_mem_memtoreg(ex_mem_memtoreg),
          .ex_mem_regwrite(ex_mem_regwrite_out),
```

```
                    . ex_mem_pc_plus_4(ex_mem_pc_plus_4_out),
                    .ex_mem_aluout(ex_mem_aluout),
                    .ex_mem_busB(ex_mem_busB),
                    .ex_mem_regwraddress(ex_mem_regwraddress),

                     .mem_wb_memtoreg_out(mem_wb_memtoreg_out),
                     .mem_wb_regwrite_out(mem_regWrite_out),
                    .mem_wb_pc_plus_4_out(mem_wb_pc_plus_4_out),
                     .mem_wb_aluout_out(mem_wb_aluout_out),
                     .mem_wb_regwraddress_out(mem_regWriteAddr_out),
                     .data_read_from_mem(mem_wb_data_read_from_mem),
                     .leds(ledmid) //汇编指令使用 sw 向特定地址存入 LED 的控制信号，然后我们直接取出来用
                                );


        WB WBreal(
                                // MEM  写入 MEM/WB 寄存器里的值
                               .mem_wb_memtoreg_out(mem_wb_memtoreg_out),
                                .mem_wb_pc_plus_4_out(mem_wb_pc_plus_4_out),
                                .mem_wb_aluout_out(mem_wb_aluout_out),
                                .data_read_from_mem(mem_wb_data_read_from_mem),
                                // choose 1 from these 3
                                .write_to_reg_data(mem_regWriteData_out)
                                      );


                assign leds=ledmid;

endmodule
//ok 7.1
```

（2）IF.v

```
`timescale 1ns / 1ps
//取指令，指令可以来源于 PC+4、branch、j、jr
//在分支指令、跳转指令跳转后，还需要 flush，因此需要把 IF/ID 寄存器里的东西存回来
//要给 IF/ID 寄存器传这个指令的 PC、instr
module IF(
input clk,
input reset,
input dojump,
input pc_keep, //flush 时候  keep PC
input dobranch,// zero&&isbranch,应该在 EX 阶段产生并送入
input [31:0]jumpaddress,
```

```
input [31:0]branchaddress,
input [1:0]next_condition,// whether to flush

input [31:0]if_id_pc4in,
input [31:0]if_id_instrin,//此时 IF/ID 寄存器存的值，如果 flush，则可能把这两个再输出

output reg [31:0]if_id_pc4out,
output reg [31:0]if_id_instrout

    );

    //IF 区的组合逻辑部分
    reg [31:0]pcaddress; //按 reset 后，赋值为 0，之后每一个周期，Pc+4
    wire [31:0]instruction;

    InstructionMemory im1(
    .Address(pcaddress),
    .Instruction(instruction)
    );
    wire [31:0]pc4;
    assign pc4=pcaddress+32'd4;
    wire [31:0]pc_next;
    assign pc_next=pc_keep?pcaddress:
                   dobranch?branchaddress:
                   dojump?jumpaddress:pc4;
                   //确定好下一个 PC 到底是谁


     always@(posedge clk or posedge reset)begin
     if(reset)begin
     pcaddress<=32'd0;
     if_id_pc4out<=32'd0;
     if_id_instrout<=32'd0;
     end
     else begin
     pcaddress<=pc_next;

     case(next_condition)
     2'b00 :begin // go on
     if_id_pc4out<=pc4;//如果要是 go on，此处的 PCnext 应该就等于 pc+4
         if_id_instrout<=instruction;
     end
     2'b01:begin// flush to zero
     if_id_pc4out<=32'd0;
```

```
            if_id_instrout<=32'd0;
        end
        2'b10:begin//keep    for load-use
        if_id_pc4out<=if_id_pc4in;
            if_id_instrout<=if_id_instrin;
        end
        endcase
        end
        end
endmodule
//ok 7.1
```

(3) instructionMemory.v

```
module InstructionMemory(
    input          [32 -1:0] Address,
    output reg [32 -1:0] Instruction
);

    always @(*)
        case (Address[9:2])

            // -------- Paste Binary Instruction Below (Inst-q1-1/Inst-q1-2.txt)



            8'd0: Instruction <= 32'h24080000;
            8'd1: Instruction <= 32'h8d100000;
            8'd2: Instruction <= 32'h8d110004;
            8'd3: Instruction <= 32'h8d120008;
            8'd4: Instruction <= 32'h8d13000c;
            8'd5: Instruction <= 32'h21140010;
            8'd6: Instruction <= 32'h0013a880;
            8'd7: Instruction <= 32'h02b4a820;
            8'd8: Instruction <= 32'h0013b0c0;
            8'd9: Instruction <= 32'h02d4b020;
            8'd10: Instruction <= 32'h22170001;
            8'd11: Instruction <= 32'h0017b880;
            8'd12: Instruction <= 32'h02f6b820;
            8'd13: Instruction <= 32'h00004021;
            8'd14: Instruction <= 32'h72124802;
            8'd15: Instruction <= 32'h22ea0190;
            8'd16: Instruction <= 32'had400000;
            8'd17: Instruction <= 32'h214a0004;
```

```
8'd18: Instruction <= 32'h21080001;
8'd19: Instruction <= 32'h1509fffc;
8'd20: Instruction <= 32'h00004021;
8'd21: Instruction <= 32'h00004821;
8'd22: Instruction <= 32'h00005021;
8'd23: Instruction <= 32'h8ecb0000;
8'd24: Instruction <= 32'h8ecc0004;
8'd25: Instruction <= 32'h000b4821;
8'd26: Instruction <= 32'h00156821;
8'd27: Instruction <= 32'h00147021;
8'd28: Instruction <= 32'h20010004;
8'd29: Instruction <= 32'h71217802;
8'd30: Instruction <= 32'h01af6820;
8'd31: Instruction <= 32'h01cf7020;
8'd32: Instruction <= 32'h8dad0000;
8'd33: Instruction <= 32'h8dce0000;
8'd34: Instruction <= 32'h00005021;
8'd35: Instruction <= 32'h22ef0190;
8'd36: Instruction <= 32'h7112c002;
8'd37: Instruction <= 32'h030ac020;
8'd38: Instruction <= 32'h20010004;
8'd39: Instruction <= 32'h7301c002;
8'd40: Instruction <= 32'h030f7820;
8'd41: Instruction <= 32'h8df90000;
8'd42: Instruction <= 32'h71b2c002;
8'd43: Instruction <= 32'h030ac020;
8'd44: Instruction <= 32'h20010004;
8'd45: Instruction <= 32'h7301c002;
8'd46: Instruction <= 32'h0317c020;
8'd47: Instruction <= 32'h8f180000;
8'd48: Instruction <= 32'h730ec002;
8'd49: Instruction <= 32'h0338c820;
8'd50: Instruction <= 32'hadf90000;
8'd51: Instruction <= 32'h214a0001;
8'd52: Instruction <= 32'h0152082a;
8'd53: Instruction <= 32'h1420ffed;
8'd54: Instruction <= 32'h21290001;
8'd55: Instruction <= 32'h012c082a;
8'd56: Instruction <= 32'h1420ffe1;
8'd57: Instruction <= 32'h21080001;
8'd58: Instruction <= 32'h22d60004;
8'd59: Instruction <= 32'h0110082a;
8'd60: Instruction <= 32'h1420ffda;
8'd61: Instruction <= 32'h22ef0190;
```

```
8'd62: Instruction <= 32'h20180000;
8'd63: Instruction <= 32'h7212c802;
8'd64: Instruction <= 32'h3c0a4000;
8'd65: Instruction <= 32'h254a0010;
8'd66: Instruction <= 32'h13190039;
8'd67: Instruction <= 32'h8dee0000;
8'd68: Instruction <= 32'h200d07d0;
8'd69: Instruction <= 32'h0c100049;
8'd70: Instruction <= 32'h21ef0004;
8'd71: Instruction <= 32'h23180001;
8'd72: Instruction <= 32'h08100042;
8'd73: Instruction <= 32'h000e6302;
8'd74: Instruction <= 32'h318c000f;
8'd75: Instruction <= 32'h200b0010;
8'd76: Instruction <= 32'h21880024;
8'd77: Instruction <= 32'h00084080;
8'd78: Instruction <= 32'h8d090000;
8'd79: Instruction <= 32'h000b89c0;
8'd80: Instruction <= 32'h02299020;
8'd81: Instruction <= 32'had520000;
8'd82: Instruction <= 32'h24130940;
8'd83: Instruction <= 32'h2673ffff;
8'd84: Instruction <= 32'h1660fffe;
8'd85: Instruction <= 32'h000e6202;
8'd86: Instruction <= 32'h318c000f;
8'd87: Instruction <= 32'h200b0008;
8'd88: Instruction <= 32'h21880024;
8'd89: Instruction <= 32'h00084080;
8'd90: Instruction <= 32'h8d090000;
8'd91: Instruction <= 32'h000b89c0;
8'd92: Instruction <= 32'h02299020;
8'd93: Instruction <= 32'had520000;
8'd94: Instruction <= 32'h24130940;
8'd95: Instruction <= 32'h2673ffff;
8'd96: Instruction <= 32'h1660fffe;
8'd97: Instruction <= 32'h000e6102;
8'd98: Instruction <= 32'h318c000f;
8'd99: Instruction <= 32'h200b0004;
8'd100: Instruction <= 32'h21880024;
8'd101: Instruction <= 32'h00084080;
8'd102: Instruction <= 32'h8d090000;
8'd103: Instruction <= 32'h000b89c0;
8'd104: Instruction <= 32'h02299020;
8'd105: Instruction <= 32'had520000;
```

```
            8'd106: Instruction <= 32'h24130940;
            8'd107: Instruction <= 32'h2673ffff;
            8'd108: Instruction <= 32'h1660fffe;
            8'd109: Instruction <= 32'h31cc000f;
            8'd110: Instruction <= 32'h200b0002;
            8'd111: Instruction <= 32'h21880024;
            8'd112: Instruction <= 32'h00084080;
            8'd113: Instruction <= 32'h8d090000;
            8'd114: Instruction <= 32'h000b89c0;
            8'd115: Instruction <= 32'h02299020;
            8'd116: Instruction <= 32'had520000;
            8'd117: Instruction <= 32'h24130940;
            8'd118: Instruction <= 32'h2673ffff;
            8'd119: Instruction <= 32'h1660fffe;
            8'd120: Instruction <= 32'h21adffff;
            8'd121: Instruction <= 32'h11a00001;
            8'd122: Instruction <= 32'h08100049;
            8'd123: Instruction <= 32'h03e00008;




            // -------- Paste Binary Instruction Above

            default: Instruction <= 32'h00000000;
        endcase

endmodule
// OK 7.1
// 55MHZ
```

（4）hazard.v

```
//stall 可能来自于  load-use     j/jr/jal    beq/bne
module hazard(
input reset,
input id_ex_memread,
input [4:0]id_ex_rt,
input [4:0]if_id_rs,
input [4:0]if_id_rt,      // for load-use
input dojump, // for j/jal/jr
input dobranch,// for beq bne
```

```
output reg pckeep,
output reg [1:0]control_if_id, // 00 go on      ; 01 flush    ;10 keep
output reg flush_id_ex
      );

    //                assign                pckeep=                (id_ex_memread                &&
(id_ex_rt==if_id_rs)&&(id_ex_rt==if_id_rt))?1'b1:1'b0; // only when load-use ,keep pc
   //    assign flush_id_ex=


      always@(*)begin
      if(reset)begin
        pckeep=1'b0;
              control_if_id=2'b00;
              flush_id_ex=1'b0;
      end


      else begin

      if (id_ex_memread && ((id_ex_rt==if_id_rs)||(id_ex_rt==if_id_rt)) ) // load-use
      begin
      pckeep=1'b1;
      control_if_id=2'b10;
      flush_id_ex=1;
      end

      else if(dobranch)begin // branch flush IF/ID and ID/EX
      pckeep=1'b0;
      control_if_id=2'b01;
      flush_id_ex=1'b1;
      end
      else if(dojump) // for j/jal/jr
      begin
      pckeep=1'b0;
      control_if_id=2'b01;
      flush_id_ex=1'b0;
      end
      else begin // normal
      pckeep=1'b0;
            control_if_id=2'b00;
            flush_id_ex=1'b0;
      end
```

```
    end

    end


endmodule
// OK 7.1
```

（5）ID.v

```
//内含一个 registerfile 和 control
module ID(
input clk,
input reset,

input whetherflush,
//IF/ID 级间寄存器数据
input [31:0]pc4,
input [31:0]instruction,
//不需要传入 ID/EX 寄存器的数据，因为 ID/EX 要不然正常 go on，要不然 flush 清零

//可能有数据需要写回到寄存器堆中
input regWrite, // WB 带来的控制信号
input [4 : 0] regWriteAddr, // WB 带来的写入地址
input [31 : 0] regWriteData, // WB 带来的写入数据

//是否 jump
output whetherjump,
output [31:0]jumpaddress,
//输出数据，包括一大堆控制信号+指令中包含的数据信息
//控制信号
output reg[1:0] pcsrc_out,// 可能来自 PC+4 00  PC+4 再偏移 01  $rs（jr） 10, 11 j/jal
//output reg extop_out,
//output reg luop_out, // for lui
output reg alusrc1_out, //判断使用来自 busB 的数据 0 or 立即数 1
output reg alusrc2_out, // 针对 srl,sll,sra,是否使用 shamet
output reg [1:0]regdst_out,// 可能要写到 $rt 00 $rd 01 $ra 10
output reg memwr_out,
output reg branch_out,
output reg [1:0]memtoreg_out, // 可能要把 存储器中的值 01、ALU 计算的值 00、PC+4 的值
（jal）写到寄存器堆中 10
output reg regwr_out,
output reg memread_out, // used in hazard unit
output  reg[1:0]btz_out, // 00 for bltz ;01 for bgtz; 10 for blez
```

```
//output    reg[3:0]aluop_out,
output reg[4:0] alucon,
output reg sign,
//指令中包含的数据信息
output reg[4:0]rs,
output reg[4:0]rt,
output reg[4:0]rd,
output reg[31:0]ext_imm,
output reg[5:0]fuc,
output reg[31:0]shamt,
output reg[31:0] registerbusA,
output reg[31:0] registerbusB,
output reg[31:0]id_pc4,
output reg[5:0]opcode_out
      );

         wire[1:0] pcsrc;// 可能来自 PC+4 00    PC+4 再偏移 01    $rs（jr） 10, 11 j/jal
         wire extop;
         wire luop;// for lui
         wire alusrc1; //判断使用来自 busB 的数据 0 or 立即数 1
         wire alusrc2; // 针对 srl,sll,sra,是否使用 shamet
         wire [1:0]regdst;// 可能要写到 $rt 00 $rd 01 $ra 10
         wire memwr;
         wire branch;
         wire [1:0]memtoreg; // 可能要把 存储器中的值 01、ALU 计算的值 00、PC+4 的值（jal）
写到寄存器堆中 10
        wire    regwr;
        wire    memread; // used in hazard unit
        wire     [1:0]btz; // 00 for bltz ;01 for bgtz; 10 for blez
        wire    [3:0]aluop;
        wire [31:0]EXTIMM;
        wire [31:0]IMMOUT;
      wire [4:0]rsmid;
      wire [4:0]rtmid;
    wire [31:0]busA;
    wire [31:0]busB;
    wire [4:0]aluconcon;
    wire signmid;
         assign rsmid=instruction[25:21];
         assign rtmid=instruction[20:16];

      wire[5:0] inopcode;
      assign inopcode=instruction[31:26];
      wire [5:0]funct;
```

```
assign funct=instruction[5:0];

control controlunit(
.opcode(inopcode),
.functcode(funct),
  .pcsrc(pcsrc),// 可能来自 PC+4 00   PC+4 再偏移 01   $rs（jr） 10, 11 j/jal
    .extop(extop),
      .luop(luop), // for lui
        .alusrc1(alusrc1), //判断使用来自 busB 的数据 0 or 立即数 1
        .alusrc2(alusrc2), // 针对 srl,sll,sra,是否使用 shamet
        .regdst(regdst),// 可能要写到 $rt 00 $rd 01 $ra 10
        .memwr(memwr),
        .branch(branch),
        .memtoreg(memtoreg), // 可能要把 存储器中的值 01、ALU 计算的值 00、PC+4 的值（jal）写到寄存器堆中 10
        .regwr(regwr),
        .memread(memread),// used in hazard unit
        .btz(btz),
        .aluop(aluop)
);


RegisterFile RF(
     .reset(reset)    ,
     .clk(clk) ,
      .RegWrite    (regWrite),
      .Read_register1(rsmid) ,
      .Read_register2(rtmid) ,
      .Write_register(regWriteAddr) ,
      .Write_data (regWriteData)       ,
      .Read_data1 (busA),
      .Read_data2 (busB)
);

ALUControl alucccccc(
      .ALUOp(aluop)          ,
      .Funct(instruction[5:0])          ,
     .ALUCtl(aluconcon) ,
      . Sign(signmid)
);

assign EXTIMM = extop ? {{16{instruction[15]}}, instruction[15 : 0]} : {16'h0000, instruction[15 : 0]};
assign IMMOUT = luop ? {instruction[15 : 0], 16'h0000} : EXTIMM;
```

```
assign whetherjump=((pcsrc==2'b11)||(pcsrc==2'b10))?1'b1:1'b0; // 11 for j and jal  ; 10 for jr
assign jumpaddress= (pcsrc==2'b11)?{pc4[31 : 28], instruction[25 : 0], 2'b00}:busA;
//如果是 j or jal, then 伪直接寻址； jr then 根据 rs 中的值选址


always@(posedge clk or posedge reset)begin
if(reset)begin
pcsrc_out<=2'b00;// 可能来自 PC+4 00  PC+4 再偏移 01  $rs（jr） 10, 11 j/jal

alusrc1_out<=1'b0; //判断使用来自 busB 的数据 0 or 立即数 1
alusrc2_out<=1'b0;// 针对 srl,sll,sra,是否使用 shamet
regdst_out<=2'd0;// 可能要写到 $rt 00 $rd 01 $ra 10
memwr_out<=1'b0;
branch_out<=1'b0;
memtoreg_out<=2'd0;// 可能要把 存储器中的值 01、ALU 计算的值 00、PC+4 的值（jal）写到寄存器堆中 10
regwr_out<=1'b0;
memread_out<=1'b0; // used in hazard unit
btz_out<=2'd0; // 00 for bltz ;01 for bgtz; 10 for blez
alucon<=5'd0;
sign<=1'b0;
rs<=5'd0;
rt<=5'd0;
rd<=5'd0;
ext_imm<=32'd0;
fuc<=6'd0;
shamt<=32'd0;
registerbusA<=32'd0;
registerbusB<=32'd0;
id_pc4<=32'd0;
opcode_out<=6'd0;
end
else begin
if(whetherflush)begin

 pcsrc_out<=2'b00;// 可能来自 PC+4 00  PC+4 再偏移 01  $rs（jr） 10, 11 j/jal

    alusrc1_out<=1'b0; //判断使用来自 busB 的数据 0 or 立即数 1
    alusrc2_out<=1'b0;// 针对 srl,sll,sra,是否使用 shamet
    regdst_out<=2'd0;// 可能要写到 $rt 00 $rd 01 $ra 10
    memwr_out<=1'b0;
    branch_out<=1'b0;
    memtoreg_out<=2'd0;// 可能要把 存储器中的值 01、ALU 计算的值 00、PC+4 的值
```

```
（jal）写到寄存器堆中 10
        regwr_out<=1'b0;
        memread_out<=1'b0; // used in hazard unit
        btz_out<=2'd0; // 00 for bltz ;01 for bgtz; 10 for blez
         alucon<=5'd0;
            sign<=1'b0;
        rs<=5'd0;
        rt<=5'd0;
        rd<=5'd0;
        ext_imm<=32'd0;
        fuc<=6'd0;
        shamt<=32'd0;
        registerbusA<=32'd0;
        registerbusB<=32'd0;
        id_pc4<=32'd0;
        opcode_out<=6'd0;
    end
    else begin
     pcsrc_out<=pcsrc;// 可能来自 PC+4 00   PC+4 再偏移 01   $rs（jr） 10, 11 j/jal

        alusrc1_out<=alusrc1; //判断使用来自 busB 的数据 0 or 立即数 1
        alusrc2_out<=alusrc2;// 针对 srl,sll,sra,是否使用 shamet
        regdst_out<=regdst;// 可能要写到 $rt 00 $rd 01 $ra 10
        memwr_out<=memwr;
        branch_out<=branch;
        memtoreg_out<=memtoreg;// 可能要把 存储器中的值 01、ALU 计算的值 00、PC+4
的值（jal）写到寄存器堆中 10
        regwr_out<=regwr;
        memread_out<=memread; // used in hazard unit
        btz_out<=btz; // 00 for bltz ;01 for bgtz; 10 for blez
        alucon<=aluconcon;
        sign<=signmid;
        rs<=rsmid;
        rt<=rtmid;
        rd<=instruction[15:11];
        ext_imm<=IMMOUT;
        fuc<=instruction[5:0];
        shamt<= {27'h0, instruction[10 : 6]};
        registerbusA<=busA;
        registerbusB<=busB;
        id_pc4<=pc4;
        opcode_out<=inopcode;
```

```
    end
```

```
    end
    end
```

```
endmodule
//OK 7.1
```

（6）control.v

```
`timescale 1ns / 1ps


module control(
input [5:0]opcode,
input [5:0]functcode,
output reg[1:0] pcsrc,// 可能来自 PC+4 00   PC+4 再偏移 01   $rs（jr） 10, 11 j/jal
output reg extop,
output reg luop, // for lui
output reg alusrc1, //判断使用来自 busB 的数据 0 or 立即数 1
output reg alusrc2, // 针对 srl,sll,sra,是否使用 shamet(1)
output reg [1:0]regdst,// 可能要写到 $rt 00 $rd 01 $ra 10
output reg memwr,
output reg branch,
output reg [1:0]memtoreg, // 可能要把 存储器中的值 01、ALU 计算的值 00、PC+4 的值（jal）
写到寄存器堆中 10
output reg regwr,
output reg memread, // used in hazard unit
output   [1:0]btz, // 00 for bltz ;01 for bgtz; 10 for blez
output   [3:0]aluop //判断 alu 该执行何种运算，加、乘、。。。。。
    );
    //由指令的 opcode 和 functcode 解析出该生成何种控制指令
    // 在 ID 阶段使用 control 模块，译码生成控制信号

    always@(*)begin
        if(opcode==6'h0)
        begin //R instruction
            case(functcode)
```

```
                    6'h20,6'h22,6'h21,6'h23,6'h25,6'h24,6'h26,6'h27          ://add          ,addu,
sub,subu,and,or,xor,nor
                     begin
                     pcsrc=2'b00;
                     extop=1'b0;
                     luop=1'b0;
                     alusrc1=1'b0;
                     alusrc2=1'b0;
                     regdst=2'b01;
                     memwr=1'b0;
                     branch=1'b0;
                     memtoreg=2'b00;
                     regwr=1'b1;
                     memread=1'b0;
                     end

                    6'h0,6'h2,6'h3: // sll,srl,sra
                    begin
                    pcsrc=2'b00;
                    extop=1'b1;
                    luop=1'b0;
                    alusrc1=1'b0;
                    alusrc2=1'b1;// use shamt!!!
                    regdst=2'b01; // rd
                    memwr=1'b0;
                    branch=1'b0;
                    memtoreg=2'b00;
                    regwr=1'b1;
                    memread=1'b0;

                    end

                     6'h8: // jr
                       begin
                       pcsrc=2'b10;
                       extop=1'b1;
                       luop=1'b0;
                       alusrc1=1'b0;
                       alusrc2=1'b0;
                       regdst=2'b00;
                       memwr=1'b0;
                       branch=1'b0;
                       memtoreg=2'b00;
                       regwr=1'b0;
```

```
                memread=1'b0;

                end

                6'h9: // jalr
                //也许有问题？？？
                  begin
                  pcsrc=2'b10;
                  extop=1'b0;
                  luop=1'b0;
                  alusrc1=1'b0;
                  alusrc2=1'b0;
                  regdst=2'b00;
                  memwr=1'b0;
                  branch=1'b0;
                  memtoreg=2'b10;// PC+4
                  regwr=1'b1;
                  memread=1'b0;

                  end



                  6'h2a,6'h2b: // slt,sltu
                begin
                pcsrc=2'b00;
                extop=1'b0;
                luop=1'b0;
                alusrc1=1'b0;
                alusrc2=1'b0;
                regdst=2'b01;
                memwr=1'b0;
                branch=1'b0;
                memtoreg=2'b00;
                regwr=1'b1;
                memread=1'b0;

                end
                endcase


    end
else begin // NOT R
if(opcode==6'h23)begin // Lw
```

```
    pcsrc=2'b00;
        extop=1'b1;
        luop=1'b0;
        alusrc1=1'b1;    // imm number
        alusrc2=1'b0;
        regdst=2'b00;//rt
        memwr=1'b0;
        branch=1'b0;
        memtoreg=2'b01; // from mem
        regwr=1'b1;
        memread=1'b1;
  end
  // checked!

  if(opcode==6'h2b)begin // sw
    pcsrc=2'b00;
        extop=1'b1;
        luop=1'b0;
        alusrc1=1'b1;    // imm number
        alusrc2=1'b0;
        regdst=2'b00;//rt
        memwr=1'b1;
        branch=1'b0;
        memtoreg=2'b01; // from mem
        regwr=1'b0;
        memread=1'b0;
  end
  // checked!

   if((opcode==6'h8)||(opcode==6'h9))begin // addi ,addiu
   pcsrc=2'b00;
      extop=1'b1;
      luop=1'b0;
      alusrc1=1'b1;    // imm number
      alusrc2=1'b0;
      regdst=2'b00;//rt
      memwr=1'b0;
      branch=1'b0;
      memtoreg=2'b00; // from alu
      regwr=1'b1;
      memread=1'b0;
end
  // checked!
   if(opcode==6'hc)begin // andi , in which i is Zero-ext-imm
```

```
                pcsrc=2'b00;
                   extop=1'b0;
                   luop=1'b0;
                   alusrc1=1'b1;    // imm number
                   alusrc2=1'b0;
                   regdst=2'b00;//rt
                   memwr=1'b0;
                   branch=1'b0;
                   memtoreg=2'b00; // from alu
                   regwr=1'b1;
                   memread=1'b0;
             end
    // checked!
    if((opcode==6'hb)||(opcode==6'ha))begin // sltiu and slti
   pcsrc=2'b00;
      extop=1'b1;
      luop=1'b0;
      alusrc1=1'b1;    // imm number
      alusrc2=1'b0;
      regdst=2'b00;//rt
      memwr=1'b0;
      branch=1'b0;
      memtoreg=2'b00; // from alu
      regwr=1'b1;
      memread=1'b0;
end
    // checked!!
      if(opcode==6'hf)begin // lui
        pcsrc=2'b00;
          extop=1'b1;
          luop=1'b1;
          alusrc1=1'b1;    // imm number
          alusrc2=1'b0;
          regdst=2'b00;//rt
          memwr=1'b0;
          branch=1'b0;
          memtoreg=2'b00; // from alu
          regwr=1'b1;
          memread=1'b0;
    end
              // checked!!

    if(opcode==6'h2)begin // j
      pcsrc=2'b11;
```

```
        extop=1'b0;
        luop=1'b0;
        alusrc1=1'b1;    // imm number
        alusrc2=1'b0;
        regdst=2'b00;//rt
        memwr=1'b0;
        branch=1'b0;
        memtoreg=2'b00; // from alu
        regwr=1'b0;
        memread=1'b0;
 end
            // checked!!
    if(opcode==6'h3)begin // jal
      pcsrc=2'b11;
        extop=1'b0;
        luop=1'b0;
        alusrc1=1'b1;    // imm number
        alusrc2=1'b0;
        regdst=2'b10;//ra
        memwr=1'b0;
        branch=1'b0;
        memtoreg=2'b10; // from PC+4
        regwr=1'b1;
        memread=1'b0;
  end
             // checked!!
    if(opcode==6'h4)begin // beq
  pcsrc=2'b00; //先认为不会跳，按照 PC+4 准备
  //如果到了 EX 阶段，发现其实需要跳转，再由 EX 阶段改 PC
      extop=1'b0;
      luop=1'b0;
      alusrc1=1'b0;    // imm number
      alusrc2=1'b0;
      regdst=2'b00;//ra
      memwr=1'b0;
      branch=1'b1;//证明是分支指令
      memtoreg=2'b00; // from PC+4
      regwr=1'b0;
      memread=1'b0;
end
          // checked!!

      if(opcode==6'h5)begin // bne
pcsrc=2'b00; //先认为不会跳，按照 PC+4 准备
```

```
    //如果到了 EX 阶段，发现其实需要跳转，再由 EX 阶段改 PC
        extop=1'b0;
        luop=1'b0;
        alusrc1=1'b0;    // imm number
        alusrc2=1'b0;
        regdst=2'b00;//ra
        memwr=1'b0;
        branch=1'b1;//证明是分支指令
        memtoreg=2'b00; // from PC+4
        regwr=1'b0;
        memread=1'b0;
end
          // checked!!
        if(opcode==6'h1c)begin // mul
           pcsrc=2'b00;
              extop=1'b1;
              luop=1'b0;
              alusrc1=1'b0;    // imm number
              alusrc2=1'b0;
              regdst=2'b01;//rd
              memwr=1'b0;
              branch=1'b0;
              memtoreg=2'b00; // fromalu
              regwr=1'b1;
              memread=1'b0;
        end

        if(opcode==6'h1||opcode==6'h7)begin // bgtz bltz
                        pcsrc=2'b00;
                           extop=1'b0;
                           luop=1'b0;
                           alusrc1=1'b0;    // imm number
                           alusrc2=1'b0;
                           regdst=2'b00;//rd
                           memwr=1'b0;
                           branch=1'b1;
                           memtoreg=2'b00; // fromalu
                           regwr=1'b0;
                           memread=1'b0;
                     end


                    // checked!!
      end
```

```
    end

     assign btz =    (opcode==6'h6)?2'b10:
     (opcode==6'h7)?2'b01:
     (opcode==6'h1)?2'b00:2'b00;
    // 00 for bltz ;01 for bgtz; 10 for blez
    //下面开始处理 aluop，得出 alu 该进行什么操作

    assign aluop[2:0] =
            (opcode == 6'h0)? 3'b010:
            ((opcode == 6'h4)||(opcode == 6'h5)||(opcode == 6'h1)||(opcode == 6'h6)||(opcode
== 6'h7))? 3'b001:   //分支类的，beq+bne+bltz+bgtz+blez
            (opcode == 6'h0c)? 3'b100: // andi
            (opcode == 6'h0a || opcode == 6'h0b)? 3'b101: //slti 和 sltiu
            (opcode == 6'h1c && functcode == 6'h02)? 3'b110://ori
            3'b000; //mul others

        assign aluop[3] = opcode[0];

endmodule
// OK 7.1
```

(7) forward_unit.v

```
module forward_unit(
input ex_mem_regwrite,
input [4:0]ex_mem_regwraddr,
input mem_wb_regwrite,
input [4:0]mem_wb_regwraddr,
input [4:0]id_ex_rs,
input [4:0]id_ex_rt,
output [1:0]forwardA,
output [1:0]forwardB
    );

    assign forwardA=(ex_mem_regwrite && (ex_mem_regwraddr!=5'd0) &&
(ex_mem_regwraddr==id_ex_rs))?2'b10:
                    (mem_wb_regwrite &&
(mem_wb_regwraddr!=5'd0)&&(mem_wb_regwraddr==id_ex_rs))?2'b01:2'b00;
    assign forwardB=(ex_mem_regwrite && (ex_mem_regwraddr!=5'd0) &&
(ex_mem_regwraddr==id_ex_rt))?2'b10:
```

(mem_wb_regwrite && (mem_wb_regwraddr!=5'd0)&&(mem_wb_regwraddr==id_ex_rt))?2'b01:2'b00;

endmodule

// OK 7.1

（8）EX.v

```
// 包含 alu  forward  hazard
module EX(
input clk,
input reset,
// 数据旁路
input [1:0]forwardA,
input [1:0]forwardB,
input [31:0]ex_mem_data,
input [31:0]mem_wb_data,

input [5:0]opcode,


// ID 写到 ID/EX 寄存器里的数据
input [1:0] pcsrc_out,// 可能来自 PC+4 00  PC+4 再偏移 01  $rs（jr） 10, 11 j/jal
input  alusrc1_out, //判断使用来自 busB 的数据 0 or 立即数 1
input  alusrc2_out, // 针对 srl,sll,sra,是否使用 shamet
input  [1:0]regdst_out,// 可能要写到 $rt 00 $rd 01 $ra 10
input  memwr_out,
input  branch_out,
input [1:0]memtoreg_out, // 可能要把 存储器中的值 01、ALU 计算的值 00、PC+4 的值（jal）写到寄存器堆中 10
input  regwr_out,
input  memread_out, // used in hazard unit
input  [1:0]btz_out, // 00 for bltz ;01 for bgtz; 10 for blez
input [4:0] alucon,
input  sign,
//指令中包含的数据信息
input [4:0]rs,
input [4:0]rt,
input [4:0]rd,
input [31:0]ext_imm,
input [5:0]fuc,
input [31:0]shamt,
```

```
input [31:0] registerbusA,
input [31:0] registerbusB,
input [31:0]id_pc4,

// for branch
output whether_branch,
output [31:0]branch_address,

output reg ex_mem_memwr,
output reg ex_mem_memread,
output reg [1:0] ex_mem_memtoreg,
output reg ex_mem_regwrite_out,

output reg [31 : 0] ex_mem_pc_plus_4_out,
output reg [31 : 0] ex_mem_aluout,
output reg [31 : 0] ex_mem_busB,
output reg [4 : 0] ex_mem_regwraddress

    );
    /*wire ex_mem_memwr_tmp;
    wire ex_mem_memread_tmp;
    wire [1:0] ex_mem_memtoreg_tmp;
    wire ex_mem_regwrite_out_tmp;
    wire [31 : 0] ex_mem_pc_plus_4_out_tmp;*/
    wire [31 : 0] ex_mem_aluout_tmp;
    /*wire [31 : 0] ex_mem_busB_tmp;
    wire [4 : 0] ex_mem_regwraddress_tmp;*/

    wire [31:0]forwardAdata;
    wire [31:0]forwardBdata;
    wire [31:0]in11;
    wire [31:0]in22;
    wire zero_tmp;
    wire [4:0]towhichreg;
    assign forwardAdata=(forwardA==2'b10)?ex_mem_data:
                        (forwardA==2'b01)?mem_wb_data:registerbusA;
    assign forwardBdata=(forwardB==2'b10)?ex_mem_data:
                          (forwardB==2'b01)?mem_wb_data:registerbusB;
    assign in11 = alusrc2_out ? shamt :forwardAdata;
    assign in22 = alusrc1_out ? ext_imm : forwardBdata;
    // regdst_out,//  可能要写到  $rt 00 $rd 01 $ra 10
    assign towhichreg=(regdst_out==2'b00)?rt:
                                (regdst_out==2'b01)?rd:5'd31;
                wire g;
```

```
                wire L;
    ALU aluhhh(
        .in1( in11)          ,
         .in2( in22)          ,
         . ALUCtl(alucon)      ,
         .Sign(sign)                    ,
         .out(ex_mem_aluout_tmp) ,
         .zero(zero_tmp),
         .greaterthanzero(g),
              .lower(L)
    );

    //assign          whether_branch=((opcode==6'h4)          &&          zero_tmp)||((opcode==6'h5)
&& !zero_tmp)||((btz_out==2'b01)&&g)||((btz_out==2'b00)&&L);
          assign        whether_branch=((opcode==6'h4)        &&        zero_tmp)||((opcode==6'h5)
&& !zero_tmp);
   // 00 for bltz ;01 for bgtz; 10 for blez
    // beq and bne
    assign branch_address= id_pc4 + {ext_imm[29 : 0], 2'b00};


    always @(posedge clk or posedge reset)begin
    if(reset)begin
    ex_mem_memwr<=1'b0;
    ex_mem_memread<=1'b0;
    ex_mem_memtoreg<=2'd0;
    ex_mem_regwrite_out<=1'b0;
    ex_mem_pc_plus_4_out<=32'd0;
    ex_mem_aluout<=32'd0;
    ex_mem_busB<=32'd0;
    ex_mem_regwraddress<=32'd0;
    end
    else begin
    ex_mem_memwr<=memwr_out;
    ex_mem_memread<=memread_out;
    ex_mem_memtoreg<=memtoreg_out;
    ex_mem_regwrite_out<=regwr_out;

    ex_mem_pc_plus_4_out<=id_pc4;
    ex_mem_aluout<=ex_mem_aluout_tmp;
    ex_mem_busB<=forwardBdata;
    ex_mem_regwraddress<=towhichreg;

    end
```

```
    end


endmodule
// OK 7.1
```

（9）MEM.v

```
module MEM(
input reset,
input clk,

input ex_mem_memwr,
input ex_mem_memread,
input [1:0] ex_mem_memtoreg,
input ex_mem_regwrite,

input [31 : 0] ex_mem_pc_plus_4,
input [31 : 0] ex_mem_aluout,
input [31 : 0] ex_mem_busB,
input [4 : 0] ex_mem_regwraddress,

output reg[1:0] mem_wb_memtoreg_out,
output    reg mem_wb_regwrite_out,

output reg[31 : 0] mem_wb_pc_plus_4_out,
output reg[31 : 0] mem_wb_aluout_out,
output reg[4 : 0] mem_wb_regwraddress_out,
output reg[31:0] data_read_from_mem,
output [11:0] leds //汇编指令使用 sw 向特定地址存入 LED 的控制信号，然后我们直接取出来
用
    );

    wire [31:0] readfrommem;
  DataMemory DM(
        .reset(reset)       ,
        .clk(clk)           ,
        .MemRead(ex_mem_memread)    ,
        .MemWrite(ex_mem_memwr) ,
        .Address(ex_mem_aluout)       ,
```

```
        . Write_data(ex_mem_busB) ,
        .Read_data(readfrommem),
        .leds(leds)
    );

    always @(posedge clk or posedge reset)begin
    if(reset)begin
    mem_wb_memtoreg_out<=2'b00;
    mem_wb_regwrite_out<=1'b0;
   mem_wb_pc_plus_4_out<=32'd0;
   mem_wb_aluout_out<=32'd0;
   mem_wb_regwraddress_out<=32'd0;
   data_read_from_mem<=32'd0;
     end
     else begin
       mem_wb_memtoreg_out<=ex_mem_memtoreg;
        mem_wb_regwrite_out<=ex_mem_regwrite;
        mem_wb_pc_plus_4_out<=ex_mem_pc_plus_4;
        mem_wb_aluout_out<=ex_mem_aluout;
        mem_wb_regwraddress_out<=ex_mem_regwraddress;
        data_read_from_mem<=readfrommem;

    end
    end



    // OK 7.1




endmodule
```

（10）WB.v

```
`timescale 1ns / 1ps

module WB(
// MEM  写入 MEM/WB 寄存器里的值
input[1:0] mem_wb_memtoreg_out,
input[31 : 0] mem_wb_pc_plus_4_out,
input[31 : 0] mem_wb_aluout_out,
input[31:0] data_read_from_mem,
```

```
// choose 1 from these 3
output [31:0]write_to_reg_data
    );

    assign write_to_reg_data=(mem_wb_memtoreg_out==2'b00)?mem_wb_aluout_out:

(mem_wb_memtoreg_out==2'b01)?data_read_from_mem:mem_wb_pc_plus_4_out;

endmodule
// OK 7.1
```

五、仿真结果及分析

为 MYCPU 模块编写 testbench 文件，对处理器进行前仿。

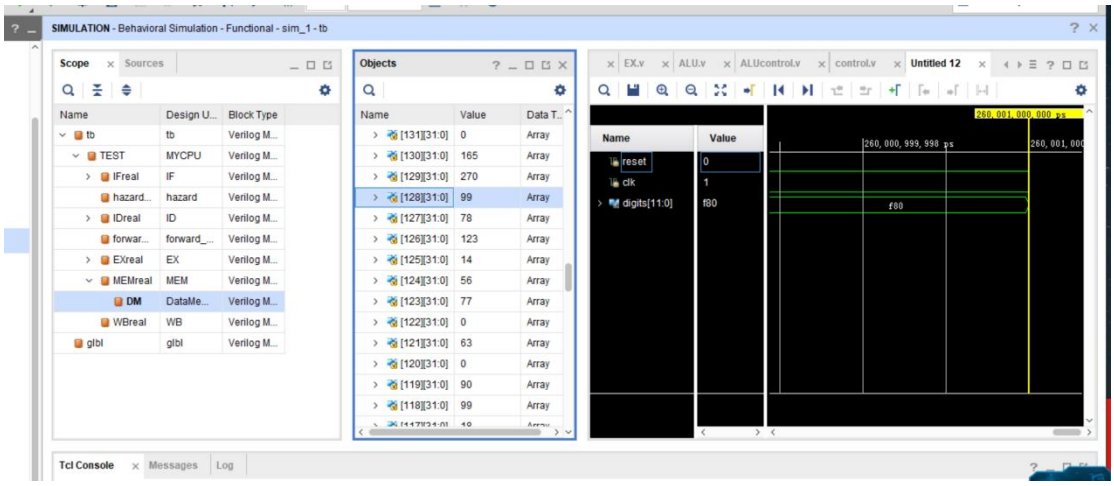


图 11 仿真结果

如图 11 所示，处理器正确计算出 C 矩阵的各个元素，并将元素值存到了 DataMemory 中。

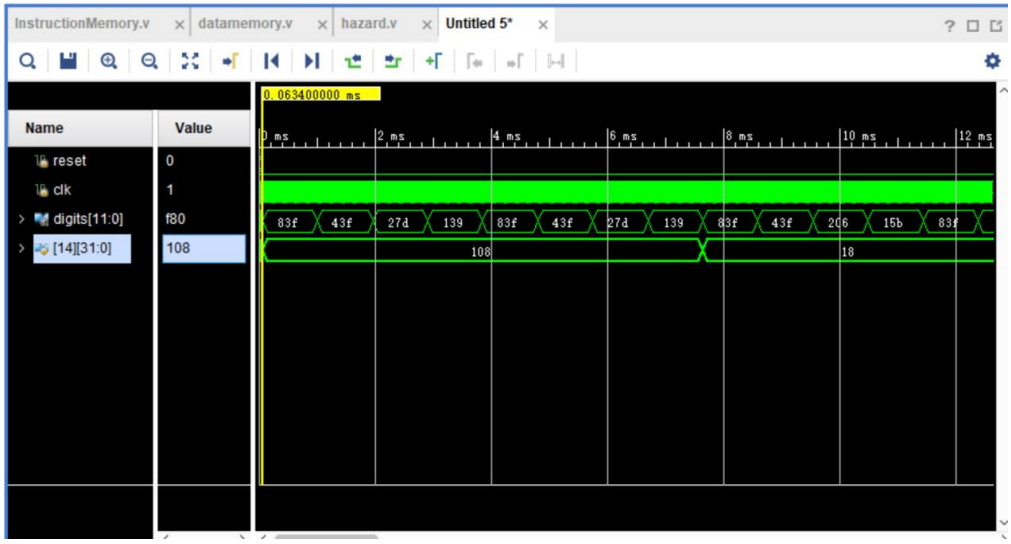


图 12 仿真结果

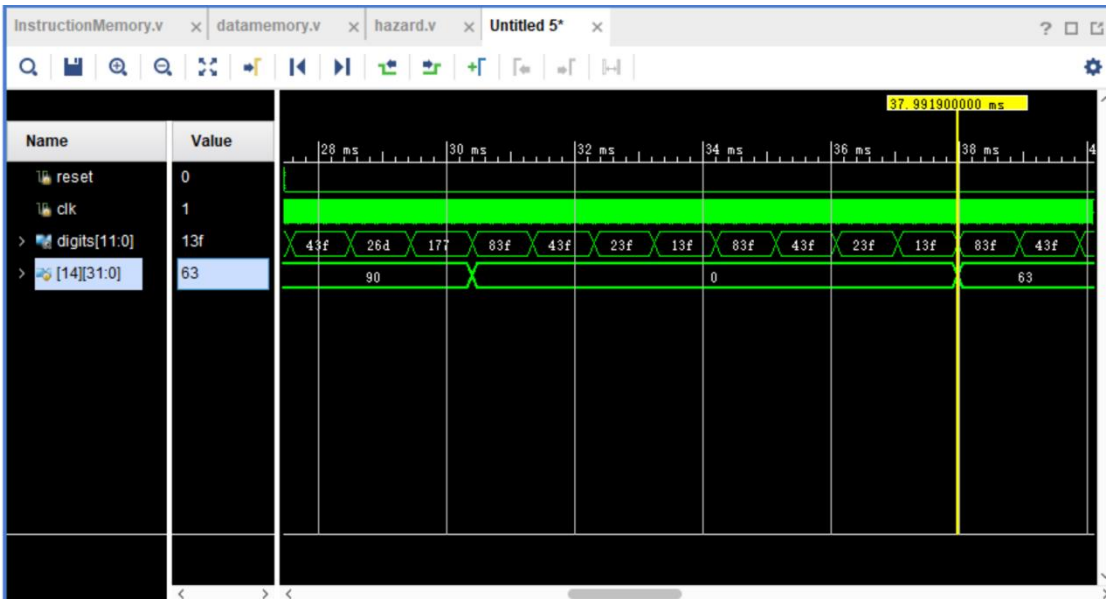


图 13 仿真结果

如图 12 和图 13，BCD 控制代码部分正确地逐个读取 C 矩阵中的元素值，并且数码管的 12 位控制信号 digits 正确地、逐个地显示 4 个十六进制数字。需要说明的是，这里为了仿真的便捷，我令每个元素只显示 8ms，避免仿真时间太长。可以看到仿真结果正确，之后我便将代码烧录到板子上进行实际测试。

六、综合情况

（一）时序性能

（1）时钟频率（时钟周期）

将 IP 核的输出时钟频率设定在 55MHz，即时钟周期为 18.18ns。Design Timing Summary 如下：

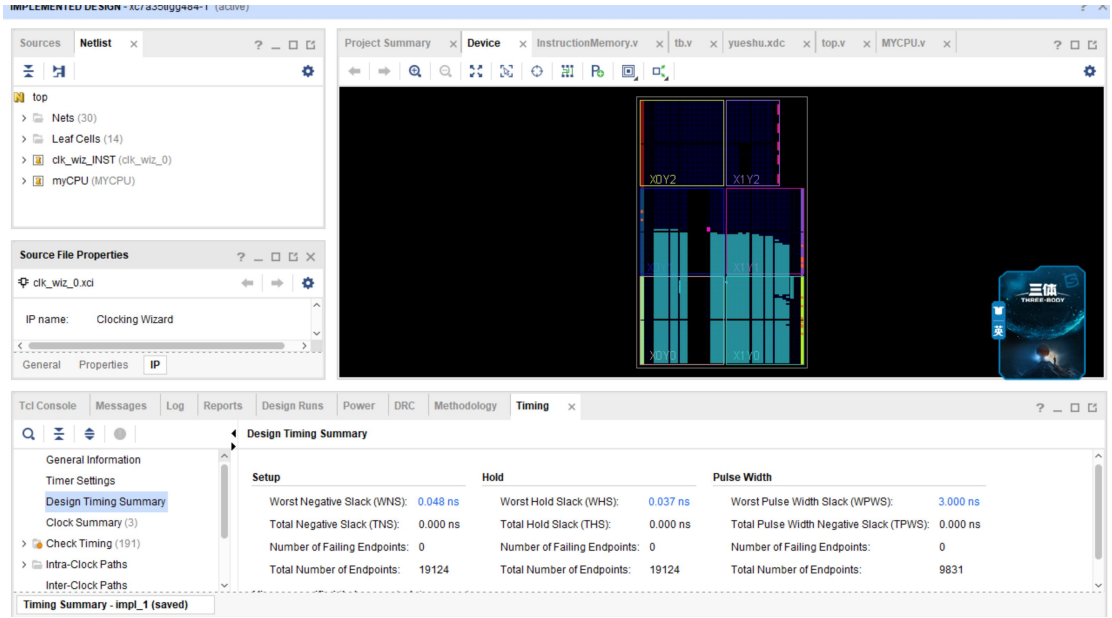


图 14

因此**最短时钟周期为 18.18-0.048=18.132ns，最高时钟频率为 55.15MHz。**

因此需要使用 IP 核对系统时钟进行分频，分出 55MHz 的频率，令处理器工作在 55MHz

的频率下。

（2）CPI

由于最终在处理器上运行的代码是直接从 DataMemory 中读取数据，但这在 Mars 中无法实现，因此我仍令 Mars 中的汇编代码使用 syscall 从文件中读取数据。不过由于 verilog 中运行的指令不包括从文件中读取矩阵数据这一操作，因此计算 CPI 时会将从文件中读取矩阵数据这一部分的指令减去，以保证 verilog 中执行的指令数目和 Mars 统计的指令数目尽可能一致。

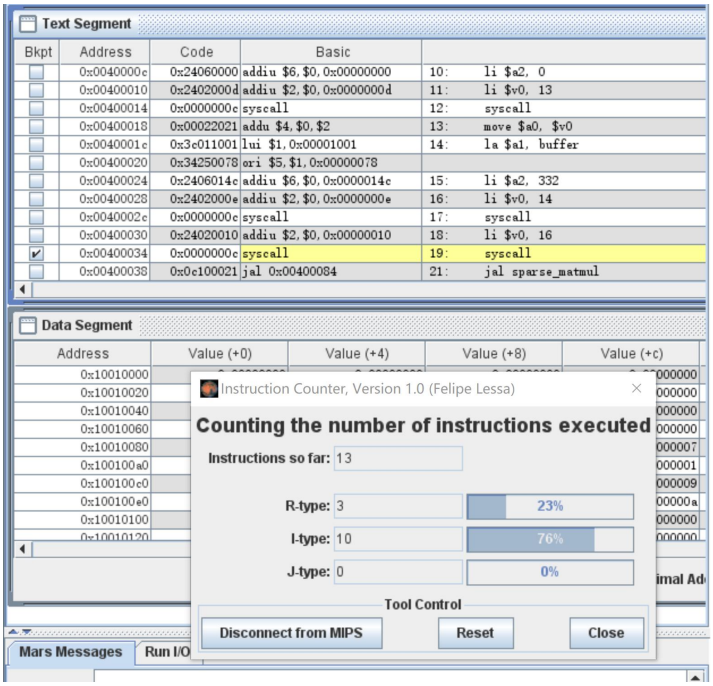


图 15

由图 15，可以看到调用稀疏矩阵计算函数 sparse_matmul 之前，共运行了 13 条指令。

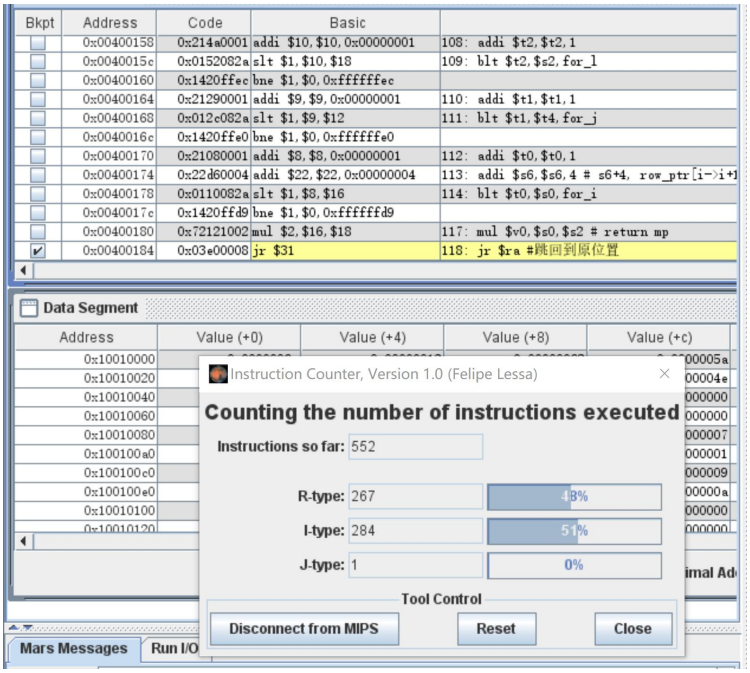


图 16

由图 16，可以看到稀疏矩阵计算函数 sparse_matmul 刚结束时，共运行了 552 条指令。

因此**共运行了 552-13=539 条指令。**

之后，我们计算执行这些指令共用了多少个时钟周期。首先，我在汇编代码执行完 C 矩阵计算后、执行 BCD 控制代码之前，加入了一条指令（如图 17），将$v0 赋值为 1，标志着 C 矩阵计算完成，与之前执行的指令数量保持基本一致，使之误差较小。

```
addi $t2,$t2,1
blt $t2,$s2,for_l
addi $t1,$t1,1
blt $t1,$t4,for_j
addi $t0,$t0,1
addi $s6,$s6,4 # s6+4, row_ptr[i->i+1]
blt $t0,$s0,for_i

li $v0,1

    # 显示结果矩阵C
    addi $t7, $s7, 400 # C矩阵地址
    addi $t8, $zero, 0 # 计数器
```

图 17

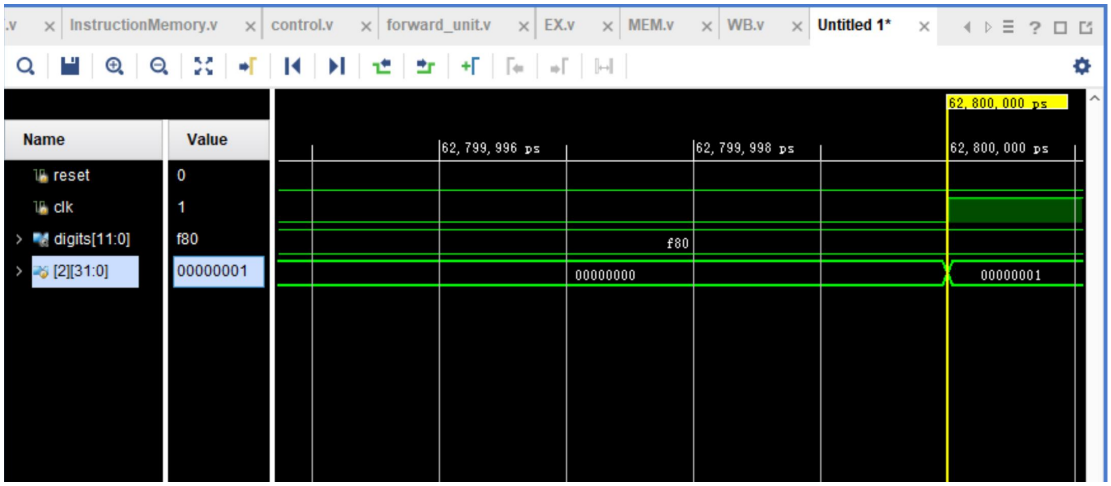


图 18

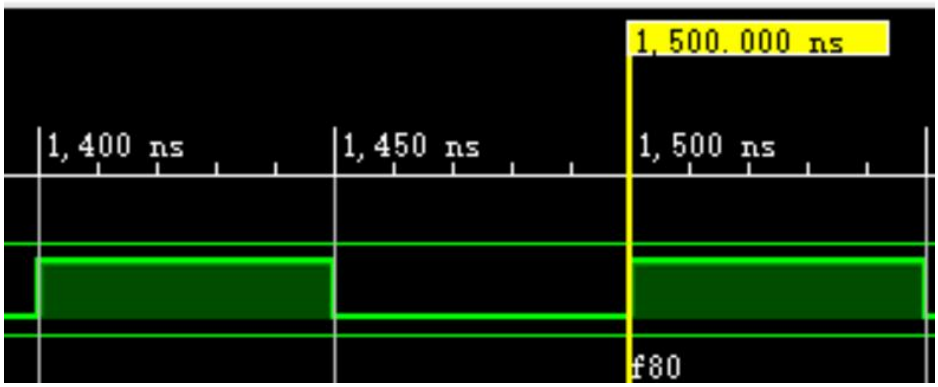


图 19

如图 18 所示，在 62.8us 时，$v0 变为 1，标志 C 矩阵计算结束。因此共用了 62.8us/100ns（仿真中的时钟周期，图 19）=628 个周期。

因此 **CPI=628/539=1.165**。这与流水线处理器的理论 CPI=1 较为接近，超过 1 的那一部分应当主要是由于分支跳转、load-use、jump 导致的 stall 造成的。可以在图 16 中看到 I 型指令占了 51%，而这中间又很多是分支跳转指令，因此就会造成很多 stall。

（二）面积性能

总资源占用情况及占比如下图所示：

Summary

Resource	Utilization	Available	Utilization %
LUT	4341	20800	20.87
FF	9838	41600	23.65
DSP	3	90	3.33
IO	15	250	6.00
PLL	1	5	20.00

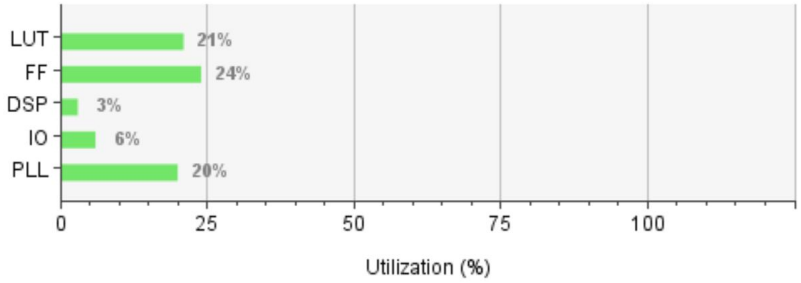


图 20

（1）LUT 占用情况

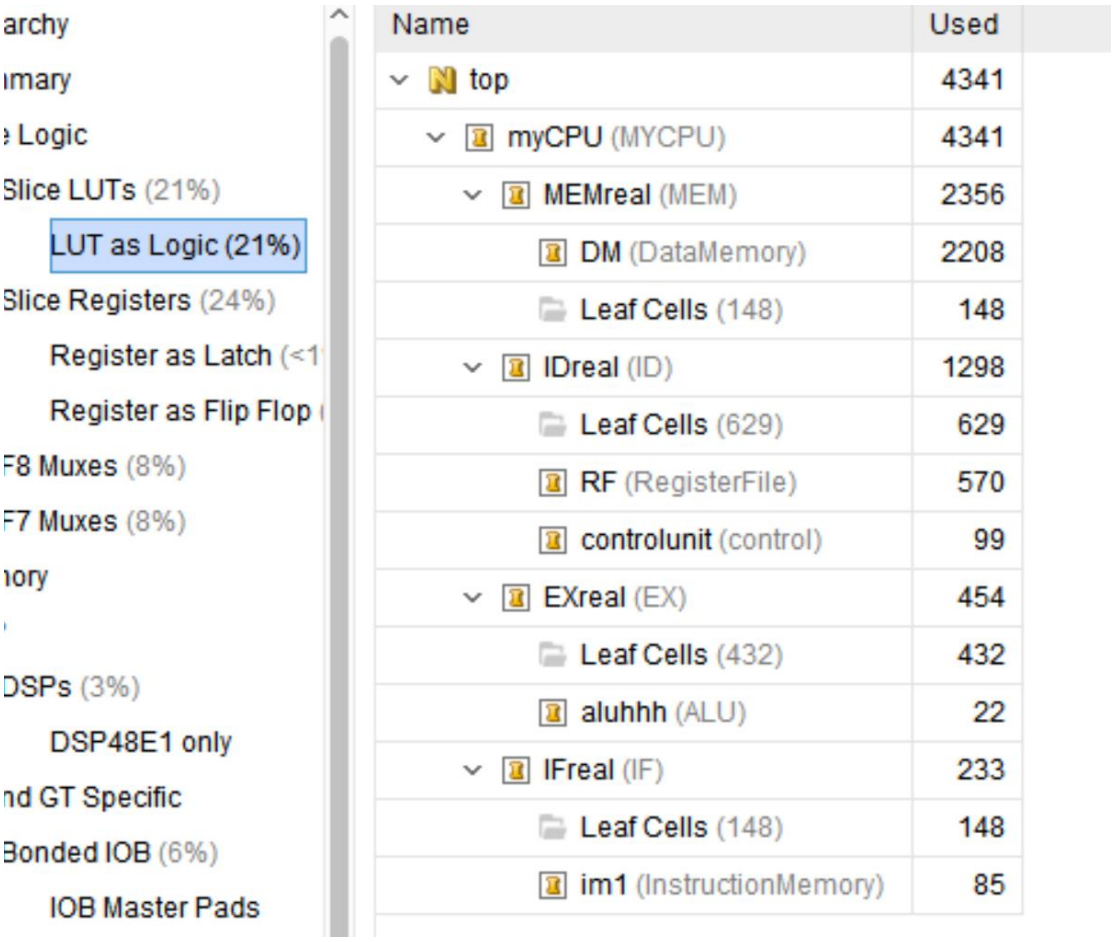


图 21

（2）Register 占用情况

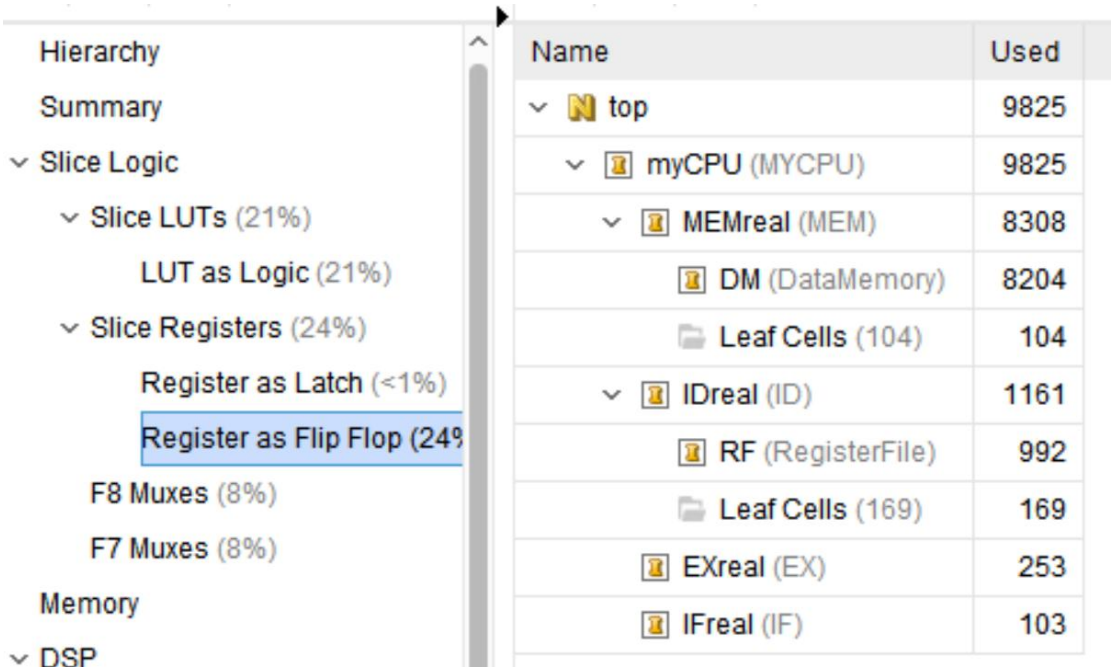


图 22

七、硬件调试情况

在编写完流水线之后，我满怀期待地开始调试工作，希望自己的代码基本没有错误。但是事与愿违，我输入了几个极其简单的 R 型指令，但是仿真中并没有出现正确的结果。我抱着“宁可详细地全部检查一次，也不要查一个试一次，然后查好多次”的想法，将刚刚写完的流水线代码又耐着性子全部仔细地看了一遍，并且每看完一个文件，就在后面标注好“OK”。果然，通过这一次详细的检查，我发现了若干错误。

（1）alusrc1 和 alusrc2 用反了。我在 control 模块将 alusrc1 作为是否使用立即数的标志，将 alusrc2 作为是否使用 shamt 的标志，但是在 EX 阶段实际使用时，却将二者记反了，导致我试验的 I 型指令全部没有跑出正确结果。

（2）sll 指令的目标寄存器设置错误。在 control 模块，我想当然地认为 sll 指令的目标寄存器是 rt，但是认真阅读指导书后发现 sll 是将结果存储到 rd 寄存器之中。

（3）slt 指令的 regwrite 指令误设为 0，应当设为 1

（4）IF 模块中，用于存储下一指令地址的 pc_next 没有写位宽[31:0]。

（5）我额外实现了 bne 指令，但是想当然地认为 bne 的控制信号与 beq 的控制信号应当一样。这导致到了 EX 阶段没有办法区分这是 beq 还是 bne 指令。因此我在 EX 阶段加了补丁，根据 opcode 区分出 beq 和 bne 指令。

检查出这些错误后，我编写了若干简短程序，来检测数据冒险、load-use、beq、跳转等情况，均得到了正确结果。

之后我开始修改汇编代码，使之直接从 DataMemory 中读取数据。这一修改是顺利的，简单修改之后，我便成功跑出了正确地 C 矩阵结果，并将它存储到了 DataMemory 中。

最后我开始编写 BCD 控制代码，但是这一过程却比我想的艰难。一开始第一个元素显示的十分正确，让我误以为能一遍过，但是后面的几个数据却全都显示错误。之后经过多次检查，我认为“没有问题”的代码却始终显示不出正确的结果。于是我最终索性将原先的 BCD 代码删掉，静下心来重新编写了一遍代码，并且这次尽量少使用复杂的循环结构，避免逻辑链条过于复杂。最终终于在仿真中跑出了正确的显示结果。

最后上板子后，又微调了一下每个数字的显示时间，便成功完成了流水线的搭建与上板子显示！

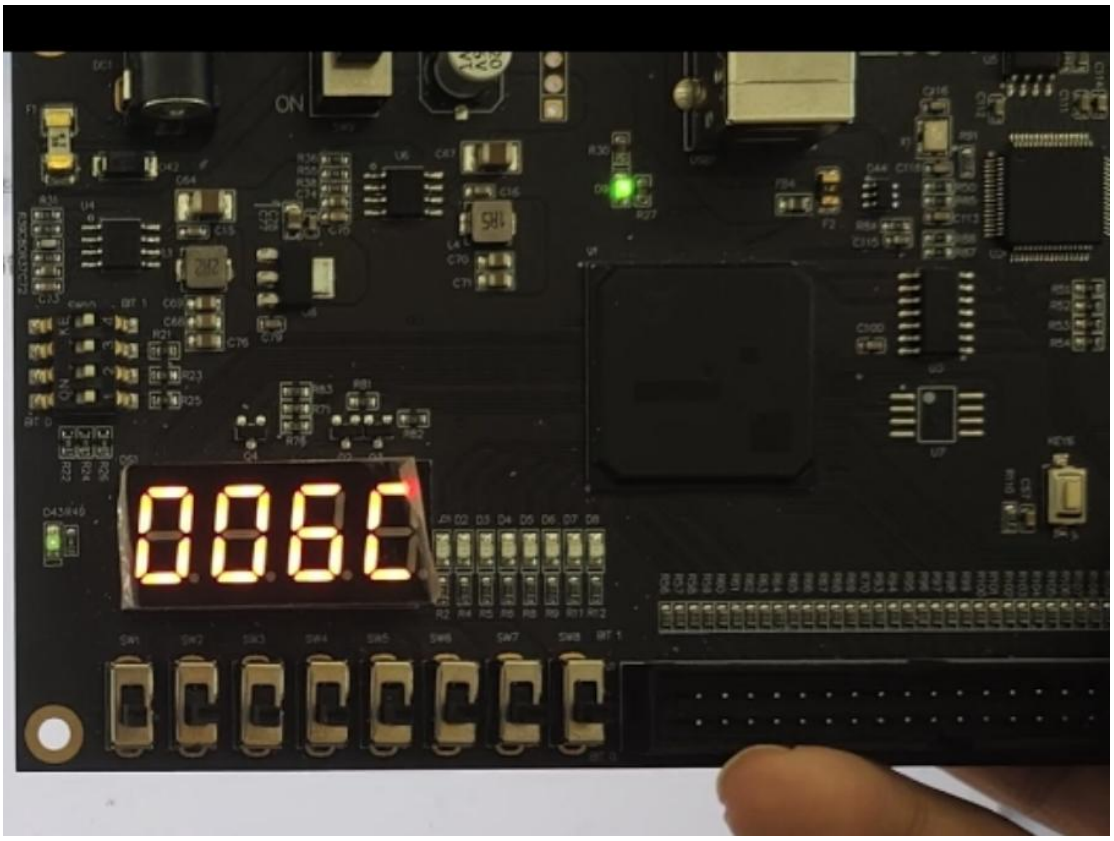


图 23

八、思想体会

（一）

这次的流水线处理器大作业极大地锻炼了我们的能力，令我们将数逻理论课上学习到的知识和数逻实验春季课学习到的知识融会贯通并付诸实践，亲自动手去做一个处理器让我们意识到，除了深厚的理论功底，细致而稳健的动手能力也是我们必不可缺的。

在编写处理器的过程中，我深刻地体会到一个优秀的顶层设计的重要性。如果我们能够先分析好处理器的整体框架，并分析好各个部分之间的关系，做到各部分尽量独立不耦合，带着一个清晰的头脑去完成各部分，那么我们编写的过程一定会是较为顺利的。这启示我们，在以后的工作和科研中，不必急于上手，先冷静地搭好框架能够事半功倍。

此外，我也意识到拥有一个细致的内心的重要性。有许多错误与 bug 完全是有粗心导致，并不是真正具有极高的难度。因此我们应当在学习中保持一个细致的心，避免粗心此类低级错误！

（二）

在最后编写软件译码的代码的过程中，我十分不理解为什么不能像之前一样使用硬件译码直接进行显示。编写完之后，我逐渐理解到，硬件译码确实工作量小，但是却需要消耗更多的硬件资源，并且灵活性不高；而软件译码则不需要额外的硬件资源，并且可以灵活地改变译码方式，不过确实会更复杂一些。这也算是让我们尝试一下软件译码，让我们对硬件译码和软件译码都有所了解吧。

综上，我认为大作业确实总是让人快速成长，这次的数逻大作业更是如此，亲手写出一个流水线处理器后，我对它的理解加深了许多，从这个过程中也收获了许多！也感谢老师和学长学姐对我们的帮助与付出！