



哈尔滨工业大学

微电子科学与技术

课程设计报告

课程名称： 微处理器结构

设计题目： 基于 MIPS 的单周期和流水线微处理器设计

院 系： 航天学院

专 业： 电子信息科学与技术

组内成员： 刘睿智、陈明虎、林博、郭鸿量

指导教师： 李晓明

日 期： 2019 年 12 月

一、概述

1.1 设计题目

采用公用电话传递数据，数据是一组 3 个四位整数，在传递过程中是加密的，加密规则如下：每位数字都加上 5，然后用加法结果循环左移 2 位代替该数字，再将第一位和第四位交换，第二位和第三位交换。将得到的结果存储到新的数组。

1.2 设计要求

按每组指定的设计题目采用 C 语言实现并验证，给出验证结果。

将验证过的 C 程序转化为 MIPS 汇编程序，并转化为机器码。

面向该汇编程序采用的指令集，采用 Verilog 分别实现为单周期处理器、多周期处理器及流水线处理器（至少实现两种处理器）。每种处理器均需实现课件里的 7 条指令。

采用 Verilog 设计上述处理器的验证环境，并在仿真器上进行验证，最终给出验证波形图（要求截屏给出关键输入和输出结果）。

每种处理器的实现要求按照 top-down 的设计方法，进行模块划分。

设计结果要求充分考虑成本、性能及存储器存储空间优化。

1.3 设计成果

将指定题目采用 C 语言实现并验证，采用了利于减小运算量的数据结构。

C 程序转化为了 MIPS 汇编程序，针对指令集成本、性能、存储空间、流水线结构，设计了不同的汇编程序。

编写了 Python 脚本，用于 MIPS 汇编程序转化为机器码。

采用 Verilog 分别实现了单周期处理器和流水线处理器。每种处理器实现了 add、sub、or、and、slt、addi、ori、beq、jump、lw、sw 指令。

采用 Verilog 设计了验证环境，得到验证波形图。

所有代码上传至 Github。详见 https://github.com/Reach666/MIPS_processor

1.3 组内分工

考虑到考研同学忙于复习，给考研同学分配的任务相对较少。

刘睿智：汇编优化、主控制器、单周期 MIPS 总装、单周期 MIPS 测试、冒险检测单元、前递单元、流水线 Jump、MEM/WB 寄存器、流水线 MIPS 测试

陈明虎：单周期取址单元、Python 脚本生成机器码、流水线 IF/ID ID/EX EX/MEM 寄存器、流水线 MIPS 总装、流水线 MIPS 测试

林博：汇编语言（全指令、最简指令集）、指令集确定、寄存器堆、ALU 控制器、流水线 PC

郭鸿量：题目要求分析、C 语言代码及调试、汇编优化、指令集优化、数据/指令存储器、ALU

二、C 软件代码、测试结果及性能分析

2.1 最初方案

```
#include <stdio.h>
void main()
{
    int iden[3][4];                //输入
    int o_iden[3];
    int target[3][4];             //输出
    int u_target[3];
    int j,k;
    for(j=0;j<=2;j++)            //C 语言输入
    {
        scanf("%d",&o_iden[j]);
    }
    for(j=0;j<=2;j++)            //格式转换
    {
        iden[j][3]=o_iden[j]%10;
        iden[j][2]=(o_iden[j]%100)/10;
        iden[j][1]=(o_iden[j]%1000)/100;
        iden[j][0]=o_iden[j]/1000;
    }
    for(j=0;j<=2;j++)            //加密
    {
        target[j][0]=(iden[j][1]+5)%10;
        target[j][1]=(iden[j][0]+5)%10;
        target[j][2]=(iden[j][3]+5)%10;
        target[j][3]=(iden[j][2]+5)%10;
    }
    for(j=0;j<=2;j++)            //格式转换
    {
        u_target[j]=target[j][0]*1000+target[j][1]*100+target[j][2]*10+target[j][3];
    }
    for(j=0;j<=2;j++)            //C 语言输出结果检查
    {
        printf("%d",u_target[j]);
    }
    for(j=0;j<=2000000000;j++);
}
```

该方案汇编实现难度较高（需要额外的除法运算）且由于题目并未要求输入输出的具体形式，故不采用这种方案。

2.2 性能分析及方案优化

数据为一组三个四位整数。考虑到后面需要按位对数据进行操作，亦考虑到汇编语言实现难度，因此将数据按位存入一个二维数组中，便于按位操作。该二维数组为一个 3 行 4 列的数组，每个元素代表数据中的一位。如果输入输出不采用二维数组，

则需要对输入数据进行按位拆分，其算法内部仍然是将输入输出拆分成二维数组的形式，不利于算法的简洁。

每位数字都加上 5，因此该部分可用函数表示为：

$$y[j][i] = (x[j][i] + 5) \bmod 10;$$

用加法结果循环左移 2 位代替该数字，再将第一位和第四位交换，第二位和第三位交换，将得到的结果存储到新的数组。这一部分操作可以简化为为每一行元素的第二个元素和第一个元素调换，第四个元素和第三个元素调换，存到新的数组中。该部分可用函数表示为：

$$y[j][0] = x[j][1];$$

$$y[j][1] = x[j][0];$$

$$y[j][2] = x[j][3];$$

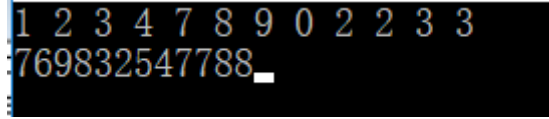
$$y[j][3] = x[j][2];$$

由此编写出 C 语言代码：

```
#include <stdio.h>
void main()
{
    int iden[3][4];           //输入
    int target[3][4];         //输出
    int j,k;
    for(j=0;j<=2;j++)        //C 语言输入
    {
        for(k=0;k<=3;k++)
        {
            scanf("%d",&iden[j][k]);
        }
    }
    for(j=0;j<=2;j++)        //加密
    {
        target[j][0]=(iden[j][1]+5)%10;
        target[j][1]=(iden[j][0]+5)%10;
        target[j][2]=(iden[j][3]+5)%10;
        target[j][3]=(iden[j][2]+5)%10;
    }
    for(j=0;j<=2;j++)        //C 语言输出结果检查
    {
        for(k=0;k<=3;k++)
        {
            printf("%d",target[j][k]);
        }
    }
    for(j=0;j<=2000000000;j++);
}
```

2.3 测试结果

设输入为 1234 7890 2233。经计算，输出应为 7698 3254 7788。在命令窗口中输入，检查结果是否符合。结果如下图。



输出结果准确无误。依此法重复调试多组数据均未出现问题。逐行调试，每行运算结果也与预期符合。

三、汇编代码和机器码

3.1 汇编代码（C 语言直接翻译）

将 C 语言直接翻译，例如将取余%翻译为 divu 和 mfhi，将 for 循环翻译为 sltiu 等，得到以下汇编代码：

	addi \$s4,\$zero,0	;reset \$s4,assign j=0
ForLoop:	sltiu \$t0,\$s4,2	;j<2 ? \$t0=1 : \$t0=0
	beq \$t0,\$zero,Exit	;j=2, goto Exit
	add \$t1,\$s4,\$s4	
	add \$t1,\$t1,\$t1	;j multiple 4
	add \$t1,\$t1,\$s0	;\$t1 address of iden[j][0]
	lw \$t0,0(\$t1)	;\$t0=iden[j][0]
	addi \$s0,\$t0,5	;\$s0=iden[j][0]+5
	addi \$t0,\$zero,10	;assign \$t0=10
	divu \$s0,\$t0	;hi is the remainder
	mfhi \$s0	;(iden[j][0]+5)%10;
	add \$s6,\$s6,\$s0	;target[j][1]=(iden[j][0]+5)%10;
	add \$t1,\$t1,\$s1	;\$t1 address of iden[j][1]
	lw \$t0,0(\$t1)	;\$t0=iden[j][1]
	addi \$s1,\$t0,5	;\$s1=iden[j][1]+5
	addi \$t0,\$zero,10	;assign \$t0=10
	divu \$s1,\$t0	;hi is the remainder
	mfhi \$s1	;(iden[j][1]+5)%10;
	add \$s5,\$s5,\$s0	;target[j][0]=(iden[j][1]+5)%10;
	add \$t1,\$t1,\$s2	;\$t1 address of iden[j][2]
	lw \$t0,0(\$t1)	;\$t0=iden[j][2]
	addi \$s2,\$t0,5	;\$s2=iden[j][2]+5
	addi \$t0,\$zero,10	;assign \$t0=10
	divu \$s2,\$t0	;hi is the remainder
	mfhi \$s2	;(iden[j][2]+5)%10;
	add \$s8,\$s8,\$s0	;target[j][3]=(iden[j][2]+5)%10;
	add \$t1,\$t1,\$s3	;\$t1 address of iden[j][3]
	lw \$t0,0(\$t1)	;\$t0=iden[j][3]
	addi \$s3,\$t0,5	;\$s3=iden[j][3]+5

addi \$t0,\$zero,10	;assign \$t0=10
divu \$s3,\$t0	;hi is the remainder
mfhi \$s3	;(iden[j][3]+5)%10;
add \$s7,\$s7,\$s0	;target[j][0]=(iden[j][3]+5)%10;
addi \$s4,\$s4,1	;j++
j ForLoop	

Exit:

3.2 汇编代码（最简指令集）

上述代码使用到了 divu、mfhi、sltiu 等指令，但 divu 对于 ALU 难以实现，mfhi、sltiu 也会增加处理器的设计难度，可以将他们用最简指令集代替。

免除除法操作的思路：由于程序在数字排列上有着特殊性，可以通过 beq 指令进行比较，如果数字超过十进行减十操作，如果数字小于十，直接保留，避免了在 ALU 中加入除法分支。

由此得到的代码，只用到了 addi、beq、lw、sw、add、j 指令。代码如下：

	addi \$s0,\$zero,0	;\$s0=&iden[0][0]
	addi \$s1,\$zero,1	;\$s1=&iden[0][1]
	addi \$s2,\$zero,2	;\$s2=&iden[0][2]
	addi \$s3,\$zero,3	;\$s3=&iden[0][3]
	addi \$s4,\$zero,0	;reset \$s4,assign j=0
ForLoop:	addi \$t0,\$zero,3	
	beq \$s4,\$t0,Exit	;j=3, goto Exit
	add \$t1,\$s4,\$s4	
	add \$t1,\$t1,\$t1	;j multiple 4
	add \$t2,\$t1,\$s0	;\$t1 address of iden[j][0]
	lw \$t0,0(\$t2)	;\$t0=iden[j][0]
	addi \$t0,\$t0,5	;\$s0=iden[j][0]+5
	addi \$t7, \$zero, 5	
	beq \$t0, \$t7, exitmod	
	addi \$t7, \$zero, 6	
	beq \$t0, \$t7, exitmod	
	addi \$t7, \$zero, 7	
	beq \$t0, \$t7, exitmod	
	addi \$t7, \$zero, 8	
	beq \$t0, \$t7, exitmod	
	addi \$t7, \$zero, 9	
	beq \$t0, \$t7, exitmod	
	addi \$t0, \$t0, -10	
exitmod:	sw \$t0,13(\$t1)	;target[j][1]=(iden[j][0]+5)%10;
	add \$t2,\$t1,\$s1	;\$t1 address of iden[j][1]
	lw \$t0,0(\$t2)	;\$t0=iden[j][1]
	addi \$t0,\$t0,5	;\$s1=iden[j][1]+5
	addi \$t7, \$zero, 5	

	beq \$t0, \$t7, exitmod2	
	addi \$t7, \$zero, 6	
	beq \$t0, \$t7, exitmod2	
	addi \$t7, \$zero, 7	
	beq \$t0, \$t7, exitmod2	
	addi \$t7, \$zero, 8	
	beq \$t0, \$t7, exitmod2	
	addi \$t7, \$zero, 9	
	beq \$t0, \$t7, exitmod2	
	addi \$t0, \$t0, -10	;(iden[j][1]+5)%10;
exitmode2:	sw \$t0,12(\$t1)	; target[j][0]=(iden[j][1]+5)%10;
	 add \$t2,\$t1,\$s2	;\$t1 address of iden[j][2]
	lw \$t0,0(\$t2)	;\$t0=iden[j][2]
	addi \$t0,\$t0,5	;\$s2=iden[j][2]+5
	addi \$t7, \$zero, 5	
	beq \$t0, \$t7, exitmod3	
	addi \$t7, \$zero, 6	
	beq \$t0, \$t7, exitmod3	
	addi \$t7, \$zero, 7	
	beq \$t0, \$t7, exitmod3	
	addi \$t7, \$zero, 8	
	beq \$t0, \$t7, exitmod3	
	addi \$t7, \$zero, 9	
	beq \$t0, \$t7, exitmod3	
	addi \$t0, \$t0, -10	;(iden[j][2]+5)%10;
exitmod3:	sw \$t0,15(\$t1)	; target[j][3]=(iden[j][2]+5)%10
	 add \$t2,\$t1,\$s3	;\$t1 address of iden[j][3]
	lw \$t0,0(\$t2)	;\$t0=iden[j][3]
	addi \$t0,\$t0,5	;\$s1=iden[j][3]+5
	addi \$t7, \$zero, 5	
	beq \$t0, \$t7, exitmod4	
	addi \$t7, \$zero, 6	
	beq \$t0, \$t7, exitmod4	
	addi \$t7, \$zero, 7	
	beq \$t0, \$t7, exitmod4	
	addi \$t7, \$zero, 8	
	beq \$t0, \$t7, exitmod4	
	addi \$t7, \$zero, 9	
	beq \$t0, \$t7, exitmod4	
	addi \$t0, \$t0, -10	;(iden[j][3]+5)%10;
exitmod4:	sw \$t0,14(\$t1)	; target[j][0]=(iden[j][3]+5)%10;
	addi \$s4,\$s4,1	;j++
	j ForLoop	
Exit:		

3.3 汇编代码（用于所设计的流水线处理器、减小占用空间）

在流水线处理器中，为了解决 beq 控制冒险问题，我们决定在 beq 指令后加入空指令，对应代码中的 add \$zero,\$zero,\$zero。为了减小占用空间，我们将程序改为二级循环，这样做的代价是增加了几句 lw、sw 用于交换数据位置，增加了程序运行时间。同时，由于处理器实现了 slt 指令，我们将大量 beq 改回了 slt。代码如下所示，去掉空指令也是单周期处理器的汇编代码。

```

                                addi $s0,$zero,0           ;$s0=&iden[0][0]
                                addi $s1,$zero,1           ;$s1=&iden[0][1]
                                addi $s2,$zero,2           ;$s2=&iden[0][2]
                                addi $s3,$zero,3           ;$s3=&iden[0][3]
                                addi $s4,$zero,0           ;reset $s4,assign j=0
ForLoop:  addi $t0,$zero,3
                                beq $s4,$t0,Exit           ;j=3, goto Exit
                                add $zero,$zero,$zero
                                add $zero,$zero,$zero
                                add $zero,$zero,$zero
                                add $t4,$s4,$s4
                                add $t4,$t4,$t4           ;j multiple 4
                                addi $s0,$zero,0           ;reset $s0,assign i=0
forloop:  addi $t0,$zero,4
                                beq $s0,$t0,exit           ;i=4, goto Exit 38h
                                add $zero,$zero,$zero
                                add $zero,$zero,$zero
                                add $zero,$zero,$zero
                                add $t2,$t4,$s0           ;$t1 address of iden[j][0]
                                lw $t0,0($t2)              ;$t0=iden[j][0]h4c
                                addi $t0,$t0,5             ;$s0=iden[j][0]+5
                                addi $t7,$zero,9
                                slt $t6,$t7,$t0
                                beq $t6,$zero,exitmod
                                add $zero,$zero,$zero
                                add $zero,$zero,$zero
                                add $zero,$zero,$zero
                                addi $t0,$t0,-10
exitmod:  sw $t0,12($t2)           ; target
                                addi $s0,$s0,1             ;i++
                                j forloop
exit:     lw $t0,12($t4)
                                lw $t1,13($t4)
                                lw $t2,14($t4)
                                lw $t3,15($t4)
                                sw $t0,13($t4)
                                sw $t1,12($t4)
                                sw $t2,15($t4)
                                sw $t3,14($t4)
```



```

addi $s4,$s4,1          ;j++
j ForLoop

```

Exit:

3.3 Python 脚本

由于将汇编转换机器码的过程繁琐复杂，而且考虑到未来可能会有多个修改版本的汇编代码，每次都进行人工转换工程量巨大，于是计划写一个 python 脚本来对其进行转换。该脚本支持的指令有 addi,add,slt,lw,sw,beq,j。能够识别 0-15 的立即数，能够识别 label 并保留，并将 beq 的跳转 label 放在注释后以便人工填写地址。以下是编写的 python 脚本的代码：

```

###editor: cmh
###function: change assembly instruction into machine instruction
###支持的指令有 addi,add,slt,lw,sw,beq,j
###支持所有所有 zero,t,s 寄存器
###支持 0-15 的立即数
###Label 不要和指令放在同一行
###将此文件和 mips.txt 放在同一目录下
source_file='mips3.txt' #源汇编代码文件名
dir_file='mips3_1.txt'  #产生的目标文件名
code = 'utf-8'         #需要在此输入源汇编代码文件的编码方式

f=open(source_file,encoding=code)
ins = f.readlines()
fl=open(dir_file,'w')

def w_op(i,index_n):
    global op
    op = ""
    if 'addi' in i[0:index_n]:
        op = '001000 '
    if 'beq' in i[0:index_n]:
        op = '000100 '
    if 'add' in i[0:index_n] and \
        'addi' not in i[0:index_n]:
        op = '000000 '
    if 'lw' in i[0:index_n]:
        op = '100011 '
    if 'j' in i[0:index_n]:
        op = '000010 '
    if 'slt' in i[0:index_n]:
        op = '000000 '
    if 'sw' in i[0:index_n]:
        op = '101011 '
    return op
def w_func(i,fir_comma):
    global func

```

```

func="
if 'add' in i[0:fir_comma] and \
    'addi' not in i[0:fir_comma]:
    func = '100000\n'
if 'slt' in i[0:fir_comma]:
    func = '101010\n'
return func
def w_r(i,fir_index,sec_index):
    #global rt
    rt = "
    if 'zero' in i[fir_index:sec_index]:
        rt = '00000 '
    if 't0' in i[fir_index:sec_index]:
        rt = '01000 '
    if 't1' in i[fir_index:sec_index]:
        rt = '01001 '
    if 't2' in i[fir_index:sec_index]:
        rt = '01010 '
    if 't3' in i[fir_index:sec_index]:
        rt = '01011 '
    if 't4' in i[fir_index:sec_index]:
        rt = '01100 '
    if 't5' in i[fir_index:sec_index]:
        rt = '01101 '
    if 't6' in i[fir_index:sec_index]:
        rt = '01110 '
    if 't7' in i[fir_index:sec_index]:
        rt = '01111 '
    if 's0' in i[fir_index:sec_index]:
        rt = '10000 '
    if 's1' in i[fir_index:sec_index]:
        rt = '10001 '
    if 's2' in i[fir_index:sec_index]:
        rt = '10010 '
    if 's3' in i[fir_index:sec_index]:
        rt = '10011 '
    if 's4' in i[fir_index:sec_index]:
        rt = '10100 '
    if 's5' in i[fir_index:sec_index]:
        rt = '10101 '
    if 's6' in i[fir_index:sec_index]:
        rt = '10110 '
    if 's7' in i[fir_index:sec_index]:
        rt = '10111 '
    if 't8' in i[fir_index:sec_index]:
        rt = '11000 '

```

```

    if 't9' in i[fir_index:sec_index]:
        rt = '11001 '
    return rt
def w_imm(i,sec_comma,index_n):
    global imm
    if '0' in i[sec_comma:index_n]:
        imm = '0000000000000000\n'
    if '1' in i[sec_comma:index_n]:
        imm = '0000000000000001\n'
    if '2' in i[sec_comma:index_n]:
        imm = '0000000000000010\n'
    if '3' in i[sec_comma:index_n]:
        imm = '0000000000000011\n'
    if '4' in i[sec_comma:index_n]:
        imm = '0000000000000100\n'
    if '5' in i[sec_comma:index_n]:
        imm = '0000000000000101\n'
    if '6' in i[sec_comma:index_n]:
        imm = '0000000000000110\n'
    if '7' in i[sec_comma:index_n]:
        imm = '0000000000000111\n'
    if '8' in i[sec_comma:index_n]:
        imm = '0000000000001000\n'
    if '9' in i[sec_comma:index_n]:
        imm = '0000000000001001\n'
    if '10' in i[sec_comma:index_n]:
        imm = '0000000000001010\n'
    if '11' in i[sec_comma:index_n]:
        imm = '0000000000001011\n'
    if '12' in i[sec_comma:index_n]:
        imm = '0000000000001100\n'
    if '13' in i[sec_comma:index_n]:
        imm = '0000000000001101\n'
    if '14' in i[sec_comma:index_n]:
        imm = '0000000000001110\n'
    if '15' in i[sec_comma:index_n]:
        imm = '0000000000001111\n'
    if '-' in i[sec_comma:index_n]:
        imm = '1'+imm[1:]
    if 'exitmod2' in i[sec_comma:index_n]:
        imm = 'xxxxxxxxxxxxxxxx //exitmod2_addr\n'
    if 'exitmod3' in i[sec_comma:index_n]:
        imm = 'xxxxxxxxxxxxxxxx //exitmod3_addr\n'
    if 'exitmod' in i[sec_comma:index_n]:
        imm = 'xxxxxxxxxxxxxxxx //exitmod4_addr\n'
    if 'Exit' in i[sec_comma:index_n]:

```

```

        imm = 'xxxxxxxxxxxxxxxxxx //Exit_addr\n'
    if 'exit' in i[sec_comma:index_n] and \
        'exitmod' not in i[sec_comma:index_n]:
        imm = 'xxxxxxxxxxxxxxxxxx //exit_addr\n'
    return imm
for i in ins:
    if i == '\n':
        fl.write('\n')
    else:
        fir_comma = i.find(',')
        sec_comma = i.find(',',fir_comma+1)
        if ';' in i:
            index_n = i.find(';')
        else:
            index_n = i.find('\n')
        if ':' in i:
            fl.write(i[0:index_n]+'\\n')
        else:
            op = w_op(i,index_n)
            rs = w_r(i,fir_comma,sec_comma)
            rd = w_r(i,0,fir_comma)
            rt = w_r(i,sec_comma,index_n)
            imm = w_imm(i,sec_comma,index_n)
            fl.write(op)
            if 'j' not in i[0:index_n]:
                if op == '000000 ':
                    fl.write(rs)
                    fl.write(rt)
                    fl.write(rd)
                    fl.write('00000 ')
                    func = w_func(i,fir_comma)
                    fl.write(func)
                elif op == '100011 ' or op == '101011 ':
                    index_brac = i.find('(')
                    rs = w_r(i,index_brac,index_n)
                    rd = w_r(i,0,fir_comma)
                    imm = w_imm(i,fir_comma,index_brac)
                    fl.write(rs)
                    fl.write(rd)
                    fl.write(imm)
            else:
                fl.write(rs)
                fl.write(rd)
                fl.write(imm)
        elif 'j' in i[0:index_n]:
            if 'f' in i[0:index_n]:

```

```

        fl.write('xxxxxxxxxxxxxxxxxxxxxxxxxxxxx //forloop_addr\n')
    elif 'F' in i[0:index_n]:
        fl.write('xxxxxxxxxxxxxxxxxxxxxxxxxxxxx //Forloop_addr\n')
    else:
        fl.write('xxxxxxxxxxxxxxxxxxxxxxxxxxxxx //???_addr\n')

fl.close()

```

3.4 机器码

由 Python 脚本转换得到的流水线处理器机器码如下所示，其中的 000000000000 0000000000000100000 代表空指令。如果去掉所有的空指令，得到的就是单周期处理器机器码。

```

00100000000100000000000000000000//0x0
00100000000100010000000000000001//0x4
00100000000100100000000000000010//0x8
00100000000100110000000000000011//0xc
00100000000101000000000000000000//0x10
00100000000010000000000000000011//0x14
00010001000101000000000000100010//Exit_addr//0x18//+34
0000000000000000000000000100000//0x1c
0000000000000000000000000100000//0x20
0000000000000000000000000100000//0x24
00000010100101000110000000100000//0x28
00000001100011000110000000100000//0x2c
00100000000100000000000000000000//0x30
00100000000010000000000000000100//0x34
0001000100010000000000000010000//0x38//+16
0000000000000000000000000100000//0x3c
0000000000000000000000000100000//0x40
0000000000000000000000000100000//0x44
00000001100100000101000000100000//0x48
10001101010010000000000000000000//0x4c
00100001000010000000000000000101//0x50
0010000000001111000000000001001//0x54
00000001111010000111000000101010//0x58
000100000000111000000000000100//exitmod_addr//0x5c//+4
0000000000000000000000000100000//0x60
0000000000000000000000000100000//0x64
0000000000000000000000000100000//0x68
001000010000100011111111110110//0x6c-10
1010110101001000000000000001100//0x70
00100010000100000000000000000001//0x74
0000100000000000000000000001101//forloop_addr//0x78//13
1000110110001000000000000001100//0x7c
1000110110001001000000000001101//0x80
1000110110001010000000000001110//0x84
1000110110001011000000000001111//0x88

```

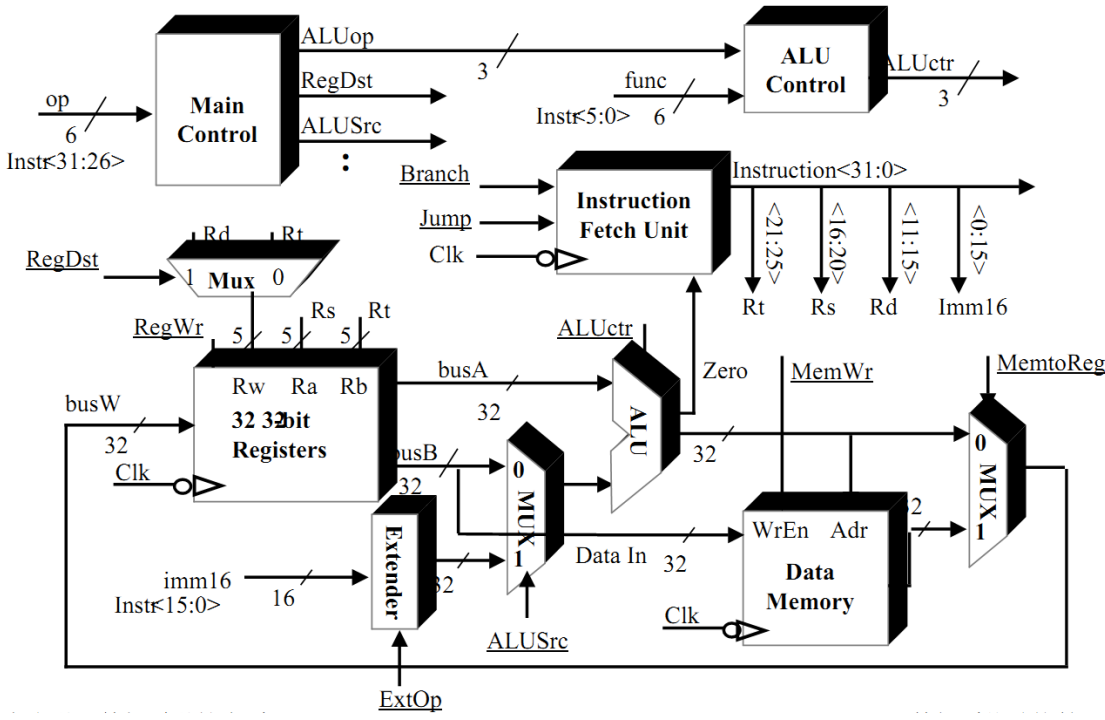
```
1010110110001000000000000000001101//0x8c
1010110110001001000000000000001100//0x90
1010110110001010000000000000001111//0x94
1010110110001011000000000000001110//0x98
001000101001010000000000000000001//0x9c
00001000000000000000000000000000101//Forloop_addr//0xa0//5
00000000000000000000000000000000000//Exit://0xa4
00000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
```

四、指令集描述

分类	指令	示例	含义
R-Type	add	add \$s1, \$s2, \$s3	\$s1 = \$s2 + \$s3
	slt	slt \$s1, \$s2, \$s3	\$s1 = \$s2 < \$s3 ? 1:0
	sub(可选)	sub \$s1, \$s2, \$s3	\$s1 = \$s2 - \$s3
	or(可选)	sub \$s1, \$s2, \$s3	\$s1 = \$s2 \$s3
	and(可选)	sub \$s1, \$s2, \$s3	\$s1 = \$s2 & \$s3
I-Type	lw	lw \$s1, 100(\$s2)	\$s1 = Mem[\$s2 + 100]
	sw	sw \$s1, 100(\$s2)	Mem[\$s2 + 100] = \$s1
	beq	beq \$s1, \$s2, L	If (\$s1 == \$s2) PC+=4*L
	addi	addi \$s1, \$s2, L	\$s1 = \$s2 + sign_ex(L)
	ori(可选)	ori \$s1, \$s2, L	\$s1 = \$s2 or ex(L)
J-Type	j	j 2500	goto 10000

五、模块示意图及端口描述；验证平台描述

5.1 单周期处理器模块图



结果显示：使用了两次\$display，方便比较输入和处理器的运算结果。

Verilog 代码如下所示：

```
`timescale 10ns/1ns
//`include "MIPS_top.v"
module MIPS_test;
reg clk,reset;

MIPS_top MIPS_top(clk,reset);

initial
begin
    clk=0;
    reset=0;
end

initial
begin
    #5 reset=1;
end

initial
begin
    repeat(1000)
        #2 clk=~clk;
end

initial
begin
    #1
    $readmemb("mips3.txt",MIPS_top.ins_fetch.Ins_Mem.data);
    // MIPS_top.ins_fetch.Ins_Mem.data[0]=32'b001000010000100000000000000000001;
    //32'b001000 01000 01000 0000000000000000001
    // MIPS_top.ins_fetch.Ins_Mem.data[1]=32'b001000010000100000000000000000001;
    // MIPS_top.ins_fetch.Ins_Mem.data[2]=32'b001000010000100000000000000000001;
    // MIPS_top.ins_fetch.Ins_Mem.data[3]=32'b001000010000100000000000000000001;
    // MIPS_top.ins_fetch.Ins_Mem.data[4]=32'b001000010000100000000000000000001;
    // MIPS_top.ins_fetch.Ins_Mem.data[5]=32'b001000010000100000000000000000001;
    // MIPS_top.ins_fetch.Ins_Mem.data[6]=32'b001000010000100000000000000000001;
    // MIPS_top.ins_fetch.Ins_Mem.data[7]=32'b001000010000100000000000000000001;
    // MIPS_top.ins_fetch.Ins_Mem.data[8]=32'b001000010000100000000000000000001;
    // MIPS_top.ins_fetch.Ins_Mem.data[9]=32'b001000010000100000000000000000001;
    // MIPS_top.ins_fetch.Ins_Mem.data[10]=32'b001000010000100000000000000000001;
    // MIPS_top.ins_fetch.Ins_Mem.data[11]=32'b001000010000100000000000000000001;
end

initial
```



```

begin
    #1
    MIPS_top.Data_Mem.data[0]=32'h1;//5'b00000
    MIPS_top.Data_Mem.data[1]=32'h3;
    MIPS_top.Data_Mem.data[2]=32'h5;
    MIPS_top.Data_Mem.data[3]=32'h7;
    MIPS_top.Data_Mem.data[4]=32'h9;
    MIPS_top.Data_Mem.data[5]=32'h2;
    MIPS_top.Data_Mem.data[6]=32'h4;
    MIPS_top.Data_Mem.data[7]=32'h6;
    MIPS_top.Data_Mem.data[8]=32'h8;//5'b01000
    MIPS_top.Data_Mem.data[9]=32'h0;//5'b01001
    MIPS_top.Data_Mem.data[10]=32'h0;
    MIPS_top.Data_Mem.data[11]=32'h0;
end

initial
begin
    #10
    $display("Input  %1d%1d%1d%1d %1d%1d%1d%1d %1d%1d%1d%1d",MIPS_top.Data_M
em.data[0],MIPS_top.Data_Mem.data[1],
    MIPS_top.Data_Mem.data[2],MIPS_top.Data_Mem.data[3],MIPS_top.Data_Mem.data[4],MIPS_t
op.Data_Mem.data[5],
    MIPS_top.Data_Mem.data[6],MIPS_top.Data_Mem.data[7],MIPS_top.Data_Mem.data[8],MIPS_t
op.Data_Mem.data[9],
    MIPS_top.Data_Mem.data[10],MIPS_top.Data_Mem.data[11]);
    #1000
    $display("Output %1d%1d%1d%1d %1d%1d%1d%1d %1d%1d%1d%1d",MIPS_top.Data_M
em.data[12],MIPS_top.Data_Mem.data[13],
    MIPS_top.Data_Mem.data[14],MIPS_top.Data_Mem.data[15],MIPS_top.Data_Mem.data[16],MIP
S_top.Data_Mem.data[17],
    MIPS_top.Data_Mem.data[18],MIPS_top.Data_Mem.data[19],MIPS_top.Data_Mem.data[20],MIP
S_top.Data_Mem.data[21],
    MIPS_top.Data_Mem.data[22],MIPS_top.Data_Mem.data[23]);
end
endmodule

```

流水线处理器与单周期处理器验证代码基本相同，只是时钟频率和读取文件略有改变。

六、硬件模块描述及源代码

6.1 单周期处理器

6.1.1 ALU

由于我们的汇编里使用了 `add`,`addi`,`slt`,`lw`,`sw`,`beq`,`jump` 等指令，所以 ALU 中实现的

运算有加，减，与，或，以及比较大小五种运算。根据编码规则对其控制信号进行编码得到这四种运算的控制信号分别为。与：000，或：001，加：010，减：110，比较：111。在 Verilog 中使用 case 语句对这四种运算进行控制。并且在减法指令时，如果结果为 0 需要让输出信号 zero 为 1。ALU 的端口有：3 位的输入控制端 ALUctr，两个 32 位的数据输入端 busA，busB，一个 32 位的结果输出 busC，一个 zero 标志位输出。其 Verilog 如下：

```
module ALU(busC,zero,busA,busB,ALUctr);
```

```
input busA,busB,ALUctr;
```

```
output zero,busC;
```

```
reg [31:0] busC;
```

```
reg zero;
```

```
wire [31:0] busA, busB;
```

```
wire [2:0] ALUctr;
```

```
parameter myAND = 3'b000;
```

```
parameter myOR  = 3'b001;
```

```
parameter myADD = 3'b010;
```

```
parameter mySUB = 3'b110;
```

```
parameter myLESS= 3'b111;
```

```
always@(*)
```

```
begin
```

```
zero = 0;
```

```
case(ALUctr)
```

```
    myAND : busC = busA & busB;
```

```
    myOR  : busC = busA | busB;
```

```
    myADD : busC = busA + busB;
```

```
    mySUB : begin
```

```
        busC = busA - busB;
```

```
        if(busC == 0)
```

```
            zero = 1;
```

```
        end
```

```
    myLESS: begin
```

```
        if(busA<busB)
```

```
            busC = 1;
```

```
        else
```

```
            busC = 0;
```

```
        end
```

```
endcase
```

```
end
```

```
endmodule
```

6.1.2 位扩展器

对于位数扩展器，该扩展器用于 I 类指令，需要有两种扩展方式。一种是符号扩展，一种是 0 扩展。由控制信号 ExtOp 决定。当 ExtOp 是 1 时，为符号扩展。当 ExtOp 为 0 时，为 0 扩展。该扩展器的端口有：16 位的立即数输入端 imm16，一个扩展方式控制输入端 extop，结果输出端 ext_imm。其 Verilog 如下：

```
module extender(ext_imm,extop,imm16);

input imm16,extop;
output ext_imm;

wire [31:0]ext_imm;
wire [15:0]imm16;
wire extop;

assign ext_imm = extop? {{16{imm16[15]}}},imm16:{16'b0,imm16};

endmodule
```

6.1.3 取值部件

对于取指部件，由于需要有 beq 和 jump。下一条 PC 的情况有三种，分别是当前 PC+4，beq 跳转和 jump 跳转。由于 PC 的变化与低 2 位无关，所以可以只对高 30 位进行处理。这三种情况的 PC 的 2-31 位分别要进行加一，符号扩展加上 PC，与当前 PC 的 28-31 拼接的操作。且在确定下一条 PC 之前需要有两轮判断，最终确定下一条 PC，并输入指令存储器中，取出下一条指令。取指部件的端口有：输入端：Branch 控制端，zero 标志位，Jump 控制端，clk，reset；输出端口有：32 位的 Instruction 指令输出。其 Verilog 如下：

```
module ins_fetch(Instruction,zero,Branch,Jump,clk,reset);

input clk,reset,zero,Branch,Jump;
output[31:0] Instruction;

reg [31:0] PC;
wire [15:0]imm16;
wire [25:0]target;
wire [29:0]PC4,ext_imm30,PC_branch,PC_4_or_beq,PC_jump,next_addr;
wire [31:0]Instruction;

//initial begin
//  PC = 0;
//end

adder_30    u0(PC4,PC[31:2],30'b01);
sign_ext30 u2(ext_imm30,imm16);
adder_30    u1(PC_branch,ext_imm30,PC4);
```

```

mux_2x1 #(.DATA_WIDTH(30)
        )mux_2x1_u0(.in1(PC_branch
        ),
        .in0(PC4
        ),
        .sel(Branch & zero
        ),
        .out(PC_4_or_beq
        )
        );
mux_2x1 #(.DATA_WIDTH(30)
        )mux_2x1_u1(.in1(PC_jump
        ),
        .in0(PC_4_or_beq
        ),
        .sel(Jump
        ),
        .out(next_addr
        )
        );
assign PC_jump = {PC[31:28],target};
assign imm16 = Instruction[15:0];
assign target = Instruction[25:0];

always@(negedge clk)
begin
    if(!reset) begin
        PC=0;
    end
    else begin
        PC = {next_addr,2'b00};
    end
end

memorys      Ins_Mem(32'b0,1'b0,{2'b00,PC[31:2]},Instruction,clk);//memorys(DataIn,WrEn,Adr,
DataOut, Clk);

endmodule

```

6.1.4 寄存器堆

寄存器堆包含 32 个寄存器，两条 32 位输出总线 busA 和 busB，一条 32 位输入总线 busW，它们由 Ra、Rb、Rw 决定寄存器的选择。RegWr 为写使能。clk 为时钟输入，只有在写操作中，clk 才有作用，读操作中，寄存器的行为和组合逻辑电路一样。要注意，zero 寄存器读数永远为 32'b0。其 Verilog 如下：

```

module registers(busA,busB,busW,Ra,Rb,Rw,RegWr,Clk);
    input [4:0] Ra,Rb,Rw;
    input [31:0]busW;
    input RegWr,Clk;
    output [31:0]busA,busB;

    reg [31:0]data[31:0];
    wire[31:0] busA,busB;

    assign busA = (Ra==5'b0)? 32'b0:data[Ra];

```

```

assign busB = (Rb==5'b0)? 32'b0:data[Rb];

always@(negedge Clk)
begin

    if(RegWr)
    begin
        data[Rw] = busW;
    end
end
endmodule

```

6.1.5 指令/数据存储器

指令/数据存储器为理想化的存储器, 有一条输入总线 DataIn 和一条输出总线 DataOut, 地址 Adr 用于选择存储字, WrEn 为写使能。clk 为时钟输入, 只有在写操作中, clk 才有作用, 读操作中, 其行为和组合逻辑电路一样。其 Verilog 如下:

```

module memorys(DataIn,WrEn,Adr,DataOut,Clk);
    input [31:0]DataIn;
    input [31:0]Adr;
    input WrEn;
    input Clk;
    output [31:0]DataOut;

    reg [31:0]data[127:0];
    wire [31:0]DataOut;

    always@(negedge Clk)
    begin
        if(WrEn)
        begin
            data[Adr] = DataIn;
        end
    end

    assign DataOut=data[Adr];

endmodule

```

6.1.6 数据选择器

单周期所用到的 MUX 都是二选一的, 但位数涉及 32 位、30 位、5 位, 为了统一格式, 将位宽作为参数写在模块中。其 Verilog 如下:

```

module mux_2x1 #(parameter DATA_WIDTH = 32) (in1, in0, sel, out);

    input [DATA_WIDTH-1:0] in1;
    input [DATA_WIDTH-1:0] in0;
    input sel;

```

```
output [DATA_WIDTH-1:0] out;
```

```
assign out=sel?in1:in0;
```

```
endmodule
```

6.1.7 主控制器

主控制器为逻辑电路，输入为指令的前六位 opcode，输出为 RegWr,RegDst,ExtOp,AluSrc,ALUop,MemWr,MemtoReg,Branch,Jump 控制信号，其 Verilog 如下：

```
module MIPS_control(op,RegWr,RegDst,ExtOp,AluSrc,ALUop,MemWr,MemtoReg,Branch,Jump);
```

```
input [5:0] op;
```

```
output RegWr,RegDst,ExtOp,AluSrc,MemWr,MemtoReg,Branch,Jump;
```

```
output [2:0] ALUop;
```

```
assign RegWr=(!op[5]&!op[4]&!op[3]&!op[2]&!op[1]&!op[0])
```

```
    | (!op[5]&!op[4]&op[3]&op[2]&!op[1]&op[0])
```

```
    | (op[5]&!op[4]&!op[3]&!op[2]&op[1]&op[0])
```

```
    | (!op[5]&!op[4]&op[3]&!op[2]&!op[1]&!op[0]);//R-type ori lw addi
```

```
assign RegDst=!op[5]&!op[4]&!op[3]&!op[2]&!op[1]&!op[0];//R-type
```

```
assign ExtOp=(op[5]&!op[4]&!op[3]&!op[2]&op[1]&op[0])
```

```
    | (op[5]&!op[4]&op[3]&!op[2]&op[1]&op[0])
```

```
    | (!op[5]&!op[4]&op[3]&!op[2]&!op[1]&!op[0]);//lw sw addi
```

```
assign AluSrc=(!op[5]&!op[4]&op[3]&op[2]&!op[1]&op[0])
```

```
    | (op[5]&!op[4]&!op[3]&!op[2]&op[1]&op[0])
```

```
    | (op[5]&!op[4]&op[3]&!op[2]&op[1]&op[0])
```

```
    | (!op[5]&!op[4]&op[3]&!op[2]&!op[1]&!op[0]);//ori lw sw addi
```

```
assign ALUop[2]=!op[5]&!op[4]&!op[3]&!op[2]&!op[1]&!op[0];//R-type
```

```
assign ALUop[1]=!op[5]&!op[4]&op[3]&op[2]&!op[1]&op[0];//ori
```

```
assign ALUop[0]=!op[5]&!op[4]&!op[3]&op[2]&!op[1]&!op[0];//beq
```

```
assign MemWr=op[5]&!op[4]&op[3]&!op[2]&op[1]&op[0];//sw
```

```
assign MemtoReg=op[5]&!op[4]&!op[3]&!op[2]&op[1]&op[0];//lw
```

```
assign Branch=!op[5]&!op[4]&!op[3]&op[2]&!op[1]&!op[0];//beq
```

```
assign Jump=!op[5]&!op[4]&!op[3]&!op[2]&op[1]&!op[0];//jump
```

```
endmodule
```

6.1.8 ALU 控制器

分离 ALU 控制器进行局部译码，可以减小主控单元的大小，提升控制速度。ALU 控制器也是逻辑电路，有两个输入 func,ALUop 和一个输出 ALUctr，其 Verilog 如下：

```
module ALU_control(func,ALUop,ALUctr);
```

```
input [5:0] func;
```

```
input [2:0] ALUop;
```

```
output [2:0] ALUctr;
```

```
wire[2:0] ALUctr;
```

```

assign ALUctr[2]=(!ALUop[2]&ALUop[0])|(ALUop[2]&!func[2]&func[1]&!func[0]);
assign ALUctr[1]=(!ALUop[2]&!ALUop[1])|(!ALUop[2]&ALUop[0])|(ALUop[2]&!func[2] &
!func[0]);
assign ALUctr[0]=(!ALUop[2]&ALUop[1])|(ALUop[2]&!func[3]&func[2]&!func[1]&func[0])|(
ALUop[2]&func[3]&!func[2]&func[1]&!func[0]);

endmodule

```

6.1.9 总装

按照单周期处理器结构将上述模块连接在一起，对各信号合理命名，得到如下的顶层模块：

```

module MIPS_top(clk,reset);
input clk,reset;
wire[31:0] Instruction,busA,busB,busW,ext_imm,busB_MUX,ALUout,DataOut;
wire[5:0] op,func;
wire[4:0] Rt,Rs,Rd,Rw;
wire[15:0] imm16;
wire[2:0] ALUctr,ALUop;

ins_fetch ins_fetch(Instruction,zero,Branch,Jump,clk,reset);
assign op=Instruction[31:26];
assign Rs=Instruction[25:21];
assign Rt=Instruction[20:16];
assign Rd=Instruction[15:11];
assign imm16=Instruction[15:0];
assign func=Instruction[5:0];

MIPS_control MIPS_control(op,RegWr,RegDst,ExtOp,AluSrc,ALUop,MemWr,MemtoReg,Branch,
Jump);
ALU_control ALU_control(func,ALUop,ALUctr);

mux_2x1 #(5) mux_2x1_RegDst(Rd,Rt,RegDst,Rw);//mux_2x1(in1, in0, sel, out);
registers registers(busA,busB,busW,Rs,Rt,Rw,RegWr,clk);
//registers(busA,busB,busW,Ra,Rb,Rw,RegWr,Clk);
extender extender(ext_imm,ExtOp,imm16);
mux_2x1 #(32) mux_2x1_Alusrc(ext_imm,busB,AluSrc,busB_MUX);
ALU ALU(ALUout,zero,busA,busB_MUX,ALUctr);
memorys Data_Mem(busB,MemWr,ALUout,DataOut,clk);
//memorys(DataIn,WrEn,Adr,DataOut,Clk);
mux_2x1 #(32) mux_2x1_MemtoReg(DataOut,ALUout,MemtoReg,busW);

endmodule

```

6.2 流水线处理器

流水线处理器所用的寄存器堆、指令/数据存储器、位扩展器、ALU、ALU 控制器、主控制器均与单周期处理器相同。

6.2.1 IF/ID 寄存器

IF_ID 寄存器用来寄存取指阶段的指令和 PC+4 的结果，一个周期后向取操作数，译码阶段输出指令和 PC+4。同时还需要加入一个控制端控制该寄存器的保持功能以解决冒险问题。其 Verilog 代码如下：

```
module IF_ID_reg(Instruction_out,PC_add4_out,Instruction_in,PC_add4_in,ctrl,clk,reset);
input clk,Instruction_in,PC_add4_in,ctrl,reset;
output Instruction_out,PC_add4_out;

reg [31:0]Instruction_out,PC_add4_out;
wire [31:0]Instruction_in,PC_add4_in;
wire clk,ctrl;

//assign myclk = ~ctrl | clk;

always@(negedge clk)
begin
    if(!reset)
    begin
        PC_add4_out <= 32'b0;
        Instruction_out <= 32'b0;
    end
    else if(!ctrl)
    begin
        PC_add4_out <= PC_add4_out;
        Instruction_out <= Instruction_out;
    end
    else
    begin
        PC_add4_out <= PC_add4_in;
        Instruction_out <= Instruction_in;
    end
end

endmodule
```

6.2.2 ID/EX 寄存器

ID_EX 寄存器用来寄存取操作数的结果和控制信号 WB,M,EX。他需要将 PC+4，数据寄存器的输出寄存下来。在寄存控制信号时，需要加入一个选择器来选择寄存完整的控制信号还是清零控制信号，以此来加入空泡。同时，还需要将当前指令的 Rs，Rt，Rd 寄存下来，以便在后续判断是否发生冒险，同时还需要通过 Rt 和 Rd 来选择回写路径。其 Verilog 代码如下：

```
module ID_EX_reg(
busA_out,busB_out,PC_add4_out,EX_out,M_out,WB_out,Ext_op_out,
imm16_out,Instruction25_21_out,Instruction20_16_out,Instruction15_11_out,
```



```

busA_in,busB_in,PC_add4_in,EX_in,M_in,WB_in,Ext_op_in,
imm16_in,Instruction25_21_in,Instruction20_16_in,Instruction15_11_in,clk,reset);

input busA_in,busB_in,PC_add4_in,EX_in,M_in,WB_in,Ext_op_in,Ext_op_in,
imm16_in,Instruction25_21_in,Instruction20_16_in,Instruction15_11_in,clk,reset;
output busA_out,busB_out,PC_add4_out,EX_out,M_out,WB_out,Ext_op_out,
imm16_out,Instruction25_21_out,Instruction20_16_out,Instruction15_11_out;

wire [31:0]busA_in,busB_in,PC_add4_in;
wire [4:0]Instruction25_21_in,Instruction20_16_in,Instruction15_11_in;
wire [15:0]imm16_in;
wire [1:0]WB_in;
wire [4:0]EX_in;
wire [2:0]M_in;
wire clk,Ext_op_in;

reg [31:0]busA_out,busB_out,PC_add4_out;
reg [4:0]Instruction25_21_out,Instruction20_16_out,Instruction15_11_out;
reg [15:0]imm16_out;
reg [1:0]WB_out;
reg [4:0]EX_out;
reg [2:0]M_out;
reg Ext_op_out;

always@(negedge clk)
begin
    if(!reset)
    begin
        {busA_out,busB_out,PC_add4_out,EX_out,M_out,WB_out,Ext_op_out,Ext_op_out,
imm16_out,Instruction25_21_out,Instruction20_16_out,Instruction15_11_out} <= 500'b0;
    end
    else
    begin
        {busA_out,busB_out,PC_add4_out,EX_out,M_out,WB_out,Ext_op_out,Ext_op_out,
imm16_out,Instruction25_21_out,Instruction20_16_out,Instruction15_11_out} <=
        {busA_in,busB_in,PC_add4_in,EX_in,M_in,WB_in,Ext_op_in,Ext_op_in,
imm16_in,Instruction25_21_in,Instruction20_16_in,Instruction15_11_in};
    end
end

endmodule

```

6.2.3 EX/MEM 寄存器

EX_MEM 寄存器用来存储后续控制信号 WB,M 和 ALU 的结果以及最后要写入的目标寄存器。其 Verilog 代码如下:

```

module EX_MEM_reg(

```

```

ALUout_out,zero_out,busB_out,Rw_out,WB_out,M_out,ADDout_out,
ALUout_in,zero_in,busB_in,Rw_in,WB_in,M_in,ADDout_in,clk,reset);

input ALUout_in,zero_in,busB_in,Rw_in,WB_in,M_in,ADDout_in,clk,reset;
output ALUout_out,zero_out,busB_out,Rw_out,WB_out,M_out,ADDout_out;

wire [31:0]busB_in,ALUout_in,ADDout_in;
wire [4:0]Rw_in;
wire [1:0]WB_in;
wire [2:0]M_in;

reg [31:0]busB_out,ALUout_out,ADDout_out;
reg [4:0]Rw_out;
reg [1:0]WB_out;
reg [2:0]M_out;
reg zero_out;

always@(negedge clk)
begin
    if(!reset)
    begin
        {ALUout_out,zero_out,busB_out,Rw_out,WB_out,M_out,ADDout_out} <= 500'b0;
    end
    else
    begin
        {ALUout_out,zero_out,busB_out,Rw_out,WB_out,M_out,ADDout_out} <=
        {ALUout_in,zero_in,busB_in,Rw_in,WB_in,M_in,ADDout_in};
    end
end

endmodule

```

6.2.4 MEM/WB 寄存器

MEM_WB 寄存器用来保存与回写相关的控制信号 WB，还有存储器的输出和上一级 ALU 的结果，同时还保存回写的目标寄存器。其 Verilog 代码如下：

```

module
MEM_WB_reg(WB_out,Rw_out,ALUout_out,DataMem_out,WB_in,Rw_in,ALUout_in,DataMem_in,clk,reset);
    output reg[1:0] WB_out;
    output reg[4:0] Rw_out;
    output reg[31:0] ALUout_out,DataMem_out;
    input[1:0] WB_in;
    input[4:0] Rw_in;
    input[31:0] ALUout_in,DataMem_in;
    input clk,reset;

```

```

always@(negedge clk)
begin
    if(!reset) begin
        WB_out<=2'b0;
        Rw_out<=5'b0;
        ALUout_out<=32'b0;
        DataMem_out<=32'b0;
    end
    else begin
        WB_out<=WB_in;
        Rw_out<=Rw_in;
        ALUout_out<=ALUout_in;
        DataMem_out<=DataMem_in;
    end
end

endmodule

```

6.2.5 PC 寄存器

相比于单周期处理器, 为了实现冒险检测并加入空泡, PC 需要有暂停功能。PCWr 为 PC 的使能控制, 用于暂停 PC。其 Verilog 代码如下:

```

module PC(PC_in,PC_out,PCWr,clk,reset);
output reg[31:0] PC_out;
input[31:0] PC_in;
input PCWr,clk,reset;

always@(negedge clk)
begin
    if(!reset) begin
        PC_out<=0;
    end
    else if(!PCWr) begin
        PC_out<=PC_out;
    end
    else begin
        PC_out<=PC_in;
    end
end

endmodule

```

6.2.6 左移模块

左移模块用来移位 beq 的扩展后的立即数, 其结果加上 PC+4 等于即将分支的地址。其内部功能为输入左移两位。其 Verilog 代码如下:

```

module shiftleft2(out,in);
input in;

```

```

output out;
wire [31:0]in,out;
assign out = in <<2;
endmodule

```

6.2.7 三选一选择器

由于流水线处理器有前递单元，ALU 的输入信号需要增加三选一的 MUX，其 Verilog 代码如下：

```

module mux_3x1 #(parameter DATA_WIDTH = 32) (in10, in01, in00, sel, out);

input [DATA_WIDTH-1:0] in10;
input [DATA_WIDTH-1:0] in01;
input [DATA_WIDTH-1:0] in00;
input [1:0] sel;
output [DATA_WIDTH-1:0] out;

assign out=sel[1]?in10:(sel[0]?in01:in00);

endmodule

```

6.2.8 前递单元

为了解决大多数的数据冒险，可以从某流水段向其他流水段前递结果。前递单元输入为 ID/EX、EX/MEM、MEM/WB 的部分寄存器，输出为 ALU 三选一 MUX 的控制信号 ForwardA,ForwardB。相比于课件中的前递单元，本单元添加了一些代码，用于解决 lw 加入空泡导致的错误前递。其 Verilog 代码如下：

```

module Forwarding_unit(ForwardA,ForwardB,ID_EX_Rs,ID_EX_Rt,EX_MEM_Rd,
MEM_WB_Rd,EX_MEM_RegWr,MEM_WB_RegWr);
output reg[1:0] ForwardA,ForwardB;
input[4:0] ID_EX_Rs,ID_EX_Rt,EX_MEM_Rd,MEM_WB_Rd;
input EX_MEM_RegWr,MEM_WB_RegWr;

always@(ID_EX_Rs or ID_EX_Rt or EX_MEM_Rd or MEM_WB_Rd
or EX_MEM_RegWr or MEM_WB_RegWr)
begin
    if(EX_MEM_RegWr && EX_MEM_Rd!=5'b0 && EX_MEM_Rd==ID_EX_Rs)//EX hazard
        ForwardA<=2'b10;
    else if(MEM_WB_RegWr && MEM_WB_Rd!=5'b0 && MEM_WB_Rd==ID_EX_Rs &&
(EX_MEM_Rd!=ID_EX_Rs || !EX_MEM_RegWr) )//MEM hazard
        ForwardA<=2'b01;
    else
        ForwardA<=2'b00;

    if(EX_MEM_RegWr && EX_MEM_Rd!=5'b0 && EX_MEM_Rd==ID_EX_Rt)
        ForwardB<=2'b10;
    else if(MEM_WB_RegWr && MEM_WB_Rd!=5'b0 && MEM_WB_Rd==ID_EX_Rt &&
(EX_MEM_Rd!=ID_EX_Rt|| !EX_MEM_RegWr) )

```

```

        ForwardB<=2'b01;
    else
        ForwardB<=2'b00;
    end

endmodule

```

6.2.9 冒险检测单元

装入指令 lw 数据冒险无法只通过数据前递解决，需要冒险检测单元，用于检测 lw 冒险，暂停 PC、IR/ID 并清除 EX/MEM，其 Verilog 代码如下：

```

module Hazard_detection(PCWr,IF_IDWr,ID_EXMux,ID_EX_MemtoReg,ID_EX_Rt,
IF_ID_Rs,IF_ID_Rt);
output reg PCWr,IF_IDWr,ID_EXMux;
input ID_EX_MemtoReg;
input[4:0] ID_EX_Rt,IF_ID_Rs,IF_ID_Rt;

always@(ID_EX_MemtoReg or ID_EX_Rt or IF_ID_Rs or IF_ID_Rt)
begin
    if(ID_EX_MemtoReg && (ID_EX_Rt==IF_ID_Rs || ID_EX_Rt==IF_ID_Rt) )
    begin
        PCWr<=0;
        IF_IDWr<=0;
        ID_EXMux<=0;
    end
    else begin
        PCWr<=1;
        IF_IDWr<=1;
        ID_EXMux<=1;
    end
end
end

endmodule

```

6.2.10 总装

按照流水线处理器结构将上述模块连接在一起，对各信号合理命名，得到如下的顶层模块：

```

module MIPS_top(clk,reset);
input clk,reset;
wire[31:0]
Instruction,busA,busB,busW,ext_imm,busB_MUX,ALUout,DataOut,PC_add4_in1,PC_add4_out,PC_ad
d4;
wire[5:0] op,func;
wire[4:0] Rt,Rs,Rd,Rw,Rw_in,Rd_out,Rs_out,Rt_out,Rw_out;
wire[15:0] imm16,imm16_out;
wire[2:0] ALUctr,ALUop,ALUop_ctrl;
wire[1:0] WB,WB1,WB2,WB3,ForwardA,ForwardB,WB_mux_in;

```

```

    wire[4:0] EX,EX1,EX_mux_in;
    wire[2:0] M,M1,M2,M_mux_in;
    wire[31:0]
imm_shift,ADDout,ALU_out,ALUB_in,ALUA_in,busA_out,busB_out,ALUB_mux_in,busA_in,busB_i
n,Regw_data;
    wire[31:0]
PC_branch,PCorBranch,PC,PC_jump,PC_in,Instruction_out,ALUout_out,Write_datas,Dataout_muxin,A
LUout_muxin;
    wire RegW,Mem2Reg,Branch,Jump,MemR,MemW,RegDst,ALUSrc,zero;
    wire[9:0] ID_EX_ctrl_in;

    assign op=Instruction[31:26];
    assign Rs=Instruction[25:21];
    assign Rt=Instruction[20:16];
    assign Rd=Instruction[15:11];
    assign imm16=Instruction[15:0];
    assign func=imm16_out[5:0];
    assign WB_mux_in = {RegW,Mem2Reg};
    assign M_mux_in = {Branch,1'b1,MemW};
    assign EX_mux_in = {RegDst,ALUop,ALUSrc};
    assign WB = ID_EX_ctrl_in[9:8];
    assign M = ID_EX_ctrl_in[7:5];
    assign EX = ID_EX_ctrl_in[4:0];
    assign RegDst_ctrl = EX1[4];
    assign ALUop_ctrl = EX1[3:1];
    assign ALUSrc_ctrl = EX1[0];
    assign Branch_ctrl = M2[2];
    assign MemR_ctrl = M2[1];
    assign MemW_ctrl = M2[0];
    assign RegW_ctrl = WB3[1];
    assign EX_MEM_RegWr = WB2[1];
    assign Mem2Reg_ctrl = WB3[0];
    assign ID_EX_MemtoReg = WB1[0];
    assign
Jump=!Instruction_out[31]&!Instruction_out[30]&!Instruction_out[29]&!Instruction_out[28]&Instructio
n_out[27]&!Instruction_out[26];
    assign PC_jump = {PC[31:28],Instruction_out[25:0],2'b00};
    assign PCSrc = Branch_ctrl & zero;

    mux_2x1 #(32) mux_2x1_PC(.in1(PC_branch),.in0(PC_add4),.sel(PCSrc),.out(PCorBranch));
//add4 or branch
    mux_2x1 #(32) mux_2x1_Jump(PC_jump,PCorBranch,Jump,PC_in);//mux_2x1(in1, in0, sel, out);
//jump or add4

    PC PC_u0(.PC_in(PC_in),.PC_out(PC),.PCWr(PCWr),.clk(clk),.reset(reset));
    adder_32 adder_32_PC(.C(PC_add4),.A(PC),.B(32'd4));

```

```

    memorys
Ins_Mem(32'b0,1'b0,{2'b00,PC[31:2]},Instruction_out,clk);//memorys(DataIn,WrEn,Adr,DataOut,Clk);

    IF_ID_reg
IF_ID_reg_u0(.Instruction_out(Instruction),.PC_add4_out(PC_add4_in1),.Instruction_in(Instruction_out
),.PC_add4_in(PC_add4),.ctrl(IF_IDWr),.clk(clk),.reset(reset));

    Hazard_detection
Hazard_detection_u0(.PCWr(PCWr),.IF_IDWr(IF_IDWr),.ID_EXMux(ID_EXMux),.ID_EX_MemtoRe
g(ID_EX_MemtoReg),.ID_EX_Rt(Rt_out),.IF_ID_Rs(Rs),.IF_ID_Rt(Rt));

    MIPS_control
MIPS_control_u0(op,RegW,RegDst,ExtOp,ALUSrc,ALUop,MemW,Mem2Reg,Branch);

    registers
registers_u0(.busA(busA_in),.busB(busB_in),.busW(Regw_data),.Ra(Rs),.Rb(Rt),.Rw(Rw),.RegWr(Reg
W_ctrl),.Clk(clk));

    ID_EX_reg ID_EX_reg_u0(
    .busA_out(busA_out),.busB_out(busB_out),.PC_add4_out(PC_add4_out),.EX_out(EX1),.M_out(M
1),.WB_out(WB1),.Ext_op_out(ExtOp_ctrl),
    .imm16_out(imm16_out),.Instruction25_21_out(Rs_out),.Instruction20_16_out(Rt_out),.Instruction
15_11_out(Rd_out),
    .busA_in(busA_in),.busB_in(busB_in),.PC_add4_in(PC_add4_in1),.EX_in(EX),.M_in(M),.WB_in
(WB),.Ext_op_in(ExtOp),
    .imm16_in(imm16),.Instruction25_21_in(Rs),.Instruction20_16_in(Rt),.Instruction15_11_in(Rd),.cl
k(clk),.reset(reset));

    extender extender_u0(.ext_imm(ext_imm),.extop(ExtOp_ctrl),.imm16(imm16_out));
    shiftright2 shiftright2_u0(.out(imm_shift),.in(ext_imm));
    adder_32 adder_32_u0(.C(ADDout),.A(PC_add4_out),.B(imm_shift));

    mux_3x1 #(32)
mux_3x1_a(.in10(ALUout_out),.in01(Regw_data),.in00(busA_out),.sel(ForwardA),.out(ALUA_in));
    mux_3x1 #(32)
mux_3x1_b(.in10(ALUout_out),.in01(Regw_data),.in00(busB_out),.sel(ForwardB),.out(ALUB_mux_
in));
    mux_2x1 #(32)
mux_2x1_AluSrc(.in1(ext_imm),.in0(ALUB_mux_in),.sel(ALUSrc_ctrl),.out(ALUB_in));
    ALU_control ALU_control_u0(func,ALUop_ctrl,ALUctr);
    ALU
ALU_u0(.busC(ALU_out),.zero(zero_in),.busA(ALUA_in),.busB(ALUB_in),.ALUctr(ALUctr));

    Forwarding_unit
Forwarding_unit_u0(.ForwardA(ForwardA),.ForwardB(ForwardB),.ID_EX_Rs(Rs_out),.ID_EX_Rt(Rt_
out),.EX_MEM_Rd(Rw_out),.MEM_WB_Rd(Rw),.EX_MEM_RegWr(EX_MEM_RegWr),.MEM_WB
_RegWr(RegW_ctrl));

```

```

mux_2x1 #(5) mux_2x1_RegDst(.in1(Rd_out),.in0(Rt_out),.sel(RegDst_ctrl),.out(Rw_in));

EX_MEM_reg EX_MEM_reg_u0(
    .ALUout_out(ALUout_out),.zero_out(zero),.busB_out(Write_datas),.Rw_out(Rw_out),.WB_out(W
B2),.M_out(M2),.ADDout_out(PC_branch),
    .ALUout_in(ALU_out),.zero_in(zero_in),.busB_in(ALUB_mux_in),.Rw_in(Rw_in),.WB_in(WB1)
,.M_in(M1),.ADDout_in(ADDout),.clk(clk),.reset(reset));

memorys
Data_Mem(Write_datas,MemW_ctrl,ALUout_out,DataOut,clk);//memorys(DataIn,WrEn,Adr,DataOut,C
lk);

MEM_WB_reg
MEM_WB_reg_u0(.WB_out(WB3),.Rw_out(Rw),.ALUout_out(ALUout_muxin),.DataMem_out(Datao
ut_muxin),
    .WB_in(WB2),.Rw_in(Rw_out),.ALUout_in(ALUout_out),.DataMem_in(DataOut),.clk(clk),.reset(
reset));
mux_2x1 #(32)
mux_2x1_Regwrite(.in1(Dataout_muxin),.in0(ALUout_muxin),.sel(Mem2Reg_ctrl),.out(Regw_data));

mux_2x1 #(10)
mux_2x1_CtlMux(.in1({WB_mux_in,M_mux_in,EX_mux_in}),.in0(10'b0),.sel(ID_EXMux),.out(ID_E
X_ctrl_in));

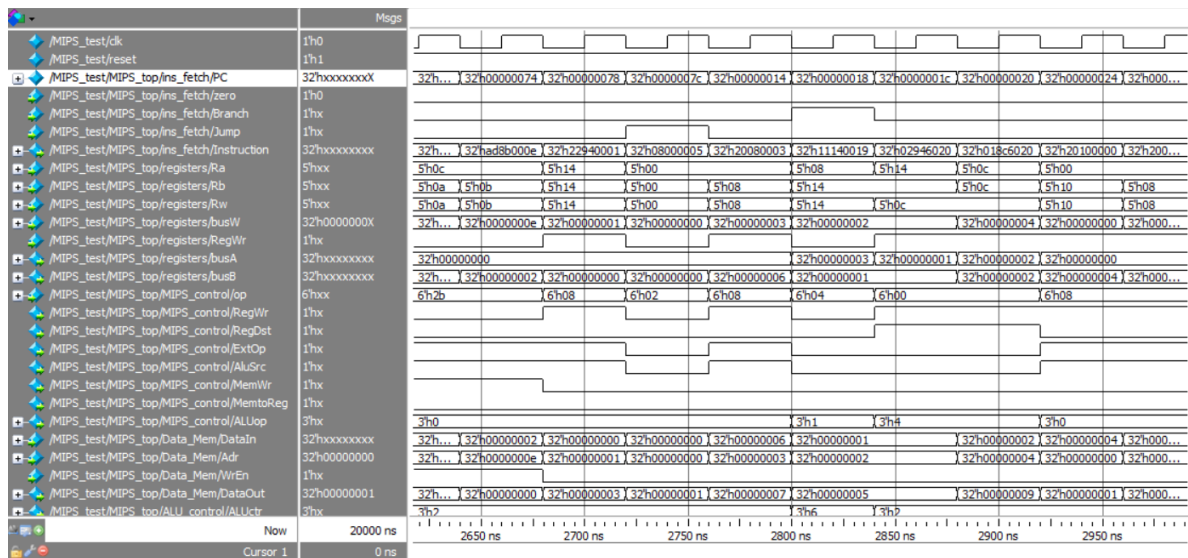
/*MIPS_control
MIPS_control(op,RegWr,RegDst,ExtOp,AluSrc,ALUop,MemWr,MemtoReg,Branch,Jump);
ins_fetch ins_fetch(Instruction,zero,Branch,Jump,clk,reset);
ALU_control ALU_control(func,ALUop,ALUctr);
mux_2x1 #(5) mux_2x1_RegDst(Rd,Rt,RegDst,Rw);//mux_2x1(in1, in0, sel, out);
registers
registers(busA,busB,busW,Rs,Rt,Rw,RegWr,clk);//registers(busA,busB,busW,Ra,Rb,Rw,RegWr,Clk);
extender extender(ext_imm,ExtOp,imm16);
mux_2x1 #(32) mux_2x1_Alusrc(ext_imm,busB,AluSrc,busB_MUX);
ALU ALU(ALUout,zero,busA,busB_MUX,ALUctr);
memorys
Data_Mem(busB,MemWr,ALUout,DataOut,clk);//memorys(DataIn,WrEn,Adr,DataOut,Clk);
mux_2x1 #(32) mux_2x1_MemtoReg(DataOut,ALUout,MemtoReg,busW);*/

endmodule

```


七、硬件仿真结果

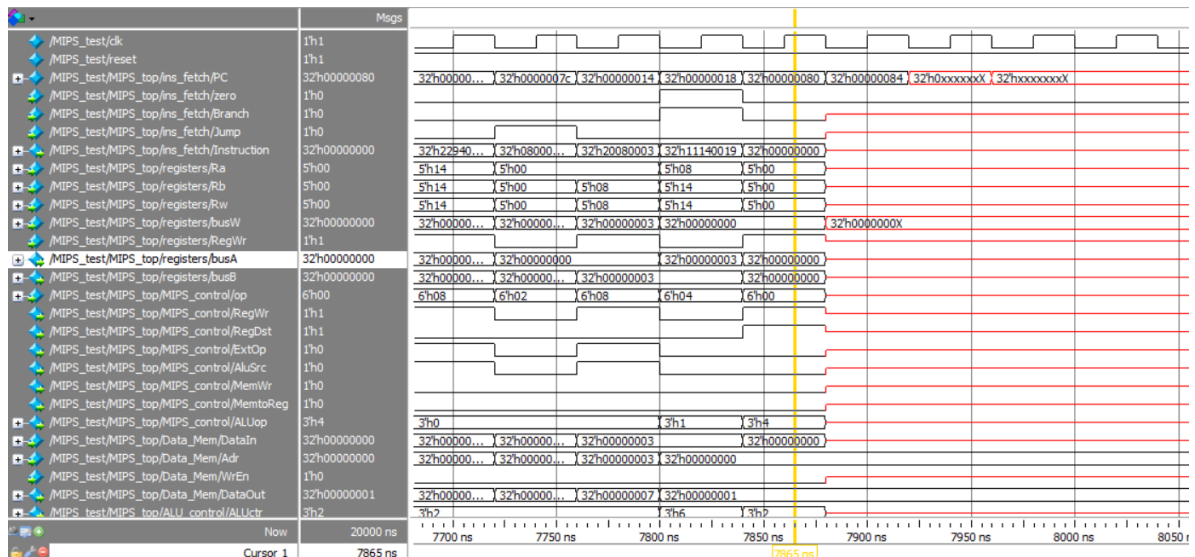
7.1 单周期处理器



如上图所示, PC 正常情况下每个周期加 4, 当发生 Branch 或 Jump 时, PC 跳变, 指令 Instruction 随 PC 而变, 控制信号随 Instruction 而变, 说明处理器大部分功能可以正常运行。

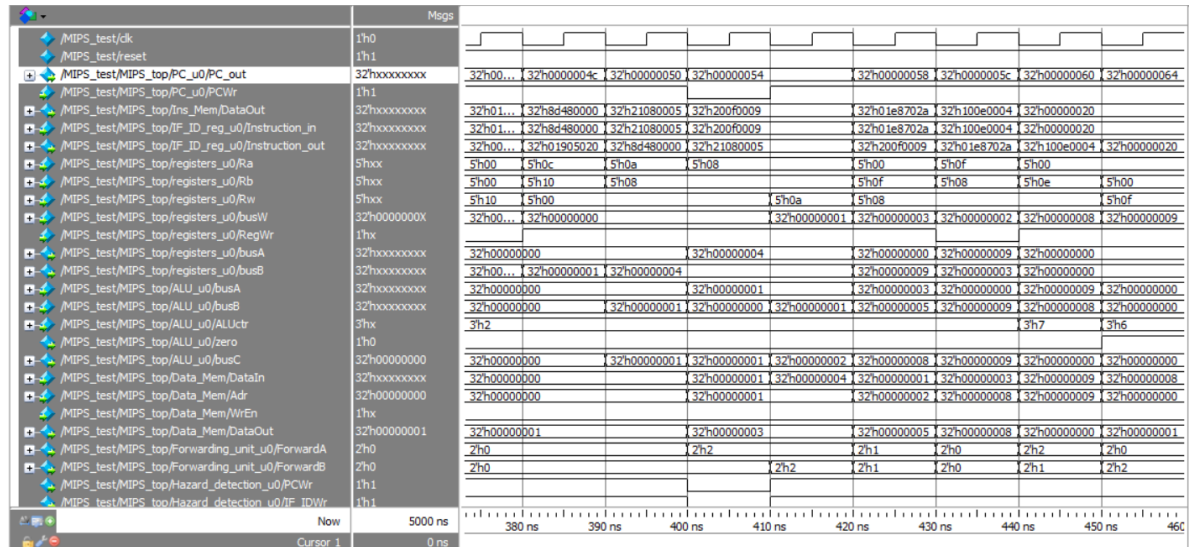
```
VSIM 21> run -all
# Input  1357 9246 8000
# Output 8620 7419 5355
```

总的运行结果如上图所示, 输出结果与预期相同, 说明成功完成了数据加密, 所设计的单周期处理器可以正常工作。



假设时钟 clk 周期为 40ns, 单周期处理器运行完所有代码, 所需时间约为 7900ns, 结束时间如上图所示。

7.2 流水线处理器

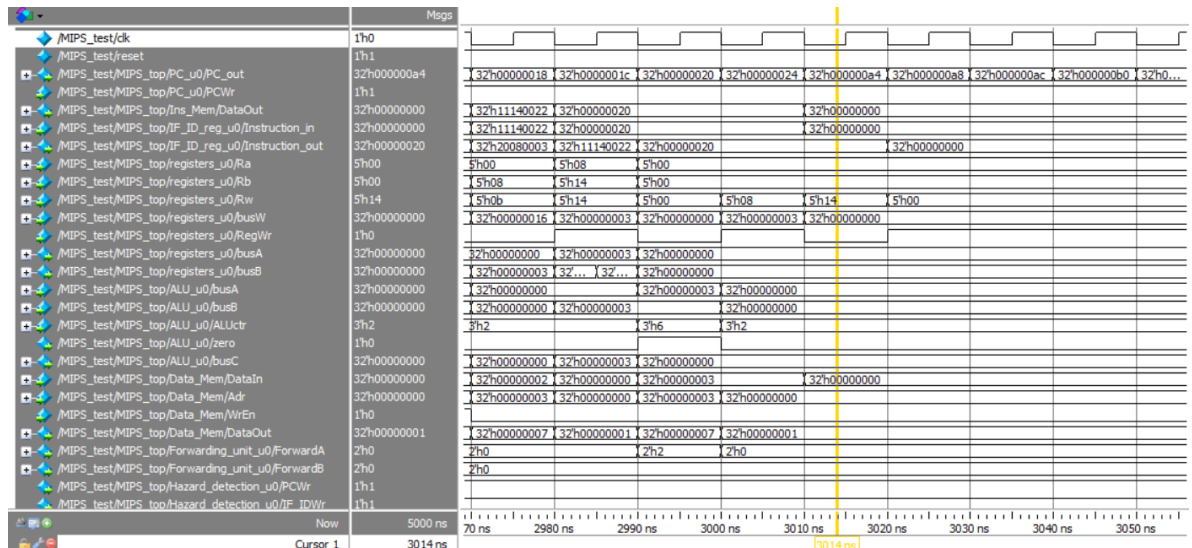


如上图所示，PC 正常情况下每个周期加 4，当发生 1w 数据冒险时，PC 和 IF/ID 被暂停，3 个周期后 `RegWr` 被置 0，说明 Bubble 作用成功。

也可以看出前递单元的输出 `ForwardA/B` 经常不为 0，说明发出了前递信号，并且数据冒险经常发生。

```
VSIM 13> run -all
# Input  1357 9246 8000
# Output 8620 7419 5355
```

总的运行结果如上图所示，输出结果与预期相同，说明成功完成了数据加密，所设计的流水线处理器可以正常工作。



假设在流水线结构下，时钟 `clk` 频率可以加快四倍，周期为 10ns，流水线处理器运行完所有代码，所需时间约为 3000ns，结束时间如上图所示。

可以看出流水线结构完成同样的任务，运行速度约为单周期结构的 2.6 倍，运行速度明显加快。