

中山大学移动信息工程学院本科生实验报告

(2013 学年秋季学期)

课程名称：数字电路实验

任课教师：王军

助教：黄秋鹏

年级&班级	12 级 1201 班	专业（方向）	软件工程（移动信息工程）
学号	12353022	姓名	陈胜杰
电话	13631203625	Email	1109197209@qq.com
开始日期	2013.12.9	完成日期	2014.1.4

实验题目：16-bit CPU by Pipeline

一、实验目的

- 1、使用 ISE 软件设计并仿真。
- 2、理解并熟悉五级流水线 CPU 的设计原理。
- 3、体会自主设计的流程与相关方法。

二、实验内容

• 实验步骤

- 1、理解并熟悉五级流水线CPU的设计原理，掌握基础理论知识。
- 2、编写Verilog文件并编译（未解决hazard）。
- 3、软件仿真与调试。
- 4、改进与优化设计（hazard的解决）。

• 实验原理

（一）流水线工作原理

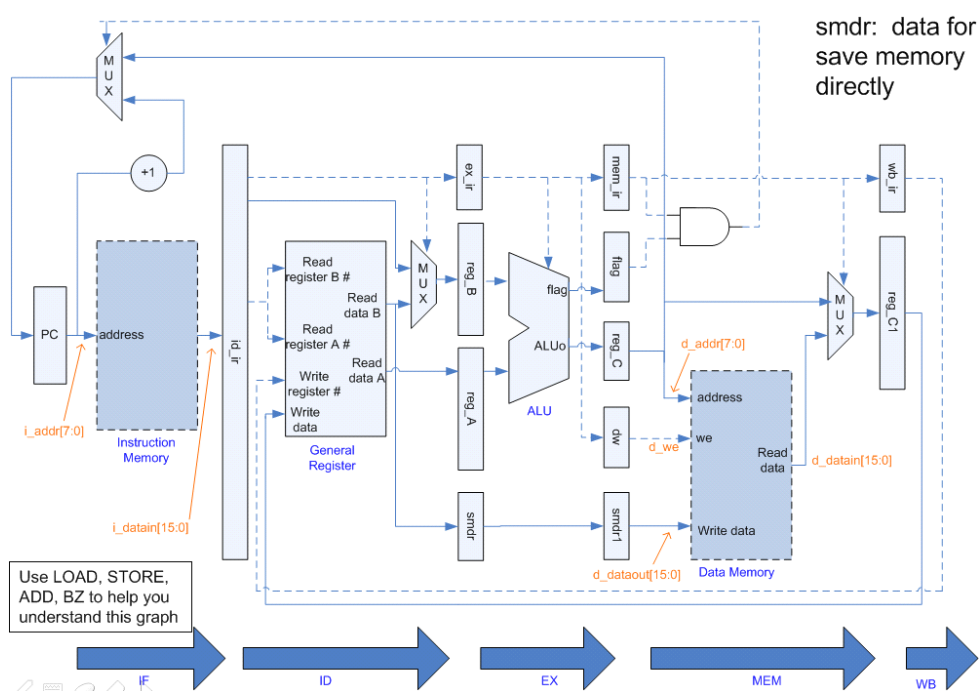


Figure1.block diagram

1. 流水线是数字系统中一种提高系统稳定性和工作速度的方法，广泛应用于高档 CPU 的架构中。根

据 MIPS 处理器的特点，将整体的处理过程分为取指令（IF）、指令译码（ID）、执行（EX）、存储器访问（MEM）和寄存器回写（WB）五级流水线，对应多周期的五个处理阶段。如图 Figure1. 和 Figure2. 所示，一个指令的执行需要 5 个时钟周期，每个时钟周期的上升沿来临时，此指令所代表的一系列数据和控制信息将转移到下一级进行处理。

	1	2	3	4	5	6	7	8	9
I1	IF	ID	EX	MEM	WB				
I2		IF	ID	EX	MEM	WB			
I3			IF	ID	EX	MEM	WB		
I4				IF	ID	EX	MEM	WB	
I5					IF	ID	EX	MEM	WB

Figure2. 流水线流水作业示意图

2. 流水线中的数据流。

（1）IF 级：取指令级。从 Instruction-Memory 中读取指令，并在下一个时钟上升沿到来时把指令送到 ID 级的指令缓冲器 id_ir 中。该级控制信号决定下一个指令指针的 pc 信号（即 Instruction-Memory 的指令地址 i_addr）。

（2）ID 级：指令译码器。对 IF 级的指令进行译码，根据指令操作码获取操作数 reg_A、reg_B 或者要直接储存的数据内容 smdr，并在下一个时钟上升沿到来前把指令 id_ir（前 8 位，操作码+operand1）送到 EX 级的指令缓冲器 ex_ir 中。

（3）EX 级：执行级。该级进行算术运算（加、减）、简单传输（JUMP 操作）、逻辑运算（与、或、异或）或移位操作（逻辑左移、逻辑右移、算术左移、算术右移）。算术逻辑单元 ALU 根据指令对两个操作数 reg_A、reg_B 进行操作，将获得的结果 ALUo 送到下一级的 reg_C，在此过程中，控制标志信号 cf、nf、zf 并将其传到相应的缓冲寄存器；或者产生存储数据的使能信号 d_we，同时将要直接储存的数据内容 smdr 传到 MEM 级的 smdr1。在下一个时钟上升沿到来前把指令 ex_ir 送到 MEM 级的指令缓冲器 mem_ir 中。

（4）MEM 级：数据存储器访问级。根据指令处理 reg_C 获取需要的内容存储到缓冲器 reg_C1，并在下一个时钟上升沿到来前把指令 mem_ir 送到 WB 级的指令缓冲器 wb_ir 中。只有在执行 LOAD、STORE 指令时才对存储器进行读、写操作，对于此之外的其他指令，MEM 级只起到一个周期的作用。

（5）WB 级：写回级。对于需要刷新通用寄存器的操作，WB 级把指令执行的结果回写到通用寄存器中。

3. 命名的规范。

由于在流水线中，数据和控制信息将在时钟周期的上升沿到来前转移到下一级，这在一定程度上增加了寄存器数量，更重要的是相同功能的寄存器的数量。在顶层文件中，类似的变量名称有近百个，因此规范命名能起到很好的识别作用。规定流水线转移变量命名遵守如下格式：

流水线级名称_变量名；（如：id_ir、ex_ir、mem_ir、wb_ir）

这在大型工程中是一个风格较好的习惯。

（二）实现操作与指令译码格式

				15	11	10	8	7	4	3	0
				Op code (5 bit)		Operand 1 (3 bit)		Operand 2 (4 bit)		Operand 3 (4 bit)	
						r1		r2		r3	
R type (register type)	r1	r2	r3	000: gr[0]		x000: gr[0]		x000: gr[0]			
				001: gr[1]		x001: gr[1]		x001: gr[1]			
				010: gr[2]		x010: gr[2]		x010: gr[2]			
				011: gr[3]		x011: gr[3]		x011: gr[3]			
				100: gr[4]		x100: gr[4]		x100: gr[4]			
I type (Immediate type)	r1	val2	val3	101: gr[5]		x101: gr[5]		x101: gr[5]			
		-	val2	val3	110: gr[6]		x110: gr[6]				
				111: gr[7]		x111: gr[7]					
RI type	r1	r2	val3			val2		val3			
						Immediate data (4 bit)		Immediate data (4 bit)			

gr: general register
(16 bit X 8)

Figure 3. Operation field

1. 指令译码格式。

如上图所示，指令为 16 位 $ir[15:0]$ ，其中 $ir[15:11]$ 为 5 位操作码； $ir[10:8]$ 为通用寄存器 gr 的编号（一共有 8 个通用寄存器）； $ir[7:4]$ 既可以是通用寄存器 gr 的编号，表示 $gr[ir[6:4]]$ （忽略 $ir[7]$ ），也可以是立即数 $ir[7:4]$ ；同样， $ir[3:0]$ 也可以是通用寄存器 gr 的编号，表示 $gr[ir[2:0]]$ （忽略 $ir[3]$ ），亦或是立即数 $ir[3:0]$ 。

2. 相关操作说明（结合 Figure 3.operation field）。








① 第一类 Data transfer & Arithmetic 操作

操作名	操作编码	Operand 1	Operand 2	Operand 3	操作说明
LOAD	11010	$gr[r1]$	$gr[r2]$	$val3$	$gr[r1] = m[gr[r2]+val3]$ ，将 data memory 里面地址为 $[gr[r2]+val3]$ 的值加载到 $gr[r1]$ 。
STORE	00010	$gr[r1]$	$gr[r2]$	$val3$	$m[gr[r2]+val3] = gr[r1]$ ，将 $gr[r1]$ 的内容存到 data memory 里面地址为 $[gr[r2]+val3]$ 处。
LDIH	00011	$gr[r1]$	$val2$	$val3$	$gr[r1]=gr[r1]+\{val2, val3, 0000_0000\}$ 将 $gr[r1]$ 的内容更新为 $gr[r1]+\{val2, val3, 0000_0000\}$ ($val2, val3$ 作为高八位)，产生进位时 CF 置 1，否则置 0，结果为 0 时，ZF 置 1，否则置 0；结果 <0 时，NF 置 1，否则置 0。
ADD	00100	$gr[r1]$	$gr[r2]$	$gr[r3]$	$gr[r1] = gr[r2]+gr[r3]$ ，将 $gr[r2]$ 、 $gr[r3]$ 相加，将结果写到 $gr[r1]$ ，产生进位时 CF 置 1，否则置 0；结果为 0 时，ZF 置 1，否则置 0；结果 <0 时，NF 置 1，否则置 0。
ADDI	00101	$gr[r1]$	$val2$	$val3$	$gr[r1] = gr[r1]+\{0000_0000, val2, val3\}$ 将 $gr[r1]$ 更新为 $gr[r1]+\{0000_0000, val2, val3\}$ ($val2, val3$ 作为低八位)，产生进位时 CF 置 1，否则置 0；结果为 0 时，ZF 置 1，否则置 0；结果 <0 时，NF 置 1，否则置 0。
ADDC	00110	$gr[r1]$	$gr[r2]$	$gr[r3]$	$gr[r1] = gr[r2]+gr[r3]+CF$ 实现带有进位 CF 的 ADD，产生进位时 CF 置 1，否则置 0；结果为 0 时，ZF 置 1，否则置 0；结果 <0 时，NF 置 1，否则置 0。
SUB	00111	$gr[r1]$	$gr[r2]$	$gr[r3]$	$gr[r1] = gr[r2]-gr[r3]$ ，将 $gr[r2]$ 、 $gr[r3]$ 相减，将结果写到 $gr[r1]$ ，需要借位时 CF 置 1，否则置 0；结果为 0 时，ZF 置 1，否则置 0；结果 <0 时，NF 置 1，否则置 0。
SUBI	01000	$gr[r1]$	$val2$	$val3$	$gr[r1] = gr[r1]-\{0000_0000, val2, val3\}$ 将 $gr[r1]$ 内容更新为 $gr[r1]-\{0000_0000, val2, val3\}$ ($val2, val3$ 作为低八位)，需要借位时 CF 置 1，否则置 0；结果为 0 时，ZF 置 1，否则置 0；结果 <0 时，NF 置 1，否则置 0。
SUBC	01001	$gr[r1]$	$gr[r2]$	$gr[r3]$	$gr[r1] = gr[r2]-gr[r3]-CF$ 实现带有借位 CF 的 SUB，需要借位时 CF 置 1，否则置 0；结果为 0 时，ZF 置 1，否则置 0；结果 <0 时，NF 置 1，否则置 0。
CMP	01010	$gr[r1]$	$gr[r2]$	$gr[r3]$	判断 $gr[r2]-gr[r3]$ ，需要借位时 CF 置 1，否则置 0；结果为 0 时，ZF 置 1，否则置 0；结果 <0 时，NF 置 1，否则置 0。

② 第二类 Logical & Shift 操作

操作名	操作编码	Operand 1	Operand 2	Operand 3	操作说明
AND	01011	gr[r1]	gr[r2]	gr[r3]	$gr[r1] = gr[r2] \& gr[r3]$, 结果为 0 时, ZF 置 1, 否则置 0; 结果<0 时, NF 置 1, 否则置 0; 产生进位时 CF 置 1, 否则置 0。
OR	01100	gr[r1]	gr[r2]	gr[r3]	$gr[r1] = gr[r2] gr[r3]$, 结果为 0 时, ZF 置 1, 否则置 0; 结果<0 时, NF 置 1, 否则置 0; 产生进位时 CF 置 1, 否则置 0。
XOR	01101	gr[r1]	gr[r2]	gr[r3]	$gr[r1] = gr[r2] \wedge gr[r3]$, 结果为 0 时, ZF 置 1, 否则置 0; 结果<0 时, NF 置 1, 否则置 0; 产生进位时 CF 置 1, 否则置 0。
SLL	01110	gr[r1]	gr[r2]	val3	$gr[r1] = gr[r2] \ll val3$, gr[r2] 逻辑左移 val3 位, 将结果写入 gr[r1], 结果为 0 时, ZF 置 1, 否则置 0; 结果<0 时, NF 置 1, 否则置 0; 产生进位时 CF 置 1, 否则置 0。
SRL	01111	gr[r1]	gr[r2]	val3	$gr[r1] = gr[r2] \gg val3$, gr[r2] 逻辑右移 val3 位, 将结果写入 gr[r1], 结果为 0 时, ZF 置 1, 否则置 0; 结果<0 时, NF 置 1, 否则置 0; 产生进位时 CF 置 1, 否则置 0。
SLA	10000	gr[r1]	gr[r2]	val3	$gr[r1] = gr[r2] \lll val3$, gr[r2] 算术左移 val3 位 (与逻辑左移一样), 将结果写入 gr[r1], 结果为 0 时, ZF 置 1, 否则置 0; 结果<0 时, NF 置 1, 否则置 0; 产生进位时 CF 置 1, 否则置 0。
SRA	10001	gr[r1]	gr[r2]	val3	$gr[r1] = gr[r2] \ggg val3$, gr[r2] 算术右移 val3 位, 将结果写入 gr[r1], 结果为 0 时, ZF 置 1, 否则置 0; 结果<0 时, NF 置 1, 否则置 0; 产生进位时 CF 置 1, 否则置 0。

③ 第三类 Control 操作

操作名	操作编码	Operand 1	Operand 2	Operand 3	操作说明
NOP	00000				无任何操作。
HALT	00001				终止程序。
JUMP	10010		val2	val3	跳转到地址为 {0000_0000, val2, val3} 这一条指令。
JMPR	10011	gr[r1]	val2	val3	跳转到地址为 gr[r1]+{0000_0000, val2, val3} 这一条指令。
BZ	10100	gr[r1]	val2	val3	如果 ZF 为 1, 跳转到地址为 gr[r1]+{0000_0000, val2, val3} 这一条指令。
BNZ	10101	gr[r1]	val2	val3	如果 ZF 为 0, 跳转到地址为 gr[r1]+{0000_0000, val2, val3} 这一条指令。
BN	10110	gr[r1]	val2	val3	如果 NF 为 1, 跳转到地址为 gr[r1]+{0000_0000, val2, val3} 这一条指令。
BNN	10111	gr[r1]	val2	val3	如果 NF 为 0, 跳转到地址为 gr[r1]+{0000_0000, val2, val3} 这一条指令。
BC	11000	gr[r1]	val2	val3	如果 CF 为 1, 跳转到地址为 gr[r1]+{0000_0000, val2, val3} 这一条指令。

BNC	11001	gr[r1]	val2	val3	如果 CF 为 0，跳转到地址为 gr[r1]+{0000_0000, val2, val3} 这一条指令。
-----	-------	--------	------	------	--------------------------------------------------------

3. 指令译码出 reg_A、reg_B 两个操作数。

① 指令译码出 reg_A。

reg_A 译码类型	相关操作
gr[r1]/gr[ir[10:8]]	LDIH、ADDI、SUBI、JMPR、BZ、BNZ、BN、BNN、BC、BNC.
gr[r2]/gr[ir[6:4]]	LOAD、STORE、ADD、ADDC、SUB、SUBC、CMP、AND、OR、XOR、SLL、SRL、SLA、SRA.
无需译码出 reg_A（实现时让其保持原值）	NOP、HALT、JUMP.

② 指令译码出 reg_B。

reg_B 译码类型	相关操作
gr[r3]/gr[ir[2:0]]	ADD、ADDC、SUB、SUBC、CMP、AND、OR、XOR.
{0000_0000_0000, val3}	LOAD、STORE、SLL、SRL、SLA、SRA.
{0000_0000, val2, val3}	ADDI、SUBI、JUMP（ALU 处理 JUMP 时直接将 reg_B 作为结果传递给顶层模块）、JMPR、BZ、BNZ、BN、BNN、BC、BNC.
{val2, val3, 0000_0000}	LDIH.
无需译码出 reg_B（实现时让其保持原值）	NOP、HALT.

4. 其他分类。

其他情况	相关操作
WB 级时需要将运算结果 reg_C1 写到 gr[r1] (gr[wb_ir[10:8]])	LOAD、LDIH、ADD、ADDI、ADDC、SUB、SUBI、SUBC、AND、OR、XOR、SLL、SRL、SLA、SRA.
需要读取 data memory	LOAD.
需要写 data memory	STORE（需要在 ID 级解码，将 gr[r1] 赋给 smdr）。
需要 ALU 进行运算	除 HALT、NOP 操作外，其他 24 种操作。

（三）流水线各级的具体实现

1. IF 级：

IF 模块由指令指针寄存器 pc、指令存储器子模块 Instruction Memory、指令指针选择器 MUX 和一个加法器组成，IF 级接口信息如下表所示：

变量名称	方向	说明
clk	Input	系统时钟信号。
r_st		系统复位信号，高电平有效。
men_ir (7:0) 以及 zf、nf、cf		指令中的跳转操作以及标志操作的三个标志 zf、nf、cf.
i_datain (15:0)		从 instruction memory 读取的当前地址指针 pc 对应的指令。
state		CPU 当前状态。
reg_C (15:0)		下一条指令地址。
pc (15:0)	Output	指令指针寄存器。
id_ir (15:0)		保存当前时钟读取的指令。

2. ID 级：

ID 模块的主要作用是从指令码中解析出指令，并根据解析结果输出操作数及控制指令信号，ID 模

块的接口信息如下表所示：

变量名称	方向	说明
clk	Input	系统时钟信号。
r_st		系统复位信号，高电平有效。
id_ir (15:0)		当前需要执行的指令(获取操作，寄存器地址或者立即数)。
gr (15:0) (7:0)		八个 15 位的通用寄存器。
state		CPU 当前状态。
reg_A(15:0)	Output	ALU 的第一个操作数。
reg_B(15:0)		ALU 的第二个操作数。
smdr(15:0)		用于直接存储的数据。
ex_ir(7:0)		指令缓冲器，保存当前执行指令的重要的高八位。

3. EX 级：

EX 模块主要有 ALU 模块，执行算术、逻辑运算。EX 模块的接口信息如下表所示：

变量名称	方向	说明
clk	Input	系统时钟信号。
r_st		系统复位信号，高电平有效。
ex_ir (7:0)		当前需要执行的指令(获取操作)。
reg_A(15:0)		ALU 的第一个操作数。
reg_B(15:0)		ALU 的第二个操作数。
state		CPU 当前状态。
zf、nf、cf	Output	保存作为操作的三种标志，三种标志均用于指令跳转，cf 同时还用于 ALU 运算，作为加法的进位，减法的借位标志。
reg_C(15:0)		保存 ALU 运算的结果。
smdr1(15:0)、dw		smdr1 保存上一级用于直接存储的数据，dw 用于指示下一级 data_memory 模块将 smdr1 储存。
mem_ir(7:0)		指令缓冲器，保存当前执行指令重要的高八位。

4. MEM 级：

MEM 模块主要由数据内存模块（用于存储数据或读取数据）还有数据选择器组成。MEM 模块的接口信息如下表所示：

变量名称	方向	说明
clk	Input	系统时钟信号
r_st		系统复位信号，高电平有效
mem_ir (7:0)		当前需要执行的指令(获取操作)
reg_C(15:0)		上一级 ALU 运算的结果，data_memory 模块要存储的内容的地址或者下一级用于回写寄存器的内容

dw、smdr1(15:0)	Output	dw 指示 data_memory 模块将 smdr1 内容储存下来
state		CPU 当前状态
reg_C1(15:0)		储存下一级将要用于回写寄存器的内容
wb_ir(7:0)		指令缓冲器，保存当前执行指令重要的高八位

5. WB 级:

WB 模块比较简单，执行通用寄存器的回写，将结果存储到通用寄存器上。WB 模块的接口信息如下表所示:

变量名称	方向	说明
clk	Input	系统时钟信号。
r_st		系统复位信号，高电平有效。
wb_ir (7:0)		当前需要执行的指令(获取操作和要回写的寄存器的地址)。
reg_C1(15:0)		回写寄存器的内容。
gr(15:0) (7:0)		CPU 八个 15 位的通用寄存器。
state		CPU 当前状态。

(四) 处理 Hazards

1. **Summary:** Pipelining provides high throughput, but does not handle data dependences easily. Data dependences cause data hazards.

2. 种类:

① **Structure hazards:** A required resource is busy. 因为此次的 CPU 将指令与数据放在不同的内存区域里面 (instruction memory & data memory)，所以不存在这个问题。

② **Data hazard:** Need to wait for previous instruction to complete its data read/write. 产生这一冲突的原因是前后指令的相关性，当前指令的执行需要用到之前指令执行的结果，但之前指令未执行完毕，其结果仍未存入相应的通用寄存器，导致当前指令访问相应的寄存器时寄存器的值未刷新，仍为旧值，从而导致当前指令执行出错。

Data hazards can be solved by:

- ◆ **Software:** Insert 3 NOPs. (编译器处理时解决)
- ◆ **Hardware:** Data forward, for arithmetic operations.
Data forward & Stall, for LOAD.

为了给编译器减负，本次试验将从硬件层面上解决数据冲突。

③ **Control hazard:** Deciding on control action depends on previous instruction. 倘若出现跳转指令，其后面的若干条指令本来不应该被执行，但由于跳转指令从执行到完毕需要五个周期，导致在 pc 跳转之前，跳转指令后面的几条指令被执行，而这有可能改变 CPU 内部相应寄存器的值 (主要是通用寄存器 gr)，最终有可能导致执行出现异常，这是我们不希望看到的。

3. 如何解决 hazard:

本次试验主要解决的是上述三种冲突中的 data hazard。首先，应该明确，只有那些有刷新通用寄存器的操作在某些需要访问通用寄存器的操作之前被执行才有可能出现数据冲突。

需要访问通用寄存数完成操作的相关操作:

访问寄存器	相关操作
gr[r1]	STORE、LDIH、ADDI、SUBI、JMPR、BZ、BNZ、BN、BNN、BC、BNC.
gr[r2]	LOAD、STORE、ADD、ADDC、SUB、SUBC、CMP、AND、OR、XOR、SLL、SRL、SLA、SRA.
gr[r3]	ADD、ADDC、SUB、SUBC、CMP、AND、OR、XOR.

备注	STORE 操作需要先后访问 gr[r1]以及 gr[r2]; ADD、ADDC、SUB、SUBC、CMP、AND、OR、XOR 操作需要先后访问 gr[r2]以及 gr[r3].
----	-----------------------------------------------------------------------------------------------

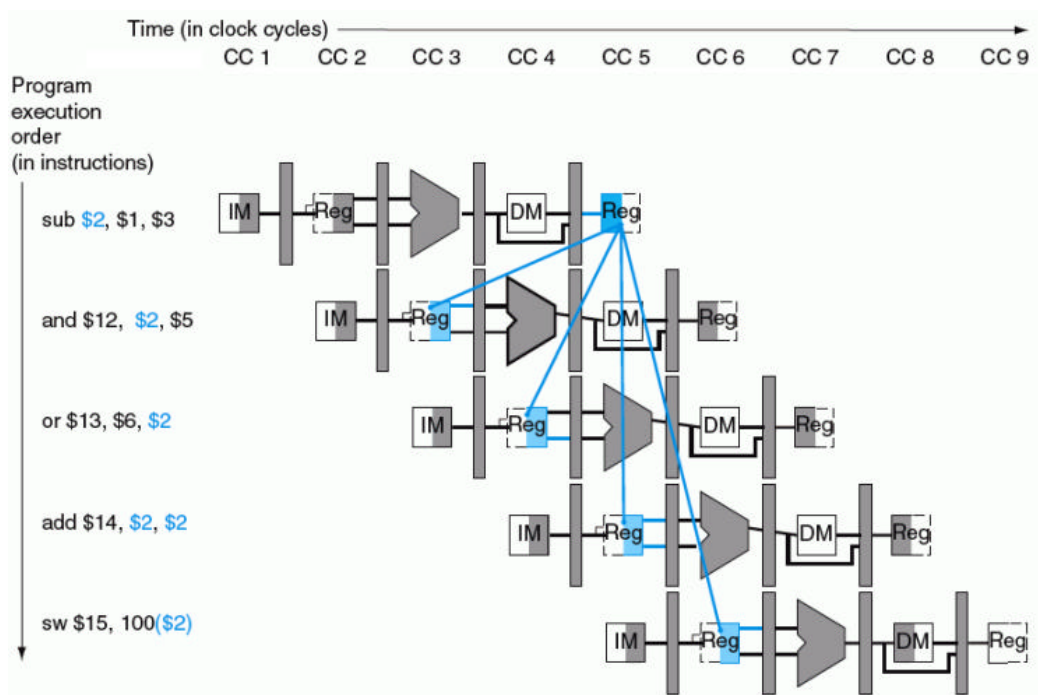
需要刷新寄存器的操作:

情况	相关操作
WB 级时需要将结果写到 gr[r1](gr[wb_ir[10:8]])	LOAD、LDIH、ADD、ADDI、ADDC、SUB、SUBI、SUBC、AND、OR、XOR、SLL、SRL、SLA、SRA.
备注	刷新的寄存器的编号永远是 operand1, 清楚这一点可以为 data forwarding 的实现排除很多种不用考虑的情况。

解决 data hazard 的一个核心就是 data forwarding, 说白了, 就是提前使用那些已经出现的运算结果 (最终会被写回到相应寄存器), 使得后面的操作都能获取正确的操作数, 得到正确的运行结果。

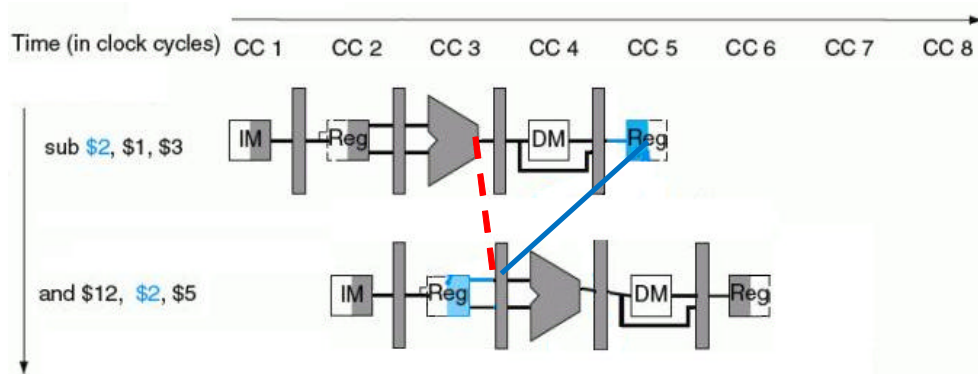
① 之前刷新寄存器的操作不为 LOAD 操作:

通过数据转发解决, 实际上是在原有基础上改进 ID 级 (代码实现上是对 smdr、reg_A、reg_B 这些缓冲寄存器在赋值做进一步选择)。如下图:



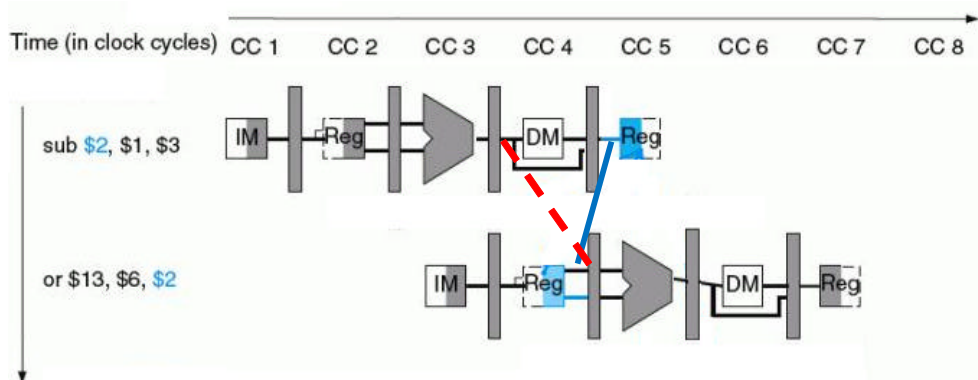
【分析】sub 操作的结果要存入 gr[2], 但是前后紧接着 sub 操作的三条指令 and、or、add 都需要访问 gr[2] 的值, 数据发生冲突。

- 一阶数据相关, 第 I 条指令需要访问的寄存器与第 I - 1 条指令 (即上一条指令) 刷新的寄存器相重, 导致数据冲突。(如下图):



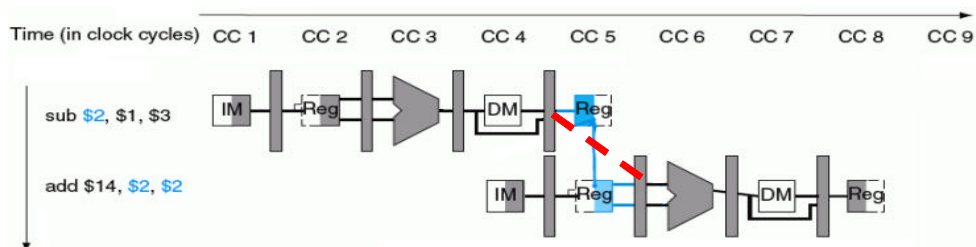
解决的办法是在 ID 级获取操作数时, 直接将 ALUo (即将在两个周期后被写会相应通用寄存器的值) 转发过来。

- b. 二阶数据相关，第 I 条指令需要访问的寄存器与第 I - 2 条指令（即上上条指令）刷新的寄存器相重，导致数据冲突。（如下图）：



解决的办法是在 ID 级获取操作数时，直接将 reg_C（即将在一个周期后被写会相应通用寄存器的值）转发过来。

- c. 三阶数据相关，第 I 条指令需要访问的寄存器与第 I - 3 条指令（即上上上条指令）刷新的寄存器相重，导致数据冲突。（如下图）：



解决的办法是在 ID 级获取操作数时，直接将 reg_C1（将在本周期后被写会相应通用寄存器的值）转发过来。

② 之前刷新寄存器的操作为 **LOAD** 操作：

我们知道，LOAD 操作是将 data memory 里面某个地址的数据写入通用寄存器，这个数据最早出现在 MEM 级的 d_datain，其次是在 WB 级的 reg_C1(如下图)：

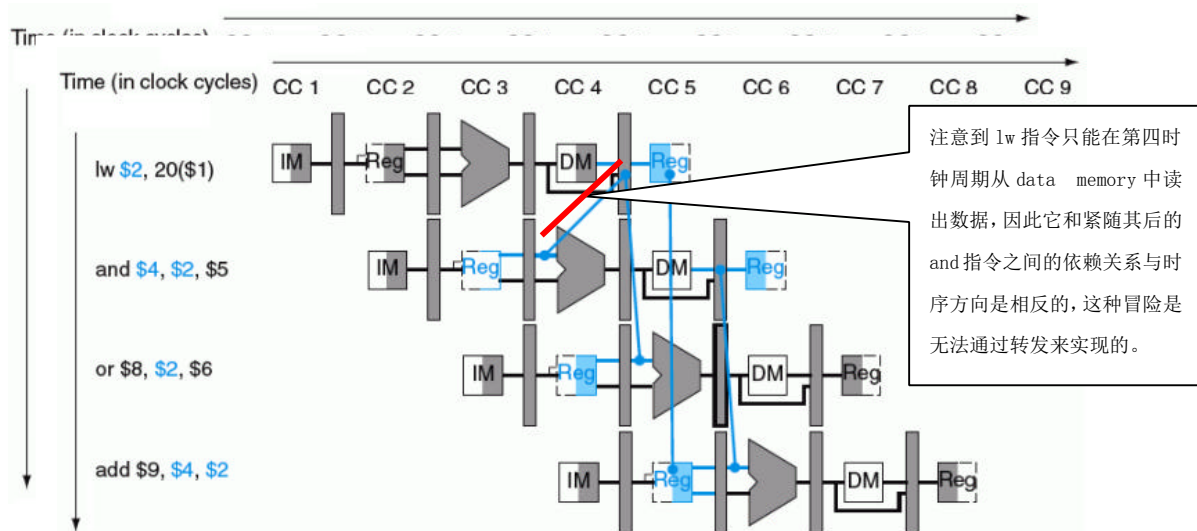


Figure. 数据冒险与阻塞实例图

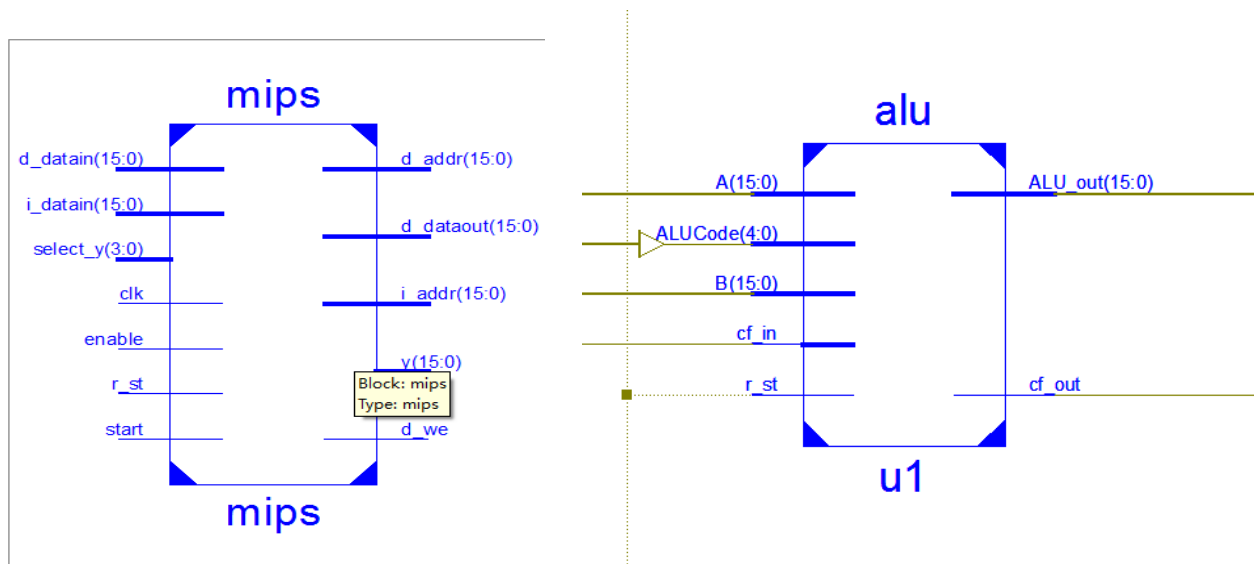
为解决此类数据冒险，我们引入流水线阻塞。当冒险条件成立时，在 LOAD 指令和下一条指令之间插入阻塞，即流水线气泡（bubble），使后一条指令延迟一个时钟周期执行，然后通过 d_datain、reg_C1 的数据转发解决剩下的问题。其中 d_datain 为二阶相关的数据冲突、reg_C1 为三阶相关的数据冲突。

实现上述过程需要在原有的设计基础上改进 IF、ID 两级：

- a. IF 级：当检测到上述冲突发生，需要阻塞流水线时，hold 住指令计数器 pc 的值，同时插入气泡，即一条没有实际意义的操作；
- b. ID 级：根据是二阶还是三阶数据相关冲突对 smdr、reg_A、reg_B 的赋值做进一步选择。

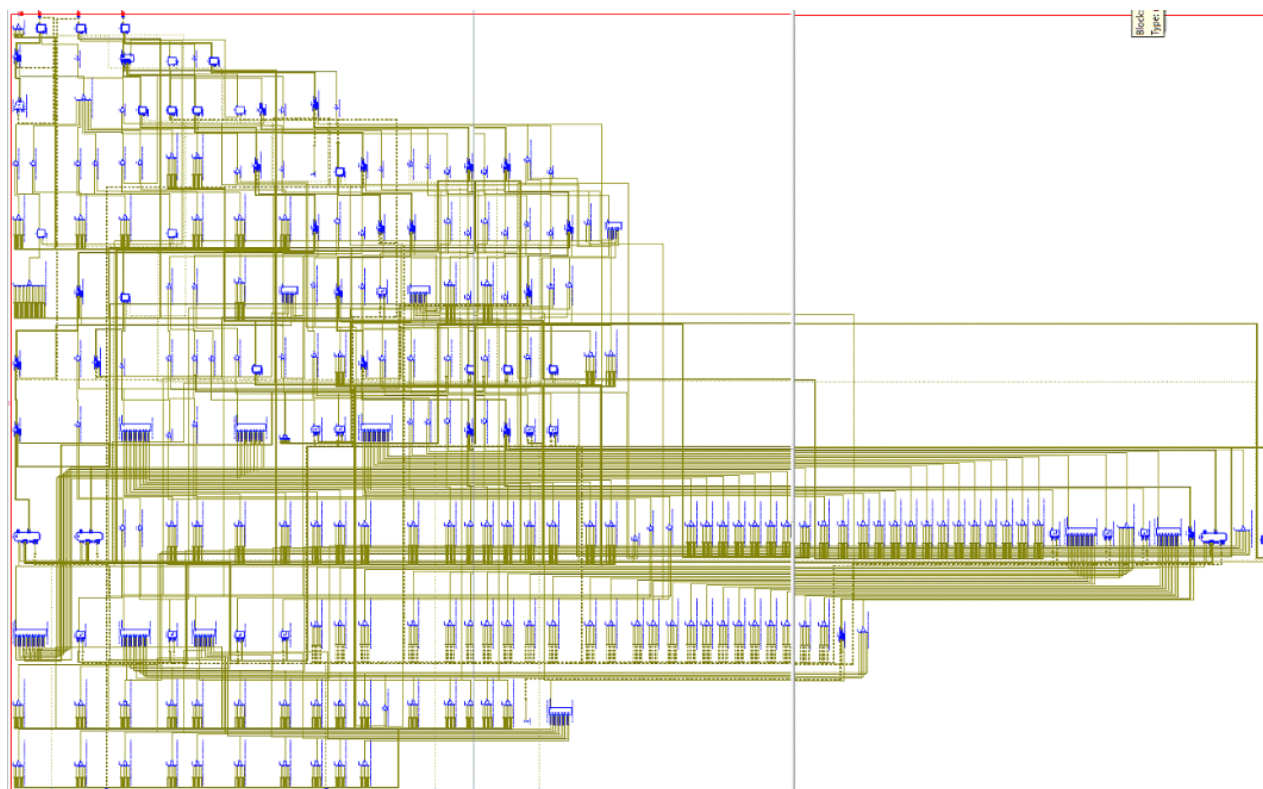
三、实验结果

1、综合的 RTL 电路图：



顶层模块 mips—端口声明

alu 模块—端口声明



整体的 RTL 图

如上图所示，RTL 图比较复杂，因为 mips 本身就涉及成百个元器件，本次试验更加侧重与逻辑上的正确性，即仿真结果的验证。

2、文本仿真数据显示：

为更好的测试 mips 的功能，建立新模块 cpu_top，模块由三个子模块组成：

1. 核心模块 mips，我们需要测试的模块；
2. Instruction memory 模块，存放我们测试的所有指令（见 **Figure. 指令模块**）；
3. Data memory 模块，存放我们需要用到的数据（见 **Figure. 数据模块**）。

```
begin
memory[0] <= {`LOAD, `gr1, 1'b1, `gr0, 4'b0000};
memory[1] <= {`LOAD, `gr2, 1'b1, `gr0, 4'b0001};
memory[2] <= {`ADD, `gr3, 1'b0, `gr1, 1'b0, `gr2};
memory[3] <= {`LOAD, `gr4, 1'b1, `gr0, 4'b0010};
memory[4] <= {`LOAD, `gr5, 1'b1, `gr0, 4'b0011};
memory[5] <= {`SUB, `gr6, 1'b1, `gr4, 1'b0, `gr5};
memory[6] <= {`ADD, `gr3, 1'b0, `gr1, 1'b0, `gr2};
memory[7] <= {`ADDC, `gr3, 1'b0, `gr1, 1'b0, `gr2};
memory[8] <= {`NOP, 11'b000_0000_0000}; // reset cf.
memory[9] <= {`SUB, `gr6, 1'b1, `gr4, 1'b0, `gr5};
memory[10] <= {`SUBC, `gr6, 1'b1, `gr4, 1'b0, `gr5};
memory[11] <= {`AND, `gr7, 1'b0, `gr1, 1'b0, `gr2};
memory[12] <= {`OR, `gr7, 1'b0, `gr1, 1'b0, `gr2};
memory[13] <= {`XOR, `gr7, 1'b0, `gr1, 1'b0, `gr2};
memory[14] <= {`SLL, `gr7, 1'b0, `gr2, 4'b0001};
memory[15] <= {`SRL, `gr7, 1'b0, `gr2, 4'b0001};
memory[16] <= {`SLA, `gr7, 1'b0, `gr2, 4'b0001};
memory[17] <= {`SRA, `gr7, 1'b0, `gr2, 4'b0001};
memory[18] <= {`STORE, `gr5, 1'b0, `gr0, 4'b1000};
memory[19] <= {`NOP, 11'b000_0000_0000};
memory[20] <= {`NOP, 11'b000_0000_0000};
memory[21] <= {`LOAD, `gr7, 1'b0, `gr0, 4'b1000};
memory[22] <= {`ADDI, `gr0, 4'b0000, 4'b1111};
memory[23] <= {`ADDC, `gr4, 1'b0, `gr3, 1'b0, `gr0};
memory[24] <= {`SUB, `gr3, 1'b0, `gr4, 1'b0, `gr2};
memory[25] <= {`SUBI, `gr3, 4'b0000, 4'b0000};
memory[26] <= {`CMP, `gr7, 1'b0, `gr3, 1'b0, `gr0};
memory[27] <= {`ADD, `gr3, 1'b0, `gr1, 1'b0, `gr2};
memory[28] <= {`ADD, `gr3, 1'b0, `gr1, 1'b0, `gr2};
memory[31] <= {`HALT, 11'b000_0000_0000};
```

Figure. 指令内存模块

```
(r_st)
begin
data_o <= 16'b0000_0000_0000_0000;
data[0] <= 16'h3c00;
data[1] <= 16'hffff; // 3c00 + ffff = 3bff(进位1)
data[2] <= 16'h3cab;
data[3] <= 16'haaaa; // 3cab - aaaa = 9201(借位1)
data[4] <= 16'b0000_0000_0000_0000;
data[5] <= 16'b0000_0000_0000_0000;
data[6] <= 16'b0000_0000_0000_0000;
data[7] <= 16'b0000_0000_0000_0000;
data[8] <= 16'b0000_0000_0000_0000;
data[9] <= 16'b0000_0000_0000_0000;
data[10] <= 16'b0000_0000_0000_0000;
data[11] <= 16'b0000_0000_0000_0000;
data[12] <= 16'b0000_0000_0000_0000;
data[13] <= 16'b0000_0000_0000_0000;
data[14] <= 16'b0000_0000_0000_0000;
data[15] <= 16'b0000_0000_0000_0000;
end
```

Figure. 数据内存模块

【分析】

1. 根据指令，data[0]、data[1]将分别被 LOAD 进 gr[1]、gr[2]，然后用于加法运算，结果存入 gr[3]；data[2]、data[3]将分别被 LOAD 进 gr[4]、gr[5]，然后用于加法运算，结果存入 gr[6]；
2. 上述 **Figure. 指令内存模块**所有指令文本仿真结果如下图（从左到右每一列分别表示：仿真时间 Time(ps)、指令计数器 pc、当前执行指令 id_ir、ALU 的两个操作数 reg_A、reg_B，结果 reg_C，存储数据的使能信号 d_we、reg_C1、八个通用寄存器 gr[0]、gr[1]、gr[2]、gr[3]、gr[4]、gr[5]、gr[6]、gr[7]，ALU 产生的 cf、zf、nf、指令地址 i_addr、下一条指令 i_datain、数据地址 d_addr、数据内容 d_datain、用于直接存储的数据内容 smdr）：

Time/ps	IF-pc	IF-idir	ID-reg_A	ID-reg_B	EX-reg_C	EX-d_we	MEM-reg_C1	gr0	gr1	gr2	gr3	gr4	gr5	gr6	gr7	EX-cf	EX-zf	EX-nf	IF-i_addr	IF-i_datain	MEM-d_addr	MEM-d_datain	ID-smdr
10000	xxxx	xxxxxxxxxxxxxxxx	xxxx	xxxx	xxxx	x	xxxx	0000	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	x	x	x	xxxx	xxxxxxxxxxxxxxxx	xxxx	xxxx	xxxx
11000	xxxx	xxxxxxxxxxxxxxxx	xxxx	xxxx	xxxx	x	xxxx	0000	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	x	x	x	xxxx	xxxxxxxxxxxxxxxx	xxxx	0000	xxxx
11500	0000	0000000000000000	0000	0000	0000	0	0000	0000	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	0	0	0	0000	1101000110000000	0000	0000	0000
12000	0000	0000000000000000	0000	0000	0000	0	0000	0000	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	0	0	0	0000	1101000110000000	0000	3c00	0000
15500	0001	1101000110000000	0000	0000	xxxx	0	0000	0000	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	x	0	0	0001	1101000110000000	xxxx	xxxx	0000
16500	0002	1101000110000000	0000	0000	xxxx	0	0000	0000	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	x	0	0	0002	0010001100010010	xxxx	xxxx	0000
17500	0002	1111010100000001	0000	0001	0000	0	0000	0000	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	0	1	0	0002	0010001100010010	0000	3c00	0000
18500	0003	0010001100010010	0000	0001	0001	0	3c00	0000	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	0	0	0	0003	1101010010000010	0001	ffff	0000
19500	0004	1101010010000010	3c00	ffff	xxxx	0	ffff	0000	3c00	ffff	xxxx	xxxx	xxxx	xxxx	xxxx	x	0	0	0004	1101010110000011	xxxx	xxxx	0000
20500	0005	1101010110000011	0000	0002	3bff	0	3bff	0000	3c00	ffff	xxxx	xxxx	xxxx	xxxx	xxxx	0	0	0	0005	0011110110001010	3bff	xxxx	0000
21500	0005	1111101100000011	0000	0003	0002	0	0002	0	3bff	0000	3c00	ffff	3bff	xxxx	xxxx	0	0	0	0005	0011110110001010	0002	3cab	0000
22500	0006	0011110110001001	0000	0003	0003	0	3cab	0000	3c00	ffff	3bff	xxxx	xxxx	xxxx	xxxx	0	0	0	0006	0010001100010010	0003	aaaa	0000
23500	0007	0010001100010010	3cab	aaaa	xxxx	0	aaaa	0000	3c00	ffff	3bff	3cab	xxxx	xxxx	xxxx	x	0	0	0007	0011001100010010	xxxx	xxxx	0000
24500	0008	0011001100010010	3c00	ffff	9201	0	aaaa	0000	3c00	ffff	3bff	3cab	aaaa	xxxx	xxxx	1	0	1	0008	0000000000000000	9201	xxxx	0000
25500	0009	0000000000000000	3c00	ffff	3bff	0	9201	0000	3c00	ffff	3bff	3cab	aaaa	xxxx	xxxx	1	0	0	0009	0011110110001010	3bff	xxxx	0000
26500	000a	0011110110001010	3c00	ffff	3c00	0	3bff	0000	3c00	ffff	3bff	3cab	aaaa	9201	xxxx	1	0	0	000a	0100111011000101	3c00	xxxx	0000
27500	000b	0100111011000101	3cab	aaaa	xxxx	0	3c00	0000	3c00	ffff	3bff	3cab	aaaa	9201	xxxx	x	0	0	000b	0101111000100010	xxxx	xxxx	0000
28500	000c	0101111000100010	3cab	aaaa	9201	0	3c00	0000	3c00	ffff	3c00	3cab	aaaa	9201	xxxx	1	0	1	000c	0110011000100010	9201	xxxx	0000
29500	000d	0110011000100010	3c00	ffff	9200	0	9201	0000	3c00	ffff	3c00	3cab	aaaa	9201	xxxx	1	0	1	000d	0110111000100010	9200	xxxx	0000
30500	000e	0110111000100010	3c00	ffff	3c00	0	9200	0000	3c00	ffff	3c00	3cab	aaaa	9201	xxxx	0	0	0	000e	0111011000100001	3c00	xxxx	0000
31500	000f	0110111000100001	3c00	ffff	ffff	0	3c00	0000	3c00	ffff	3c00	3cab	aaaa	9200	xxxx	0	0	1	000f	0111110010000001	ffff	xxxx	0000
32500	0010	0111110110001000	ffff	0001	c3ff	0	ffff	0000	3c00	ffff	3c00	3cab	aaaa	9200	3c00	0	0	1	0010	1000011001000001	c3ff	xxxx	0000
33500	0011	1000011001000001	ffff	0001	ffffa	0	c3ff	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff	1	0	1	0011	1000111001000001	ffffa	xxxx	0000
34500	0012	1000111001000001	ffff	0001	7fff	0	ffffa	0000	3c00	ffff	3c00	3cab	aaaa	9200	c3ff	0	0	0	0012	0001010100001000	7fff	xxxx	0000
35500	0013	0001010100001000	ffff	0001	ffffa	0	7fff	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff	1	0	1	0013	0000000000000000	ffffa	xxxx	0000
36500	0014	0000000000000000	0000	0008	ffff	0	ffff	0000	3c00	ffff	3c00	3cab	aaaa	9200	7fff	1	0	1	0014	0000000000000000	ffff	xxxx	aaaa
37500	0015	0000000000000000	0000	0008	0008	1	ffff	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff	0	0	0	0015	1101011000010000	0008	xxxx	aaaa
38500	0016	1101011000010000	0000	0008	xxxx	0	ffff	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff	x	0	0	0016	0010100000000000	xxxx	xxxx	aaaa
39500	0017	0010100000000111	0000	0008	xxxx	0	ffff	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff	x	0	0	0017	0011010000110000	xxxx	xxxx	aaaa
40500	0018	0011010000110000	0000	000f	0008	0	ffff	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff	0	0	0	0018	0011010100000010	000f	aaaa	aaaa
41500	0019	0011101010000010	3c00	000f	000f	0	aaaa	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff	0	0	0	0019	0100001100000000	000f	0000	aaaa
42500	001a	0100001100000000	3c0f	ffff	3c0f	0	000f	0000	3c00	ffff	3c00	3cab	aaaa	9200	aaaa	0	0	0	001a	0101011001100000	3c0f	xxxx	aaaa
43500	001b	0101011001100000	3c10	0000	3c10	0	3c0f	000f	3c00	ffff	3c00	3cab	aaaa	9200	aaaa	1	0	0	001b	0010001100010010	3c10	xxxx	aaaa
44500	001c	0010001100010010	3c10	000f	3c10	0	3c10	000f	3c00	ffff	3c00	3c0f	aaaa	9200	aaaa	0	0	0	001c	0010001100010010	3c10	xxxx	aaaa

Figure. 仿真结果

【总体分析】

- a. id_ir 与 i_datain 前后总是相隔一个时钟周期；
- b. d_addr 与 reg_C1 应相互对应，总是符合已有的数据内存中的数据；

- c. d_datain 应与数据内存中的数据相对应;
d. reg_C 在大多数情况下 (操作 LOAD 除外) 总是与 reg_C1 相对应。

【具体解析】

Finished circuit initialization process.					
Time/ps	IF-pc	IF-id_ir	ID-reg_A	ID-reg_B	EX-reg_C
100000	xxxx	xxxxxxxxxxxxxxxx	xxxx	xxxx	xxxx
110000	xxxx	xxxxxxxxxxxxxxxx	xxxx	xxxx	xxxx
115000	0000	0000000000000000	0000	0000	0000
120000	0000	0000000000000000	0000	0000	0000
155000	0001	1101000110000000	0000	0000	xxxx
165000	0002	1101001010000001	0000	0000	xxxx
175000	0002	1111101010000001	0000	0001	0000
185000	0003	0010001100010010	0000	0001	0001
195000	0004	1101010010000010	3c00	ffff	xxxx
205000	0005	1101010110000011	0000	0002	3bff

Figure. hold pc

gr0	gr1	gr2	gr3	gr4	gr5
0000	xxxx	xxxx	xxxx	xxxx	xxxx
0000	xxxx	xxxx	xxxx	xxxx	xxxx
0000	xxxx	xxxx	xxxx	xxxx	xxxx
0000	xxxx	xxxx	xxxx	xxxx	xxxx
0000	xxxx	xxxx	xxxx	xxxx	xxxx
0000	xxxx	xxxx	xxxx	xxxx	xxxx
0000	3c00	xxxx	xxxx	xxxx	xxxx
0000	3c00	ffff	xxxx	xxxx	xxxx
0000	3c00	ffff	xxxx	xxxx	xxxx
0000	3c00	ffff	3bff	xxxx	xxxx
0000	3c00	ffff	3bff	3cab	xxxx
0000	3c00	ffff	3bff	3cab	aaaa
0000	3c00	ffff	3bff	3cab	aaaa

Figure. hold pc 's gr

1) 见上图 **Figure. hold pc**, At 155000ps, id_ir = 16' b11010_001_1000_0000, 为 LOAD 操作, 因为 gr[0] = 16' b0, 这一条指令解码的两个操作数 reg_A、reg_B 均为 0, at 165000ps, 可以看出, 两个操作数解码正确。pc = 0002, id_ir = 16' b11010_010_1000_0001, 连续两条 LOAD 操作, 后一条指令解码的两个操作数 reg_A、reg_B 分别为 0 和 1, 两条指令最终将 data[0] 的值 LOAD 进 gr[1], 将 data[1] 的值 LOAD 进 gr[2]。图 **Figure. hold pc 's gr**, gr[1]、gr[2] 先后被刷新, 仿真正确。At 175000ps, pc 被 hold 住, 根据 at 185000ps, 原来紧跟着两条 LOAD 指令后, 要执行 ADD 操作 (pc = 0003, id_ir = 16' b00100_110_0001_0010), 这一条指令解码的两个操作数 reg_A、reg_B 分别为 gr[1], gr[2], 我们知道 gr[2] 刚好是后一条 LOAD 指令要刷新的通用寄存器, 这就产生了冲突, 解决的办法是 hold 住 pc, 同时让下一条指令无效执行。图 **Figure. hold pc** 刚好反映了这一点。最终 ADD 指令成功执行, 两个操作数 reg_A、reg_B 成功解码, 运算结果为 16' h3bff, 进位 cf 变为 1 (见图 **Figure. 仿真结果**), 最终运算结果成功存入 gr[3]。仿真正确。

2)

Time/ps	IF-pc	IF-id_ir	ID-reg_A	ID-reg_B	EX-reg_C
235000	0007	0010001100010010	3cab	aaaa	xxxx
245000	0008	0011001100010010	3c00	ffff	9201
255000	0009	0000000000000000	3c00	ffff	3bff
265000	000a	0011111011000101	3c00	ffff	3c00
275000	000b	0100111011000101	3cab	aaaa	xxxx
285000	000c	010111100010010	3cab	aaaa	9201
295000	000d	0110011100010010	3c00	ffff	0000

Figure. 指令解码

gr0	gr1	gr2	gr3	gr4	gr5	gr6	gr7
0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff
0000	3c00	ffff	3c00	3cab	aaaa	9200	7fff
0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff
0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff
0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff
0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff
0000	3c00	ffff	3c00	3cab	aaaa	9200	aaaa
000f	3c00	ffff	3c00	3cab	aaaa	9200	aaaa

Figure. gr[0]-gr[7]

见上图 **Figure. 指令解码**, at 255000ps, 操作为 NOP, 不做任何操作, 用于将进位 cf 重置 (见图 **Figure. 仿真结果**), at 265000ps, id_ir = 16' b00111_011_0001_0010, 执行 SUB 操作, 这一条指令解码的两个操作数 reg_A、reg_B 分别为 gr[1]、gr[2], 结合图 **Figure. 指令解码** 和图 **Figure. gr[0]-gr[7]**, 即 16' h3cab, 16' haaaa 解码正确, 同时运算也正确, 为 9201, 借位 cf 置一。仿真正确。

3)

Time	pc	id_ir	reg_A	reg_B	reg_C	gr0	gr1	gr2	gr3	gr4	gr5	gr6	gr7
200000	000a	0011111011000101	3c00	1111	3c00	0000	3c00	ffff	3c00	3cab	aaaa	9200	xxxx
275000	000b	0100111011000101	3cab	aaaa	xxxx	0000	3c00	ffff	3c00	3cab	aaaa	9200	3c00
285000	000c	010111100010010	3cab	aaaa	9201	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff
295000	000d	0110011100010010	3c00	ffff	9200	0000	3c00	ffff	3c00	3cab	aaaa	9200	c3ff
305000	000e	0110111100010010	3c00	ffff	3c00	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff
315000	000f	0111011100100001	3c00	ffff	ffff	0000	3c00	ffff	3c00	3cab	aaaa	9200	7fff
325000	0010	0111111100100001	ffff	0001	c3ff	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff
335000	0011	1000011100100001	ffff	0001	ffff	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff
345000	0012	1000111100100001	ffff	0001	7fff	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff
355000	0013	0001010100001000	ffff	0001	ffff	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff
365000	0014	0000000000000000	0000	0008	ffff	0000	3c00	ffff	3c00	3cab	aaaa	9200	aaaa
375000	0015	0000000000000000	0000	0008	0008	000f	3c00	ffff	3c00	3cab	aaaa	9200	aaaa

Figure. 指令解码

Figure. gr[0]-gr[7]

见上图 **Figure. 指令解码**, from 285000ps to 355000ps, 连续执行 AND、OR、XOR、SLL、SRL、SLA、SRA 操作: 三个位逻辑运算的操作数 reg_A、reg_B 均为 16' h3c00、16' hffff, 运行结果分别为 16' h3c00、16' hffff、16' bc3ff, cf 分别为 0、0、1, 通过计算器运算可知, 运行结果正确; 四个移位操作的操作数 reg_A、reg_B 均为 16' hffff 和 1, 运行结果分别为 16' hffffe、16' h7fff、16' hffffe、16' hffff, 进位分别为 1、0、1、1, 运行结果同样正确。最后 (见上图 **Figure. gr[0]-gr[7]**), 运算结果前后不断刷新通用寄存器 gr7 的值。仿真正确。

4)

Time	pc	id_ir	reg_A	reg_B	reg_C	gr0	gr1	gr2	gr3	gr4	gr5	gr6	gr7	cf
375000	0015	0000000000000000	0000	0008	0008	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff	0
385000	0016	1101011100001000	0000	0008	xxxx	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff	x
395000	0017	0010100000001111	0000	0008	xxxx	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff	0
405000	0018	0011010000110000	0000	000f	0008	0000	3c00	ffff	3c00	3cab	aaaa	9200	ffff	0
415000	0019	0011101101000010	3c00	000f	000f	0000	3c00	ffff	3c00	3cab	aaaa	9200	aaaa	0
425000	001a	0100001100000000	3c0f	ffff	3c0f	000f	3c00	ffff	3c00	3cab	aaaa	9200	aaaa	1
435000	001b	0101011100110000	3c10	0000	3c10	000f	3c00	ffff	3c00	3c0f	aaaa	9200	aaaa	0
445000	001c	0010001100010010	3c10	000f	3c10	000f	3c00	ffff	3c10	3c0f	aaaa	9200	aaaa	0
455000	001d	0010001100010010	3c00	ffff	3c01	000f	3c00	ffff	3c10	3c0f	aaaa	9200	aaaa	1
465000	001e	xxxxxxxxxxxxxxxx	3c00	ffff	3bff	000f	3c00	ffff	3c10	3c0f	aaaa	9200	aaaa	1
475000	001f	xxxxxxxxxxxxxxxx	3c00	ffff	3bff	000f	3c00	ffff	3bff	3c0f	aaaa	9200	aaaa	x
485000	0020	0000100000000000	3c00	ffff	xxxx	000f	3c00	ffff	3bff	3c0f	aaaa	9200	aaaa	x

Figure. 指令解码

Figure. gr[0]-gr[7]

见图 **Figure. 指令解码**, from 395000ps to 435000ps, 连续执行 ADDI、ADDC、SUB、SUBI、CMP 五条操作 (395000ps: {`ADDI, `gr0, 4'b0000, 4'b1111};
405000ps: {`ADDC, `gr4, 1'b0, `gr3, 1'b0, `gr0};
415000ps: {`SUB, `gr3, 1'b0, `gr4, 1'b0, `gr2};
425000ps: {`SUBI, `gr3, 4'b0000, 4'b0000};
435000ps: {`CMP, `gr7, 1'b0, `gr3, 1'b0, `gr0};),

可以明显看出前后指令间都存在冲突, 这一块主要用测试能否正确处理一般数据冲突。

ADDI 指令的操作数 reg_A、reg_B 为 16' h0000、16' h000f, 运行结果为 16' h000f, cf = 0;

ADDC 指令的操作数 reg_A、reg_B 为 16' h3c00、16' h000f, 运行结果为 16' h3c0f, cf = 0;

SUB 指令的操作数 reg_A、reg_B 为 16' h3c0f、16' hffff, 运行结果为 16' h3c10, cf = 1;

SUBI 指令的操作数 reg_A、reg_B 应为 16' h3c10、16' h0000, 运行结果为 16' h3c10, cf = 0;

CMP 指令的操作数 reg_A、reg_B 应为 16' h3c10、16' h000f, 运行结果为 16' h3c01, cf = 0, zf = 0, nf = 0; 存在数据冲突的前后指令的操作数均能成功解码。通过计算器运算也可知, 运行结果正确; 最后 (见上图 **Figure. gr[0]-gr[7]**), 运算结果先后刷新通用寄存器的值。仿真正确。

3、资源开销及时序报告

a. 时序报告:

```
Timing Summary:
Speed Grade: -3

Minimum period: 8.727ns (Maximum Frequency: 114.584MHz)
Minimum input arrival time before clock: 9.034ns
Maximum output required time after clock: 5.218ns
Maximum combinational path delay: No path found

Timing Details:
All values displayed in nanoseconds (ns)

=====
Timing constraint: Default period analysis for Clock 'clk'
Clock period: 8.727ns (frequency: 114.584MHz)
Total number of paths / destination ports: 65366 / 485
=====
Delay: 8.727ns (Levels of Logic = 12)
Source: reg_B_3 (FF)
Destination: zf (FF)
Source Clock: clk rising
Destination Clock: clk rising
```

Figure. Timing Report-1

```
Timing Summary:
Speed Grade: -3

Minimum period: 5.884ns (Maximum Frequency: 169.964MHz)
Minimum input arrival time before clock: 5.873ns
Maximum output required time after clock: 5.442ns
Maximum combinational path delay: No path found

Timing Details:
All values displayed in nanoseconds (ns)

=====
Timing constraint: Default period analysis for Clock 'clk'
Clock period: 5.884ns (frequency: 169.964MHz)
Total number of paths / destination ports: 9827 / 493
=====
```

Figure. Timing Report-2

【分析】Timing Report-1 所示，实现的 mips 的最高频率为 114.584MHz。Timing Report-2 为 ALU 模块中的 case 语句中的 default 注释后的时序报告，疯狂提升至 169.964MHz。ISE 提示 Latch，问过 TA，解释是：对于产生 Latch 的该条路径被忽略，使得最短路径缩短，频率提升。ALU 模块是组合逻辑，case 语句没有 default 的话，确实会产生 latch，但这样使得 EX 级的 reg_C 以及 cf 缓冲寄存器的值不发生改变，如 NOP 操作就本意就是要系统的所有寄存器在当前操作维持不变。但考虑到对于产生 latch 的设计，在板上仿真时容易产生时序混乱。权衡后还是采用注释掉 default 的设计，此处需要进一步改进以及进行板上调试。

b. 资源开销报告：

```
Device utilization summary:
-----
Selected Device : 6s1x16csg324-3

Slice Logic Utilization:
Number of Slice Registers:      195 out of 18224    1%
Number of Slice LUTs:          598 out of 9112      6%
    Number used as Logic:      550 out of 9112      6%
    Number used as Memory:     48 out of 2176       2%
    Number used as RAM:        48
Slice Logic Distribution:
Number of LUT Flip Flop pairs used: 659
Number with an unused Flip Flop: 464 out of 659    70%
Number with an unused LUT: 61 out of 659           9%
Number of fully used LUT-FF pairs: 134 out of 659   20%
Number of unique control sets: 8
IO Utilization:
Number of IOs: 105
Number of bonded IOBs: 85 out of 232             36%
Specific Feature Utilization:
Number of BUFG/BUFGCTRLs: 1 out of 16            6%
```

【分析】原先的 ex_ir, mem_ir, wb_ir 均为 16 位，将 id_ir 完全传递下去，后来发现，id_ir 的低八位在后面三级的操作中毫无意义，所以最终的 ex_ir, mem_ir, wb_ir 均取 8 位，保存 id_ir[15:11]——5 位操作码、id_ir[10:8]——需要刷新的寄存器的编号。改进后资源开销减少，频率也有较小幅度的提升。

四、实验感想

◆ 本次实验代码编写部分较为简单，实验指导中已经很明确的给出了各个模块中的工作原理与信号之间的逻辑关系，但由于连接信号众多，信号名称相近，很容易由于录入粗心而产生错误，这也是本次实验最主要的错误来源。

◆ 我认为本次实验的难点在于对流水线 CPU 多种工作状态的整体把握，以及顶层仿真中的纠错工作。流水线 CPU 速度更快，多条指令同时运行，这就要求我们对不同指令的运行以及数据的传递有一定的认识。在理论课程中已经对流水线 CPU 有了一定的了解学习，通过实验可以更直观更透彻的学习其具体流程和实现方式。

◆ 本次试验采用与之前所有试验都不同的仿真方法，前面都通过波形分析，显然在较大型，含有许多变量的工程中是不适用的。本次试验使用文本仿真的方法，大大提高仿真的效率。学会了提取内部变量进行跟踪，以及如何选取关键变量进行跟踪。同时，根据五级流水线的原理，一级一级跟踪变量，对其进行分析，判断正确与否，然后将 bug 逐层反馈到设计文件，从而对设计进行调试、修正。这一过程中，一个比较有趣且耗时巨大的 bug 是：由于在仿真的 delay 后面直接添上注释(见下图)，导致文本仿真数据奇奇怪怪，经过千辛万苦才发现这一个令人哭笑不得的 bug。

```
#10 enable = 1;
#10 start = 1;
#10 start = 0; // LOAD gr0 gr0 0
    i_datain = {\LOAD, `gr0, 1'b1, `gr0, 4'b0000};
#10 i_datain = {\LOAD, `gr1, 1'b1, `gr0, 4'b0001}; // LOAD gr0 gr0 1
#10 i_datain = {\NOP, 11'b000_0000_0000};
#10 i_datain = {\NOP, 11'b000_0000_0000}; // 3 clock after Load
    d_datain = 16'hf000;
#10 i_datain = {\NOP, 11'b000_0000_0000}; // 3 clock after Load
    d_datain = 16'hf00f;
#10 i_datain = {\ADD, `gr2, 1'b0, `gr0, 1'b0, `gr1};
#10 i_datain = {\NOP, 11'b000_0000_0000};
#10 i_datain = {\NOP, 11'b000_0000_0000}; |
#10 i_datain = {\NOP, 11'b000_0000_0000};
```

◆ 回顾整个实验过程，最重要的是严谨细致的态度。笔误这种错误最容易避免，也最容易发生，如：“==”错写成“=”；LDIH错写成LIDH...后者语法检查时已经报错，比较容易解决，但前者就比较难了，隐藏在一堆判断条件之中，排查起来相当费力。所以，只有再细心一点，实验才能更轻松一点。

◆ 实验中的不足之处在于对 ISE 软件的应用还是很生涩，很多功能仍未学习到，一些错误提示也不理解，时序报告、资源开销报告未能够透彻理解与分析，还需要进一步学习。

◆ 一些心得：对于 ID 中 reg_A、reg_B 的 if 语句里面的判断条件越具体越好，因为综合的时候这些语句是生成多路选择器，将具体操作与具体条件对应起来，通过时序报告发现，频率有明显的提升；在设计过程中，对于那些对缓冲寄存器的值的变化可有可无的操作，尽量让其保持原值，这样可以减少功耗，同时也提高了效率。

◆ 本次试验的缺陷：没有较为高效的解决 Control Hazards，想到的一个办法就是（如下图）：

```
// make id_ir useless and hold pc, the lowest 11bits hold its previous value.
else if( ( (ex_ir[7:3] == `BZ)|| (ex_ir[7:3] == `BNZ)|| (ex_ir[7:3] == `BN)
|| (ex_ir[7:3] == `BNN)|| (ex_ir[7:3] == `BC)|| (ex_ir[7:3] == `BNC)
|| (ex_ir[7:3] == `JUMP)|| (ex_ir[7:3] == `JMPR) )
|| ( (id_ir[15:11] == `BZ)|| (id_ir[15:11] == `BNZ)
|| (id_ir[15:11] == `BN)|| (id_ir[15:11] == `BNN)
|| (id_ir[15:11] == `BC)|| (id_ir[15:11] == `BNC)
|| (id_ir[15:11] == `JUMP)|| (id_ir[15:11] == `JMPR) ) )
id_ir[15:11] <= 5'b1111_1;
// make id_ir useless and hold pc, the lowest 11bits hold its previous value.
else
```

对于跳转指令，在未知其是否满足跳转条件之前，hold 住 pc，即下一条要执行的指令的地址，同时连续产生三条无意义的操作指令，直到 pc 跳转或者判断出跳转条件不满足，则执行原本要执行的下一条指令。

◆ 对于本次试验的设计以及设计上的缺陷将会在日后学习中得到优化、改进与解决。最后一句，写硬件，就得把自己当做机器，熟练认识其中的整套流程，然后一切都按部就班，机器怎么做，我们就按照这个流程来执行与编写。

四、附录

- | | |
|---------------------------|---------------|
| ◆ 12353022_CPU-Report.pdf | 本次试验的实验报告 |
| ◆ Simulation Code | 仿真文件文件夹 |
| 1. cpu_top.v | 用于仿真的顶层模块代码 |
| 2. mips.v | 核心 mips 模块源代码 |
| 3. alu.v | ALU 子模块源代码 |
| 4. instruction_memory.v | 指令内存模块代码 |
| 5. data_memory.v | 数据内存模块代码 |
| 6. cpu_tb.v | 仿真文件代码 |
| ◆ Source Code | 设计文件文件夹 |
| 1. mips.v | 核心 mips 模块源代码 |
| 2. alu.v | ALU 子模块源代码 |
| ◆ MIPS | ISE 设计文件夹 |