

目 录

目录

1 实验目的和意义	4
1.1 实验目的	4
2 实验环境介绍	5
2.1 Verilog HDL	5
2.2 MARS	5
2.3 ModelSim	5
3 概要设计	6
3.1 总体设计	6
3.1.1 单周期 CPU 总体设计	6
3.1.2 流水线 CPU 总体设计	10
3.2 PC（程序计数器）	14
3.2.1 功能描述	14
3.2.2 模块接口	15
3.3 RF（寄存器文件）	15
3.3.1 功能描述	15
3.3.2 模块接口	16
3.4 ALU（算数逻辑单元）	17
3.4.1 功能描述	17
3.4.2 模块接口	17
3.5 IM（指令存储器）	18
3.5.1 功能描述	18
3.5.2 模块接口	18
3.6 DM（数据存储器）	19
3.6.1 功能描述	19
3.6.2 模块接口	19

3.7 MUX（数据选择器）	20
3.7.1 功能描述	20
3.7.2 模块接口	20
3.8 EXT（数据拓展器）	21
3.8.1 功能描述	21
3.8.2 模块接口	21
3.9 CONTROL UNIT（控制单元）	21
3.9.1 功能描述	21
3.10 级间寄存器(流水线 CPU)	25
3.11 32 位 d 触发器(流水线 CPU)	26
3.11.1 功能描述	26
3.11.2 模块接口	26
4 详细设计	28
4.1 单周期 CPU 详细设计	28
4.1.1 CPU 总体结构	28
4.1.2 PC（程序计数器）	28
4.1.3 RF（寄存器文件）	29
4.1.4 ALU（算数逻辑单元）	29
4.1.5 IM（指令寄存器）	30
4.1.6 DM（数据存储器）	31
4.1.7 MUX（数据选择器）	33
4.1.8 EXT（数据拓展器）	33
4.1.9 Control Unit（控制中心）	34
4.2 流水线 CPU 详细设计	36
4.2.1 IF 阶段模块	36
4.2.2 ID 阶段模块	37
4.2.3 EXE 阶段模块	41

4.2.4 MEM 阶段模块.....	44
4.2.5 WB 阶段模块.....	45
5 测试及结果分析.....	47
5.1 单周期 CPU 测试及结果分析.....	47
5.1.1 代码仿真及分析.....	47
5.1.2 仿真测试结果.....	50
5.2 流水线 CPU 测试及结果分析.....	52
5.2.1 代码仿真及分析.....	52
5.2.2 仿真测试结果.....	55
6 实验遇到的问题和心得.....	60

1 实验目的和意义

1.1 实验目的

1. 掌握单周期 CPU 和流水线 CPU 数据通路图的构成、原理及其设计方法；
2. 掌握单周期 CPU 和流水线 CPU 的实现方法、代码实现方法；
3. 认识和掌握指令与 CPU 设计的关系；
4. 掌握在流水线 CPU 中消除冒险的方法：旁路、空指令等；
5. 掌握合理测试单周期 CPU 和流水线 CPU 的方法。

2 实验环境介绍

2.1 Verilog HDL

Verilog HDL 是一种硬件描述语言，用于从算法级、门级到开关级的多种抽象设计层次的数字系统建模。在本实验中利用该语言的模块化特性，将各个功能板块抽象出来，可以将数据通路及其流程模块化，降低设计的难度。

2.2 MARS

MARS 软件是一款用来运行 MIPS 汇编代码的软件，在其中编入汇编指令并运行，可以监测各寄存器的值变化，理解各指令的实际运行效果。此外，根据汇编指令在 MARS 中的运行结果，可以用于对比我们设计的 CPU，以检验其功能正确性。

2.3 ModelSim

ModelSim 是常用的 HDL 语言仿真软件，在本实验中用来调试和运行我们的 Verilog HDL 代码、观察各时期代码的实际运行结果以及用来 debug。在 ModelSim 中调试完毕的代码，在仿真层面已经完成了相应的功能，降低了此后在板子上调试失败的概率。

3 概要设计

3.1 总体设计

3.1.1 单周期 CPU 总体设计

设计基本原理和架构：

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期。在本实验的测试文件中，每个振荡间隔为 50ns，所以周期为 100ns。

CPU 在处理指令时，一般需要经过以下几个步骤：

(1) 取指令(IF)：根据程序计数器 PC 中的指令地址，从存储器中取出一条指令，同时，PC 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 PC，而转移的地址根据指令的不同又会有所不同。

(2) 指令译码(ID)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作和目的，从而产生相应的操作控制信号，用于驱动和指导执行状态中的各种操作。

(3) 指令执行(EXE)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态，一般是数值的计算，包括地址和数据等。

(4) 存储器访问(MEM)：所有需要访问存储器的操作都将在这个步骤中执行，包括存储器的读和写操作，根据该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(5) 写回(WB)：如果有相关要求，则指令会将执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

单周期 CPU 指令处理过程：



图 1 单周期 CPU 指令处理过程

单周期 CPU 数据通路与控制线路图:

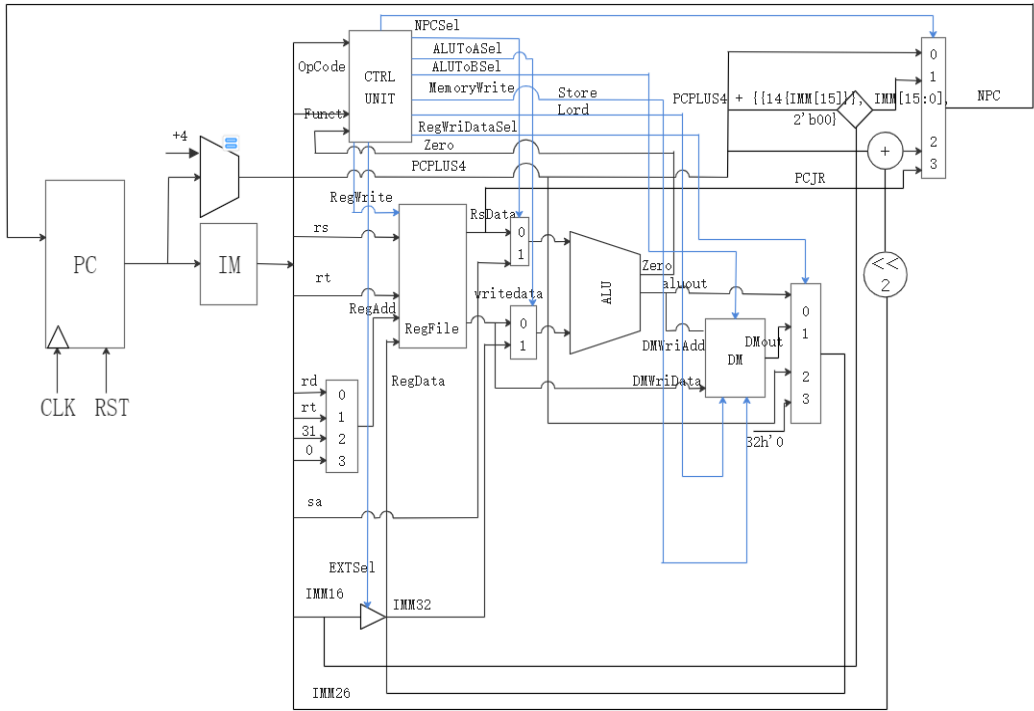


图 2 单周期 CPU 数据通路与控制路线图

图 2 是一个简单的基本上能够在单周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储不同的存储器中，即有数据存储器 and 指令存储器。访问存储器时，先给出内存地址，然后根据读或者写信号控制操作。对于寄存器组，先给出寄存器地址，读操作时和写操作各自在时钟上升沿和下降沿完成，此外写操作还需要写使能信号为 1，才能顺利实现。图中的控制信号作用如表 1 所示，表 2 是 ALU 运算功能表。

相关部件信号说明：

控制信号	含义	状态‘0’	状态‘1’
ALUToASel	操作数 A 选择	取寄存器 Rs 中数据	取偏移量 sa
ALUToBSel	操作数 B 选择	取寄存器 Rt 中数据	取拓展后的 32 位立即数
EXTSel	是否进行符号拓展	对 16 位立即数零拓展	对 16 位立即数符号拓展

MemWrite	存储器写信号	无关存储器写操作	存储器写使能
RegWrite	寄存器写信号	无关寄存器写操作	寄存器写使能
ALUOpertion[3:0]	ALU 运算操作	16 种运算功能选择 [0000–1111]，见功能表	
NPCSel[1:0]	PC 更新值选择	00: $NPC=PC+4$ ，一般情况 01: $NPC=PC+4+\text{偏移}\times 4$ ，如 bne、beq 等 10: $NPC=(PC+4)[31:28]+\text{IMM}[25:0]+2'b00$ 11: $NPC=RsData$ ，如 JR, JLR 等	
RegWriAddSel[1:0]	写寄存器地址选择	00: 写进 Rd 01: 写进 Rt 10: 写进 31 号寄存器 11: 暂未设置	
RegWriDatSel[1:0]	写寄存器数据选择	00: ALU 的计算结果 01: 从 DM 中读出来的结果 10: $PC + 4$ 11: 暂未设置	
Load[2:0]	加载数据选择	000: lw, 加载全字 001: lb, 加载字节 010: lbu, 加载字节（无符号） 011: lh, 加载半字 100: lhu, 加载半字（无符号） 101–111: 暂未设置	
Store[1:0]	存储数据选择	00: sw, 存储全字 01: sb, 存储字节 10: sh, 存储半字 11: 暂未设置	

表 1 控制信号及其作用

ALU 操作码功能表

ALUOp	操作	描述
0000	ALU_NOP	无操作, $C = A$

0001	ALU_ADD	加法操作
0010	ALU_SUB	减法操作
0011	ALU_AND	逻辑与操作
0100	ALU_OR	逻辑或操作
0101	ALU_SLT	小于则置位
0110	ALU_SLTU	小于则置位, 无符号数
0111	ALU_NOR	逻辑或非操作
1000	ALU_SLL	向左移位操作
1001	ALU_SRL	向右移位操作
1010	ALU_SRA	向右算术移位操作
1011	ALU_SLLV	逻辑可变左移操作
1100	ALU_SRLV	逻辑可变右移操作
1101	ALU_SLL16	向做移位 (16 位) 操作, 使用与指令 LUI
1110	ALU_XOR	逻辑异或操作
1111	ALU_SRAV	算数可变右移操作

表 2 ALU 操作码功能对应表

单周期 CPU 支持的指令操作有以下几种类型。

加载指令: lb, lbu, lh, lhu, lw

OpCode	rs(base)	rt	offset
6	5	5	16

R-R 运算指令: add, addu, and, nor, or, sll, sllv, slt, sltu, sra, srav, srl, srlv, sub, subu, xor

OpCode	rs	rt	rd	sa	FuncCode
6(000000)	5	5	5	5	6

R-I 运算指令: addi, addiu, andi, lui, ori, slti, sltiu, xori

OpCode	rs	rt	immediate
6	5	5	16

分支指令: beq, bne

OpCode	rs	rt	offset
6	5	5	16

跳转指令: j, jal, jalr, jr

OpCode	instr_index
6	26

OpCode	rs	0	rd	0	FuncCode
6	5	5	5	5	6

另：单周期 CPU 支持的指令集为：

add/sub/and/or/slt/sltu/addu/subu
addi/ori/lw/sw/beq
j/jal
sll/nor/lui/slti/bne/andi/srl/sllv/srlv/jr/jalr
xor/sra/srav
lb/lh/lbu/lhu/sb/sh

以上共 35 条指令。

3.1.2 流水线 CPU 总体设计

设计原理和结构：

流水线 CPU 同单周期一致，将指令的实现分为 5 个阶段，每个阶段实现的功能也同单周期 CPU 大致相同，因为涉及到跳转、分支等指令以及数据冒险、控制冒险等操作，在每个阶段的功能都必须进行丰富，以使其顺利完成相应的功能。

(1) 取指令(IF)：根据程序计数器 PC 中的指令地址，从存储器中取出一条指令，将指令中的内容传送至 IF-ID 级寄存器。此外，IF 级还需要同时处理 PC 的更迭，因为无法翻译指令(ID 级)的内容在此阶段无法完成，因此这里只能将 PC 自增 4，默认无跳转发生。在后面的阶段，若 PC 值有所更新，会改变 IF 阶段的转移地址。

(2) 指令译码(ID)：对 IF-ID 级寄存器得到的指令进行分析并译码，确定这条指令需要完成的操作和目的，从而产生相应的操作控制信号，用于驱动和指导执行状态中的各种操作。同时接收来自 EXE 和 MEM 级传来的信号，包括(mm, mm2reg, mwreg, ern, em2reg, ewreg)，这些信号和 ID 级控制单元的其他控制信号，如寄存器号，目的寄存器号等信号共同决定数据冒险的类别(exe-alu、mem-alu、mem-lw)。在两个目的操作数的选择上，根据已经确定的冒险类别，

设置多选器决定使用的数据，因为涉及到之前指令的数据，同样需要将 exe/mem 级的相关数据传送到对应的多选器上，由上所述，ID 级解决了数据冒险的问题。不仅是数据冒险，ID 级同样是控制冒险解决的阶段，在本阶段译指，得到的数据即可分析跳转指令的目的 PC 地址，此外利用在 ID 级的 NOR 门，可以直接判断分支指令是否发生，以上所述跳转/分支指令若发生，则将跳转目的 PC 传入 IF 级，调整指令执行顺序，同时生成一个清空指令 flush，将其传送到 IF 级，使得预测不发生而导致多余执行的一条指令使能失败，完成控制冒险的实现。最后，寄存器堆模块也在本阶段中，用于完成寄存器的读写操作。

(3) 指令执行(EXE)：根据指令译码(在物理上来自 ID-EXE 寄存器)得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态，一般是数值的计算，包括地址和数据等。

(4) 存储器访问(MEM)：需要访问存储器的操作都将在这个步骤中执行，包括存储器的读和写操作，根据该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(5) 写回(WB)：如果有相关要求，则指令会将执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

流水线 CPU 数据通路和控制路线图：

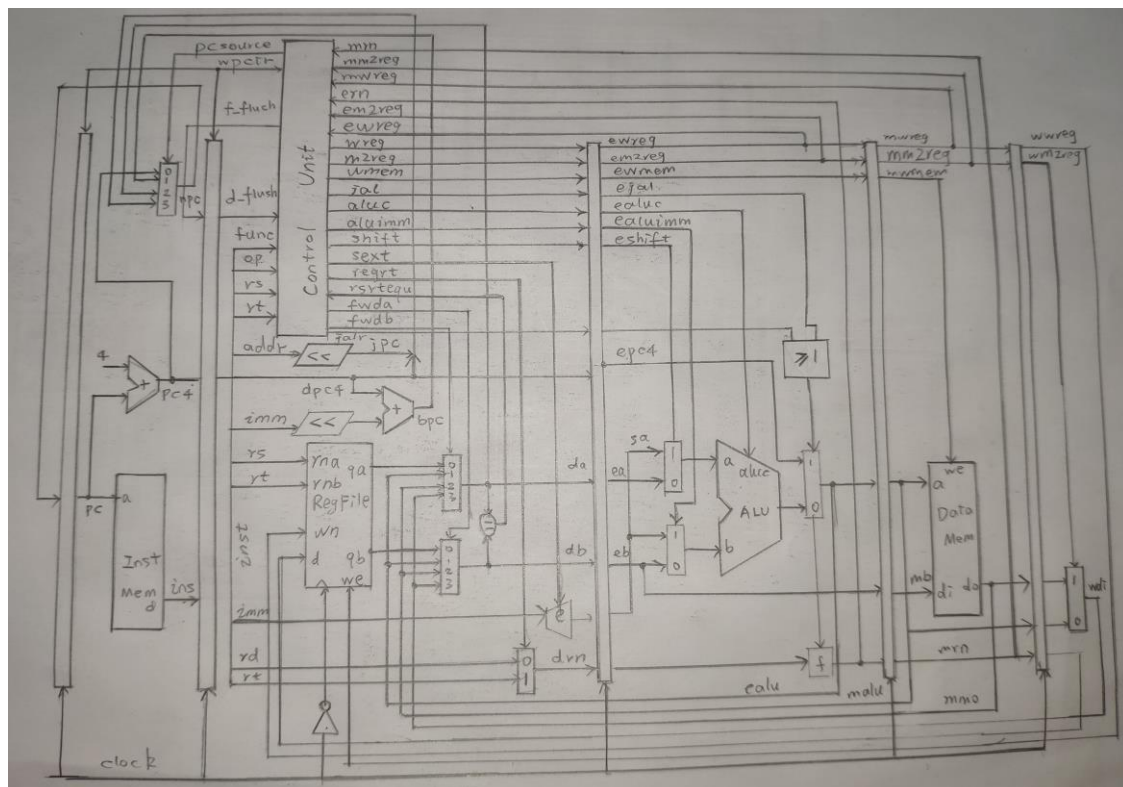


图 3 流水线 CPU 数据通路和控制路线图

相关控件信号说明:

控制信号	含义	状态‘0’	状态‘1’
pcsource	PC 更新值	00: PC + 4 01: bpc(PC+4+imm*4) 10: rs(供 JALR 及 JA 使用) 11: jpc(addr)	
wpcir	PC 写使能	PC 不更新	PC 更新
flush	是否清空该指令	指令正常	指令清空
wreg	寄存器写信号	寄存器不可写	寄存器写使能
m2reg	写寄存器数据来自 memory	写寄存器数据不来自 memory	写寄存器数据来自 memory
wmem	memory 写信号	memory 不可写	memory 写使能
jal	指令是否为 jal	指令不是 jal	指令为 jal
jalr	指令是否为 jalr	指令不是 jalr	指令是 jalr
aluc	alu 执行操作类型	详见 alu 操作表	
aluimm	操作数 B 是否为立即数	操作数 B 来自寄存器	操作数 B 来自立即数
shift	操作数 A 是否是 sa	操作数 A 来自寄存器	操作数 A 来自 sa
sext	对立即数是否进行符号拓展	对立即数进行零拓展	对立即数进行符号拓展
regrt	写寄存器是否是 rt	目的寄存器不是 rt	目的寄存器是 rt
rsrtequ	两原寄存器内容相等	两原寄存器内容不相等	两原寄存器内容相等
fwda	操作数 A 的来源	00: 操作数 A 来自原寄存器 01: 操作数 A 来自 EXE 阶段 alu 结果 10: 操作数 A 来自 MEM 阶段 alu 结果 11: 操作数 A 来自 MEM 阶段 mem 读取值	
fwdb	操作数 B 的来源	00: 操作数 B 来自原寄存器	

		01: 操作数 B 来自 EXE 阶段 alu 结果
		10: 操作数 B 来自 MEM 阶段 alu 结果
		11: 操作数 B 来自 MEM 阶段 mem 读取值

表 3: 流水线 CPU 相关控件信号说明

流水线 CPU 支持的指令如下:

add/sub/and/or/slt/sltu/addu/subu
addi/ori/lw/sw/beq
j/jal
sll/nor/lui/slti/bne/andi/srl/sllv/srlv/jr/jalr
xor/sra/srav/xori

以上共 30 条指令，以及不支持冒险的 muti、mfhi、mflo3 条指令。

其中各指令所属的类别同单周期一致，此处不再赘述。

在讨论流水线 CPU 的设计架构之前，我先对流水线 CPU 设计中不可避免的问题——冒险问题提出解决方案，以使后面的介绍更加明了。因为结构冒险问题通常通过硬件解决，所以在这里只考虑数据冒险和控制冒险的处理。

数据冒险的处理:

数据冒险分为根据数据来源分为两类，即:

- 1. 某条指令使用到上一条指令的 ALU 结果，譬如

add r3, r1, r2
sub r4, r9, r3//指令使用了上条指令的运算结果（ALU 结果）

这种数据冒险通过使用旁路的方式，可以从 EXE / MEM 级的运算结果推送到下一条指令的 ID 级，因为二者从时间上处于同一周期，所以不需要对流水线采取延迟措施。

- 2. 某条指令使用到上一条指令的加载结果，譬如

lw r3, 0(r1)
sub r4, r9, r3//指令使用了上条指令的加载结果

这种数据冒险同样使用旁路的方式，将 MEM 级读取到数据推送到后面指令的 ID 级，因为在时间上，前者仍然要延后一个周期，所以需要使整个流水线延迟一个周期。

控制冒险的处理:

控制冒险的处理可以有多种方案，考虑到实现难度和流水线效率的问题，我在此选择了实现起来较为简单的预测机制，即默认跳转或分支指令不发生，若发生则清空已经执行的指令，否则按照原顺序执行。

为了统一清空指令的问题，因为跳转指令(j/jal/jr/jalr)在 ID 级即可判断能否发生跳转，我们将分支指令(bne/beq)指令的判断同样移至 ID 级，具体实现方法是在 ID 级中增加一个异或门，比较两个寄存器数据是否相等。

以上所述冒险的处理，都需要一定条件的判定，在后面代码中会具体分析

3.2 PC（程序计数器）

3.2.1 功能描述

单周期 CPU: 如图 3 所示，PC 模块控制当前读取的指令，通过对 PC 值的更迭，实现 MIPS 代码要求实现的逻辑顺序。当然，在一个周期内，PC 总是能够根据所取指令的正确要求，实现 PC 的更迭：一般情况下增加 4，若分支结构发生，则跳转到目的分支地址，同理，跳转指令条件下，PC 发生跳转。

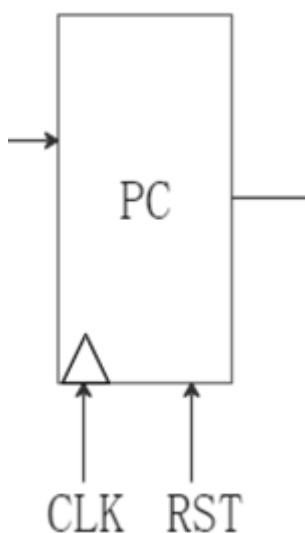


图 4 单周期 CPU PC 模块示意图

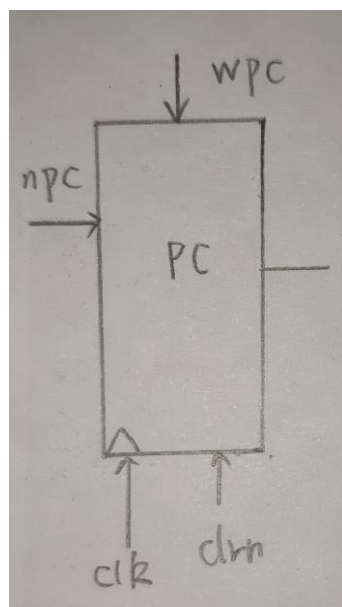


图 5 流水线 CPU PC 模块示意图

流水线 CPU: PC 模块控制当前周期 IF 级读取的指令，通过对 PC 的更迭，由于周期缩短至阶段的时钟长度，因此，PC 在发生更迭时，不能获得当前指令实

际的效果，无法判断其中的分支/跳转指令是否发生，所以，在这里，默认 PC 更迭的效果是自增 4，效果是：在一般情况下，顺利执行；如果指令是分支 / 跳转指令，则预测分支 / 跳转不发生。当然，若分支 / 跳转发生，则在后面的 PC 中再进行更迭。此外，因为某些时候需要流水线的延迟，因此 PC 不可更迭，所以 PC 模块中增加一个写使能信号。

3.2.2 模块接口

信号名	方向	描述
clk	input	Clk 上升沿时，若 rst 信号为 0，将 NPC 写入 PC，完成更迭。
rst	input	Rst 上升沿时，将 PC 置零。
NPC [31:0]	input	输入的将要更迭的 PC
PC [31:0]	output	当前 PC

表 4 单周期 CPU PC 模块接口示意图

信号名	方向	描述
clk	input	Clk 上升沿时，若 rst 信号为 0，将 NPC 写入 PC，完成更迭。
rst	input	Rst 上升沿时，将 PC 置零。
NPC [31:0]	input	输入的将要更迭的 PC
wpc	input	PC 写使能，为 1 则顺利写，否则写失败
PC [31:0]	output	当前 PC

表 5 流水线 CPU PC 模块接口示意图

3.3 RF（寄存器文件）

3.3.1 功能描述

单周期 CPU 和流水线 CPU 的寄存器组模块功能基本一致，因为流水线 CPU 的寄存器组模块是在 ID 阶段中，在该模块内，因此两者的输入/输出信号描述不同，但是其功能完全一致，在功能描述中，我们将二者放在一起来说。

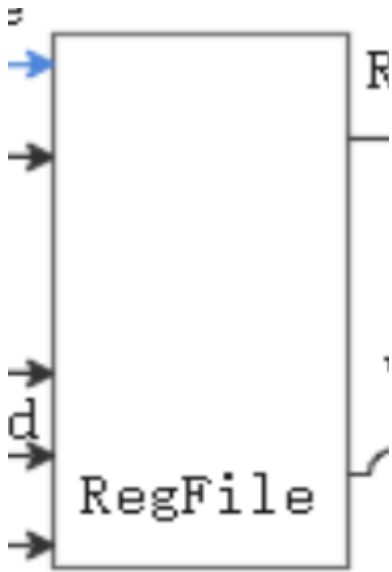


图 6 单周期 CPU RegFile 示意图

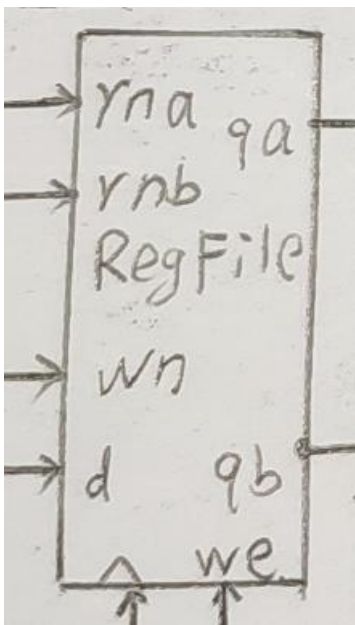


图 7 流水线 CPU Reg File 示意图

寄存器组文件(register file)用来存储 MIPS 指令集中 32 个寄存器，作为一个单独的模块，通过读写端口输入的地址和数据来更新寄存器的内容。其中寄存器读是组合逻辑的功能，而写寄存器堆是时序逻辑，前者用 assign 赋值，而后者则需要在时钟的边沿触发写入。

3.3.2 模块接口

信号名	方向	描述
clk	input	Clk 上升沿时，开始写入工作。
rst	input	Rst 上升沿时，将寄存器组内容全部置零。
RFWr	input	若 RFWr 信号为 1， 则数据成功写入。
A1[4:0]	input	读寄存器端口 1 的地址
A2[4:0]	input	读寄存器端口 2 的地址
RegAdd[4:0]	input	写寄存器端口的地址
RegWriData[31:0]	input	写寄存器端口的数据
RD1[31:0]	output	读寄存器端口 1 的数据
RD2[31:0]	output	读寄存器端口 2 的数据

表 6 单周期 CPU 寄存器模块的接口

信号名	方向	描述
clk	input	Clk 上升沿时，开始写入工作。
clrn	input	clrn 上升沿时，将寄存器组内容全部置零。
we	input	若 we 信号为 1， 则数据成功写入。

rna[4:0]	input	读寄存器端口 1 的地址
rnb[4:0]	input	读寄存器端口 2 的地址
wn[4:0]	input	写寄存器端口的地址
d[31:0]	input	写寄存器端口的数据
qa[31:0]	output	读寄存器端口 1 的数据
qb[31:0]	output	读寄存器端口 2 的数据

表 7 流水线 CPU 寄存器模块的接口

3.4 ALU（算数逻辑单元）

3.4.1 功能描述

对接受到的两个操作数进行算术逻辑运算，其中具体运算的操作由接收到的操作码 (ALUOP) 决定，将算术逻辑运算的结果输出，同时输出的还有 0 标志位。

对于单周期和流水线，两个的 ALU 模块完全一致，此处只给出其中一个接口表。

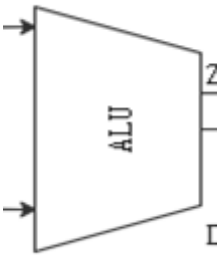


图 8 单周期 CPU ALU 示意图

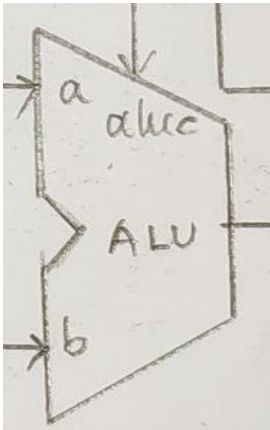


图 9 流水线 CPU ALU 示意图

3.4.2 模块接口

信号名	方向	描述
A[31:0]	input	算数逻辑单元的操作数 1
B[31:0]	input	算数逻辑单元的操作数 2
ALUOp[3:0]	input	指示算术逻辑单元执行的操作
C[31:0]	output	算数逻辑单元的运算结构
Zero	output	零标志位

表 8 单周期/流水线 CPU ALU 模块的接口

3.5 IM（指令存储器）

3.5.1 功能描述

IM(Instruction Memory)指令寄存器是组合逻辑，内部设计为含有 128 个字的 ROM。在 IM 中存储的是 MIPS 指令的机器码。机器码的生成借助于 MARS 软件，在其中编写 MIPS 汇编代码之后即可转成相应的机器码。该模块由两个接口，分别是访问 IM 的地址，输出接口则是地址对应的数据(指令)。在整个电路模拟仿真的初始化时，需要借助 Verilog HDL 的\$readmemh 函数将目的文件中的数据(一般是工程文件中的.dat 文件)读入 IM。

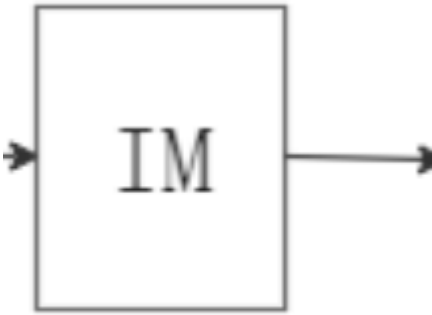


图 10 单周期 CPU IM 示意图

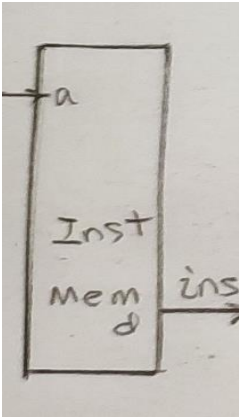


图 11 流水线 CPU IM 示意图

3.5.2 模块接口

单周期和流水线 CPU 的 IM 模块基本一致，二者只是接口的描述不同，功能基本一致。

信号名	方向	描述
RegRedAdd[8:2]	input	输入的访问位置，取其字地址（故舍弃后两位）
RegRedData[31:0]	output	输出的指令数据

表 9 单周期 CPU 的 IM 模块接口

信号名	方向	描述
addr[8:2]	input	输入的访问位置，取其字地址（故舍弃后两位）
output[31:0]	output	输出的指令数据

表 10 流水线 CPU 的 IM 模块接口

3.6 DM（数据存储器）

3.6.1 功能描述

单周期 CPU 的 DM：原本地，数据寄存器和寄存器组一样，读存储器为组合逻辑，写寄存器位时序逻辑，但是单周期 CPU 的实现加入了 lb/lbu/lh/ldu 等指令，于是加载指令的输出也随指令的变化而变化，更改设计为：读寄存器在下降沿读出，而写寄存器要在时钟的上升沿写入。同时，因为 sb/sh 等指令的加入，使得存储器的最小分割单元由字变成了字节，即 8 位为原子单位。其内部设计为含有 1024 个字节的 RAM，即大小为 256 个字。

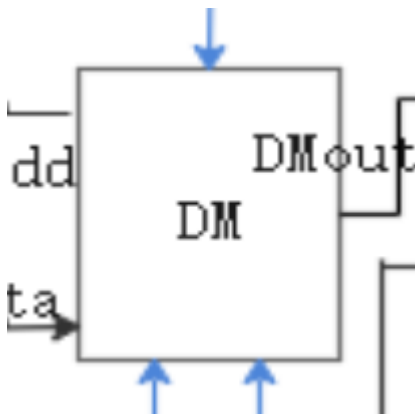


图 12 单周期 CPU DM 示意图

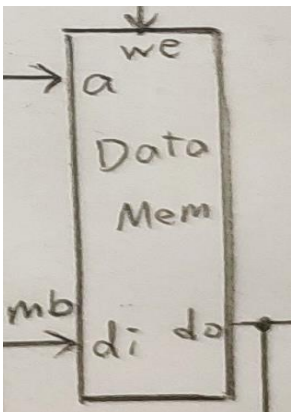


图 13 流水线 CPU DM 示意图

流水线 CPU 的 DM：考虑到流水线 CPU 的复杂程度和实现难度，在此，我没有加入可能会影响整体结构的 lb/lbu/lh/ldu/sb/sh 指令，加载指令和存储指令均只有一条，即 lw 和 sw，因此其整体结构也得到简化。读存储器为组合逻辑，写寄存器位时序逻辑。此外，因为涉及的最小原子单位是字，所以以 32 位为基本单位，内部被设计为含有 128 个字的 RAM。同样地，因为加载和存储指令只有一种，因此关于加载和存储的控制信号 (Load 和 Store) 在流水线的 DM 模块中也不必出现。

3.6.2 模块接口

单周期 CPU：

信号名	方向	描述
clk	input	clk 上升沿完成写入，下降沿完成读出

MemWrite	input	存储器写信号
MemAdd[9:0]	input	写存储器的位置（字节）
MemWriData[31:0]	input	写存储器的数据
Load[2:0]	input	加载指令的格式
Store[1:0]	input	存储指令的格式
MemRedData[31:0]	output	读寄存器的数据

表 11 单周期 CPU 的 DM 模块接口

流水线 CPU:

信号名	方向	描述
clk	input	clk 上升沿完成写入
DMWr	input	存储器写信号
addr[8:2]	input	写存储器的位置（字节）
din[31:0]	input	写存储器的数据
dout[31:0]	output	读寄存器的数据

表 12 流水线 CPU 的 DM 模块接口

3.7 MUX（数据选择器）

3.7.1 功能描述

数据选择器是电子电路中常见的逻辑器件，在 CPU（单周期/流水线）的数据通路里出现多次，因为数据选择器涉及的数据大小以及选项的多寡皆有不同，在这里我只给出其中一个规格（4 选 1 32 位选择器）的其中一种实现方式，其他规格的数据选择器均可以有本格式变更而来，不再详述。功能：根据输入的控制信号的不同，从不同的输入中选择合适的输出。

3.7.2 模块接口

信号名	方向	描述
WIDTH	input	数据位宽
d0[WIDTH-1:0]	input	写入数据 1
d1[WIDTH-1:0]	input	写入数据 2
d2[WIDTH-1:0]	input	写入数据 3
d3[WIDTH-1:0]	input	写入数据 4
s[1:0]	input	选择信号

y[WIDTH-1:0]	output	输出数据
--------------	--------	------

表 13 数据选择器的模块接口

3.8 EXT（数据拓展器）

3.8.1 功能描述

同数据选择器一样，数据拓展器同样是电子电路中常用的逻辑模块，在单周期和流水线 CPU 中均有所作用，功能是：将输入的 16 位立即数，通过控制信号，将其零拓展或者符号拓展并且输出。

3.8.2 模块接口

信号名	方向	描述
Imm[15:0]	input	输入的 16 位立即数
EXTOp	input	控制信号，为 1 则符号拓展，否则零拓展
Imm32[31:0]	output	拓展之后得到的 32 位立即数

表 14 数据拓展器的模块接口

3.9 CONTROL UNIT（控制单元）

3.9.1 功能描述

Control Unit 是组合逻辑，是整个 CPU 逻辑产生的核心，在单周期和流水线中作用类似，但是具化到实际的功能仍然有所不同，为了使 Control Unit 的具体效用更加明确，在这里我将两者分开描述。

单周期 CPU Control Unit: 单周期 CPU 的控制中心相对而言功能比较简单，主要操作是将当前指令中的机器码(opcode、funcCode、Zero)转换为各个控制信号，从而控制不同的指令在不同的数据通路中传输。各控制信号的作用在“总体设计”单元的已经给出，详见表 1。根据表 1，可以得到单周期 CPU 各个指令与控制信号之间的关系，如下表 13：

Inst	Zero	A	B	ALUOp	EXT	NPC	MW	RW	RWA	RWD	L	S
add	x	0	0	0001	0	00	0	1	00	00	xxx	xx
sub	x	0	0	0010	0	00	0	1	00	00	xxx	xx
and	x	0	0	0011	0	00	0	1	00	00	xxx	xx
or	x	0	0	0100	0	00	0	1	00	00	xxx	xx
slt	x	0	0	0101	0	00	0	1	00	00	xxx	xx
sltu	x	0	0	0110	0	00	0	1	00	00	xxx	xx
addu	x	0	0	0001	0	00	0	1	00	00	xxx	xx
subu	x	0	0	0010	0	00	0	1	00	00	xxx	xx
jr	x	0	0	xxxx	0	11	0	1	00	00	xxx	xx
jalr	x	0	0	xxxx	0	11	00	1	00	10	xxx	xx
nor	x	0	0	0111	0	00	0	1	00	00	xxx	xx
sll	x	1	0	1000	0	00	0	1	00	00	xxx	xx
srl	x	1	0	1001	0	00	0	1	00	00	xxx	xx
sra	x	1	0	1010	0	00	0	1	00	00	xxx	xx
sllv	x	0	0	1011	0	00	0	1	00	00	xxx	xx
srlv	x	0	0	1100	0	00	0	1	00	00	xxx	xx
xor	x	0	0	1110	0	00	0	1	00	00	xxx	xx
srav	x	0	0	1111	0	00	0	1	00	00	xxx	xx
addi	x	0	1	0001	1	00	0	1	01	00	xxx	xx
ori	x	0	1	0100	1	00	0	1	01	00	xxx	xx
lw	x	0	1	0001	1	00	0	1	01	01	000	xx
sw	x	0	1	0001	1	00	1	0	xx	xx	xxx	00
beq	0	0	0	0010	x	00	0	0	xx	xx	xxx	xx
beq	1	0	0	0010	x	01	0	00	xx	xx	xxx	xx
bne	0	0	0	0010	x	01	0	0	xx	xx	xxx	xx
bne	1	0	0	0010	x	00	0	0	xx	xx	xxx	xx
slti	x	0	1	0101	1	00	0	1	01	00	xxx	xx

lui	x	0	1	1101	1	00	0	1	01	00	xxx	xx
andi	x	0	1	0011	1	00	0	1	01	00	xxx	xx
lb	x	0	1	0001	1	00	0	1	01	01	001	xx
lbu	x	0	1	0001	1	00	0	1	01	01	010	xx
lh	x	0	1	0001	1	00	0	1	01	01	011	xx
lhu	x	0	1	0001	1	00	0	1	01	01	100	xx
sb	x	0	1	0001	1	00	1	0	xx	xx	xxx	01
sh	x	0	1	0001	1	00	1	0	xx	xx	xxx	10
j	x	x	x	xxxx	0	10	0	0	xx	xx	xxx	xx
jal	x	x	x	xxxx	0	10	0	1	10	10	xxx	xx

表 15 单周期 CPU 指令与控制信号的关系

流水线 CPU 控制单元：相对于单周期 CPU，流水线 CPU 的控制中心功能要较为复杂，是因为流水线的控制中心不仅要接收来自本阶段指令的机器码，还要考虑本阶段执行的指令可能受到前面指令的影响，形成的后果包括数据冒险、控制冒险等等。由上，控制中心的输入还包括来自前面指令在本时钟周期(此时，前面指令应运行至 EXE、MEM 阶段)的控制信号，譬如寄存器写地址、是否写寄存器以及写寄存器数据等。将控制中心的作用分为 3 类，在下面具体描述：

1. 将当前指令中的机器码(opcode、funcCode、Zero)转换为各个控制信号，从而控制不同的指令在不同的数据通路中传输。类似于单周期 CPU 控制中心，控制中心的主要作用仍然是产生从而控制指令的执行，各个控制信号的作用已在表 1 中阐述，这里不再详述，控制信号的作用同单周期流水线类似，这里不再详述。

2. 数据冒险的消除：数据冒险的产生来自于后续指令的源寄存器等同于前驱指令的目的指令，因为源寄存器可能是 rt 或者 rs，而旁路的类型又分为 3 类，即：exe-alu、mem-alu、mem-lw。

假设某条指令的源寄存器 rs，正是上条指令的目的寄存器，而上条指令有写寄存器作用，这样就必须在 rs 寄存器上产生 exe-alu 旁路，用代码解释就是

```
if(ewreg & (ern != 0) & (ern == rs) & ~em2reg)
    begin
        fwda = 2'b01;
```

```
end
```

此为其中一种情况，此时根据 **fwda** 的取值，操作数 A 应该来自本周期 EXE 阶段的计算结果(aluout)。同理其他所有的情况用代码表示如下：

```
begin
    fwda = 2'b00;
    if(ewreg & (ern != 0) & (ern == rs) & ~em2reg)
    begin
        fwda = 2'b01;
    end else begin
        if(mwreg & (mrn != 0) & (mrn == rs) & ~mm2reg)
        begin
            fwda = 2'b10;
        end else begin
            if(mwreg & (mrn != 0) & (mrn == rs) & mm2reg) begin
                fwda = 2'b11;
            end
        end
    end
end

fwdb = 2'b00;
if(ewreg & (ern != 0) & (ern == rt) & ~em2reg)begin
    fwdb = 2'b01;
end else begin
    if(mwreg & (mrn != 0) & (mrn == rt) & ~mm2reg)begin
        fwdb = 2'b10;
    end else begin
        if(mwreg & (mrn != 0) & (mrn == rt) & mm2reg)begin
            fwdb = 2'b11;
        end
    end
end
end
end
```

3. 控制冒险的处理：我们采取静态预测控制冒险的不发生，若冒险发生，则需要清空一条指令。具体的控制信号由 **flush** 表示，清空信号产生的表达式如下：

```
assign flush = (i_j | i_jr | i_jal | i_jalr | i_beq & rsrtequ | i_bne & ~rsrtequ );
```

在跳转指令以及分支指令发生的情况下进行清空。

3.10 级间寄存器(流水线 CPU)

MIPS 流水线 CPU 在总体结构上要比单周期复杂得更多,尤其是不同阶段的控制信号的传递及其作用,因为时序的限制,我们必须设置不同的寄存器来保存同一指令在不同阶段的控制信号和数据,下面分别介绍不同阶段之间的寄存器模块。

- IF-ID 寄存器: 存储的值有:

instruction	f_flush	pc+4
-------------	---------	------

instruction: 从 IF 阶段读出来的指令, 用于控制信号的生成;

f_flush: ID 阶段传入的控制信号、用来控制指令的清空;

pc+4: PC 自增 4 后的结果, 用于后续的计算;

- ID-EXE 寄存器: 存储的值有:

wreg	m2reg	wmem	jal	jalr	aluc	aluimm	shift	PC+4	da	db	dimm	drn
------	-------	------	-----	------	------	--------	-------	------	----	----	------	-----

wreg~shift: 作用详见在表 3;

PC+4: PC 自增 4 的结果, 在 EXE 阶段用于部分跳转指令(JAL 和 JALR)的计算结果;

da: 操作数 A 来自寄存器的候选值;

db: 操作数 B 来自寄存器的候选值;

dimm: 操作数 B 的立即数候选值、其中部分位是操作数 A 的候选值(sa);

drn: 写目的寄存器, 在 EXE 阶段中, 部分跳转指令(JAL)会对此值进行修改。

- EXE-MEM 寄存器: 存储的值有:

ewreg	wm2reg	ewmem	ealu	eb	ern
-------	--------	-------	------	----	-----

ewreg~ewmem: 作用详见表 3;

ealu: EXE 阶段之后得到的 alu 结果, 是后续写寄存器数据的候选之一;

eb: 操作数 B 的寄存器候选值, 同时也是存储指令的 memory 数据;

ern: 写目的寄存器。

- MEM-WB 寄存器: 存储的值有:

mwreg	mm2reg	do	malu	mrn
-------	--------	----	------	-----

mwreg~em2reg: 作用详见表 3;

do: memory 中读出来的数据;

malu: 同 ealu, 在 EXE 阶段计算出来的结果;

mrn: 同 ern, 写目的寄存器。

- WB-IF 寄存器: 存储的值有:

npc	wpcir
-----	-------

npc: 待更新的 PC 值;

wpcir: PC 写使能信号。

3.11 32 位 d 触发器(流水线 CPU)

3.11.1 功能描述

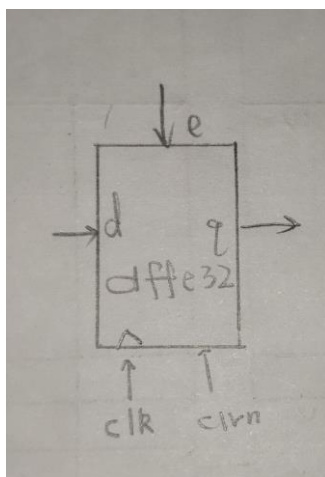


图 13 D 触发器示意图

D 触发器实际上是一个寄存器，它将寄存器的功能封装成一个模块，使得调用方便了许多。在流水线中因为涉及到 PC 的存储、instruction 的存储等，将其封装成 D 触发器会使调用起来更加便捷。D 触发器内数据在时钟信号上升沿，且写使能信号为 1 时更改，在清空信号上升沿清零。

3.11.2 模块接口

信号名	方向	描述
d	input	输入的 32 位数据
clk	input	时钟信号，上升沿时将数据存入寄存器

clrn	input	清空信号，上升沿时将数据清零
e	input	写使能信号，为 1 时才能顺利写
q	output	寄存器 q 内的数据

4 详细设计

4.1 单周期 CPU 详细设计

4.1.1 CPU 总体结构

单周期 CPU 总体结构：MIPS 单周期 CPU 可以简单地划分为五个阶段，对应着上面已经提到的五个关键的模块：IF 对应于 PC，ID 阶段对应于 IM（指令寄存器）和 RegisterFile（寄存器组），EXE 对应于 ALU，MEM 阶段对应于 DataMemory，WB 阶段对应于 RegisterFile。在数据通路中，除了这些模块之外，还需要完成具体指令的控制信号，控制信号生成集成在 CTRLUNIT（控制单元），此外，不同控制信号的选择控制具体实现还需要依赖于数据选择器（MUX）。当然，立即数的拓展不能离开数据拓展器（EXT）。

以上所提到的所有模块，包括数据通路和两个存储器（IM 和 DM），最终构成一个较为完善的 MIPS 单周期 CPU 通路。

4.1.2 PC（程序计数器）

单周期 CPU：

```
module PC( clk, rst, NPC, PC );
    input      clk;
    input      rst;
    input  [31:0] NPC;
    output reg  [31:0] PC;

    always @(posedge clk, posedge rst)
        if (rst)
            PC <= 32'h0000_0000;
        // PC <= 32'h0000_3000;
        else
            PC <= NPC;
endmodule
```

说明：在时钟上升沿到来时，更新 PC 值；充值信号上升沿时，PC 置 0

4.1.3 RF（寄存器文件）

单周期 CPU:

```

module RF( input      clk,
           input      rst,
           input      RFWr,
           input  [4:0] A1, A2, RegAdd,
           input  [31:0] RegWriData,
           output [31:0] RD1, RD2,
           input  [4:0] reg_sel,
           output [31:0] reg_data);

reg [31:0] rf[31:0];

integer i;

always @(posedge clk, posedge rst)
  if (rst) begin // reset
    for (i=1; i<32; i=i+1)
      rf[i] <= 0; // i;
    end

  else
    if (RFWr) begin
      rf[RegAdd] <= RegWriData;
    end

    assign RD1 = (A1 != 0) ? rf[A1] : 0;
    assign RD2 = (A2 != 0) ? rf[A2] : 0;
    assign reg_data = (reg_sel != 0) ? rf[reg_sel] : 0;

endmodule

```

说明：寄存器模块具体功能见 3.3 表 6

4.1.4 ALU（算术逻辑单元）

单周期 CPU:

```

module ALU(A, B, ALUOp, C, Zero);

input signed [31:0] A, B;
input      [3:0] ALUOp;
output signed [31:0] C;

```

```

output Zero;

reg [31:0] C;
integer i;

always @( * ) begin
    case ( ALUOp )
        `ALU_NOP: C = A;                // nop
        `ALU_ADD: C = A + B;            // add
        `ALU_SUB: C = A - B;            // sub
        `ALU_AND: C = A & B;            // and/andi
        `ALU_OR:  C = A | B;            // or/ori
        `ALU_SLT: C = (A < B) ? 32'd1 : 32'd0; // slt/slti
        `ALU_SLTU: C = ({1'b0, A} < {1'b0, B}) ? 32'd1 : 32'd0; //sltu
        `ALU_NOR: C = ~(A | B);         //nor
        `ALU_XOR: C = A ^ B;            //xor
        `ALU_SLL: C = B << A;            //sll
        `ALU_SRL: C = B >> A;            //srl
        `ALU_SRA: C = B >>> A;           //sra
        `ALU_SLLV: C = B << A[4:0];      //sllv
        `ALU_SRLV: C = B >> A[4:0];      //srlv
        `ALU_SRAV: C = B >>> A[4:0];     //srav
        `ALU_SLL16: C = B << 16;         //lui
        default: C = A;                  // Undefined
    endcase
end // end always

assign Zero = (C == 32'b0);

endmodule

```

说明：ALU 的具体功能见 3.4 表 8

4.1.5 IM（指令寄存器）

单周期 CPU：

```

// instruction memory
module IM(input [8:2] RegRedAdd,
          output [31:0] RegRedData );

reg [31:0] rom[127:0];

assign RegRedData = rom[RegRedAdd]; // word aligned

```

```
endmodule
```

说明：单周期 IM 具体功能见 3.5 表 10；

流水线 CPU：

```
// instruction memory
module im(input [8:2] addr,
          output [31:0] dout );

    reg [31:0] ROM[127:0];
    assign dout = ROM[addr]; // word aligned
endmodule
```

说明：流水线 IM 具体功能见 3.5 表 11；

4.1.6 DM（数据存储器）

单周期 CPU：

```
module DM(clk, MemWrite, MemAdd, MemWriData, Load, Store, MemRedData);
    input      clk;
    input      MemWrite;
    input [9:0] MemAdd;
    input [31:0] MemWriData;
    input [2:0] Load;
    input [1:0] Store;
    output reg [31:0] MemRedData;

    reg [8:0] ram[1023:0];
    always @(posedge clk)
    begin
        if (MemWrite) begin
            case(Store)
                2'b00: begin // sw
                    ram[MemAdd + 0] = MemWriData[7:0];
                    ram[MemAdd + 1] = MemWriData[15:8];
                    ram[MemAdd + 2] = MemWriData[23:16];
                    ram[MemAdd + 3] = MemWriData[31:24];
                end
                2'b01: begin // sb
                    ram[MemAdd + 0] = MemWriData[7:0];
                end
                2'b10: begin // sh
                    ram[MemAdd + 0] = MemWriData[7:0];
                    ram[MemAdd + 1] = MemWriData[15:8];
                end
            endcase
        end
    end
```

```

        end
    endcase

    $display("ram[0x%8X] = 0x%2X 0x%2X 0x%2X 0x%2X ", (MemAdd / 4) << 2, ram[((MemAdd / 4) << 2) + 3], ram[((MemAdd / 4) << 2) + 2], ram[((MemAdd / 4) << 2) + 1], ram[((MemAdd / 4) << 2)]);
    end
end

always @(negedge clk) begin
    case(Load)
        3'b000: begin // lw
            MemRedData[7:0]  = ram[MemAdd + 0];
            MemRedData[15:8] = ram[MemAdd + 1];
            MemRedData[23:16] = ram[MemAdd + 2];
            MemRedData[31:24] = ram[MemAdd + 3];
        end
        3'b001: begin // lb
            //MemRedData[7:0]  = ram[MemAdd + 0];
            MemRedData = {{24{ram[MemAdd + 0][7]}}, ram[MemAdd + 0][7: 0]};
        end
        3'b010: begin // lbu
            MemRedData = {24'd0, ram[MemAdd + 0][7: 0]};
        end
        3'b011: begin // lh
            MemRedData = {{16{ram[MemAdd + 1][7]}}, ram[MemAdd + 1][7: 0], ram[MemAdd + 0][7: 0]};
        end
        3'b100: begin // lhu
            MemRedData = {16'd0, ram[MemAdd + 1][7: 0], ram[MemAdd + 0][7: 0]};
        end
        default: begin //default
            MemRedData[7:0]  = ram[MemAdd + 0];
            MemRedData[15:8] = ram[MemAdd + 1];
            MemRedData[23:16] = ram[MemAdd + 2];
            MemRedData[31:24] = ram[MemAdd + 3];
        end
    endcase
end
endmodule

```

说明：关于单周期 CPU DM 的具体功能在 3.6 部分已详细阐述

4.1.7 MUX（数据选择器）

由于数据选择器的结构基本类似，在这里我只给出 32 位 4 选 1 数据选择器的代码，其他类型的数据选择器同理：

在我的两个实验中，单周期和流水线 CPU 的 MUX 模块设计有所不同，但基本一致，单周期 CPU 将不同的位数和选择单元的多选器集成在一起，因此，具有统一性和普适性，在此给出其代码，流水线的实现请参见工程文件；

MUX 多选器代码：

```
module mux4 #(parameter WIDTH = 8)
    (d0, d1, d2, d3,
     s, y);

    input  [WIDTH-1:0] d0, d1, d2, d3;
    input  [1:0] s;
    output [WIDTH-1:0] y;

    reg [WIDTH-1:0] y_r;

    always @( * ) begin
        case ( s )
            2'b00: y_r = d0;
            2'b01: y_r = d1;
            2'b10: y_r = d2;
            2'b11: y_r = d3;
            default: ;
        endcase
    end // end always
    assign y = y_r;
endmodule
```

说明：因为此 MUX 的设计具有普适性，在调用此模块时，应注意要主动声明位宽，如：

```
mux4 #(32) mux4_32

// 实例化 32 位 4 选 1 的数据选择器 mux4_32
```

4.1.8 EXT（数据拓展器）

同 MUX 一样，EXT 一样具有普适性，因此，在两个 CPU 实验中，他们的结构是一致的，所以这里给出其中一份代码，如下：

```
module EXT( Imm16, EXT0p, Imm32 );
    input  [15:0] Imm16;
```

```

input      EXT0p;
output [31:0] Imm32;

assign Imm32 = (EXT0p) ? {{16{Imm16[15]}}, Imm16} : {16'd0, Imm16}; //
signed-extension or zero extension

endmodule

```

说明：EXT 的具体功能在 3.8 中已经详细说明

4.1.9 Control Unit（控制中心）

单周期 CPU 控制信号比较规范，因此单周期 CPU 的控制中心使用 Verilog HDL 语言集成的 case 语句表示，而流水线的控制中心设计使用传统的逻辑门的方法表示。

单周期 CPU：根据控制单元中指令和控制信号的对应表，结合 case 语句即可生成每条指令的控制信号。由于篇幅的限制，在这里只给出 R 型和其他型的两条指令，其他指令都有着类似的结构，这里不再赘述。

```

module CTRLUNIT(OpCode, FunctCode, Zero,
                ALUtoASel, ALUtoBSel, ALUOpertion,
                EXTSEL, NPCSEL,
                MemWrite,
                RegWrite, RegWriAddSel, RegWriDatSel,
                Load, Store
                );

input [5:0] OpCode;
input [5:0] FunctCode;
input      Zero;
output reg   ALUtoASel;
output reg   ALUtoBSel;
output reg [3:0] ALUOpertion;
output reg   EXTSEL;
output reg [1:0] NPCSEL;
output reg   MemWrite;
output reg   RegWrite;
output reg [1:0] RegWriAddSel;
output reg [1:0] RegWriDatSel;
output reg [2:0] Load;
output reg [1:0] Store;

```

```

always@(OpCode or Zero or FunctCode)
begin
  case (OpCode)
    6'b000000: begin // R
      case(FunctCode)
        6'b100000: begin //add
          {ALUtoASel, ALUtoBSel, EXTsel, MemWrite, RegWrite} = 5'b00001;
          {ALUOpertion[3:0], NPCSel[1:0], RegWriAddSel[1:0],
RegWriDatSel[1:0]} = 10'b0001_00_00_00;
          {Load[2:0], Store[1:0]} = 5'bxxx_xx;
        end
        6'b100010: begin //sub
          {ALUtoASel, ALUtoBSel, EXTsel, MemWrite, RegWrite} = 5'b00001;
          {ALUOpertion[3:0], NPCSel[1:0], RegWriAddSel[1:0],
RegWriDatSel[1:0]} = 10'b0010_00_00_00;
          {Load[2:0], Store[1:0]} = 5'bxxx_xx;
        end
        【此处省略...】
      endcase
    end
  //I
  6'b001000: begin //addi
    {ALUtoASel, ALUtoBSel, EXTsel, MemWrite, RegWrite} = 5'b01101;
    {ALUOpertion[3:0], NPCSel[1:0], RegWriAddSel[1:0], RegWriDatSel[1:0]} =
10'b0001_00_01_00;
    {Load[2:0], Store[1:0]} = 5'bxxx_xx;
  end
  6'b001101: begin //ori
    {ALUtoASel, ALUtoBSel, EXTsel, MemWrite, RegWrite} = 5'b01101;
    {ALUOpertion[3:0], NPCSel[1:0], RegWriAddSel[1:0], RegWriDatSel[1:0]} =
10'b0100_00_01_00;
    {Load[2:0], Store[1:0]} = 5'bxxx_xx;
    ...
  end
endcase
end

endmodule

```

4.2 流水线 CPU 详细设计

4.2.1 IF 阶段模块

① PC 寄存器

```
module pipepc(npc, wpc, clk, clrn, pc);
    input [31:0] npc;
    input      wpc, clrn, clk;
    output [31:0] pc;
    dffe32 program_counter(npc, clk, clrn, wpc, pc);
endmodule
```

以下是 dffe32 详细代码，详细描述见 3.11

```
module dffe32(d, clk, clrn, e, q);

    input [31:0] d;
    input      clk, clrn, e;
    output [31:0] q;

    reg [31:0] q;
    always @(posedge clk or posedge clrn) begin
        if (clrn) begin
            q <= 0;
        end
        else begin
            if(e)
                q <= d;
            end
        end
    end
endmodule
```

② IF 级的组合电路（主要模块是指令存储器 IM）

```
module pipeif(pcsourse, ins, pc, bpc, rpc, jpc, npc, pc4, inst, f_flush, d_flush);
//module pipeif(pcsourse, ins, pc, bpc, rpc, jpc, npc, pc4, inst);//, f_flush,
d_flush);
    input [31:0] pc, bpc, rpc, jpc; //
    input [1:0] pcsourse;           //
    input [31:0] ins;               //
    input f_flush;
    output reg d_flush;
    output [31:0] npc, pc4, inst;   //
```

```

    assign pc4 = pc + 4;
    assign inst = ins;          //
    always@(*)
    begin
        d_flush <= f_flush;
    end
    mux4x32 next_pc(pc4, bpc, rpc, jpc, pcsource, npc);

    //cla32 pc_plus4(pc, 32'h4, 1'b0, pc4);
endmodule

```

以下是 IM 详细代码，IM 的初始化在仿真时通过\$readme 指令读入：

```

// instruction memory
module im(input  [8:2]  addr,
          output [31:0] dout );

    reg  [31:0] ROM[127:0];

    assign dout = ROM[addr]; // word aligned
endmodule

```

4.2.2 ID 阶段模块

① IF-ID 寄存器：指令保存和 PC+4 的计算

```

module pipeir(pc4, inst, wir, clk, clrn, dpc4, dinst);

    input [31:0] pc4, inst;  //
    input      wir, clk, clrn;

    output [31:0] dpc4, dinst; //pc4
    dffe32 pc_plus4(pc4, clk, clrn, wir, dpc4);
    dffe32 instruction (inst, clk, clrn, wir, dinst);

endmodule

```

② ID 级组合电路（包括控制中心和寄存器模块）

```

module pipeid(mwreg, mrn, ern, ewreg, em2reg, mm2reg, dpc4, dinst, wrn, wdi, ealu,
malu, mmo, wwreg, clk,
clrn, bpc, jpc, pcsource, nostall, wreg, m2reg, wmem, aluc, aluimm, a,b,imm, rn, shift,
jal, jalr, flush, d_flush, mult, mfhi, mflo);

    input [31:0] dpc4, dinst, wdi, ealu, malu, mmo;

    input [4:0] ern, mrn, wrn;

```

```

input      mwreg, ewreg, wwreg;//, em2reg, mm2reg, wwreg;
input [1:0] em2reg, mm2reg;
input      clk, clrn;
input      d_flush;

output [31:0] bpc, jpc, a, b, imm;
output [4:0]  rn;
output [4:0]  aluc;
output [1:0]  pcsource;
output nostall, wreg, wmem, aluimm, shift, jal, jalr, mult, mfhi, mflo;
output [1:0]  m2reg;
output flush;

wire [5:0] op, func;
wire [4:0] rs, rt, rd;
wire [31:0] qa, qb, br_offset;
wire [15:0] ext16;
wire [1:0] fwda, fwdb;
wire      regrt, sext, rsrtequ,e;

assign func = dinst[5:0];
assign op = dinst[31:26];
assign rs = dinst[25:21];
assign rt = dinst[20:16];
assign rd = dinst[15:11];
assign jpc = {dpc4[31:28], dinst[25:0], 2'b00};

pipeidcu cu(mwreg, mrn, ern, ewreg, em2reg, mm2reg, rsrtequ, func, op, rs, rt,
wreg, m2reg, wmem, aluc, regrt, aluimm, fwda, fwdb, nostall, sext, pcsource, shift,
jal, jalr, flush, d_flush, mult, mfhi, mflo);

regfile rf(rs,rt, wdi, wrn, wwreg, ~clk, clrn, qa, qb);
mux2x5 des_reg_no(rd, rt, regrt, rn);
mux4x32 alu_a(qa, ealu, malu, mmo, fwda, a);
mux4x32 alu_b(qb, ealu, malu, mmo, fwdb, b);
assign rsrtequ = ~(a^b);
assign e = sext & dinst[15];
assign ext16 = {16{e}};
assign imm = {ext16, dinst[15:0]};
assign br_offset = {imm[29:0], 2'b00};
//cla32 br_addr(dpc4, br_offset, 1'b0, bpc);
assign bpc = dpc4 + br_offset;
endmodule

```

控制单元：

考虑到控制信号的生成具有规律性，为节省篇幅，这里只给出部分指令的代码：

```
module pipeidcu (mwreg, mrn, ern, ewreg, em2reg, mm2reg, rsrtequ, func, op, rs, rt,
wreg, m2reg,
    wmem, aluc, regrt, aluimm, fwda, fwdb, nostall, sext, pcsource, shift, jal, jalr,
flush, d_flush, mult, mfhi, mflo);

    input mwreg, ewreg, rsrtequ;
    input [1:0] em2reg, mm2reg;
    input [4:0] mrn, ern, rs, rt;
    input [5:0] func, op;
    input d_flush;

    output wreg, wmem, regrt, aluimm, sext, shift, jal, jalr;
    output [1:0] m2reg;
    output [4:0] aluc;
    output [1:0] pcsource;
    output [1:0] fwda, fwdb;
    output nostall;
    output flush;
    output mult, mfhi, mflo;
    reg [1:0] fwda, fwdb;
    reg dd_flush;

    wire r_type = ~|op;
    wire i_add = r_type& func[5]&~func[4]&~func[3]&~func[2]&~func[1]&~func[0]; //
add
    wire i_sub = r_type& func[5]&~func[4]&~func[3]&func[2]& func[1]&~func[0]; //
sub
    【此处省略...】
    // i format
    wire i_addi = ~op[5]&~op[4]& op[3]&~op[2]&~op[1]&~op[0]; // addi 001000
    wire i_addiu= ~op[5]&~op[4]& op[3]&~op[2]&~op[1]& op[0]; // addiu 001001
    【此处省略...】
    // j format
    wire i_j = ~op[5]&~op[4]&~op[3]&~op[2]& op[1]&~op[0]; // j
    wire i_jal = ~op[5]&~op[4]&~op[3]&~op[2]& op[1]& op[0]; // jal

    wire i_rs = i_add | i_sub | i_and | i_or | i_xor | i_mult| i_jr | i_addi | i_addiu|
i_andi | i_ori | i_xori | i_lw | i_sw | i_beq | i_bne | i_sllv| i_srav|
i_srlv|i_slti|i_sltiu;

    wire i_rt = i_add | i_sub | i_and | i_or | i_xor | i_mult| i_sll | i_sllv | i_srl
```

```

| i_srlv | i_sra | i_srav | i_sw | i_beq | i_bne | i_nor | i_slt | i_sltu;

    assign nostall = ~(ewreg & em2reg & (ern != 0) & (i_rs & (ern == rs) | i_rt &
(ern == rt)));
    //assign nostall = ~(ewreg & em2reg & (ern != 0) & (i_rs & (ern == rs) | i_rt
& (ern == rt)) | i_j | i_jal | i_jr | i_beq & rsrtequ | i_bne & ~rsrtequ);

    always @(ewreg or mwreg or ern or mrn or em2reg or mm2reg or mm2reg or rs or
rt)
    begin
        fwda = 2'b00;
        if(ewreg & (ern != 0) & (ern == rs) & ~em2reg)
            begin
                fwda = 2'b01;
            end else begin
                if(mwreg & (mrn != 0) & (mrn == rs) & ~mm2reg)
                    begin
                        fwda = 2'b10;
                    end else begin
                        if(mwreg & (mrn != 0) & (mrn == rs) & mm2reg) begin
                            fwda = 2'b11;
                        end
                    end
                end
            end
        end

        fwdb = 2'b00;
        if(ewreg & (ern != 0) & (ern == rt) & ~em2reg)begin
            fwdb = 2'b01;
        end else begin
            if(mwreg & (mrn != 0) & (mrn == rt) & ~mm2reg)begin
                fwdb = 2'b10;
            end else begin
                if(mwreg & (mrn != 0) & (mrn == rt) & mm2reg)begin
                    fwdb = 2'b11;
                end
            end
        end
        end
        dd_flush = d_flush;
    end

    assign wreg = (i_add | i_addu | i_sub | i_subu | i_and | i_or | i_xor | i_sll | i_sllv
| i_srlv | i_srav | i_nor | i_slt | i_sltu | i_srl | i_sra | i_addi | i_addiu | i_andi
| i_ori | i_xori | i_lw | i_lw | i_lui | i_jal | i_slti | i_sltiu | i_jalr | i_mflo | i_mfhi)

```



```

& nostall & (~dd_flush);
    assign regrt = i_addi | i_addiu|i_andi | i_ori | i_xori | i_lw | i_lui
|i_slti|i_sltiu;

    assign jal = i_jal;
    assign jalr = i_jalr;
    【此处省略...】
    assign wmem = i_sw & nostall;
    assign pcsource[1] = i_jr | i_j | i_jal |i_jalr;
    assign pcsource[0] = i_beq & rsrtequ | i_bne & ~rsrtequ | i_j | i_jal;
    assign flush = (i_j | i_jr |i_jal |i_jalr| i_beq & rsrtequ | i_bne & ~rsrtequ );
endmodule

```

寄存器模块:

```

module regfile(rna, rnb, d, wn, we, clk, clrn, qa, qb);

    input [4:0] rna, rnb, wn;
    input [31:0] d;
    input      we, clk, clrn;
    output [31:0] qa, qb;

    reg [31:0] register [1:31];
    assign qa = (rna == 0) ? 0:register[rna];
    assign qb = (rnb == 0) ? 0:register[rnb];
    integer i;
    always @(posedge clk or posedge clrn) begin
        if(clrn) begin

            for(i = 1; i <32; i = i + 1)
                register[i] <= 0;
            end else if((wn != 0) && we)
                register[wn] <= d;

        end
    end
endmodule

```

4.2.3 EXE 阶段模块

① ID-EXE 寄存器

```

module pipedereg(dwreg, dm2reg, dwmem, daluc, daluimm, da, db, dimm, drn, dshift,
djal,djalr, dpc4, clk, clrn,
                ewreg,em2reg ,ewmem, ealuc, ealuimm, ea, eb, eimm, ern, eshift,
ejal,ejalr, epc4,
                dmult, dmfhi, dmflo, emult, emfhi, emflo);

```

```

input [31:0] da, db, dimm, dpc4;
input [4:0] drn;
input [4:0] daluc;
input      dwreg, dwmem, daluimm, dshift, djal, djalr;
input [1:0] dm2reg;
input dmult, dmfhi, dmflo;
input      clk, clrn;

output [31:0] ea, eb, eimm, epc4;
output [4:0] ern;
output [4:0] ealuc;
output      ewreg, ewmem, ealuimm, eshift, ejal, ejalr;
output [1:0] em2reg;
output emult, emfhi, emflo;
reg [31:0]   ea, eb, eimm, epc4;
reg [4:0]    ern;
reg [4:0]    ealuc;

reg          ewreg, ewmem, ealuimm, eshift, ejal, ejalr;
reg [1:0] em2reg;
reg emult, emfhi, emflo;
always @ (posedge clrn or posedge clk)begin
if(clrn)
begin
    ewreg <= 0;
    em2reg <= 0;
    ewmem <= 0;
    ealuc <= 0;
    ealuimm <= 0;
    ea <= 0;
    eb <= 0;
    eimm <= 0;
    ern <= 0;
    eshift <= 0;
    ejal <= 0;
    ejalr <= 0;
    epc4 <= 0;
    emult <= 0;
    emfhi <= 0;
    emflo <= 0;

end else begin
    ewreg <= dwreg;
    em2reg <= dm2reg;

```

```

    ewmem <= dwmem;
    ealuc <= daluc;
    ealuimm <= daluimm;
    ea <= da;
    eb <= db;
    eimm <= dimm;
    ern <= drn;
    eshift <= dshift;
    ejal <= djal;
    ejalr <= djalr;
    epc4 <= dpc4;
    emult <= dmult;
    emfhi <= dmfhi;
    emflo <= dmflo;

    end
    end

```

```
endmodule
```

② EXE 级的组合电路（包括 ALU）

```

module pipeexe(ealuc, ealuimm, ea, eb, eimm, eshift, ern0, epc4, ejal, ejalr, ern,
    ealu, mult_alu);

    input [31:0] ea, eb, eimm, epc4;
    input [4:0] ern0;
    input [4:0] ealuc;
    input      ealuimm, eshift, ejal, ejalr;

    output [31:0] ealu;
    output [4:0] ern;
    output [63:0] mult_alu;
    wire [31:0] alua, alub, sa, ealu0, epc8;
    wire z;
    wire [63:0] mult_alu;
    assign sa = {27'b0, eimm[10:6]};

    assign epc8 = epc4 + 4;
    mux2x32 alu_ina(ea, sa, eshift, alua);
    mux2x32 alu_inb(eb, eimm, ealuimm, alub);
    mux2x32 save_pc8(ealu0, epc4, ejal|ejalr, ealu);
    assign ern = ern0 | {5{ejal}};
    alu al_unit(alua, alub, ealuc, ealu0, z, mult_alu);

    endmodule

```

4.2.4 MEM 阶段模块

① EXE-MEM 寄存器

```

module pipeemreg(ewreg, em2reg, ewmem, ealu, eb, ern, clk, clrn, mwreg,
                 mm2reg, mwmem, malu, mb, mrn, emfhi, emflo, mmfhi, mmflo);

    input [31:0] ealu, eb;
    input [4:0] ern;
    input      ewreg, ewmem;
    input [1:0] em2reg;
    input      clk, clrn;
    input emfhi, emflo;

    output [31:0] malu, mb;
    output [4:0] mrn;
    output      mwreg, mwmem;
    output [1:0] mm2reg;
    output mmfhi, mmflo;
    reg [31:0] malu, mb;
    reg [4:0] mrn;
    reg      mwreg, mwmem;
    reg mmflo, mmfhi;
    reg [1:0] mm2reg;

    always @(posedge clrn or posedge clk) begin
        if (clrn)begin
            mwreg <= 0;
            mm2reg <= 0;
            mwmem <= 0;
            malu <= 0;
            mb <= 0;
            mrn <= 0;
            mmflo <= 0;
            mmfhi <= 0;
        end
        else begin

            mwreg <= ewreg;
            mm2reg <= em2reg;
            mwmem <= ewmem;
            malu <= ealu;
            mb <= eb;
            mrn <= ern;
        end
    end
endmodule

```

```

        mmfhi <= emfhi;
        mmflo <= emflo;
    end
end
endmodule

```

② Data Memory 模块

```

// data memory
module dm(clk, DMWr, addr, din, dout);
    input      clk;
    input      DMWr;
    input [8:2] addr;
    input [31:0] din;
    output [31:0] dout;

    reg [31:0] dmem[127:0];
    always @(posedge clk)
        if (DMWr) begin
            dmem[addr[8:2]] <= din;
            $display("dmem[0x%8X] = 0x%8X,", addr << 2, din);
        end

    assign dout = dmem[addr[8:2]];

endmodule

```

4.2.5 WB 阶段模块

① MEM-WB 寄存器

```

module pipemwreg(mwreg, mm2reg, mmo, malu, mrn, clk, clrn,
                wwreg, wm2reg, wmo, walu, wrn, mmfhi, mmflo, wmfhi, wmflo);

    input [31:0] mmo, malu;
    input [4:0] mrn;
    input      mwreg;
    input [1:0] mm2reg;
    input      clk, clrn;
    input mmfhi, mmflo;

    output [31:0] wmo, walu;
    output [4:0] wrn;
    output      wwreg;

```

```
output [1:0] wm2reg;
output wmfhi, wmflo;
reg [31:0] wmo, walu;
reg [4:0] wrn;
reg      wwreg;
reg [1:0] wm2reg;
reg wmfhi, wmflo;
always @(posedge clrn or posedge clk) begin
    if (clrn) begin
        wwreg <= 0;
        wm2reg <= 0;
        wmo <= 0;
        walu <= 0;
        wrn <= 0;
        wmfhi <= 0;
        wmflo <= 0;
    end
    else begin
        wwreg <= mwreg;
        wm2reg <= mm2reg;
        wmo <= mmo;
        walu <= malu;
        wrn <= mrn;
        wmfhi <= mmfhi;
        wmflo <= mmflo;
    end
end
endmodule
```

② WB 级的组合电路相对简单，已经集成在顶层模块中。

5 测试及结果分析

5.1 单周期 CPU 测试及结果分析

5.1.1 代码仿真及分析

单周期没有涉及冒险，在编写测试代码时，可以将所有支持指令编入同一个汇编代码文件，通过 ModelSim 软件和 Mars 软件检测我们设计的 CPU 是否能够顺利工作。

单周期 CPU 支持的指令见 3.1 节；

根据该指令编写的汇编代码如下所示(不包括半字和字节的加载指令和存储指令)：

汇编代码块 1：

#	Assembly	Description	Address	Machine
main:	addi \$2, \$0, 5	# initialize \$2 = 5	0	20020005
	addi \$3, \$0, 12	# initialize \$3 = 12	4	2003000c
	addi \$7, \$3, -9	# initialize \$7 = 3	8	2067fff7
	addi \$1, \$0, 76	# initialize \$1 = 76	c	2001004c
call_a:	jalr \$31,\$1	# jump to cal	10	0020f809
	or \$4, \$7, \$2	# \$4 <= 3 or 5 = 7	14	00e22025
	and \$5, \$3, \$4	# \$5 <= 12 and 7 = 4	18	00642824
	add \$5, \$5, \$4	# \$5 = 4 + 7 = 11	1c	00a42820
	beq \$5, \$7, word	# shouldn't be taken	20	10a70017
	slt \$4, \$3, \$4	# \$4 = 12 < 7 = 0	24	0064202a
	beq \$4, \$0, around	# should be taken	28	10800001
	addi \$5, \$0, 0	# shouldn't happen	2c	20050000
around:	slt \$4, \$7, \$2	# \$4 = 3 < 5 = 1	30	00e2202a
	add \$7, \$4, \$5	# \$7 = 1 + 11 = 12	34	00853820
	sub \$7, \$7, \$2	# \$7 = 12 - 5 = 7	38	00e23822
	sw \$7, 68(\$3)	# [80] = 7	3c	ac670044
	lw \$2, 80(\$0)	# \$2 = [80] = 7	40	8c020050
	j word	# should be taken	44	08000020
	addi \$2, \$0, 1	# shouldn't happen	48	20020001
cal:	sll \$7, \$7, 1	# \$7 << 1 = 6	4c	00073840
call_b:	jal cal2	# jump to cal2	50	0c000017
	addi \$31,\$0,20	# \$31<= 20	54	201f0014
	jr \$31	# return to call_a	58	03e00008

```

cal2:  lui $1, 0x55AA    # $1 <= 0x55AA0000          5c    3c0155aa
        slti $1, $1, 0x55AA # $1 <= 0                60    282155aa
        bne $1, $0, word  # shouldn't be taken         64    14200006
        ori $7, $7, 5    # $7 <= 6 or 5 = 7           68    34e70005
        andi $1, $7, 5    # $1 <= 7 and 5 = 5          6c    30e10005
        addu $1, $7, $1   # $1 = 7+5 = 12             70    00e10821
        subu $1, $1, $1   # $1 =12-12 = 0             74    00210823
        srl $7, $7, 1     # $7 >> 1 = 3               78    00073842
        jr $31            # return to call_b           7c    03e00008
word:   sw $2, 84($0)     # write adr 84 = 7           80    ac020054
loop:   j loop

```

汇编代码运行结果:

```

# $0 = 0 # $1 = 0 # $2 = 7 # $3 = c
# $4 = 1 # $5 = b # $7 = 7 # $31 = 14h
# [0x50] = 7 [0x54] = 7

```

MARS 运行结果:

1. 寄存器:

\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000007
\$v1	3	0x0000000c
\$a0	4	0x00000001
\$a1	5	0x0000000b
\$a2	6	0x00000000
\$a3	7	0x00000007
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x00001800
\$sp	29	0x00003ffc
\$fp	30	0x00000000
\$ra	31	0x00000014
pc		0x00000084
hi		0x00000000
lo		0x00000000

图 14 MARS 在运行代码块 1 后的寄存器结果

说明: 其中\$gp 和\$sp 为系统初始化值

2. Memory 变化:

Value (+10)	Value (+14)
0x0020f809	0x00e22025
0x00e2202a	0x00853820
0x50	0x54
0x00000007	0x00000007

图 15 MARS 在运行代码块 1 后的 memory 变化

汇编代码块 2：对存储指令和加载指令的汇编代码测试：

```
# Attention: Mars, Settings -> Memory Configuration -> Compact, Data at address 0
# lui ori subu addu add sub nor or and slt stlu addi
# sll srl sra sllv srlv srav
# sw sh sb
# lw lh lhu lb lbu

lui    $3, 0x9876      # $3=0x98760000      # 3c039876
ori    $2, $0, 0x1234  # $2=0x1234      # 34021234
subu   $8, $3, $2      # $8=0x98760000-0x1234=0x9875edcc      # 00624023
xor    $9, $8, $3      # $9=0x9875edcc^0x98760000=0x0003edcc      # 01034826
addu   $10, $9, $8      # $10=0x0003edcc+0x9875edcc=0x9879db98      # 01285021
add    $10, $10, $2      # $10=0x9879db98+0x1234=0x9879edcc      # 01425020
sub    $11, $10, $3      # $11=0x9879edcc-0x98760000=0x0003edcc      # 01435822
nor    $12, $11, $10      # $12=~(0x0003edcc|0x9879edcc)=0x67841233      # 016a6027
or     $13, $11, $10      # $13=0x0003edcc|0x9879edcc=0x987bedcc      # 016a6825
and    $14, $11, $10      # $14=0x0003edcc&0x9879edcc=0x0001edcc      # 016a7024
slt    $19, $13, $12      # $19=(0x987bedcc<0x67841233)=1      # 01ac982a
sltu   $20, $13, $12      # $20=(0x987bedcc<0x67841233)=0      # 01aca02b
sll    $8, $8, 3          # $8=0x9875edcc<<3=0xc3af6e60      # 000840c0
srl    $9, $8, 0x10       # $9=0xc3af6e60>>16=0xc3af      # 00084c02
sra    $10, $8, 0x1d      # $10=0xc3af6e60>>>29=0xffffffffe      # 00085743
ori    $11, $0, 0x3410     # $11=0x3410      # 340b3410
sllv   $12, $8, $11       # $12=0xc3af6e60<<16=0x6e600000      # 01686004
srlv   $13, $8, $11       # $13=0xc3af6e60>>16=0xc3af      # 01686806
srav   $14, $8, $11       # $14=0xc3af6e60>>>16=0xfffffc3af      # 01687007
addu   $4, $2, $3          # $4=0x1234+0x98760000=0x98761234      # 00432021
addi   $29, $0, 0x0       # $29=0      # 201d0000
sw     $4, 0x00($29)       # [0]=0x98761234      # afa40000
sw     $4, 0x04($29)       # [4]=0x98761234      # afa40004
sw     $4, 0x08($29)       # [8]=0x98761234      # afa40008
sh     $8, 0x04($29)       # [4]=0x98766e60 $8=0xc3af6e60      # a7a80004 little
endian
sh     $9, 0x0a($29)       # [8]=0xc3af1234 $9=0xc3af      # a7a9000a little
endian
sb     $10, 0x07($29)       # [4]=0xfe766e60 $10=0xffffffffe      # a3aa0007 little
endian
sb     $8, 0x09($29)       # [8]=0xc3af6034 $8=0xc3af6e60      # a3a80009 little
```

```

endian
sb    $9, 0x08($29)    # [8]=0xc3af60af $9=0xc3af          # a3a90008 little
endian
lw    $8, 0x00($29)    # $8=0x98761234                    # 8fa80000
sw    $8, 0x0c($29)    # [0xc]=0x98761234                  # afa8000c
lh    $9, 0x02($29)    # $9=0xffff9876 [0]=0x98761234      # 87a90002 little
endian, sign extension
sw    $9, 0x10($29)    # [0x10]=0xffff9876                 # afa90010
lhu   $9, 0x02($29)    # $9=0x00009876 [0]=0x98761234      # 97a90002 little
endian, zero extension
sw    $9, 0x14($29)    # [0x14]=0x00009876                 # afa90014
lb    $10, 0x03($29)   # $10=0xffffffff98 [0]=0x98761234    # 83aa0003 little
endian, sign extension
sw    $10, 0x18($29)   # [0x18]=0xffffffff98                 # afaa0018
lbu   $10, 0x03($29)   # $10=0x00000098 [0]=0x98761234      # 93aa0003 little
endian, zero extension
sw    $10, 0x1c($29)   # [0x1c]=0x00000098                 # afaa001c
lbu   $10, 0x01($29)   # $10=0x00000012 [0]=0x98761234      # 93aa0001 little
endian, zero extension
sw    $10, 0x20($29)   # [0x20]=0x00000012                 # afaa0020

```

汇编代码运行结果：

```

# $0=0          $1=0          $2=0x00001234    $3=0x98760000
# $4=0x98761234 $5=0          $6=0          $7=0
# $8=0x98761234 $9=0x00009876 $10=0x00000012 $11=0x00003410
# $12=0x6e600000 $13=0x0000c3af $14=0xffffc3af $19=0x00000001
# $29=0
# [0]=0x98761234 [4]=0xfe766e60 [8]=0xc3af60af [0xc]=0x98761234
# [0x10]=0xffff9876 [0x14]=0x00009876 [0x18]=0xffffffff98 [0x1c]=0x00000098
# [0x20]=0x00000012

```

5.1.2 仿真测试结果

对汇编代码块 1，CPU 运行结果如下：

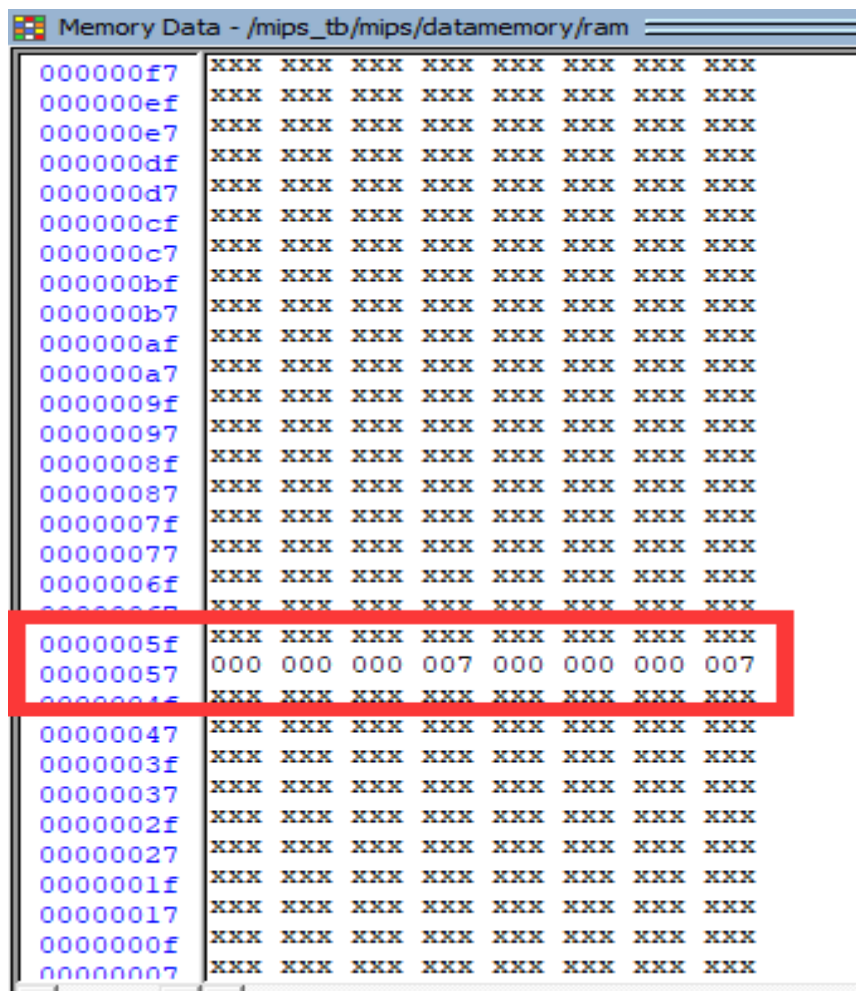
1. 寄存器如图所示：

Memory Data - /mips_tb/mips/cpu/registerfile/rf									
0000001f	00000014	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000017	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0000000f	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000007	00000007	00000000	0000000b	00000001	0000000c	00000007	00000000	00000000	00000014

图 16 单周期 CPU 在运行代码块 1 后的寄存器结果

说明：即使\$0 被置数值，但在实际使用时，从\$0 中读出的数值只能是 0，因此测试结果同 MARS 完全一致。

2. memory 如图所示：



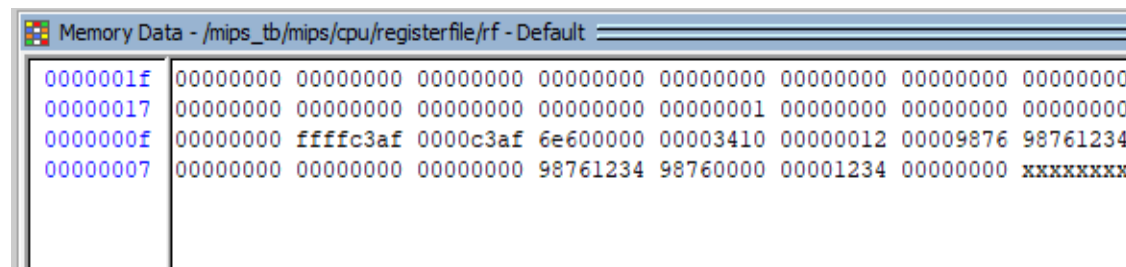
Address	Value
000000f7	xxx xxx xxx xxx xxx xxx xxx xxx
000000ef	xxx xxx xxx xxx xxx xxx xxx xxx
000000e7	xxx xxx xxx xxx xxx xxx xxx xxx
000000df	xxx xxx xxx xxx xxx xxx xxx xxx
000000d7	xxx xxx xxx xxx xxx xxx xxx xxx
000000cf	xxx xxx xxx xxx xxx xxx xxx xxx
000000c7	xxx xxx xxx xxx xxx xxx xxx xxx
000000bf	xxx xxx xxx xxx xxx xxx xxx xxx
000000b7	xxx xxx xxx xxx xxx xxx xxx xxx
000000af	xxx xxx xxx xxx xxx xxx xxx xxx
000000a7	xxx xxx xxx xxx xxx xxx xxx xxx
0000009f	xxx xxx xxx xxx xxx xxx xxx xxx
00000097	xxx xxx xxx xxx xxx xxx xxx xxx
0000008f	xxx xxx xxx xxx xxx xxx xxx xxx
00000087	xxx xxx xxx xxx xxx xxx xxx xxx
0000007f	xxx xxx xxx xxx xxx xxx xxx xxx
00000077	xxx xxx xxx xxx xxx xxx xxx xxx
0000006f	xxx xxx xxx xxx xxx xxx xxx xxx
0000005f	xxx xxx xxx xxx xxx xxx xxx xxx
00000057	000 000 000 007 000 000 000 007
0000004f	xxx xxx xxx xxx xxx xxx xxx xxx
00000047	xxx xxx xxx xxx xxx xxx xxx xxx
0000003f	xxx xxx xxx xxx xxx xxx xxx xxx
00000037	xxx xxx xxx xxx xxx xxx xxx xxx
0000002f	xxx xxx xxx xxx xxx xxx xxx xxx
00000027	xxx xxx xxx xxx xxx xxx xxx xxx
0000001f	xxx xxx xxx xxx xxx xxx xxx xxx
00000017	xxx xxx xxx xxx xxx xxx xxx xxx
0000000f	xxx xxx xxx xxx xxx xxx xxx xxx
00000007	xxx xxx xxx xxx xxx xxx xxx xxx

图 17 单周期 CPU 在运行代码块 2 后的 memory 变化

说明：在[0x50][0x54]的值变成 7，测试结果同 MARS 一致。

对汇编代码块 2，CPU 运行结果如下：

1. 寄存器结果：



Address	Value
0000001f	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000017	00000000 00000000 00000000 00000000 00000001 00000000 00000000 00000000
0000000f	00000000 ffffc3af 0000c3af 6e600000 00003410 00000012 00009876 98761234
00000007	00000000 00000000 00000000 98761234 98760000 00001234 00000000 xxxxxxxx

图 18 单周期 CPU 在运行代码块 2 后的寄存器变化

说明：同代码逻辑结果一致。

2. memory 结果：

Address	Value
000000f7	xxx xxx xxx xxx xxx xxx xxx
000000ef	xxx xxx xxx xxx xxx xxx xxx
000000e7	xxx xxx xxx xxx xxx xxx xxx
000000df	xxx xxx xxx xxx xxx xxx xxx
000000d7	xxx xxx xxx xxx xxx xxx xxx
000000cf	xxx xxx xxx xxx xxx xxx xxx
000000c7	xxx xxx xxx xxx xxx xxx xxx
000000bf	xxx xxx xxx xxx xxx xxx xxx
000000b7	xxx xxx xxx xxx xxx xxx xxx
000000af	xxx xxx xxx xxx xxx xxx xxx
000000a7	xxx xxx xxx xxx xxx xxx xxx
0000009f	xxx xxx xxx xxx xxx xxx xxx
00000097	xxx xxx xxx xxx xxx xxx xxx
0000008f	xxx xxx xxx xxx xxx xxx xxx
00000087	xxx xxx xxx xxx xxx xxx xxx
0000007f	xxx xxx xxx xxx xxx xxx xxx
00000077	xxx xxx xxx xxx xxx xxx xxx
0000006f	xxx xxx xxx xxx xxx xxx xxx
00000067	xxx xxx xxx xxx xxx xxx xxx
0000005f	xxx xxx xxx xxx xxx xxx xxx
00000057	xxx xxx xxx xxx xxx xxx xxx
0000004f	xxx xxx xxx xxx xxx xxx xxx
00000047	xxx xxx xxx xxx xxx xxx xxx
0000003f	xxx xxx xxx xxx xxx xxx xxx
00000037	xxx xxx xxx xxx xxx xxx xxx
0000002f	xxx xxx xxx xxx 000 000 000 012
00000027	000 000 000 098 0ff 0ff 0ff 098
0000001f	000 000 098 076 0ff 0ff 098 076
00000017	098 076 012 034 0c3 0af 060 0af
0000000f	0fe 076 06e 060 098 076 012 034
00000007	

图 19 单周期 CPU 在运行代码块之后的 memory 变化

说明：同代码逻辑结果一致。

总结：单周期 CPU 指令测试结束，均符合实验预期。

5.2 流水线 CPU 测试及结果分析

5.2.1 代码仿真及分析

流水线因为其中涉及冒险，所以对具体代码要分开测试并阐述。

总体测试汇编代码块 3：

```
# mipstest_pipelined.asm

### Make sure the following Settings :
# Settings -> Memory Configuration -> Compact, Text at address 0
# You could use it to test if there is data hazard and control hazard.
```

If successful, it should write value 12 to address 80 and 84, and register \$7 should be 12

#	Assembly	Description	Address	Machine
# Test if there is data hazard				
main:	addi \$2, \$0, 5	# initialize \$2 = 5	00	20020005
	ori \$3, \$0, 12	# initialize \$3 = 12	04	3403000c
	subu \$1, \$3, \$2	# \$1 = 12 - 5 = 7	08	00620823
	srl \$7, \$1, 1	# \$7 = 7 >> 1 = 3	0c	00013842
call_a:	j a	# jump to a	10	08000015
	or \$4, \$7, \$2	# \$4 = (3 or 5) = 7	14	00e22025
	and \$5, \$3, \$4	# \$5 = (12 and 7) = 4	18	00642824
	add \$5, \$5, \$4	# \$5 = 4 + 7 = 11	1c	00a42820
	beq \$5, \$7, end	# should not be taken	20	10a70018
	sltu \$4, \$3, \$4	# \$4 = (12 < 7) = 0	24	0064202b
#Test if there is control hazard				
	beq \$4, \$0, around	# should be taken	28	10800003
	addi \$5, \$0, 0	# should not happen	2c	20050000
	addi \$5, \$0, 0	# should not happen	30	20050000
	addi \$5, \$0, 0	# should not happen	34	20050000
around:	slt \$4, \$7, \$2	# \$4 = 3 < 5 = 1	38	00e2202a
	addu \$7, \$4, \$5	# \$7 = 1 + 11 = 12	3c	00853821
	sub \$7, \$7, \$2	# \$7 = 12 - 5 = 7	40	00e23822
	sw \$7, 68(\$3)	# [80] = 7	44	ac670044
	lw \$2, 80(\$0)	# \$2 = [80] = 7	48	8c020050
	j end	# should be taken	4c	08000021
	addi \$2, \$0, 1	# should not happen	50	20020001
a:	sll \$7, \$7, 2	# \$7 = 3 << 2 = 12	54	00073880
call_b:	jal b	# jump to b	58	0c000019
	addi \$31, \$0, 20	# \$31 <= 20	5c	201f0014
	jr \$31	# return to call_a	60	03e00008
b:	lui \$1, 0xFFAA	# \$1 <= 0xFFAA0000	64	3c01ffaa
	slt \$1, \$7, \$1	# \$1 <= 0	68	00e1082a
	bne \$1, \$0, end	# should not be taken	6c	14200005
	sub \$7, \$7, \$2	# \$7 = 12 - 5 = 7	70	00e23822
	srl \$7, \$7, 1	# \$7 = 7 >> 1 = 3	74	00073842
	nor \$1, \$7, \$1	# \$1 = 0xFFFFF000	78	00e10827
	sltu \$1, \$1, \$7	# \$1 <= 0	7c	0027082b
	jr \$31	# return to call_b	80	03e00008
# Test if there is load use hazard				
end:	sw \$3, 84(\$0)	# [84] = 12	84	ac030054
	lw \$7, 72(\$3)	# \$7 = [84] = 12	88	8c670048
	sw \$7, 68(\$3)	# [80] = 12	8c	ac670044
	lw \$6, 68(\$3)	# \$6 = [80] = 12	88	8c660044

add \$6, \$6, \$5	# \$6 = 12 + 11 = 23	90	00c53020
loop: j loop	# dead loop	94	08000026
汇编运行结果:			
# \$0 = 0 # \$1 = 0 # \$2 = 7 # \$3 = c			
# \$4 = 1 # \$5 = b # \$6 = 17h # \$7 = c			
# \$31 = 14h			

在 MARS 上的运行结果:

1. 寄存器结果:

Registers	Coproc 1	Coproc 0	
Name	Number	Value	
\$zero	0	0x00000000	
\$at	1	0x00000000	
\$v0	2	0x00000007	
\$v1	3	0x0000000c	
\$a0	4	0x00000001	
\$a1	5	0x0000000b	
\$a2	6	0x00000017	
\$a3	7	0x0000000c	
\$t0	8	0x00000000	
\$t1	9	0x00000000	
\$t2	10	0x00000000	
\$t3	11	0x00000000	
\$t4	12	0x00000000	
\$t5	13	0x00000000	
\$t6	14	0x00000000	
\$t7	15	0x00000000	
\$s0	16	0x00000000	
\$s1	17	0x00000000	
\$s2	18	0x00000000	
\$s3	19	0x00000000	
\$s4	20	0x00000000	
\$s5	21	0x00000000	
\$s6	22	0x00000000	
\$s7	23	0x00000000	
\$t8	24	0x00000000	
\$t9	25	0x00000000	
\$k0	26	0x00000000	
\$k1	27	0x00000000	
\$gp	28	0x00001800	
\$sp	29	0x00003ffc	
\$fp	30	0x00000000	
\$ra	31	0x00000014	
pc		0x00000098	
hi		0x00000000	
lo		0x00000000	

图 20 MARS 运行代码块 3 之后的寄存器结果

说明: 其中\$gp 和\$sp 的值为系统初始化, 与本实验无关

2. memory 的值

Value (+10)	Value (+14)
0x08000015	0x00e22025
0x20050000	0x20050000
0x0000000c	0x0000000c
0x00e22025	0x00e22025

图 21 MARS 运行代码块 3 之后的 memory 变化

说明：更改的值有：[0x50] = [0x54] = 0xc;

5.2.2 仿真测试结果

因为流水线 CPU 涉及到数据冒险和控制冒险，因此我先给出汇编代码的最终运行结果，然后再对涉及到冒险的地方进行详细说明。

CPU 运行结果：

1. 寄存器结果：

Memory Data - /pipecomp_tb/U_SCCOMP/U_PIPECPU/id_stage/rf/register								
00000001	00000000	00000007	0000000c	00000001	0000000b	00000017	0000000c	00000000
00000009	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000011	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000019	00000000	00000000	00000000	00000000	00000000	00000000	00000014	

图 22 流水线 CPU 运行代码块 3 之后的寄存器结果

说明：注意这里是从 1 开始；其中非零的几个寄存器值同代码预测以及 MARS 运行结果一致。

2. memory 结果：

Memory Data - /pipecomp_tb/U_SCCOMP/U_DM/dmem								
0000007f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000077	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000006f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000067	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000005f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000057	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000004f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000047	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000003f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000037	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000002f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000027	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000001f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000017	XXXXXXXX	XXXXXXXX	0000000c	0000000c	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0000000f	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
00000007	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX

图 23 流水线 CPU 运行代码块 3 之后的 memory 变化

说明：变化的位置分别是 0x50 和 0x54，即[0x50]=[0x54]=0xc,与代码预测以及 MARS 运行结果一致。

下面具体介绍冒险的运行情况：

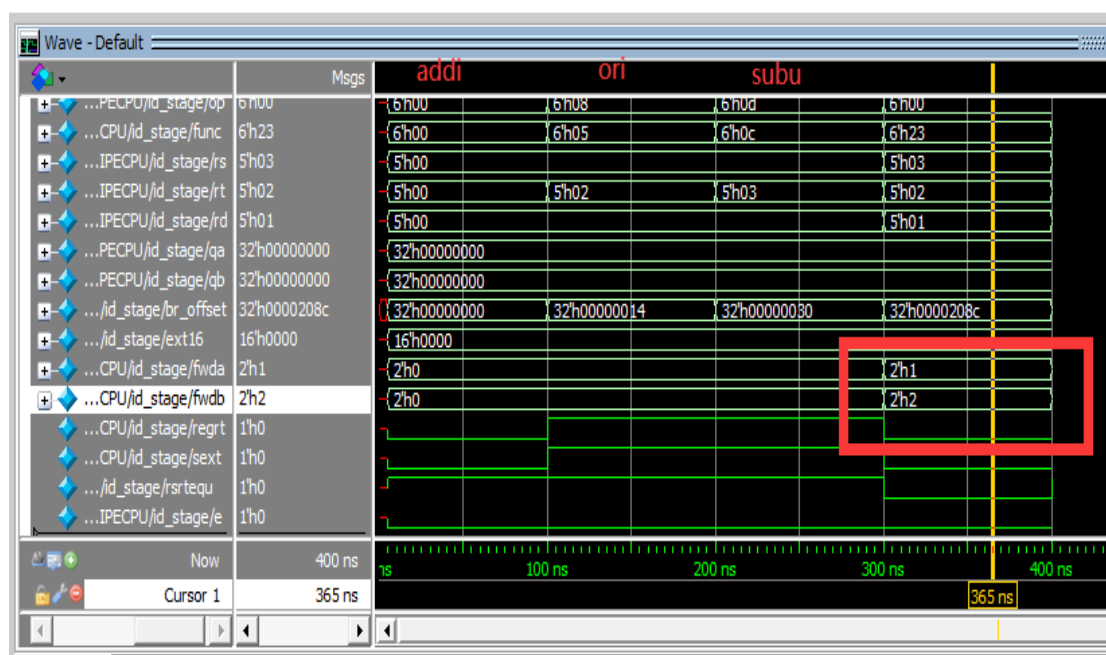
1. 数据冒险：

数据冒险有三类：exe-alu、mem-alu、mem-lw。

1) 上述代码中涉及 exe-alu、mem-alu 冒险的代码：

```
main: addi $2, $0, 5      # initialize $2 = 5    00    20020005
      ori  $3, $0, 12     # initialize $3 = 12   04    3403000c
      subu $1, $3, $2     # $1 = 12 - 5 = 7     08    00620823
```

其中，subu 利用了 exe 阶段和 mem 阶段的 alu 结果，根据前面所述控制信号的指令，此时 fwda 和 fwdb 都应发生旁路，且两值应分别是 fwda=01、fwdb = 10；代码仿真结果如下图所示：



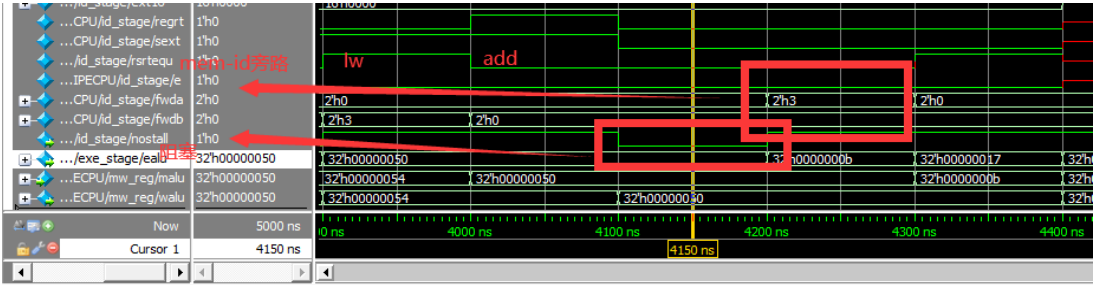
说明：subu 的 ID 阶段(如图所示的第四个周期)，产生的旁路信号 fwda 和 fwdb 分别是 01 和 10，符合实验结果。

2) 上述代码涉及 mem-lw 冒险的在：

```
# Test if there is load use hazard
end:  sw  $3, 84($0)      # [84] = 12          84    ac030054
      lw  $7, 72($3)      # $7 = [84] = 12          88    8c670048
      sw  $7, 68($3)      # [80] = 12          8c    ac670044
      lw  $6, 68($3)      # $6 = [80] = 12          88    8c660044
      add $6, $6, $5      # $6 = 12 + 11 = 23    90    00c53020
```


其中，最后一条指令 `add` 使用了上一条指令的目的寄存器，而上一条指令正是加载指令，因此按照设计的原则，应该延迟一条指令，并在 `add` 延迟之后，旁路将 `mem` 得到的值传送至本指令的 ID 阶段。

代码仿真结果如下图所示：



说明：`add` 的 ID 阶段(如图所示的第三个周期)，因为检测到是 `mem-lw` 数据冒险，所以需要阻塞一个周期，即 `nostall` 信号为 0，推迟一个周期之后，下次执行 `add` 指令时，应该产生的旁路是 `mem-id`，即 `fwda` 为 11。信号量显示如上图所示。

2. 控制冒险：

控制冒险可以分为两类(实际上跳转指令并不算控制冒险)：

1. 跳转指令的控制冒险：

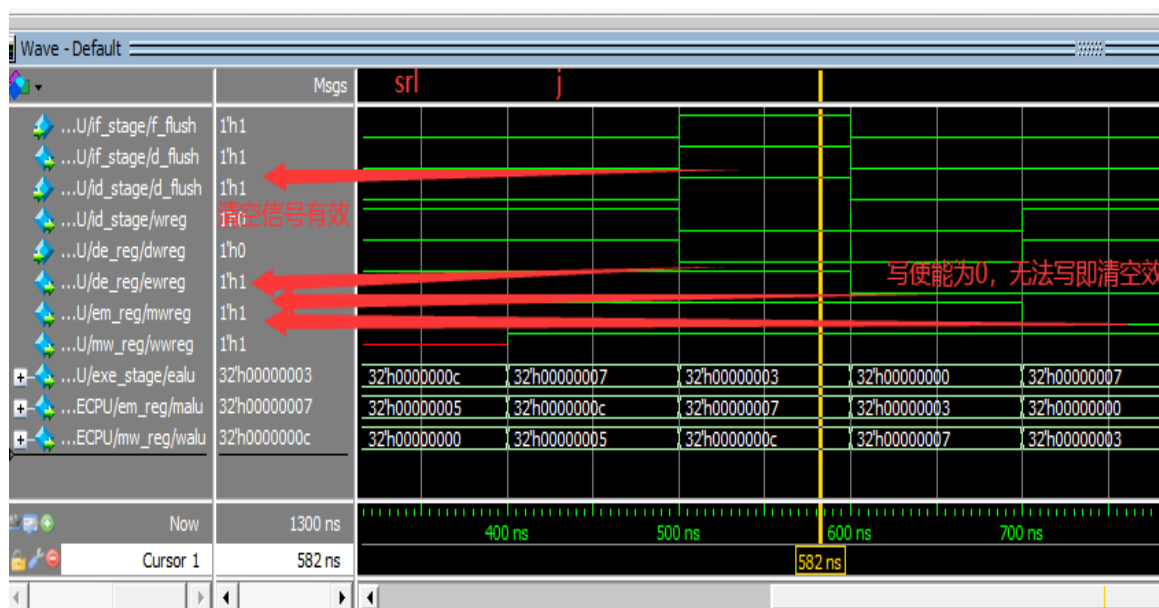
跳转指令实际上并不产生冒险，但因为和分支指令在结构上有类似的地方，而且跳转指令同样会延迟一个周期，所以将它作为冒险处理。简言之，跳转指令的预测总发生一直正确。

在检验代码中的跳转指令：

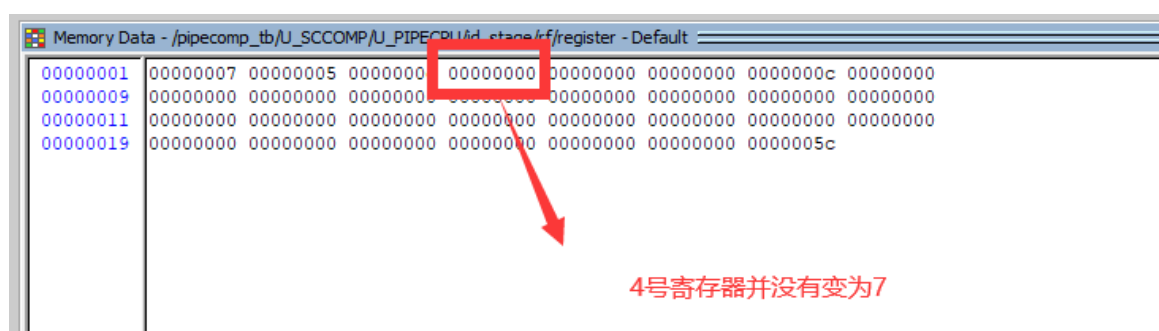
```
call_a: j    a           # jump to a           10    08000015
        or   $4, $7, $2   # $4 = (3 or 5) = 7   14    00e22025
```

说明：`j` 指令发生时，不会立即跳转，因为地址的计算在 ID 阶段完成，所以 `flush` 指令为 1，以防其后的指令 `or` 执行，从而破坏代码原有的逻辑。

代码仿真结果如下图所示：



说明：图中 j 指令的 ID 阶段(即图中的第三个周期)，产生清空信号传至 IF，下一条指令的寄存器写使能为 0，无法完成目的操作，效果同预期一致。



说明：4 号寄存器值没有变化，j 后面的指令没有执行。

2. 分支指令的控制冒险：

同跳转指令一样，分支指令的决策同样在 ID 级做出，当然，由于分支指令并不一定会发生，所以它不一定会延迟一个周期。由于算术逻辑单元在 EXE，我们需要在 ID 阶段使用一个异或门直接判断寄存器数据是否相等。若跳转发生，则同跳转指令一样，否则按照原来顺序执行。

上述测试代码中分支指令的使用有：

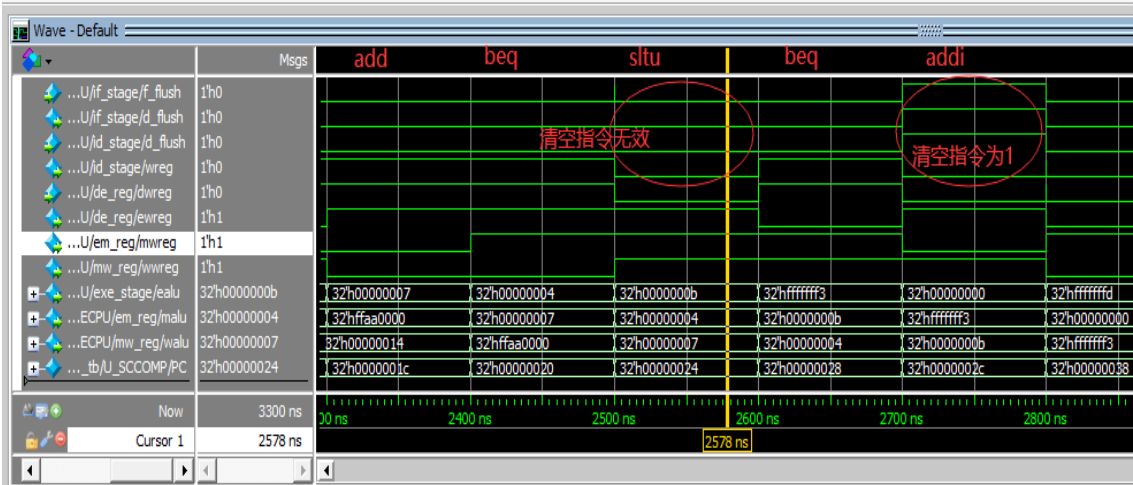
```

beq $5, $7, end    # should not be taken  20    10a70018
sltu $4, $3, $4    # $4 = (12 < 7) = 0    24    0064202b
#Test if there is control hazard
beq $4, $0, around # should be taken      28    10800003
addi $5, $0, 0     # should not happen    2c    20050000

```

说明：第一个分支指令不会发生，第二个分支指令会发生，而它之后的指令将被清空。

代码仿真结果如下图所示：



说明：对于第一个 `beq` 指令的 ID 阶段(图中所示的第三个周期)，因为分支指令没有发生，所以清空指令无效，按照原顺序执行；对于第二个指令的 ID 阶段(途中所示的第五个周期)，分支指令发生，下一条 `addi` 指令清空。

总结：流水线 CPU 支持的指令均顺利实现，数据冒险得到解决，控制冒险的预测机制保证逻辑的正确性。

6 实验遇到的问题和心得

- 整体观念。CPU 设计整体看来是一个复杂的工程，尤其是流水线 CPU 的架构，在动手写代码之前，应当首先完善数据通路图、控制信号、模块的规划等过程，各个模块包括 PC、IM、DM、ALU、Control Unit 等应当明确它们的接口和作用。本实验的难点应该是开始的架构，在得出一个完整的 CPU 架构之后，增加指令和修改应该不是困难的事，而从零到架构出一个 CPU 框架要困难的多，无从下手之时，可以参考书目《计算机原理与设计：Verilog HDL 版》，跟随作者的脚步，从零开始。
- 兼容性。虽然上述表明后期的增加指令比初期架构容易，但同样是一个繁琐的工作，如果初期的某个地方没有设计好、没有考虑到后面可能的指令，可能会带来较大的困扰。比如，alu 操作码的位数，起初因为支持的指令较少，且 alu 能够支持的操作较少，将 alu 操作码设计为 3 位，后面发现 3 位不够支持后续指令，将其修改为 4 位。所以涉及到的接口大小都要随之更改，应当说较为繁琐。此外还有，存储器的基础大小，若涉及到字的操作，应以 8 位为最小单元。实际上，考虑后期的兼容性，仍然属于前面所说的整体观念。
- 触发条件。触发条件是一个隐性的问题，一旦出错较难发现。在单周期 CPU 的控制单元控制信号的产生里，激励信号表达式如果写成 `always@(OpCode or Zero)`，在 subu 指令时可能出现错误，因为 subu 的操作码是 0，后来将其改成 `always@(*)`，顺利完成。
- 模块化处理。通过这次实验，我更加理解了 Verilog HDL 语言的模块化特性，在单周期 CPU 根据物理单元分块，最后通过顶层模块将各个模块连接起来；而在流水线 CPU 中，不仅仅要在按照物理单元模块化，还要根据阶段进行模块化，因为一个阶段对应一个周期，这样的划分有利于理清逻辑和思路。
- 关于乘法指令 mult。根据 MIPS-CPU 指令集，以及单周期 CPU 问答时临时增加乘法指令的经验，我尝试将乘法指令在流水线上实现，乘法指令顺利实现，接着将 mfhi/mflo 实现，功能顺利实现，但是，因为 HI 和 LO 寄存器的读写，

类似于 lw，需要在 MEM 阶段之后得到值，涉及到冒险且大大更新了原有数据通路，遂放弃。

- 测试平台(test bench)的编写。在单周期 CPU 设计完成之后，测试文件编写的错误，浪费了我许多时间。测试文件，主要是时钟信号的变化，需要理解之后才能完成，比如重置信号 reset 应当在第一个周期内震荡以下，将 PC 置零，然后回归，不然 PC 将无法获得初值。此外，时钟信号应当持续震荡，模拟实际场景，否则我们根本无法看到 CPU 的各种变化。
- Verilog HDL 的语法知识。这次实验，我重新温习了一年级时学过的 Verilog 知识，当时课堂上所讲，在实际应用中才能深切感受到，对阻塞赋值与非阻塞赋值、过程赋值语句与持续赋值语句、wire 类型与 reg 类型等概念有了更清晰的认识，也学会了在合适的地方使用合适的表达式和语句。

参考文献

- [1] 李亚民, 计算机原理与设计: Verilog HDL 版; 北京: 清华大学出版社, 2011. 6, ISBN: 978-7-302-25109-5.
- [2] 夏宇闻, Verilog 数字系统设计教程, 北京航空航天大学出版社, 2008. 6, ISBN: 978-7-811-24309-3.
- [3] David A. Patterson & John L. Hennessy, 《计算机组成与设计: 硬件/软件接口 (原书第 5 版)》, 机械工业出版社.
- [4] MIPS-C 指令集