

# 流水线 MIPS 处理器设计

无 08 李煜彤 2020010841

## 目录

实验目的.....	2
预备工作.....	3
指令格式表 .....	3
控制信号 .....	4
冒险处理 .....	6
结构冒险 .....	6
数据冒险 .....	6
控制冒险 .....	9
JumpRegister 冒险 .....	11
数据通路 .....	12
寄存器寄存清单 .....	13
CPU 代码编写 .....	14
模块设计 .....	14
ForwardingUnit: 转发单元 .....	15
HazardUnit: load-use 冒险单元 .....	16
BranchUnit: 分支指令处理单元 .....	17
PC: PC 寄存器及其选择.....	18
连线（顶层设计） .....	19
CPU 代码调试 .....	20
指令测试 .....	20
冒险测试 .....	29
字符串查找的实现 .....	31
仿真实现 .....	31
代码编写 .....	31
将较复杂代码写入 vivado 的指令存储器 .....	33
仿真结果 .....	33
添加外设 .....	34
添加外设后的仿真结果 .....	37
硬件调试 .....	38
性能分析.....	39
时序性能 .....	39
逻辑资源 .....	40
文件清单.....	41
心得体会.....	42

# 实验目的

运用春季学期课程《数字逻辑与处理器基础》和《数字逻辑与处理器基础实验》所学知识，在此前完成的单、多周期处理器的基础上，将其改进为流水线结构，在其上完成字符串搜索算法，并与单、多周期处理器进行性能比较。

# 预备工作

## 指令格式表

指令格式表						
	OpCode [5:0]	Rs [4:0]	Rt [4:0]	Rd [4:0]	Shamt [4:0]	Funct [5:0]
	OpCode [5:0]	Rs [4:0]	Rt [4:0]	imm/offset [15:0]		
	OpCode [5:0]	target [25:0]				
存取指令						
lw rt,offset(rs)	0x23	rs	rt	offset		
sw rt,offset(rs)	0x2b	rs	rt	offset		
lui rt,imm	0x0f	0	rt	imm		
R 型算术/逻辑指令						
add rd,rs,rt	0	rs	rt	rd	0	0x20
addu rd,rs,rt	0	rs	rt	rd	0	0x21
sub rd,rs,rt	0	rs	rt	rd	0	0x22
subu rd,rs,rt	0	rs	rt	rd	0	0x23
and rd,rs,rt	0	rs	rt	rd	0	0x24
or rd,rs,rt	0	rs	rt	rd	0	0x25
xor rd,rs,rt	0	rs	rt	rd	0	0x26
nor rd,rs,rt	0	rs	rt	rd	0	0x27
slt rd,rs,rt	0	rs	rt	rd	0	0x2a
sltu rd,rs,rt	0	rs	rt	rd	0	0x2b
sll rd,rt,shamt	0	0	rt	rd	shamt	0
srl rd,rt,shamt	0	0	rt	rd	shamt	0x02
sra rd,rt,shamt	0	0	rt	rd	shamt	0x03
I 型算术/逻辑指令						
addi	0x08	rs	rt	imm		
addiu	0x09	rs	rt	imm		
slti	0x0a	rs	rt	imm		
sltiu	0x0b	rs	rt	imm		
andi	0x0c	rs	rt	imm		
ori	0x0d	rs	rt	imm		
分支指令						
beq	0x04	rs	rt	offset		
bne	0x05	rs	rt	offset		
blez	0x06	rs	0	offset		
bgtz	0x07	rs	0	offset		
bltz	0x01	rs	0	offset		
跳转指令						
j	0x02	target				
jal	0x03	target				
jr	0	rs	0			0x08
jalr	0	rs	0	rd	0	0x09

注：

空指令：nop (0x00000000, 即 sll \$0 \$0 0)

自己补充的指令包括：slti, bne, ori

# 控制信号

Control:

Instruction	PCSrc[1:0]	RegWrite	RegDst[1:0]	MemRead	MemWrite	MemtoReg[1:0]	ALUSrc1	ALUSrc2	ExtOp	LuiOp	BranchOp[2:0]
lw rt,offset(rs)	0	1	1	1	0	1	0	1	1	0	x
sw rt,offset(rs)	0	0	x	0	1	x	0	1	1	0	x
lui rt,imm	0	1	1	0	0	0	0	1	x	1	x
add rd,rs,rt	0	1	0	0	0	0	0	0	x	x	x
addu rd,rs,rt	0	1	0	0	0	0	0	0	x	x	x
sub rd,rs,rt	0	1	0	0	0	0	0	0	x	x	x
subu rd,rs,rt	0	1	0	0	0	0	0	0	x	x	x
and rd,rs,rt	0	1	0	0	0	0	0	0	x	x	x
or rd,rs,rt	0	1	0	0	0	0	0	0	x	x	x
xor rd,rs,rt	0	1	0	0	0	0	0	0	x	x	x
nor rd,rs,rt	0	1	0	0	0	0	0	0	x	x	x
slt rd,rs,rt	0	1	0	0	0	0	0	0	x	x	x
sltu rd,rs,rt	0	1	0	0	0	0	0	0	x	x	x
sll rd,rt,shamt	0	1	0	0	0	0	1	0	x	x	x
srl rd,rt,shamt	0	1	0	0	0	0	1	0	x	x	x
sra rd,rt,shamt	0	1	0	0	0	0	1	0	x	x	x
addi	0	1	1	0	0	0	0	1	1	0	x
addiu	0	1	1	0	0	0	0	1	1	0	x
slti	0	1	1	0	0	0	0	1	1	0	x
sltiu	0	1	1	0	0	0	0	1	1	0	x
andi	0	1	1	0	0	0	0	1	0	0	x
ori	0	1	1	0	0	0	0	1	0	0	x
beq	1	0	x	0	0	x	0	0	1	0	0
bne	1	0	x	0	0	x	0	0	1	0	1
blez	1	0	x	0	0	x	0	0	1	0	2
bgtz	1	0	x	0	0	x	0	0	1	0	3
bltz	1	0	x	0	0	x	0	0	1	0	4
j	2	0	x	0	0	x	x	x	x	x	x
jal	2	1	2	0	0	2	x	x	x	x	x
jr	3	0	x	0	0	x	x	x	x	x	x
jalr	3	1	0	0	0	2	x	x	x	x	x

说明:

PCSrc: PC Source, 取指令过程控制。00-顺序执行,  $PC = PC + 4$ ; 01-表示其为分支指令; 10-对于 j 和 jal, 会跳到 target; 11-对于 jr 和 jalr, 会跳到 \$rs。

RegWrite: 是否写入寄存器。jal 和 jalr 需要, 因为要将某个寄存器写入当前 PC 位置。

RegDst: 写入的寄存器。00-对于 R 型指令和 jalr, 需写入 Rd; 01-对于 I 型指令, 要写入 Rt; 10-对于 jal 需写入 \$31。若 RegWrite 为 0, 则 RegDst 为 x。

MemRead: 是否读内存。只有 lw 需要。

MemWrite: 是否写入内存。只有 sw 需要。

MemtoReg[1:0]: 判断写回寄存器的来源。00-将 ALU 计算的结果写回寄存器; 01-将内存的结果写回寄存器, 只有 MemRead 为 1 时, MemtoReg 才有可能为 01; 10-将 PC+4 写入寄存器。若 RegWrite 为 0, 则 MemtoReg 为 x。(注: 为什么 lui 也是 alu 的结果呢? 因为 lui 是先进行了扩展后与 \$0 相加)

ALUSrc1: ALU 第一个操作数的来源。0-rs 寄存器; 1-shamt, 因此只有 sll、srl 和 sra 的 ALUSrc1 是 1。对于跳转指令, 由于无需经过 ALU, 因此为 x。

ALUSrc2: ALU 第二个操作数的来源。0-rt 寄存器; 1-立即数。

ExtOp: 立即数扩展方式。0-0 扩展, 只有 andi 是 0 扩展; 1-符号扩展。对于非 I 型指令, 为 x。Lui 由于不需要扩展后的高位, 因此也是 x。

LuiOp: 判断是否是 lui 指令。对于非立即数的指令, LuiOp 可以为 x。

BranchOp: 判断分支指令操作。若 PCSrc 不为 1, 则为 x。

### ALUControl:

Instruction	ALUOp	Sign
lw rt,offset(rs)	ADD	1
sw rt,offset(rs)	ADD	1
lui rt,imm	ADD	x
add rd,rs,rt	ADD	1
addu rd,rs,rt	ADD	0
sub rd,rs,rt	SUB	1
subu rd,rs,rt	SUB	0
and rd,rs,rt	AND	x
or rd,rs,rt	OR	x
xor rd,rs,rt	XOR	x
nor rd,rs,rt	NOR	x
slt rd,rs,rt	LT	1
sltu rd,rs,rt	LT	0
sll rd,rt,shamt	SL	0
srl rd,rt,shamt	SR	0
sra rd,rt,shamt	SR	1
addi	ADD	1
addiu	ADD	0
slti	LT	1
sltiu	LT	0
andi	AND	x
ori	OR	x
beq	SUB	x
bne	SUB	x
blez	LE	1
bgtz	GT	1
bltz	LT	1
j	x	x
jal	x	x
jr	x	x
jalr	x	x

控制信号的最后使用阶段:

IF: 生成

ID: ExtOp, LuiOp

EX: PCSrc, RegDst, ALUSrc1, ALUSrc2, BranchOp, ALUOp, Sign

MEM: MemRead, MemWrite,

WB: RegWrite, MemtoReg

# 冒险处理

## 结构冒险

现有的电路图已经解决了结构冒险，包括增加硬件资源、调整步骤执行周期等。

## 数据冒险

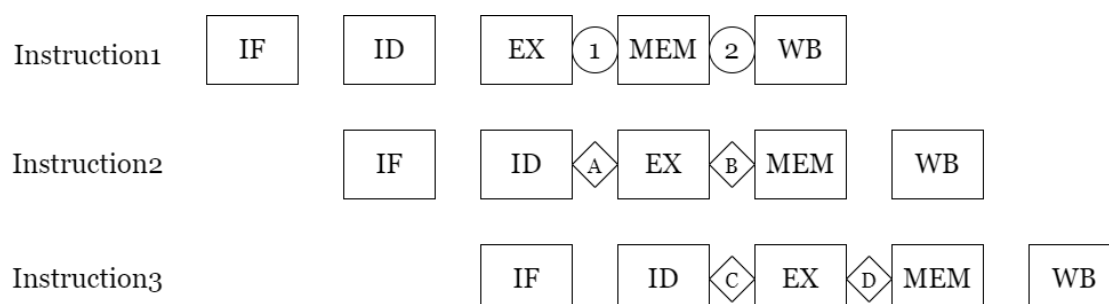
### 1) 寄存器堆的访问

由于有可能对寄存器堆进行读和写的操作，因此在硬件上希望实现先写后读。对于 RegisterFile，由于其没有读使能，因此输出数据随时更新，满足了先写后读的需求。

### 2) Read after write data hazards

下面对于寄存器数据的前后关联进行讨论，包括计算指令与 load 指令。

假设 Instruction1 产生了后续可能需要的数据，如下图所示：



可能的数据来源：1—计算指令；2—lw 指令

可能需要数据处：A/C—计算指令；B/D—sw 指令

若再往后的指令需要用到 Instruction1 产生的数据，则最早也会在其 WB 阶段结束后需要，因此不存在数据冒险。

接下来对这 2\*4 种情况进行讨论：

	1	2
A	EX/MEM->ID/EX	load-use hazard
B	与 2-B 相同	MEM/WB->EX/MEM
C	与 2-C 相同	MEM/WB->ID/EX
D	不需要转发	不需要转发

通过上表分析可知：通过转发可以解决五种数据冒险。其中 1-B 的转发可以通过 2-B 实现，1-C 的转发可以通过 2-C 实现。下面对这三种情况进行分析（在实际过程中，Forwarding Unit 生成的是控制信号）：

注：RegWriteAddr 是经过 RegDst 选择后的信号

1-A:

判断条件:

EX/MEM.RegWrite (指令 1 需要写入寄存器)

EX/MEM.RegWriteAddr (指令 1 不写入\$0)

EX/MEM.RegWriteAddr == ID/EX.rs (需要进行转发)

转发结果:

ALUSrc1 = EX/MEM.out

判断条件:

EX/MEM.RegWrite (指令 1 需要写入寄存器)

EX/MEM.RegWriteAddr (指令 1 不写入\$0)

EX/MEM.RegWriteAddr == ID/EX.rt (需要进行转发)

转发结果:

ALUSrc2 = EX/MEM.out

2-B:

判断条件:

MEM/WB.RegWrite (指令 1 需要写入寄存器)

MEM/WB.RegWriteAddr (指令 1 不写入\$0)

MEM/WB.RegWriteAddr == EX/MEM.rt (需要进行转发)

转发结果:

MemWriteData = RegWriteData

2-C:

判断条件:

MEM/WB.RegWrite (指令 1 需要写入寄存器)

MEM/WB.RegWriteAddr (指令 1 不写入\$0)

MEM/WB.RegWriteAddr == ID/EX.rs (需要进行转发)

EX/MEM.RegWriteAddr != ID/EX.rs || !EX/MEM.RegWrite (保证数据最新)

转发结果:

ALUSrc1 == MEM/WB.ALUout

判断条件:

MEM/WB.RegWrite (指令 1 需要写入寄存器)

MEM/WB.RegWriteAddr (指令 1 不写入\$0)

MEM/WB.RegWriteAddr == ID/EX.rt (需要进行转发)

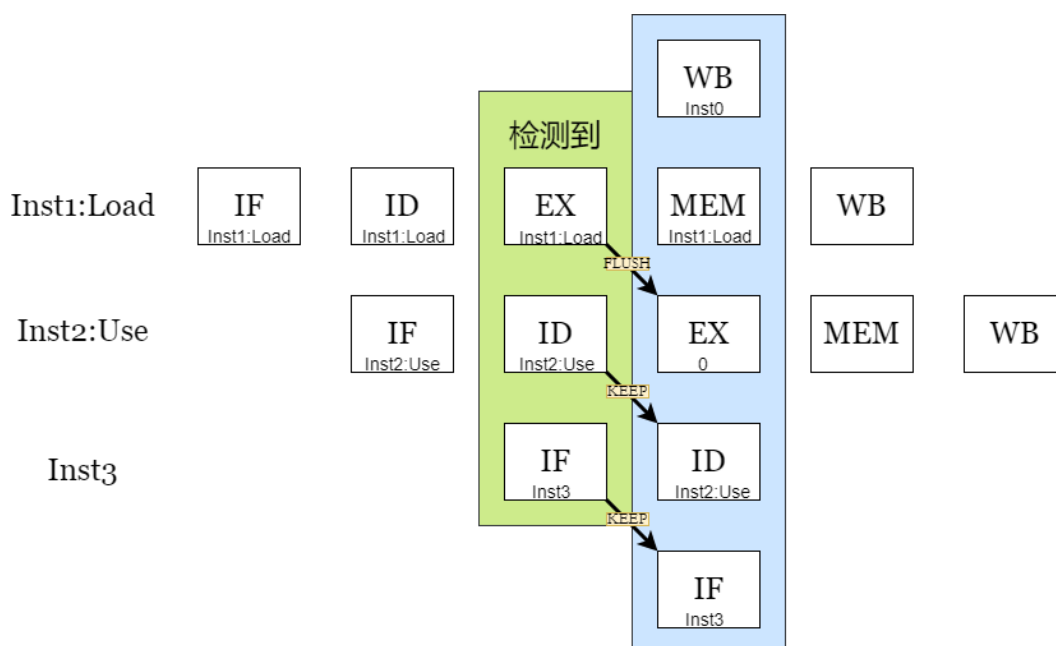
EX/MEM.RegWriteAddr != ID/EX.rt || !EX/MEM.RegWrite (保证数据最新)

转发结果:

ALUSrc2 == MEM/WB.ALUout

### 3) load-use hazard

当上一条指令为 load、下一条指令需要 use 时, 需要通过 Hazard 单元进行 stall 控制。如下图所示:



其中图中纵向的方框表示同一时间执行的周期。

若 Inst1 为 Load, Inst2 为 Use, 则当 Inst2 译码后 (绿色方块), 可以开始判断是否存在 load-use 冒险。

若检测到 load-use 冒险, 则对于下一阶段 (蓝色方块) 的 MEM 可以继续执行 Load 指令, EX 则需要进行 flush, ID、IF 则需要进行 keep, 这样可以实现一个 stall。后续可以继续进行检测和 forwarding。

EX 的 FLUSH 实现: 在 ID/EX 寄存器前放置多路选择器, 选择正常的上一路信号还是 0, 控制信号由 Hazard Unit 生成。

ID 的 KEEP 实现: 在 IF/ID 寄存器前放置多路选择器, 选择正常的上一路信号还是自己本身, 控制信号由 Hazard Unit 生成。

IF 的 KEEP 实现: 在 PC 寄存器前放置多路选择器, 选择正常的上一路信号还是自己本身的值, 控制信号由 Hazard Unit 生成。

Hazard Unit 的实现:

判断条件:

ID/EX.MemRead (判断为 Load 指令)

ID/EX.rt==rs || ID/EX.rt == rt (考察下一指令是否需要 use)

实现结果:

KEEP PC

KEEP IF/ID

FLUSH ID/EX

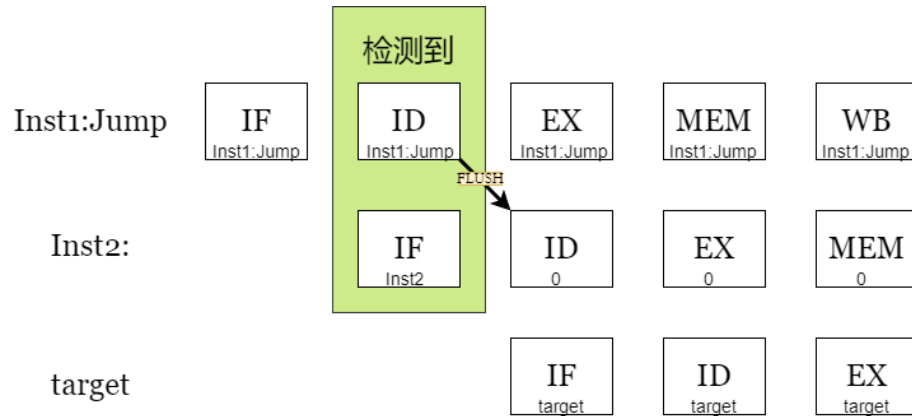


## 控制冒险

### 1) 跳转指令

跳转指令包括 j、jal、jr、jalr。

当出现跳转指令时，过程如下：



其中图中纵向的方框表示同一时间执行的周期。

若第一条指令为跳转指令，则在其 ID 阶段（绿色方框）可以被检测到，且 PCSrc 将对 PC 的值作出指示。同时应将 ID 进行 flush, flush 方法同 load-use hazard。

判断条件：

当出现跳转指令时，也即  $PCSrc[1] == 1$ 。因为在我设计的控制信号中，PCSrc 承担了标记跳转指令和分支指令的任务。

j	2
jal	2
jr	3
jalr	3

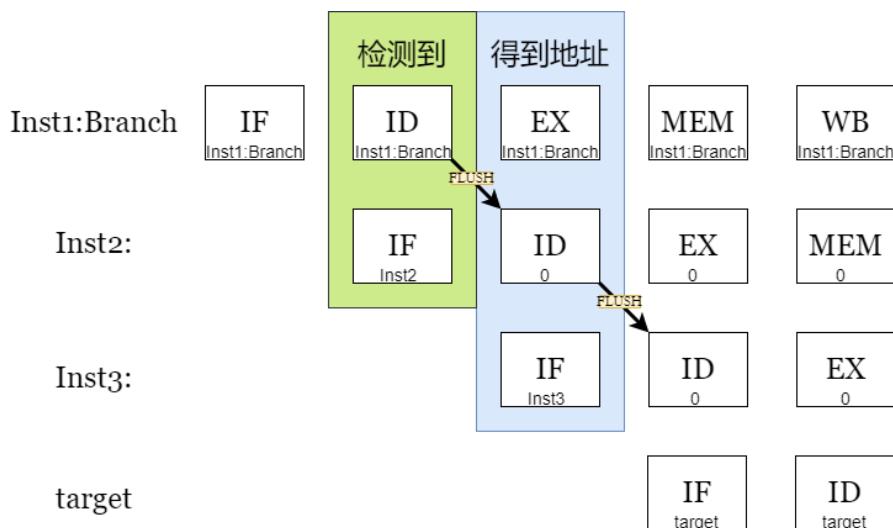
实现结果：

FLUSH IF/ID

## 2) 分支指令

跳转指令包括 beq、bne、blez、bgtz、bltz。

当出现跳转指令时，过程如下：



其中图中纵向的方框表示同一时间执行的周期。

若第一条指令为分支指令，则在其 ID 阶段（绿色方框）可以被检测到，在其 EX 阶段（蓝色方框）可以判断出跳转的地址。那么将有以下两件工作需要做：

①在其 ID、EX 阶段对 IF/ID 进行 flush

若 PCSrc 为 01，则说明现处于绿色方框阶段，需要对 ID 进行 flush。结合上面对于跳转指令的分析，可知：若 Inst1 为分支或者跳转指令，也即 PCSrc!=0 时需要 flush。

若 ID/EX.PCSrc 为 01，则说明现处于蓝色方框阶段，依然需要对 ID 进行 flush。

综上所述，IF/ID 的 flush 条件为：PCSrc || ID/EX.PCSrc == 01

②在 EX 阶段给出下一条指令的地址，并传递给 PC

由于其处理相对复杂，将其封装为 Branch Unit 来进行处理。大致如下：

输入：ID/EX.BranchOp, ALUout, ID/EX.PCplus4, BranchAddr

输出：BranchTarget

Branch Unit 可以通过 BranchOp 和 ALUout 来决定下一条指令，即 BranchTarget。

## JumpRegister 冒险

以上是根据课上所学而得出的结果。然而可能有更多种类的冒险需要进一步斟酌，例如 jr 和 jalr。二者需要在 ID 阶段拿到 rs 的值，但很有可能这时候 rs 的值还没有得到。即：若 jr/jalr 的上上条指令给出 rs 的值，则需要 EX/MEM->IF/ID 的转发；若 jr/jalr 的上条指令给出 rs 的值，则需要 ID/EX->IF/ID 的转发。

判断条件：（优先级更高）

PCSrc == 3

ID/EX.RegWrite == 1

ID/EX.RegWriteAddr != 0

ID/EX.RegWriteAddr == rs

转发结果：

JumpRegisterTarget = ALUout

判断条件：

PCSrc == 3

EX/MEM.RegWrite == 1

EX/MEM.RegWriteAddr != 0

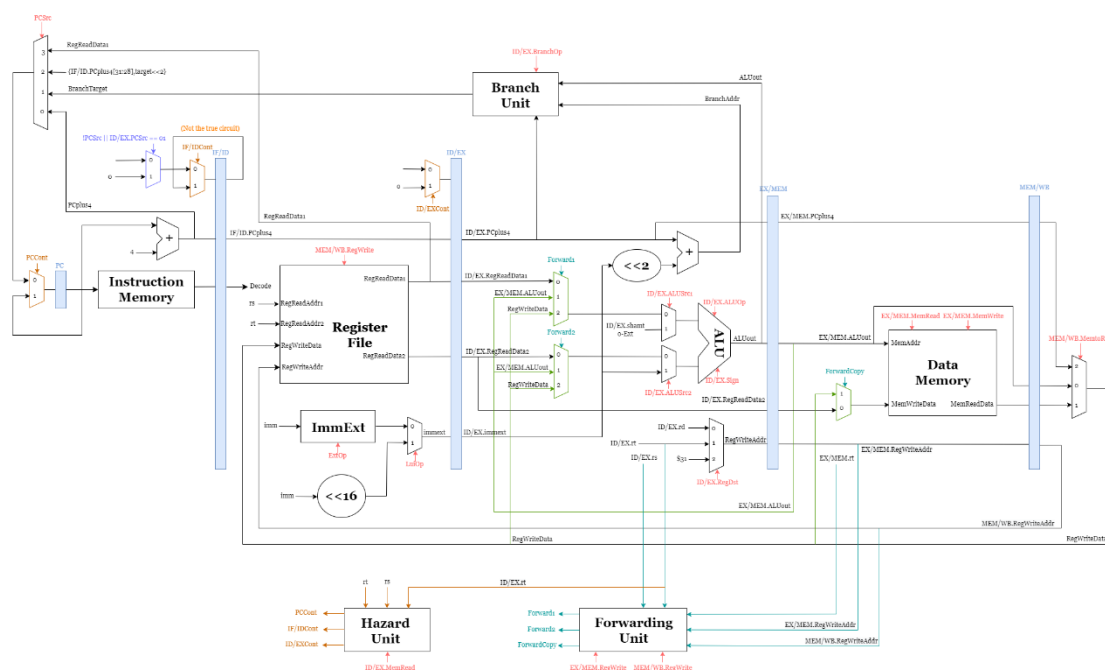
EX/MEM.RegWriteAddr == rs

转发结果：

JumpRegisterTarget = EX/MEM.ALUout

# 数据通路

综合上述分析，按照本人理解，可以得到（呕心沥血画出的）数据通路图如下：



余以为大部分地方都是比较严谨而合理的，唯独 PC 的选择之处难以用电路图完整描述，但用 verilog 代码会相对容易一些。故做如下说明：

- PC 默认为 PC+4。特殊情况则特殊判断（e.g. Jump，必须在下一个周期才能判断出其为跳转指令）。而在上图中，PC+4 却要经过 ID 阶段才出现的 PCSrc 来进行选择，对于顺序执行的指令是不合理的。
- IF/ID 寄存器中表示 KEEP 的回环型信号表示其包含的全部信号。
- 如有需要多级传递的信号，例如 rs，则在其第一次出现的阶段，如 ID，则以其本名出现；传递后，其名称为上一级间寄存器名称.本名，如 ID/EX.rs。

寄存器寄存清单

寄存器	IF/ID	ID/EX	EX/MEM	MEM/WB
控制信号	RegWrite MemtoReg MemRead MemWrite PCSrc RegDst ALUSrc1 ALUSrc2 BranchOp ALUOp Sign ExtOp LuiOp	RegWrite MemtoReg MemRead MemWrite PCSrc RegDst ALUSrc1 ALUSrc2 BranchOp ALUOp Sign	RegWrite MemtoReg MemRead MemWrite	RegWrite MemtoReg
指令信息	Instruction	rt rs rd shamt	rt	
中间数据	PCplus4	PCplus4 RegReadData1 RegReadData2 immext	PCplus4 RegReadData2 ALUout RegWriteAddr	PCplus4 ALUout RegWriteAddr MemReadData

注：IF/ID 不存储控制信号，存 Instruction

# CPU 代码编写

## 模块设计

根据上述前期准备工作，做以下模块设计：

Control: 生成控制信号

ALUControl: 生成 ALU 的控制信号

ALU: 算术逻辑单元

DataMemory: 内存

InstructionMemory: 指令内存

RegisterFile: 寄存器堆

ForwardingUnit: 转发单元

HazardUnit: load-save 冒险单元

BranchUnit: 处理分支指令单元

PC: PC 寄存器及其选择

IF\_ID: IF/ID 寄存器

ID\_EX: ID/EX 寄存器

EX\_MEM: EX/MEM 寄存器

MEM\_WB: MEM/WB 寄存器

下面对关键模块（个性化色彩较重）进行说明。

## ForwardingUnit: 转发单元

ForwardingUnit, 即转发单元, 用于判断出需要转发的情况, 并生成相应的控制信号。可以解决数据冒险中需要转发的部分。

输入:

```
input wire EX/MEM.RegWrite
input wire MEM/WB.RegWriteAddr
input wire [4:0] ID/EX.rs
input wire [4:0] ID/EX.rt
input wire [4:0] EX/MEM.rt
input wire [4:0] EX/MEM.RegWriteAddr
input wire [4:0] MEM/WB.RegWriteAddr
```

输出:

```
output reg [1:0] Forward1
output reg [1:0] Forward2
output reg ForwardCopy
```

其中三个输出信号分别负责控制 ALU 的两个输入数据和写入内存的数据。

在进行转发判断时, 使用 if-else if-else 来判断, 因而可以首先判断 EX/MEM 到 ID/EX 的转发, 保证了其优先级, 使得在都需要转发的情况下, 使数据保证最新。判断过程如下:

```
if(EX/MEM.RegWrite && EX/MEM.RegWriteAddr == ID/EX.rs) Forward1 <= 2'd1;
else if(MEM/WB.RegWrite && MEM/WB.RegWriteAddr == ID/EX.rs) Forward1 <= 2'd2;
else Forward1 <= 2'd0;

if(EX/MEM.RegWrite && EX/MEM.RegWriteAddr == ID/EX.rt) Forward2 <= 2'd1;
else if (MEM/WB.RegWrite && MEM/WB.RegWriteAddr == ID/EX.rt) Forward2 <= 2'd2;
else Forward2 <= 2'd0;

if(MEM/WB.RegWrite && MEM/WB.RegWriteAddr == EX/MEM.rt) ForwardCopy <= 1'd1;
else ForwardCopy <= 1'd0;
```

## HazardUnit: load-use 冒险单元

HazardUnit, 即 load-use 冒险单元(实际上应该负责控制所有的 FLUSH, KEEP 等等, 但其它的另有处理)。它可以在 load-use 冒险出现时, 生成相关的控制信号, 使相应寄存器发生 FLUSH 或 KEEP, 以达到 stall 的效果。

输入:

```
input wire ID/EX.MemRead
input wire [4:0] rt
input wire [4:0] rs
input wire [4:0] ID/EX.rt
```

输出:

```
output reg PCCont
output reg IF/IDCont
output reg ID/EXCont
```

其中三个输出分别控制 PC、IF/ID、ID/EX 的情况。它们不能完全判断, 需和 PCSrc 等信号共同决定其行为。

判断过程:

```
if(ID/EX.MemRead && (ID/EX.rt==rs || ID/EX.rt == rt)) begin
    PCCont <= 1'b1;
    IF/IDCont <= 1'b1;
    ID/EXCont <= 1'b1;
end
else begin
    PCCont <= 1'b0;
    IF/IDCont <= 1'b0;
    ID/EXCont <= 1'b0;
end
```



## BranchUnit: 分支指令处理单元

BranchUnit, 即分支指令处理单元, 可以在 branch 指令进行到 EX 阶段的时候给出下一指令的跳转地址, 当然有可能为 PC+4, 有可能为跳转的地址。

输入:

input wire [2:0] ID/EX.BranchOp

input wire [31:0] ALUout

input wire [31:0] BranchAddr

input wire [31:0] ID/EX.PCplus4

输出:

output reg [31:0] BranchTarget

其中 ID/EX.BranchOp 表示该分支指令是哪一条, 配合 ALUout 可以判断是否要进行分支跳转, 在 BranchAddr 和 ID/EX.PCplus4 中选择出正确的一项, 即 BranchTarget 并传递给 PC。

判断过程:

```
case(ID/EX.BranchOp)
  3'd0: begin
    // beq
    if(!ALUout) BranchTarget <= BranchAddr;
    else BranchTarget <= ID/EX.PCplus4;
  end
  3'd1: begin
    // bne
    if(ALUout) BranchTarget <= BranchAddr;
    else BranchTarget <= ID/EX.PCplus4;
  end
  3'd2: begin
    // blez
    if(ALUout) BranchTarget <= BranchAddr;
    else BranchTarget <= ID/EX.PCplus4;
  end
  3'd3: begin
    // bgtz
    if(ALUout) BranchTarget <= BranchAddr;
    else BranchTarget <= ID/EX.PCplus4;
  end
  3'd4: begin
    // bltz
    if(ALUout) BranchTarget <= BranchAddr;
    else BranchTarget <= ID/EX.PCplus4;
  end
  default: begin
    BranchTarget <= ID/EX.PCplus4;
  end
endcase
```

ALU 操作为 SUB, 其余指令都进行了相应的 ALU 操作分配, 使其为 1 时跳转。

## PC: PC 寄存器及其选择

在我设计的 PC 模块中, 不仅包括了寄存器功能, 还包括了下一个指令的选择。思路为:

- 默认  $PC \leq PC + 4$
- 若  $IF/ID.PCSrc == 2'b10$  或  $2'b11$ , 说明此时跳转指令执行到 ID 阶段, 下一周期将跳转到 target
- 若  $ID/EX.PCSrc == 2'b01$ , 说明此时 branch 指令执行到 EX 阶段, 下一周期将跳转到 BranchTarget
- 若  $PCcont == 1$ , 说明 PC 需要保持, 也即不用做任何赋值操作。

判断过程:

```
always @(posedge clk or posedge reset) begin
    if (reset) begin
        // reset
        pc <= 32'h00400000;
    end
    else if(!PCcont) begin
        // doesn't need to keep
        if(IF/ID.PCSrc == 2'b10) begin
            // j or jal
            pc <= {IF/ID.PCplus4[31:28], target << 2};
        end
        else if(IF/ID.PCSrc == 2'b11) begin
            // jr or jalr
            pc <= RegReadData1;
        end
        else if(ID/EX.PCSrc == 2'b01) begin
            // branch
            pc <= BranchTarget;
        end
        else begin
            pc <= pc + 4;
        end
    end
    else begin
        // keep
    end
end
```

其它级间寄存器大致同理, 不再赘述。

## 连线（顶层设计）

由于时序逻辑（寄存器）电路和复杂的组合逻辑部件已经封装为模块，因此只需将它们之间的线连好即可。不再赘述。

# CPU 代码调试

## 指令测试

注：此阶段测试仅测试单条指令运行情况，不考虑冒险。因此每个指令间有 4 个 nop。

**test1: addi, addiu, slti, sltiu, andi, ori**

测试指令：

addi, addiu, slti, sltiu, andi, ori

汇编指令：

addi \$a0 \$0 1

addiu \$a0 \$a0 1

slti \$a1 \$a0 3

sltiu \$a2 \$a0 -1

andi \$t0 \$a0 255

ori \$t1 \$a0 255

机器码：

0x20040001

0x24840001

0x28850003

0x2c86ffff

0x308800ff

0x348900ff

预期结果：

\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000002
\$a1	5	0x00000001
\$a2	6	0x00000001
\$a3	7	0x00000000
\$t0	8	0x00000002
\$t1	9	0x000000ff

实际结果：

> [4][31:0]	00000002
> [5][31:0]	0000000c
> [6][31:0]	ffffffc
> [7][31:0]	00000000
> [8][31:0]	000003fc
> [9][31:0]	00000000

一塌糊涂。。。

Debug:

5 号寄存器应为 1，实际为 c，说明 `slti $a1 $a0 3` 指令出现问题。

检查发现，虽然 ALU 的操作数一个是 2 一个是 3，但输出直接给了 c。检查 ALUCtrl 信号，当进行到 EX 阶段时，传给 ALU 的 ALUCtrl 是 6 (SL，即 nop 的结果) 而不是 8 (Shift Left)。发现是由于给 ALU 接线时，接的是当时的 ALUOp，而不是经过 ID/EX 寄存器之后的 ALUOp。因此前两个指令对了，也仅是一个 happy coincidence。

```
// ALU
wire [31:0] ALUout;
ALU MyALU(ALUOp, Sign, Operand1, Operand2,
```

debug 后的结果:

> [4][31:0]	00000002
> [5][31:0]	00000001
> [6][31:0]	00000001
> [7][31:0]	00000000
> [8][31:0]	00000002
> [9][31:0]	000000ff

非常 nice。

## test2: add, addu, sub, subu, and, or, xor, nor

测试指令: add, addu, sub, subu, and, or, xor, nor

汇编指令:

```
addi $1 $0 2022
addiu $2 $0 2059
addi $3 $1 -3599
add $4 $1 $2
addu $5 $2 $3
sub $6 $3 $4
subu $7 $4 $5
and $8 $5 $6
or $9 $6 $7
xor $10 $7 $8
nor $11 $8 $9
```

机器码:












```
0x200107e6
0x2402080b
0x2023f1f1
0x00222020
0x00432821
0x00643022
0x00853823
0x00a64024
0x00c74825
0x00e85026
0x01095827
```

预期结果:

1	0x000007e6
2	0x0000080b
3	0xfffff9d7
4	0x00000ff1
5	0x000001e2
6	0xfffffe9e6
7	0x00000e0f
8	0x000001e2
9	0xfffffefef
10	0x00000fed
11	0x00001010

完全一致, 不错。

实际结果:

>  [1][31:0]	000007e6
>  [2][31:0]	0000080b
>  [3][31:0]	fffff9d7
>  [4][31:0]	00000ff1
>  [5][31:0]	000001e2
>  [6][31:0]	ffffe9e6
>  [7][31:0]	00000e0f
>  [8][31:0]	000001e2
>  [9][31:0]	fffffefef
>  [10][31:0]	00000fed
>  [11][31:0]	00001010

### test3: slt, sltu

在 test2 的代码后加上两行:

```
slt $12 $8 $9
```

```
sltu $13 $8 $9
```

机器码:



```
0x0109602a
```

```
0x0109682b
```

预期结果:

12	0x00000000
13	0x00000001

实际结果:

>  [12][31:0]	00000000
>  [13][31:0]	00000001

一致。

(实际上测试的时候出过问题,发现只是把机器码 copy 错了)

**test4: lui, sll, srl, sra**

测试代码:

addi \$1 \$0 1234567890(伪代码)

sll \$2 \$1 4

srl \$3 \$2 4

sra \$4 \$2 6

机器码:

0x3c014996

0x342102d2

0x00010820

0x00011100





0x00021902

0x00022183

预期结果:

\$at	1	0x499602d2
\$v0	2	0x99602d20
\$v1	3	0x099602d2
\$a0	4	0xfe6580b4

实际结果:

>  [1][31:0]	499602d2
>  [2][31:0]	99602d20
>  [3][31:0]	099602d2
>  [4][31:0]	fe6580b4

结果一致。



test5: lw, sw

测试代码:

```
addi $2 $0 5
sw $2 12($0)
lw $3 12($0)
```

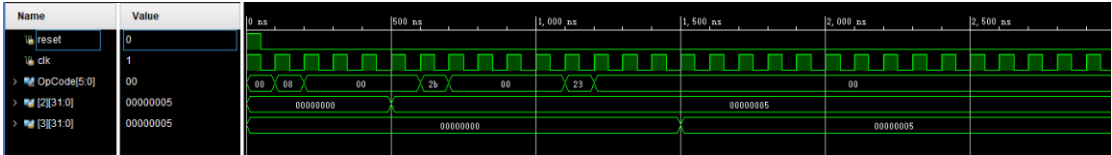
机器码:

```
0x20020005
0xac02000c
0x8c03000c
```

预期结果:

```
$2 5
$3 5
```

测试结果:



与预期一致。

test6: j, jal, jr, jalr

```
测试代码:
addi $1 $0 1
jal test1
addi $3 $0 0x0040001c
jalr $3
j end
test1:
add $2 $1 $1
jr $ra
test2:
add $4 $2 $2
jr $ra
end:
addi $5 $0 5
```

机器码:

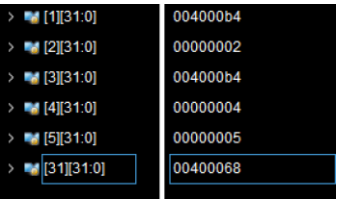
20010001  
0c100007  
3c010040  
34210024  
00011820  
0060f809  
0810000b  
00211020  
03e00008  
00422020  
03e00008  
20050005

预期结果:

\$a0	1	0x004000b4
\$v0	2	0x00000002
\$v1	3	0x004000b4
\$a0	4	0x00000004
\$a1	5	0x00000005

同时\$31 为 0x00400068

测试结果:



与预期结果一致。

## test7: beq

测试代码:

```
addi $2 $0 78
```

```
addi $3 $0 79
```

```
addi $4 $0 79
```

```
beq $2 $3 wrong
```

```
beq $3 $4 right
```

wrong:

```
addi $5 $0 -1
```

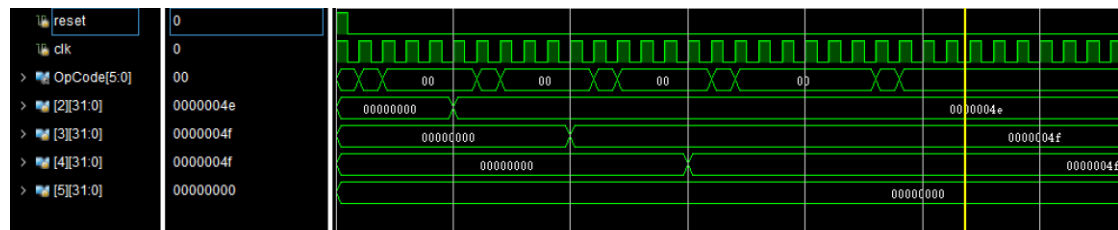
right:

```
addi $5 $0 1
```

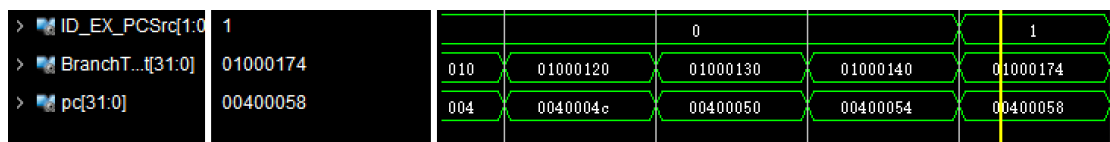
预期结果:

2	0x0000004e
3	0x0000004f
4	0x0000004f
5	0x00000001

然而仿真时:



可以看到,在执行第一个 beq 时,如期进行两个 stall,并继续进行第二个 beq。但是第二个 beq 并没有如期进行跳转。

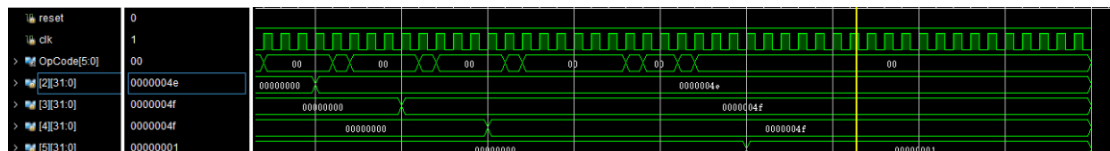


可以发现, BranchTarget 并没有给出正确的地址,而其 BranchAddr 已经有误。

```
wire [31:0] BranchAddr;
```

```
assign BranchAddr = ID_EX_PCplus4 + ID_EX_immext << 2;
```

这里是 BranchAddr 的赋值,发现 BranchAddr 之所以离谱是因为+的优先级高于<<,所以它先相加再左移,拴 Q 了



调整后,可以得到预期结果。

### test8: bne

测试代码:

```
addi $2 $0 78
addi $3 $0 79
addi $4 $0 79
bne $3 $4 wrong
bne $2 $3 right
wrong:
addi $5 $0 -1
right:
addi $5 $0 1
```

预期结果:

2	0x0000004e
3	0x0000004f
4	0x0000004f
5	0x00000001

与预期一致。

测试结果:

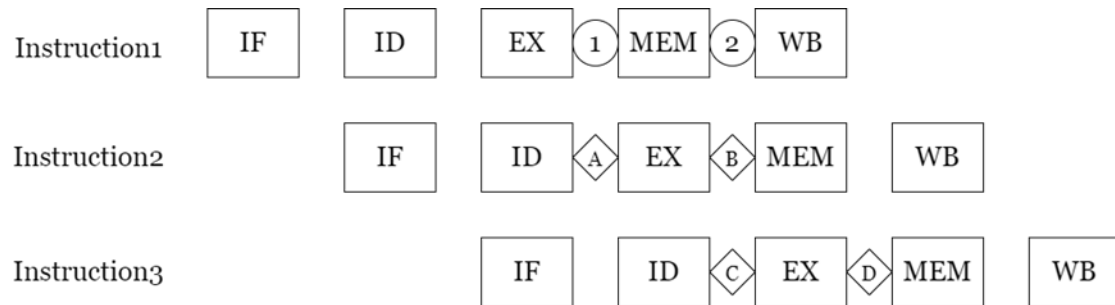
> [2][31:0]	0000004e
> [3][31:0]	0000004f
> [4][31:0]	0000004f
> [5][31:0]	00000001

### test9: blez、bgtz、bltz

对 blez、bgtz、bltz 进行类似的测试, 均获得正确的跳转结果。由于代码类似, 不再赘述。

综上, 对每一条指令均进行了单独的验证, 可以说明其可以进行正常运作 (如果愿意每个指令都接四个 nop。接下来进行冒险测试。

## 冒险测试



test1: 1-A & 2-C

测试代码:

```
addi $2 $0 2022
```

```
addi $3 $0 1700
```

```
add $4 $2 $3
```

预期结果:

2	0x000007e6
3	0x000006a4
4	0x00000e8a

实际结果:

[2][31:0]	000007e6
[3][31:0]	000006a4
[4][31:0]	000006a4

通过这个错误，找到两个 bug:

1. 在 top 文件里，Forward1 和 Forward2 都设计成了 1 位信号
2. 连线的时候，误把 MEM/WB.RegWrite 传成了 MEM/WB.RegWriteAddr

改正后，可得正确结果:

> [2][31:0]	000007e6
> [3][31:0]	000006a4
> [4][31:0]	00000e8a

test2: 2-B

测试代码:

```
addi $2 $0 5
```

```
sw $2 12($0)
```

```
lw $3 12($0)
```

预期结果:

\$2 5

\$3 5

测试结果:

与预期一致。

[2][31:0]	00000005
[3][31:0]	00000005

test3: load-use

测试代码：

```
addi $2 $0 0x10010000
addi $3 $0 1800
sw $3 0($2)
lw $4 0($2)
add $5 $4 $4
```

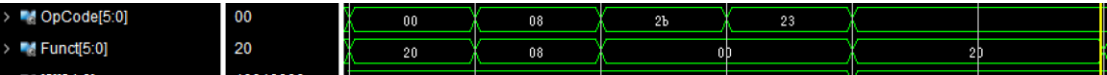
预期结果：

2	0x10010000
3	0x00000708
4	0x00000708
5	0x00000e10

测试结果：

[2][31:0]	10010000
[3][31:0]	00000708
[4][31:0]	00000708
[5][31:0]	00000e10

可见结果一致。



同时观察 Funct 信号，停留在 20 两个周期，说明由于 load-use，add 得到了 keep。

控制冒险在单独测试指令的时候可以说明设计的完整性与正确性。

# 字符串查找的实现

## 仿真实现

### 代码编写

首先写一个简单的案例在 MARS 上试运行：

```
# str: 0x10010000
# pattern: 0x10010200

# load the str and the pattern
li $a1 0x10010000 # $a1: second argument:str
li $a3 0x10010200 # $a3: fourth argument:pattern
li $t0 1
li $t1 2
sw $t0 0($a1)
sw $t1 4($a1)
sw $t0 8($a1)
sw $t1 12($a1)
sw $t0 16($a1)
sw $t1 20($a1)
sw $t0 24($a1)
sw $t0 0($a3)
sw $t1 4($a3)
sw $t0 8($a3)

# find the length of the str and the pattern
# a0 : length of the str
# a2 : length of the pattern
add $a0 $0 $0
move $t0 $a1 # $t0: pointer
For_begin_str:
lw $t1 0($t0) # $t1: element
beq $t1 $0 For_end_str
addi $a0 $a0 1
addi $t0 $t0 4
j For_begin_str
For_end_str:

add $a2 $0 $0
move $t0 $a3
For_begin_pattern:
lw $t1 0($t0) # $t1: element
beq $t1 $0 For_end_pattern
addi $a2 $a2 1
addi $t0 $t0 4
j For_begin_pattern
For_end_pattern:

sll $a0 $a0 2
sll $a2 $a2 2
```

```

jal brute_force

End:
beq $0 $0 End

brute_force:
# # # # # your code here # # # # #
li $t0 0          # $t0: i
li $t1 0          # $t1: j
li $v0 0          # $v0: cnt
bigger_for_judge:
sub $t2 $a0 $a2
blt $t2 $t0 bigger_for_exit
li $t1 0
smaller_for_judge:
slt $t2 $t1 $a2
beq $t2 $zero smaller_for_exit
add $t2 $t0 $t1    # i + j
add $t2 $t2 $a1    # address of str[i + j]
lw $t2 0($t2)
add $t3 $a3 $t1    # address of pattern[j]
lw $t3 0($t3)
bne $t2 $t3 smaller_for_exit
addi $t1 $t1 4
j smaller_for_judge
smaller_for_exit:
bne $t1 $a2 no_need_to_add_one
addi $v0 $v0 1
no_need_to_add_one:
addi $t0 $t0 4
j bigger_for_judge
bigger_for_exit:
jr $ra

```

这段代码的大致意思为：

先用 `sw` 在内存中放置好字符串和模式串。在实际完成时，可以事先在内存中初始化，以避免用代码完成过于繁琐。在这个程序中，字符串是 1212121，模式串是 121。并给出二者首地址。

之后凭借两个字符串的首地址，判断出字符串和模式串的长度。

之后调用之前数逻大作业完成的代码。但是由于在该 CPU 不支持 `lb` 运算，因此需要修改。修改的地方包括：把 `lb` 变为 `lw`，把相应的变量\*4。

最后完成的结果保存在 `$v0` 中，并循环重复一条 `beq` 指令以视结尾。

由于暴力算法、horspool 和 KMP 算法都有现成的汇编代码，因此他们之间没有本质区别，在此选用暴力算法完成任务。



## 将较复杂代码写入 vivado 的指令存储器

将指令一条条复制到指令存储器是很繁琐而浪费时间的。采取下面方法加快该进程：

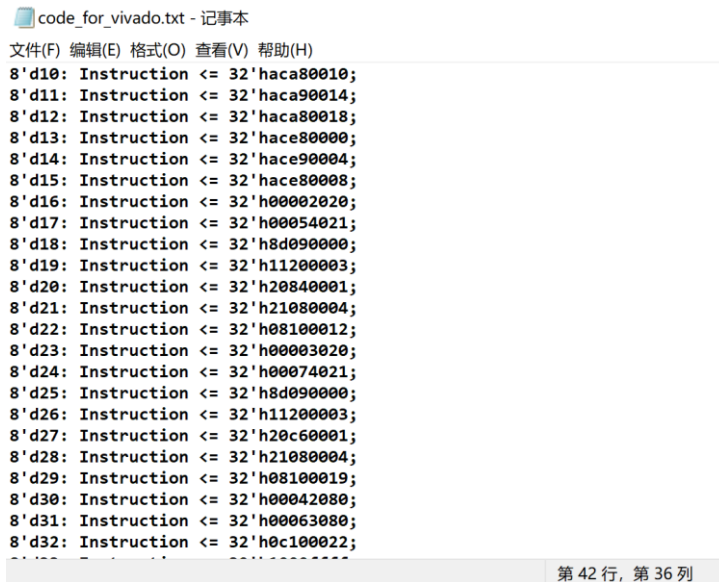
1. 将 MARS 生成的机器码以十六进制文本的形式导出。
2. 编写 python 程序，为机器码添加统一的格式。

```
file1 = open('C:\\Users\\liyutong\\Desktop\\code.txt', 'r')
asmCode = file1.read()
file1.close()

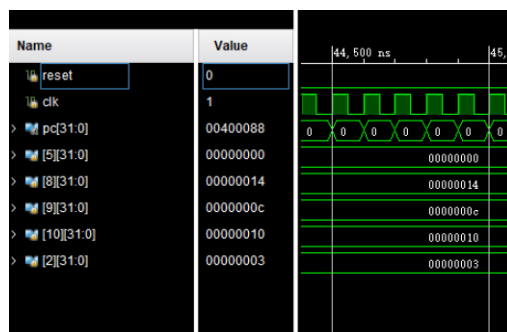
asmCode = asmCode.split("\n")

file2 = open('C:\\Users\\liyutong\\Desktop\\code_for_vivado.txt', 'w')
i = 0
for code in asmCode:
    print(file2.write("8'd10: Instruction <= 32'h" + code + ';\n'))
    i = i + 1
```

3. 复制粘贴进入指令存储器即可。



## 仿真结果



在程序运行足够长的事件后，得到了预期的结果。

## 添加外设

在完成 CPU 的基本实现之后，可以在思维上将其封装，并添加外设接口。但由于要求利用软件的方法完成外设，将 BCD 绑定于存储器上，因此需要调整内存的状况，同时在程序主体结束后添加外设部分。则修改内存如下：

```
// Read data
assign ReadData = MemRead == 1'b0 ? 32'h0 :
    Address == 32'h4000000c ? {20'h0, AN, BCD} :
    Address == 32'h40000010 ? {24'b0, leds} :
    Address == 32'h40000014 ? Counter : RAM_data[Address[9:2]];

// Write data
integer i;
always @* begin
    if(reset) begin
        AN <= 4'b0;
        BCD <= 8'b0;
        for (i = 0; i < RAM_SIZE; i = i + 1)
            RAM_data[i] <= 32'h00000000;
    end
    else if(MemWrite) begin
        if(Address == 32'h4000000c) begin
            AN <= WriteData[11:8];
            BCD <= WriteData[7:0];
        end
        else if(Address == 32'h40000010) begin
            leds <= WriteData[7:0];
        end
        else begin
            RAM_data[Address[9:2]] <= Writedata;
        end
    end
end

always @(posedge reset or posedge clk) begin
    if(reset) begin
        Counter <= 32'b0;
    end
    else begin
        Counter <= Counter + 1;
    end
end
```

注：对于内存的初始化有两种选择：一种是在程序正式开始之前，将数据 sw 放入内存；另一种是在 DataMemory 收到 reset 信号时，将数据加载进去。上述代码是第一种写法，后附的代码文件为第二种写法。

同时应在汇编程序之后添加用于显示的代码部分:

```
# Decode
# t0:
andi $a0 $v0 0x0000000f
jal Decode
move $t0 $v1
ori $t0 $t0 0x00000e00
# t1:
andi $a0 $v0 0x000000f0
srl $a0 $a0 4
jal Decode
move $t1 $v1
ori $t1 $t1 0x00000d00
# t2
andi $a0 $v0 0x00000f00
srl $a0 $a0 8
jal Decode
move $t2 $v1
ori $t2 $t2 0x00000b00
# t3
andi $a0 $v0 0x0000f000
srl $a0 $a0 12
jal Decode
move $t3 $v1
ori $t3 $t3 0x00000300

li $a0 0x10010400
Display_t0:
sw $t0 0($a0)
jal count
sw $t1 0($a0)
jal count
sw $t2 0($a0)
jal count
sw $t3 0($a0)
jal count
j Display_t0

count:
li $s0 0
li $s1 2500
add_one:
addi $s0 $s0 1
bne $s0 $s1 add_one
jr $ra

Decode:
# a0 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
beq $a0 0 zero
beq $a0 1 one
beq $a0 2 two
beq $a0 3 three
beq $a0 4 four
beq $a0 5 five
beq $a0 6 six
beq $a0 7 seven
beq $a0 8 eight
beq $a0 9 nine
beq $a0 10 aa
beq $a0 11 bb
beq $a0 12 cc
beq $a0 13 dd
beq $a0 14 ee
beq $a0 15 ff
li $v1 0xff
j ok
zero:
li $v1 0xc0
j ok
```

```

one:
li $v1 0xf9
j ok
two:
li $v1 0xa4
j ok
three:
li $v1 0xb0
j ok
four:
li $v1 0x99
j ok
five:
li $v1 0x92
j ok
six:
li $v1 0x82
j ok
seven:
li $v1 0xf8
j ok
eight:
li $v1 0x80
j ok
nine:
li $v1 0x90
j ok
aa:
li $v1 0x88
j ok
bb:
li $v1 0x83
j ok
cc:
li $v1 0xc6
j ok
dd:
li $v1 0xa1
j ok
ee:
li $v1 0x84
j ok
ff:
li $v1 0x8e
j ok
ok:
jr $ra

```

以上代码的含义为：

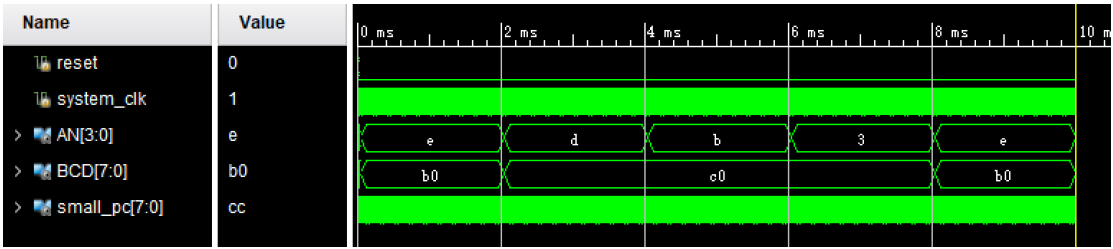
此前已将结果存在\$*v0* 中，目标将\$*v0* 的结果的十六进制低四位显示在七段数码管上。

将每一位进行相应的译码——这里采取的是一种类似于 case 的想法实现——并分别将译码后的结果加载到\$t0~\$t3 中，并利用 ori 将使能信号与其拼接起来。

每过 1ms，利用 sw 将结果写入 0x40000010 中。这里的 1ms 是利用循环计数实现的。

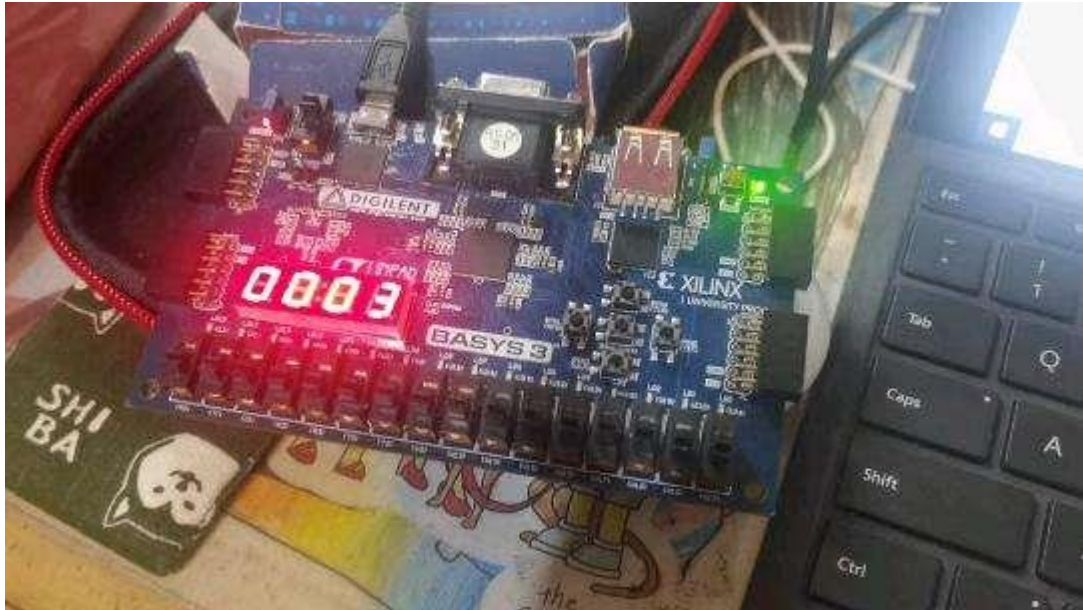
添加外设后的仿真结果

观察运行一段时间后的程序运行结果，得到预期结果：



注：可以看到，显示的数字管每 2ms 换一次数字。实际上汇编代码完成的功能是 1ms，上述结果是尝试了分频器后的结果。

## 硬件调试



程序烧录后即可显示正确结果，具体情况已在视频验收中有所汇报。

# 性能分析

## 时序性能

当设置时钟为 10ns 时，综合后的时序情况如下：

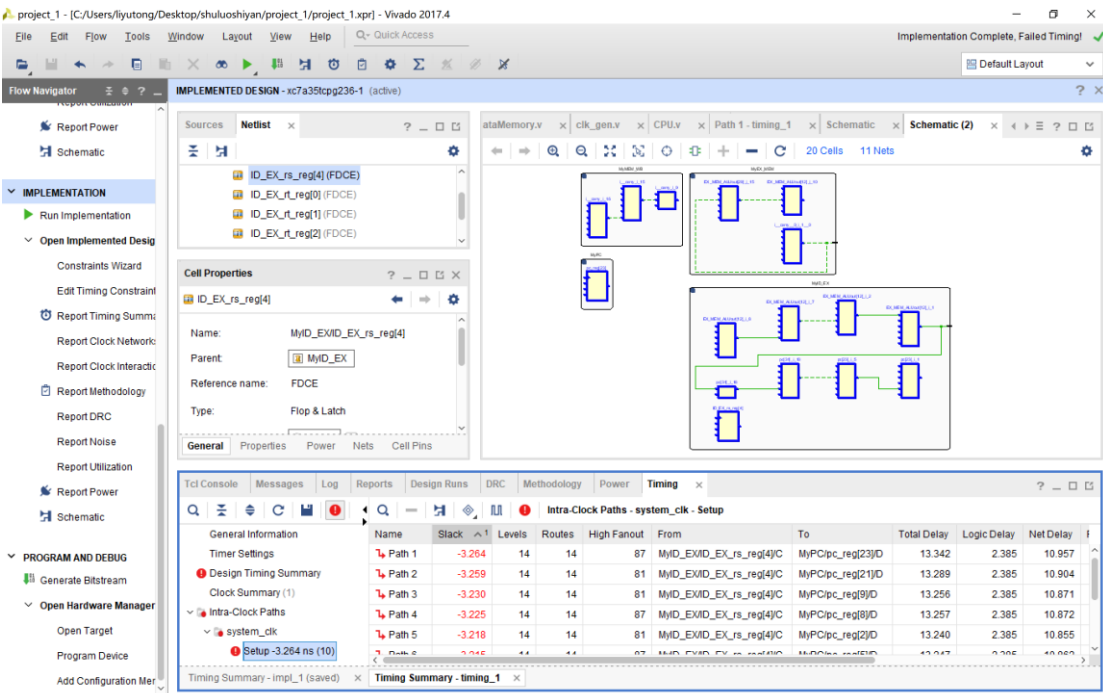
Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -4.024 ns	Worst Hold Slack (WHS): 0.148 ns	Worst Pulse Width Slack (WPWS): 4.500 ns
Total Negative Slack (TNS): -163.091 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 85	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 627	Total Number of Endpoints: 627	Total Number of Endpoints: 529

布线后的时序情况如下：

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -3.264 ns	Worst Hold Slack (WHS): 0.127 ns	Worst Pulse Width Slack (WPWS): 4.500 ns
Total Negative Slack (TNS): -152.889 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 107	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 627	Total Number of Endpoints: 627	Total Number of Endpoints: 529

则最短时钟周期约为 14.024ns，最高时钟频率约为 $\frac{1s}{14.024ns} \approx 71.3MHz$ 。

观察关键路径如下：



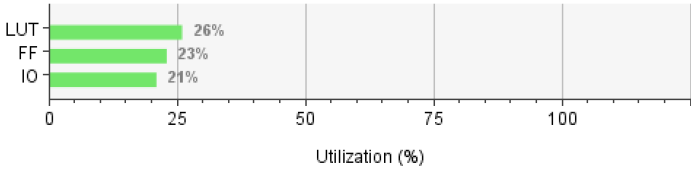
可以发现，最长路径起始是 ID/EX.rs，终点是 pc，也即对于分支指令的判断和冒险是这里的最长路径。

# 逻辑资源

该 CPU 占据的逻辑资源状况如下：

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Bonded IOB (106)	BUFGCTRL (32)
CPU	5445	9732	1345	576	4132	5445	579	22	12
MyALU (ALU)	0	0	0	0	16	0	0	0	0
MyDataMemory (DataM...	2180	8212	1088	544	3559	2180	0	0	0
MyEX_MEM (EX_MEM)	766	150	0	0	378	766	0	0	0
MyID_EX (ID_EX)	761	166	0	0	307	761	0	0	0
MyIF_ID (IF_ID)	106	67	0	0	79	106	6	0	0
MyMEM_WB (MEM_WB)	925	113	0	0	437	925	0	0	0
MyPC (PC)	163	32	1	0	73	163	0	0	0
MyRegisterFile (Regist...	592	992	256	32	508	592	17	0	0

Resource	Utilization	Available	Utilization %
LUT	5445	20800	26.18
FF	9732	41600	23.39
IO	22	106	20.75



而以下是之前做单、多处理器的逻辑资源占用状况：

单周期：

Resource	Utilization	Available	Utilization %
LUT	3928	20800	18.88
FF	9230	41600	22.19
IO	162	106	152.83

多周期：

Resource	Utilization	Available	Utilization %
LUT	4079	20800	19.61
FF	9503	41600	22.84
IO	162	106	152.83

对比可知，流水线为了照顾不同指令的兼容性和处理各种各样的冒险，导致占用的资源较多。与此同时，其将指令拆分成多个周期并行的思想使其具有更高的时钟周期，使其执行效率更高。



# 文件清单

注：要求中提到需要“关键代码及文件清单”，关键代码在上述分析过程中已有相应的展示，不再赘述。下整理文件清单。

文件夹	文件名	备注
Asm_Code	bf.asm	用于在 MARS 上调试的代码
	bf_vivado.asm	变为机器码后适用于 vivado
Constraints	CPU.xdc	约束文件
Design_Source	ALU.v	设计文件 具体含义在前均有介绍
	ALUControl.v	
	BranchUnit.v	
	clk_gen.v	
	Control.v	
	CPU.v	
	DataMemory.v	
	Display.v	
	EX_MEM.v	
	ForwardingUnit.v	
	HazardUnit.v	
	ID_EX.v	
	IF_ID.v	
	InstructionMemory.v	
	MEM_WB.v	
	PC.v	
	RegisterFile.v	
Other_Docs	asmconverter.py	给机器码加格式的 python 文件
	branch_hazard.drawio	报告中的图示
	branch_hazard.drawio.png	
	data_hazard.drawio	
	data_hazard.drawio.png	
	jump_hazard.drawio	
	jump_hazard.png	
	test.drawio	
	test.png	
	指令格式表及控制信号.xlsx	指令格式表及控制信号
Simulation_Source	test_cpu.v	仿真文件

# 心得体会

流水线作为上半个学期的学习重点，虽然已经系统学习过其原理，但真正去实操起来才发现有很多细节之前都没有想明白，尤其是种种冒险。可以说这次作业是对以往内容的一个综合、全面而完整的回顾，无论是知识上还是思维上。

对于一个较为繁重的任务，我认为我这次的处理步骤是较为合理的：先确定指令和控制信号，之后边绘制线路图边思考冒险处理和连线，最后进行测试和分析——把任务拆解成小步骤然后一步一步进行——这是这次实验给予我的一些方法论的思考。

另外，我认为这次实验中我也有很多不足的地方，例如时钟频率较低还没有着手去进行改进、有些地方思维较为繁琐不够简洁等等。

也要感谢这学期以来，张老师、助教们的指导以及同学们的帮助，让我磕磕绊绊地收获许多。