Pipeline

Q1 A 5-stage pipelined RISCV datapath is illustrated in Fig. 1. Now we have the time information for each component tabulated in Table 1. (15 points)

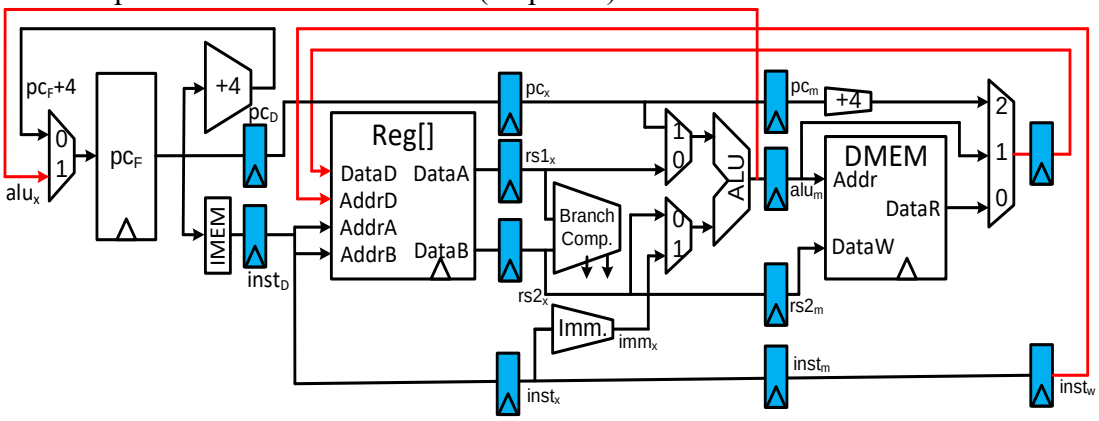

Fig. 1 5-stage pipelined RSICV datapath

Table 1

Element	Register clk-to-q	Register Setup	MUX	ALU	Mem Read	Mem Write	RegFile Read	RegFile Setup
Parameter	$t_{clk\text{-}to\text{-}q}$	t_{setup}	t_{mux}	t_{ALU}	$t_{MEMread}$	$t_{MEMwrite}$	t_{RFread}	$t_{RFsetup}$
Delay(ps)	30	20	25	250	400	500	200	50

(1) What is the clock time and frequency of a pipelined CPU (ignore the branch comp and imm.)?

Clock time:

Tclk >= the longest cycle time =

IF: tpc clk-to-q + t_imem read + t reg setup = 30+400+20=450 ps

Id: tReg clk-to-q + trf read + t reg setup = 30+200 +20 = 250 ps

EX: tReg clk-to-q + tmux +t alu + t reg setup = 30 + 25+250+20 = 325ps

MEM : tReg clk-to-q + tDMEM read + tmux + tReg setup = 30 + 400 + 25 + 20 = 475 ps

WB: tReg clk-to-q + trf setup = 30+20 =50 ps

The frequency of a pipelined CPU :

1*10^12/475=2105263157.894737 = 2.1GHz

(2) What is the purpose of the new registers?

We need to save the result of the last stage so that different instructions in the pipeline would not influence each other. What' s more, we could control each stage by generating appropriate control signals.

(3) Why do we add +4 to the PC again in the memory stage?

此处指令的执行 Execute 阶段，是第三个，而这个第三阶段的 Execute 和指令的第一个阶段的 Fetch 取指，相差的值是 3 -1 =2，即两个 CPU 的 Cycle，而每个 Cycle 都会导致 PC=+PC+4，所以，指令到了 Execute 阶段，才会发现，此时 PC 已经变成 PC=PC+8 了。

The Execute stage of the instruction here is the third of the five-stage pipeline, and the third stage of Execute and the Fetch of the first stage of the instruction are fetched. The difference is 3 -1 = 2, that is, two The cycle of the CPU, and each cycle will cause PC=+PC+4, so the instruction will not find until the Execute stage, at this time the PC has become PC=PC+8. .So we needadd +4 to the PC again.

(4) Why do we need to save the instruction in a register multiple times?

Because each pipeline stage needs to receive correct control signals for the current instruction.

Hazards

Data hazards occur due to data dependencies among instructions. Forwarding can solve many data hazards.

Q2. Spot the data dependencies in the code below and figure out how forwarding can resolve data hazards. (10 points)

Instruction	C0	C1	C2	C3	C4	C5	C6
addi t0, s0, -1	IF	REG	EX	MEM	WB		
and s2, t0, a0		IF	REG	EX	MEM	WB	
sw s0, 100(t0)			IF	REG	EX	MEM	WB

```
指令 1 和指令 2 冲突，因为指令 2 需要指令 1 的 t0，指令 3 和指令 1 冲突,指令 3 需要指令 1 的 t0.
And s2,t0,a0
Sw s0,100(t0)
```

Needs the result of addi, so after

```
ex addi(C2)
```

we needs to forward data to

```
ex And s2,t0,a0 #(C3) 转发到指令 2 的 ex 也就是 C3 开始之前
ex Sw s0,100(t0) #(C4) 执行阶段，sw 指令通过 t0+offset 算出地址，mem 阶段，把 s0 存入. 因此我们需要把 c2 的结果转发给 sw 的执行阶段 c4 之前,sw 的 reg 阶段取过来的就不要了.
```

Q3. In general, under what conditions will an EX stage need to take in forwarded inputs from previous instructions? Where should those inputs come from in regards to the current cycle? Assume you have the signals ALUout(n), rt(n), rs(n), regWrite(n), and regDst(n), where n is 0 for the signal of the current instruction being executed by the EX stage, -1 for the previous, etc.

通常，在什么条件下，EX 阶段需要从先前的指令中接收转发的输入？ 就当前周期而言，这些 input 应从何而来？假设您有信号 ALUout（n），rt（n），rs（n），regWrite（n）和 regDst（n），其中 n 为 0，表示 EX 级正在执行的当前指令的信号- 1 代表前一个，依此类推。

For example,

Forward ALUout(-1) if (rs2(0) == regDst(-1) || rs1(0) == regDst(-1)) && regWrite(-1)//如果要写回,那就直接转发
Add s2 s1 s0
Add s3 s2 s5
Please give other conditions. (10 points)

Forward ALUout(-2) if (rs2(0) == regDst(-2) || rs1(0) == regDst(-2)) && regWrite(-2)// 还在 dm,没有写回
Forward ALUout(-3) if (rs2(0) == regDst(-3) || rs1(0) == regDst(-3)) && regWrite(-3)//在 wb 但是还是来不及.

Lw 的转发不同,是在 mem 后才有结果
Forward memDst(-1) if(rs2(0) == regDst(-1) || rs1(0) == regDst(-1))&& regWrite(-1)

Q4. Spot the data dependencies in the code below and figure out why forwarding cannot resolve this hazard. What can we do to solve this data hazard? (10 points)

Instruction	C0	C1	C2	C3	C4	C5
lw t0, 20(s0)	IF	REG	EX	MEM	WB	
addiu t1, t0, t0		IF	REG	EX	MEM	WB

即使使用了旁路转发机制，仍然会产生一次阻塞，存储器访问的输出到执行级的输入之间的路径在时间上是倒着的,第二条指令需要在 C3 之前获得第一条指令 mem 的结果，而第一条指令 C3 之后才完成 mem.
我们可以插入气泡来解决这个数据冒险,或者重新排列指令.

Even if the bypass mechanism is used, a blockage will still occur, and the path from the output of the memory access to the input of the execution stage is reversed in time, which is obviously impossible.
We can insert bubbles to solve this data adventure, or rearrange the instructions,

Q5. Given the RISC-V code below and a pipelined CPU with no forwarding, how many hazards

would there be? What types are each hazard? Consider all possible hazards from all pairs of instructions. How many stalls would there need to be in order to fix the data hazard(s)? What about the control hazard(s)? (10 points)

Instruction	C0	C1	C2	C3	C4	C5	C6			
1. addi t0, s0, -1	IF	REG	EX	MEM	WB					
2. and s2, t0, a0		IF	REG	EX	MEM	WB				
3. sw s2, 100(t0)			IF	REG	EX	MEM	WB			
4. beq s0, s3, label				IF	REG	EX	MEM	WB		
5. addi t2, x0, x0					IF	REG	EX	MEM	WB	

And s2,t0,a0 有数据冒险，依赖前面的数据 t0. 要停三个时钟周期，写回后才能取 reg.
Sw s2,100(t0) 有数据冒险，依赖前面的数据 t0(停到 C4 之后再 reg)和 s2(停到 C3 之后再执行),停两个周期.
Beq s0 ,s3,label 控制冒险，因为可能跳转，但是后面的读入进流水线了.要停三个周期，直到 mem 级才知道是否执行分支，我们可以分支预测来消除.
Addi t2,x0,x0

Q6. Besides stalling, what can we do to resolve control hazards? (5 points)
1.分支预测,这样流水线可以全速执行，甚至可以动态预测，根据本次预测的结果来调整下一次的预测行为. 历史记录足够多时，正确率可以达到 90%，预测错误时，让后面的指令执行不生效，flush 流水线，在正确的分支地址处重新开始启动流水线. 一种方法是预测分支总是不发生.
2.延迟分支， 比如用一些不影响分支的指令来隐藏分支延迟，但是只有分支延迟较短的时候延迟分支才有效，对更长的延迟分支，一般都使用硬件分支预测器.

Branch prediction, one method is to predict that the branch will not occur, so that the pipeline can be executed at full speed, and can even be dynamically predicted, according to the results of this prediction to adjust the next prediction behavior. When there are enough historical records, the accuracy rate can reach 90%, and the prediction When an error occurs, the execution of the following instructions will not take effect, and the pipeline will be restarted at the correct branch address.
There is also a delayed branch.For example, some instructions that do not affect the branch are used to hide the branch delay, but the delayed branch is only effective when the branch delay is short.For longer delayed branches, hardware branch predictors are generally used.

Performance Analysis
Q7. (10 points)

Register clk-to-q 30 ps	**Branch comp.** 75 ps	**Memory write** 200 ps
Register setup 20 ps	**ALU** 200 ps	**RegFile read** 150 ps
Mux 25 ps	**Memory read** 250 ps	**RegFile setup** 20 ps

(1) With the delays provided above for each of the datapath components, what would be the fastest possible clock time for a single cycle datapath?

30+75+200+0(setup ignored)+200+150+25+250+0(setup ignored)=930ps

IF: tpc clk-to-q + t_imem read + t reg setup = 30+250+20=300 ps

Id: tReg clk-to-q + trf read + t reg setup = 30+150+20 = 200 ps

EX: tReg clk-to-q + tmux +t alu + t reg setup = 30 + 25+200+20 = 275ps

MEM : tReg clk-to-q + tDMEM read + tmux + tReg setup = 30+250+25+20=325 ps

WB: tReg clk-to-q + trf setup = 30+20 =50 ps

(2) What is the fastest possible clock time for a pipelined datapath?

the fastest possible clock time = The longest stage time = MEM =325ps

Q8. (10 points)

4.24 [10] <§4.7> Which of the two pipeline diagrams below better describes the operation of the pipeline's hazard detection unit? Why?

Choice 1:

```
ld x11, 0(x12):     IF ID EX ME WB
add x13, x11, x14:    IF ID EX..ME WB
or x15, x16, x17:      IF ID..EX ME WB
```

Choice 2:

```
ld x11, 0(x12):    IF ID EX ME WB
add x13, x11, x14:  IF ID..EX ME WB
or x15, x16, x17:    IF..ID EX ME WB
```

在 add 执行前检测，因为不一定读入的 x11 更新了.

在 or 译码前检测. 检测装载指令，如果 id 指令被阻塞, if 也必须阻塞，否则已经取到的指令就会丢失.

Q9. (20 points)

4.25 Consider the following loop.

```
LOOP: ld    x10, 0(x13)
      ld    x11, 8(x13)
      add   x12, x10, x11
      subi  x13, x13, 16
      bnez  x12, LOOP
```

Assume that perfect branch prediction is used (no stalls due to control hazards), that there are no delay slots, that the pipeline has full forwarding support, and that branches are resolved in the EX (as opposed to the ID) stage.

4.25.1 [10] <§4.7> Show a pipeline execution diagram for the first two iterations of this loop.

4.25.2 [10] <§4.7> Mark pipeline stages that do not perform useful work. How often while the pipeline is full do we have a cycle in which all five pipeline stages are doing useful work? (Begin with the cycle during which the `subi` is in the IF stage. End with the cycle during which the `bnez` is in the IF stage.)

As opposed to the ID stage.

Assume that perfect branch prediction is used(no stalls due to control hazards)是说没有 control hazards 造成的停，而不是说没有 stall.

4.25.1

Show a pipeline execution diagream for the first two iterations of this loop.

如下图所示

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ld x10,0(x13)	if	id	ex	mem(然后转发给6)	wb		if	id	ex	mem(然后	wb		if	id	ex	mem(然后
ld x11,8(x13)		if	id	ex	mem(然后	wb		if	id	ex	mem(然后	wb		if	id	ex
add x12 , x10,x11			if	stall(这时x11还没有	id	ex(5的mem转发	mem	wb	if	stall(要等n	id	ex(11的me	mem	wb	if	stall(要等n
subi x13,x13, 16					if	id	ex	mem	wb		if	id	ex	mem	wb	if
bnez x12,loop						if	id	ex	mem	wb		if	id	ex	mem	wb

显示此循环的前两次迭代的流水线执行图

4.25.2

How often 就是说占总的百分比多少.

8 和 9 是 five pipeline stages are doing useful work ， 5 到 12 总共 8 个,因此 占 25%