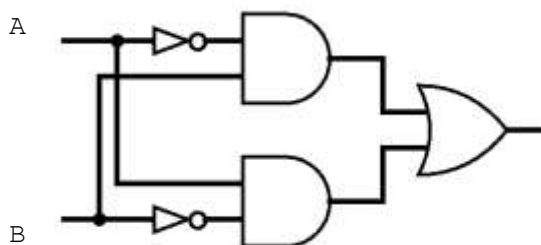


【体系结构基础】

1. 下列描述更符合（早期）RISC 还是 CISC?

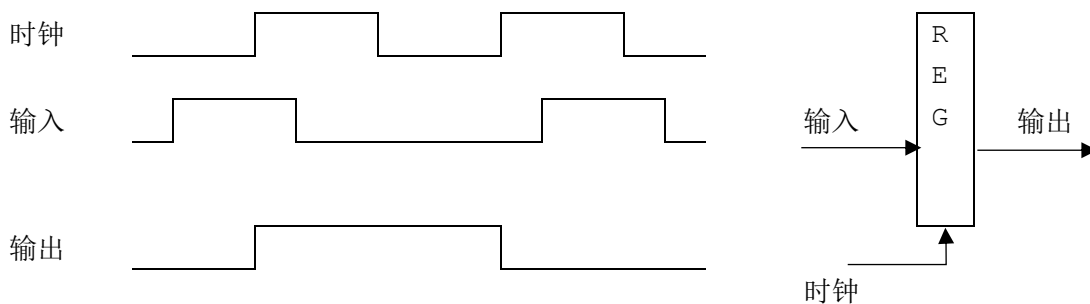
	描述	RISC	CISC
(1)	指令机器码长度固定	√	
(2)	指令类型多、功能丰富		√
(3)	不采用条件码	√	
(4)	实现同一功能，需要的汇编代码较多	√	
(5)	译码电路复杂		√
(6)	访存模式多样		√
(7)	参数、返回地址都使用寄存器进行保存	√	
(8)	x86-64		√
(9)	MIPS	√	
(10)	广泛用于嵌入式系统	√	
(11)	已知某个体系结构使用 <code>add R1, R2, R3</code> 来完成加法运算。当要将数据从寄存器 S 移动至寄存器 D 时，使用 <code>add S, #ZR, D</code> 进行操作（#ZR 是一个恒为 0 的寄存器），而没有类似于 <code>mov</code> 的指令。	√	
(12)	已知某个体系结构提供了 <code>xlat</code> 指令，它以一个固定的寄存器 A 为基地址，以另一个固定的寄存器 B 为偏移量，在 A 对应的数组中取出下标为 B 的项的内容，放回寄存器 A 中。		√

2. 写出下列电路的表达式



【答】 $(\neg A \ \&\& \ B) \ || \ (A \ \&\& \ \neg B)$

3. 下列寄存器在时钟上升沿锁存数据，画出输出的电平（忽略建立/保持时间）



【顺序处理器】

3. 根据 64 位 Y86-64 模型完成下表

		call	jXX
Fetch	icode, ifun	icode:ifun <- M1[PC]	icode:ifun <- M1[PC]
	rA, rB	\	
	valC	valC <- M8[PC+1]	valC <- M8[PC+1]
	valP	valP <- PC+9	valP <- PC+9
Decode	valA, srcA	\	\
	valB, srcB	valB <- R[%rsp]	\
Execute	valE	valE <- valB + (-8)	\
	Cond Code	\	Cnd <- Cond(CC, ifun)
Memory	valM	M8[valE] <- valP	\
Write back	dstE	R[%rsp] <- valE	\
	dstM	\	\
PC	PC	PC <- valC	PC <- Cnd?valC:valP

4. 已知 valC 为指令中的常数值, valM 为访存得到的数据, valP 为 PC 自增得到的值, 完成以下的 PC 更新逻辑:

```
int new_pc = [
    icode == ICALL : valC;
    icode == IJXX && Cnd: valC;
    icode == IRET : valM;
    1: valP;
]
```

【流水线的基本原理】

5. 判断下列说法的正确性

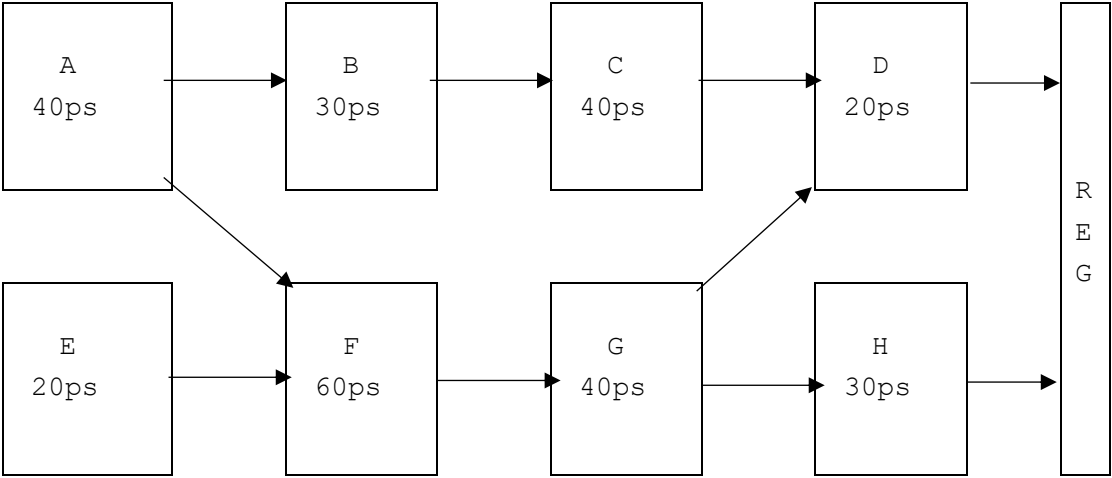
- (1) () 流水线的深度越深, 总吞吐率越大, 因此流水线应当越深越好。
- (2) () 流水线的吞吐率取决于最慢的流水级, 因此流水线的划分应当尽量均匀。
- (3) () 假设寄存器延迟为 20ps, 那么总吞吐率不可能达到或超过 50 GIPS。
- (4) () 数据冒险总是可以只通过转发来解决。
- (5) () 数据冒险总是可以只通过暂停流水线来解决。

【答】N, Y, Y, N(mrmov + add), Y

6. 一条三级流水线, 包括延迟为 50ps, 100ps, 100ps 的三个流水级, 每个寄存器的延迟为 10ps。那么这条流水线的总延迟是 _____ ps, 吞吐率是 _____ GIPS。

【答】3*(100+10)=330 ps, 1000/(100+10)=9.09 GIPS

7. A~H 为 8 个基本逻辑单元，下图中标出了每个单元的延迟，以及用箭头标出了单元之间的数据依赖关系。寄存器的延迟均为 10ps。



(1) 计算目前的电路的总延迟。

【答】40+60+40+30+10=180ps

(2) 通过插入寄存器，可以对这个电路进行流水化改造。现在想将其改造为两级流水线，为了达到尽可能高的吞吐率，问寄存器应插在何处？获得的吞吐率是多少？

【答】BC 与 FG 之间。1000/(40+60+10)=9.09GIPS

(3) 现在想将其改造为三级流水线，问最优改造所获得的吞吐率是多少？

【答】插在 AB、AF、EF、BC、FG 之间。1000/(40+30+10)=12.5GIPS

【流水线处理器】

8. 一个只使用流水线暂停、没有数据前递的 Y86 流水线处理器，为了执行以下的语句，至少需要累计停顿多少个周期？

<pre>irmovl \$1, %eax irmovl \$2, %ebx addl %eax, %ecx addl %ebx, %edx halt</pre>	<pre>rrmovl %eax, %edx mrmovl (%ecx), %eax addl %edx, %eax halt</pre>	<pre>irmovl \$0x40, %eax mrmovl (%eax), %ebx subl %ebx, %ecx halt</pre>
(1)	(2)	(3)

【答】(1) 2 个。第五个时钟周期结束以后，第三行代码才能开始译码。而原来第三行在第四个时钟周期开始译码，因此需要两周期停顿。(2) 3 个。1、3 两行，2、3 两行，均有数据相关。为了解决 1、3 两行的数据相关，需要额外的 2 个停顿；为了解决 2、3 两行的数据相关，需要额外 3 个停顿，因此需要 3 个停顿。(3) 6 个。1、2 两行的数据相关，需要额外 3 个停顿才能解决。2、3 两行的数据相关，需要额外 3 个停顿才能解决。

9. 考虑 Y86 中的 ret 与 jXX 指令。jXX 总是预测分支跳转。

(1) 写出流水线需要处理 ret 的条件 (ret 对应的常量为 IRET):

```
IRET in {D_icode, E_icode, M_icode}
```

(2) 写出发现上述条件以后, 流水线寄存器应设置的状态

	Fetch	Decode	Execute	Memory	Writeback
处理 ret	stall	bubble	normal	normal	Normal

(3) 写出流水线需要处理 jXX 分支错误的条件 (jXX 对应的常量为 IJXX):

```
E_icode == IJXX && !e_Cnd
```

(4) 写出发现上述条件以后, 流水线寄存器应设置的状态

	Fetch	Decode	Execute	Memory	Writeback
分支错误	normal	bubble	bubble	normal	Normal

(5) 写出下一条指令地址 f_pc 的控制逻辑

```
int f_pc = [
    M_icode == IJXX && !M_Cnd : M_valA;
    W_icode == IRET : W_valM;
    1 : F_predPC;
];

# 已知有如下的代码, 其中 valC 为指令中的常数值, valM 为访存得到的数据, valP
# 为 PC 自增得到的值:
int f_predPC = [
    f_icode in { IJXX, ICALL } : f_valC;
    1 : f_valP;
];

int d_valA = [
    D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
    # ...省略部分数据前递代码
    1 : d_rvalA; # Use value read from register file
];
```

【答】(1) 容易。(2) 由于第一次发现 ret 是在 Decode 阶段, 因此 Execute 阶段应当设置为 normal, 否则下一周期 ret 无法执行; 下一周期, ret 后的指令进入 Decode 阶段, 这是一条错误指令, 因此 Decode 应当设置为 bubble。(3) 容易。(4) 由于第一次发现 jXX 是在 Execute 阶段, 因此 Memory 阶段应当设置为 normal, 否则下一周期 jXX 无法执行; jXX 后面的两条指令均为错误指令, 下一周期它们将进入 Decode 和 Execute 阶段, 因此这两个阶段均为 bubble; 而 jXX 正确地址在 valP 中, 因此可以使下一周期取到正确的指令, Fetch 应当为 normal。(5) 在 Decode 阶段, valP 进了 d_valA, 在 Execute 结束以后就可以将正确的 PC (自增) 传回去, 此时它在 M_valA 里。当 Memory

阶段结束以后，`ret` 才能从内存中取出正确的地址，因此正确地址在 `w_valM` 里。

10. (2016 期中 `cret` 题)

11. (2018 期中最后一题)