

# Processor Arch

13 元培数科 常欣海

Here we go! →

2024/10/23



# Knowledge Review

# Emphasis

# RISC与CISC

- CISC: IA32, AMD64(x86-64)
- RISC: ARM64, RISC-V, MIPS

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 add R1, R2, R3 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 add S, #ZR, D 进行操作 (#ZR 是一个恒为 0 的寄存器)，而没有类似于 mov 的指令。	✓	
(12)	已知某个体系结构提供了 xlat 指令，它以一个固定的寄存器 A 为基址，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。		✓

## RISC vs. CISC

	RISC	CISC
指令(CMD)	种类少，使用频率高 指令编码长度固定(常为4个字节)	种类多，使用频率低 指令编码长度不定(x86-64的指令编码长度为1~15不等)
状态单元(Status)	寄存器数量多(最多32个) 没有条件码	寄存器数量较少 有条件码
函数调用(Procedure)	优先使用寄存器传参(也拥有用栈传参的能力) 可能隐式进行栈操作	完全依靠栈传参 只能显式进行栈操作
内存访问(Memory)	只有读取和写入两条指令 可以访问内存 只支持基址和偏移量寻址	算数运算和逻辑运算指令 也可以访问内存 支持多种寻址方法
其他(Others)	可以更充分地激发硬件性能(用体积更小的芯片跑更高的性能) 更易于进行程序性能优化	抽象程度更高，拥有很多对应高级程序语言典型操作的指令

# 程序员可见状态

**Figure 4.1**

**Y86-64 programmer-visible state.** As with x86-64, programs for Y86-64 access and modify the program registers, the condition codes, the program counter (PC), and the memory. The status code indicates whether the program is running normally or some special event has occurred.

RF: Program registers

%rax	%rsp	%r8	%r12
%rcx	%rbp	%r9	%r13
%rdx	%rsi	%r10	%r14
%rbx	%rdi	%r11	

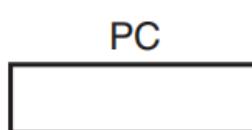
CC:  
Condition  
codes



Stat: Program status



DMEM: Memory



# Y86-64 ISA

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq rA, rB	2	0	rA	rB						
irmovq V, rB	3	0	F	rB			V			
rmmovq rA, D(rB)	4	0	rA	rB			D			
mrmovq D(rB), rA	5	0	rA	rB			D			
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn			Dest					
cmovXX rA, rB	2	fn	rA	rB						
call Dest	8	0			Dest					
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

# Y86-64 ISA

## Operations

addq 

6	0
---	---

subq 

6	1
---	---

andq 

6	2
---	---

xorq 

6	3
---	---

## Branches

jmp 

7	0
---	---

jle 

7	1
---	---

jl 

7	2
---	---

je 

7	3
---	---

## Moves

rrmovq 

2	0
---	---

cmovele 

2	1
---	---

cmovl 

2	2
---	---

cmove 

2	3
---	---

cmovne 

2	4
---	---

cmovge 

2	5
---	---

cmovg 

2	6
---	---

# Y86-64 ISA

Name	Value (hex)	Meaning
IHALT	0	Code for halt instruction
INOP	1	Code for nop instruction
IRRMVQ	2	Code for <code>rrmovq</code> instruction
IIRMOVQ	3	Code for <code>irmovq</code> instruction
IRMMOVQ	4	Code for <code>rmmovq</code> instruction
IMRMOVQ	5	Code for <code>mrmovq</code> instruction
IOPL	6	Code for integer operation instructions
IJXX	7	Code for jump instructions
ICALL	8	Code for <code>call</code> instruction
IRET	9	Code for <code>ret</code> instruction
IPUSHQ	A	Code for <code>pushq</code> instruction
IPOPQ	B	Code for <code>popq</code> instruction
FNONE	0	Default function code
RESP	4	Register ID for <code>%rsp</code>
RNONE	F	Indicates no register file access
ALUADD	0	Function for addition operation
SAOK	1	Status code for normal operation
SADR	2	Status code for address exception
SINS	3	Status code for illegal instruction exception
SHLT	4	Status code for halt

**Figure 4.26 Constant values used in HCL descriptions.** These values represent the encodings of the instructions, function codes, register IDs, ALU operations, and status codes.

Number	Register name	Number	Register name
0	<code>%rax</code>	8	<code>%r8</code>
1	<code>%rcx</code>	9	<code>%r9</code>
2	<code>%rdx</code>	A	<code>%r10</code>
3	<code>%rbx</code>	B	<code>%r11</code>
4	<code>%rsp</code>	C	<code>%r12</code>
5	<code>%rbp</code>	D	<code>%r13</code>
6	<code>%rsi</code>	E	<code>%r14</code>
7	<code>%rdi</code>	F	No register

**Figure 4.4 Y86-64 program register identifiers.** Each of the 15 program registers has an associated identifier (ID) ranging from 0 to 0xE. ID 0xF in a register field of an instruction indicates the absence of a register operand.

- IOPL改为IOPQ
- RESP改为RRSP

# push&pop指令

- push: 先将 %rsp 减 8, 再压栈
- pop: 先弹栈, 再将 %rsp 加 8
- 先这么理解, 原理会在第四章学习 (与表象并不完全一致)
- call: push + jmp, 可进行间接跳转
- ret: pop + jmp

**pushq %rsp**

效果: 让%rsp的值入栈  
入栈的值是%rsp还是%rsp - 8?

**popq %rsp**

效果: 弹出栈顶的值, 存储到%rsp中  
存储到%rsp中的值是%rsp + 8还是(%rsp)?

Instruction	Effect	Description
pushq <i>S</i>	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$	Push quad word
popq <i>D</i>	$D \leftarrow M[R[\%rsp]];$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	Pop quad word

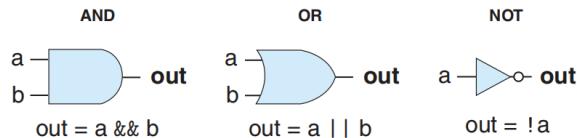
**Figure 3.8** Push and pop instructions.

# HCL

## 逻辑门

Figure 4.9

**Logic gate types.** Each gate generates output equal to some Boolean function of its inputs.



## 算术/逻辑单元ALU

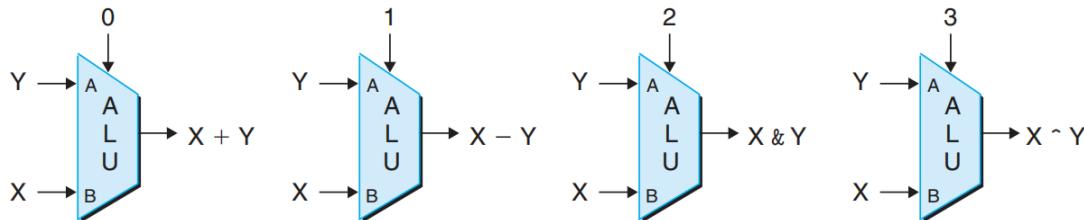


Figure 4.15 Arithmetic/logic unit (ALU). Depending on the setting of the function input, the circuit will perform one of four different arithmetic and logical operations.

# Y86-64硬件结构

## 阶段

- 取址: Fetch
- 译码: Decode
- 执行: Execute
- 访存: Memory
- 写回: Write back
- 更新 PC: PC Update

W: 只能写入valE, valM

( rrmovq, irmovq, cmovXX 需要\*\*+O\*\*做ALU计算)

## 指令处理

Stage	
Fetch	icode:ifun rA:rB valC valP
Decode	$valA \leftarrow R[rA / \%rsp]$ $valB \leftarrow R[rB / \%rsp]$
Execute	$valE \leftarrow (valB / 0) (OP / +) (valA / valC / \pm 8)$ Set CC $Cnd \leftarrow Cond(CC, ifun)$
Memory	$(M_8[valE] \leftarrow valA / valP) / (valM \leftarrow M_8[valE / valA])$
Write back	$R[rB / \%rsp] \leftarrow (if (Cnd)) valE$ $R[rA] \leftarrow valM$
PC update	$PC \leftarrow valP / (Cnd ? valC : valP) / valC / valM$

# 指令处理

Stage	$OPq\ rA, rB$	$rrmovq\ rA, rB$	$irmovq\ V, rB$
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC + 1]$ $valP \leftarrow PC + 2$	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC + 1]$ $valP \leftarrow PC + 2$	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC + 1]$ $valC \leftarrow M_8[PC + 2]$ $valP \leftarrow PC + 10$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	$valA \leftarrow R[rA]$	
Execute	$valE \leftarrow valB\ OP\ valA$ Set CC	$valE \leftarrow 0 + valA$	$valE \leftarrow 0 + valC$
Memory			
Write back	$R[rB] \leftarrow valE$	$R[rB] \leftarrow valE$	$R[rB] \leftarrow valE$
PC update	$PC \leftarrow valP$	$PC \leftarrow valP$	$PC \leftarrow valP$

# 指令处理

Stage	rmmovq rA, D(rB)	mrmovq D(rB), rA
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_8[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 10$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_8[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 10$
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	$\text{valB} \leftarrow R[\text{rB}]$
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$	$\text{valE} \leftarrow \text{valB} + \text{valC}$
Memory	$M_8[\text{valE}] \leftarrow \text{valA}$	$\text{valM} \leftarrow M_8[\text{valE}]$
Write back		$R[\text{rA}] \leftarrow \text{valM}$
PC update	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$

# 指令处理

Stage	<code>cmoveXX rA, rB</code>
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $rA:rB \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$
Decode	$\text{valA} \leftarrow R[rA]$
Execute	$\text{valE} \leftarrow 0 + \text{valA}$ $\text{Cnd} \leftarrow \text{Cond(CC, ifun)}$
Memory	
Write back	$\text{if } (\text{Cnd}) R[rB] \leftarrow \text{valE}$
PC update	$\text{PC} \leftarrow \text{valP}$

# 指令处理

Stage	pushq rA	popq rA
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$  $\text{valP} \leftarrow \text{PC} + 2$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$  $\text{valP} \leftarrow \text{PC} + 2$
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\%rsp]$	$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$
Execute	$\text{valE} \leftarrow \text{valB} + (-8)$	$\text{valE} \leftarrow \text{valB} + 8$
Memory	$M_8[\text{valE}] \leftarrow \text{valA}$	$\text{valM} \leftarrow M_8[\text{valA}]$
Write back	$R[\%rsp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$	$R[\%rsp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$
PC update	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$

# Overview

# 基本概念

1. 吞吐量：单位时间内完成的指令数（单位：GIPS，每秒十亿条指令）
2. 延迟：完成一条指令需要的时间（单位：ps，皮秒）
3. CPI：执行一条指令所需要的平均时钟周期数（单位：时钟周期）

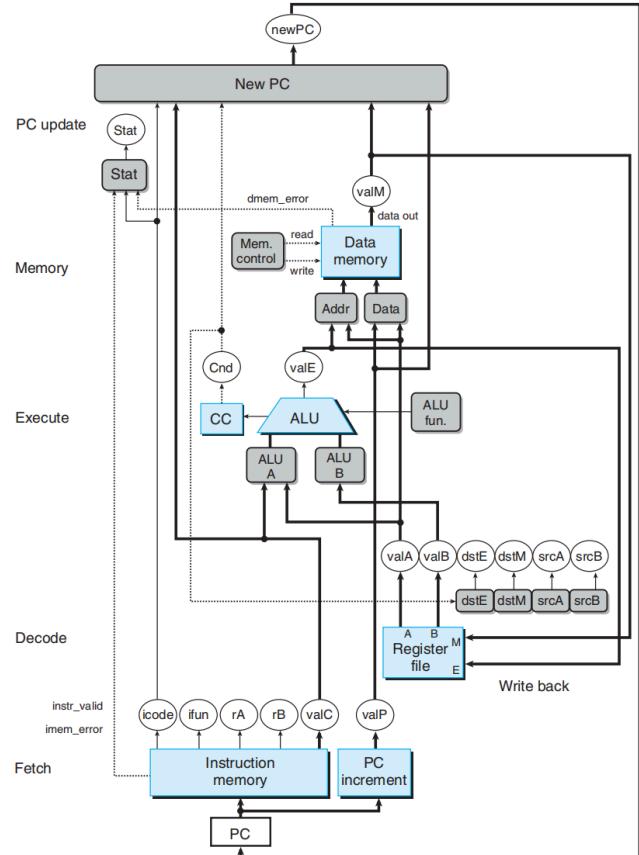
$$\text{CPI} = \frac{C_i + C_b}{C_i} = 1.0 + \frac{C_b}{C_i}$$

$$\text{CPI} = 1.0 + lp + mp + rp$$

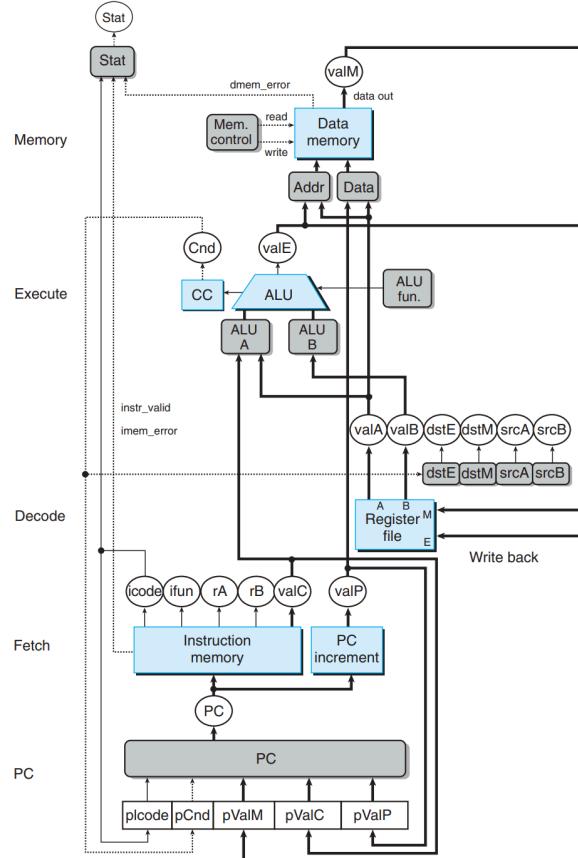
- $lp$  - 加载处罚,  $mp$  - 预测错误分支处罚,  $rp$  - 返回处罚

Cause	Name	Instruction frequency	Condition frequency	Bubbles	Product
Load/use	$lp$	0.25	0.20	1	0.05
Mispredict	$mp$	0.20	0.40	2	0.16
Return	$rp$	0.02	1.00	3	0.06
Total penalty					0.27

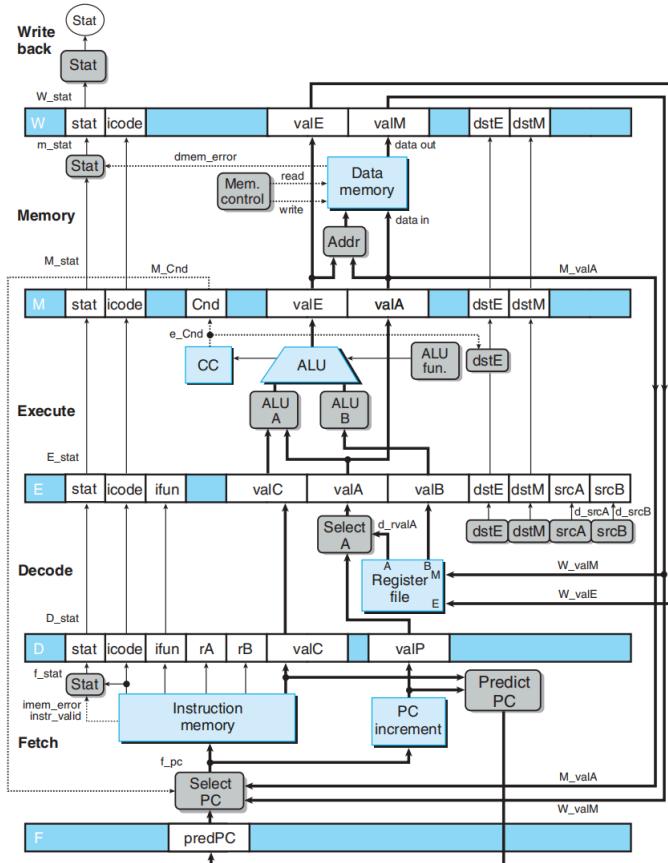
4. IPC：每周期执行指令平均数（单位：指令数）



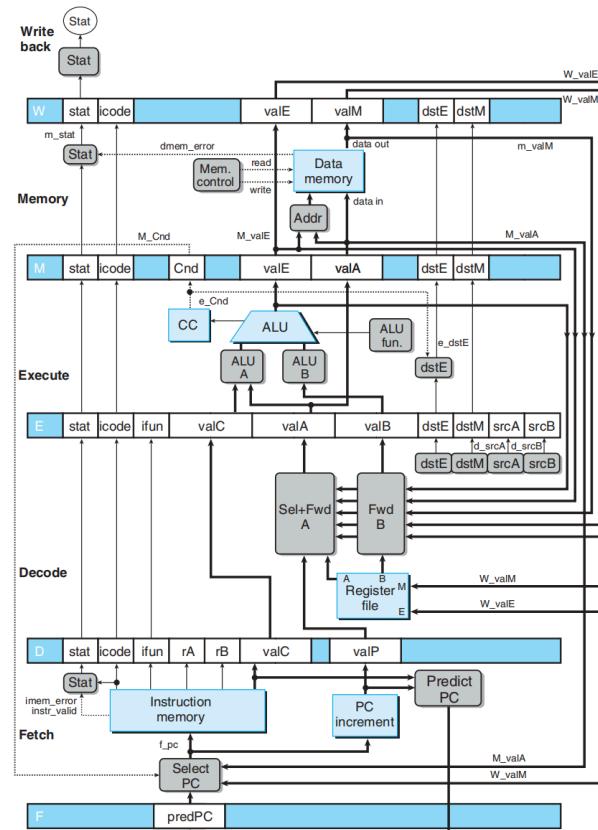
**Figure 4.23** Hardware structure of SEQ, a sequential implementation. Some of the control signals, as well as the register and control word connections, are not shown.



**Figure 4.40** SEQ+ hardware structure. Shifting the PC computation from the end of the clock cycle to the beginning makes it more suitable for pipelining.



**Figure 4.41** Hardware structure of PIPE-, an initial pipelined implementation. By inserting pipeline registers into SEQ+ (Figure 4.40), we create a five-stage pipeline. There are several shortcomings of this version that we will deal with shortly.



**Figure 4.52** Hardware structure of PIPE, our final pipelined implementation. The additional bypassing paths enable forwarding the results from the three preceding instructions. This allows us to handle most forms of data hazards without stalling the pipeline.

# Details

SEQ、PIPE的具体实现图、HCL代码都需要详细掌握

此处不再说明，详见【补充资料】部分

# SEQ->SEQ+

## 前置PC

- 电路重定时：改变状态表示而不改变逻辑
- 目的：平衡一个流水线各个阶段之间的延迟
- 在 SEQ+ 的实现里，PC update 的时期从周期的最后被提到了最前，更加接近流水线的形态

# SEQ+ -> PIPE-

## 阶段划分

在 Y86-64 的实现中：

- 正常指令指令默认预测 PC 为下一条指令的地址；
- call 指令和 jxx 指令默认预测 PC 为跳转后地址；
- ret 指令不进行任何预测，直到其对应的写回完成

插入流水线寄存器：分别插入了5个流水线寄存器用来保存后续阶段所需的信号，编号为 F、D、E、M 和 W

- **Fetch**: Select current PC; Read instruction; Compute incremented PC
- **Decode**: Read program registers
- **Execute**: Operate ALU
- **Memory**: Read or write data memory
- **Write Back**: Update register file

寄存器顺序：F—f—D—d—E—e—M—m—W

# PIPE->PIPE

## 处理冒险

- 硬件：暂停和气泡
  - stall 能将指令阻塞在某个阶段
  - bubble 能使得流水线继续运行，但是不会改变当前阶段的寄存器、内存、条件码或程序状态
- 结构冒险
  - 计算的多时钟周期：采用独立于主流水线的特殊硬件功能单元来处理较为复杂的操作（一个功能单元执行整数乘法和除法，一个功能单元执行浮点操作）
  - 访存的多时钟周期：
    - 翻译后备缓冲器 (TLB) + 高速缓存 (Cache)：实现一个时钟周期内读指令并读或写数据
    - 缺页 (page fault) 异常信号：指令暂停+磁盘到主存传送+指令重新执行

# PIPE->PIPE

## 处理冒险

### ■ 数据冒险

- **前后使用数据冒险**: 在处理器中, `valA` 和 `valB` 一共有5个转发源:
  - `e_valE` : 在执行阶段, ALU中计算得到的结果 `valE` , 通过 `E_dstE` 与 `d_srcA` 和 `d_src_B` 进行比较决定是否转发。
  - `M_valE` : 将ALU计算的结果 `valE` 保存到流水线寄存器M中, 通过 `M_dstE` 与 `d_srcA` 和 `d_src_B` 进行比较决定是否转发。
  - `m_valM` : 在访存阶段, 从内存中读取的值 `valM` , 通过 `M_dstM` 与 `d_srcA` 和 `d_src_B` 进行比较决定是否转发。
  - `W_valM` : 将内存中的值 `valM` 保存到流水线寄存器W中, 通过 `W_dstM` 与 `d_srcA` 和 `d_src_B` 进行比较决定是否转发。
  - `W_valE` : 将ALU计算的结果 `valE` 保存到流水线寄存器W中, 通过 `W_dstE` 与 `d_srcA` 和 `d_src_B` 进行比较决定是否转发。

# PIPE->PIPE

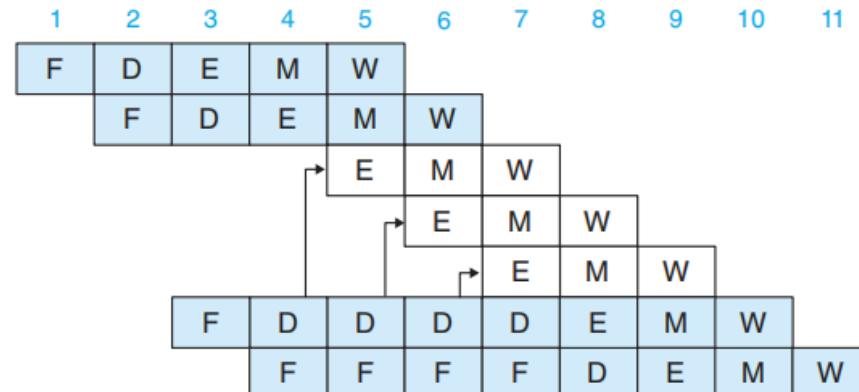
## 处理冒险

- 数据冒险

- 用暂停来避免数据冒险

- 插入一段自动产生的 `nop` 指令
    - 该方法指令要停顿最少一个最多三个时钟周期，严重降低整体的吞吐量

```
# prog4
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
      bubble
      bubble
      bubble
0x014: addq %rdx,%rax
0x016: halt
```



# PIPE->PIPE

## 处理冒险

- 数据冒险
  - 加载/使用数据冒险

Condition	Trigger
Processing ret	$IRET \in \{D\_icode, E\_icode, M\_icode\}$
Load/use hazard	$E\_icode \in \{IMRMOVQ, IPOPQ\} \&& E\_dstM \in \{d\_srcA, d\_srcB\}$
Mispredicted branch	$E\_icode = IJXX \&& !e\_Cnd$
Exception	$m\_stat \in \{SADR, SINS, SHLT\} \mid W\_stat \in \{SADR, SINS, SHLT\}$

Condition	Pipeline register				
	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal

# PIPE->PIPE

## 处理冒险

- 控制冒险

- ret指令（不预测）：删除后续操作——插入3个bubble

```
# prog7
 1 2 3 4 5 6 7 8 9 10 11
0x000: irmovq Stack,%edx
0x00a: call proc
0x020: ret
    bubble
    bubble
    bubble
0x013: irmovq $10,%edx # Return point
```

The diagram illustrates the insertion of three bubbles after the ret instruction. The pipeline stages are F (Fetch), D (Decode), E (Execute), M (Memory Access), and W (Write). The original code has a ret instruction at stage 6. Three bubbles are inserted after it, pushing the subsequent irmovq instruction to stage 11.

- 跳转指令（预测）：删除后续操作——插入2个bubble

```
# prog7
 1 2 3 4 5 6 7 8 9 10
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovl $2,%rdx # Target
    bubble
0x020: irmovl $3,%rbx # Target+
    bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```

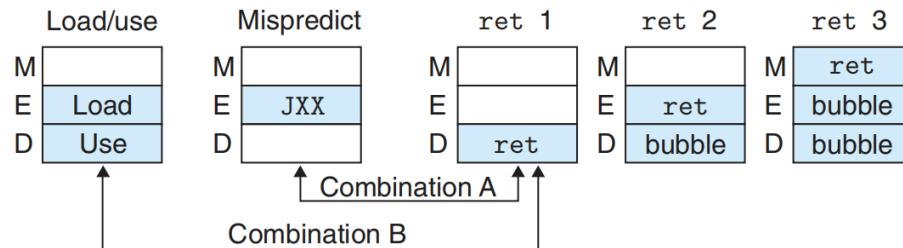
The diagram illustrates the insertion of two bubbles after the jne instruction. The pipeline stages are F (Fetch), D (Decode), E (Execute), M (Memory Access), and W (Write). The original code has a jne instruction at stage 2. Two bubbles are inserted after it, pushing the subsequent irmovl instruction to stage 10.

# PIPE->PIPE

## 处理冒险

- 检验自洽

- 控制条件组合——有限性组合： **Combination A + B**
  - Combination A: \*\*ret位于不选择分支\*\* ——简单叠加
  - Combination B: \*\*加载/使用+ret\*\* ——取”**stall**”
  - 加载互锁核心思想：通过暂停+转发组合实现
  - 合理性： Install后，下一条指令无法进入寄存器，当前指令因为bubble并未成功下传
  - 有效性： Install后，当前指令ret依然存在于流水线中，加载/使用语句后可进一步执行



# PIPE->PIPE

## 处理冒险

- 检验自治

Condition	Pipeline register				
	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal
Combination	stall	bubble	bubble	normal	normal

Condition	Pipeline register				
	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Combination	stall	bubble+stall	bubble	normal	normal
Desired	stall	stall	bubble	normal	normal

# PIPE->PIPE

## 处理冒险

- 异常处理
  - 内部异常：
    - HLT：执行halt指令
    - ADR：从非法内存地址读或向非法内存地址写
    - INS：非法指令
  - 外部异常
    - 系统重启
    - I/O设备请求
    - 硬件故障
- 要求：异常指令之前的所有指令已经完成，后续的指令都不能修改条件码寄存器和内存。

# PIPE- ->PIPE

## 处理冒险

- 异常处理

1. 当同时多条指令引起异常时，处理器应该向操作系统报告哪个异常？

**基本原则：**由流水线中最深的指令引起的异常，表示该指令越早执行，优先级最高。

2. 在分支预测中，当预测分支中出现了异常，而后由于预测错误而取消该指令时，需要取消异常。
3. 如何处理不同阶段更新系统状态不同部分的问题？

- 异常发生时，记录指令状态，继续取指、译码、执行
- 异常到达 **访存阶段**：
  1. 执行阶段，禁止设置条件码 ( $set\_cc \leftarrow m\_stat, W\_stat$ )
  2. 访存阶段，插入气泡，禁止写入内存
  3. 写回阶段，暂停写回，即暂停流水线

# Homework Review

# HW4

学号	HW-分数
2300012932	10
2300012935	10
2300012950	10
2300012951	10
2300013083	10
2300013115	10
2300013158	10
2300013201	10
2300013222	10
2300013230	10
2300013272	10
2300017788	10
2300094610	10

- 可能确实不存在很好的办法验证Y86-64的正确性
- 暂不存在 C 到 Y86-64 的转换工具
- 但可以通过手动模拟的方式，通过Y86-64模拟器验证正确性（如果你真的想这么干）

# Exercises

# Processor Arch: ISA & Logic

**E1**

10. 下面有三组对于指令集的描述，它们分别符合 ①\_\_\_\_\_，②\_\_\_\_\_，③\_\_\_\_\_ 的特点。

- ① 某指令集中，只有两条指令能够访问内存。
- ② 某指令集中，指令的长度都是 4 字节。
- ③ 某指令集中，可以只利用一条指令完成字符串的复制，也可以只利用一条指令查找字符串中第一次出现字母 K 的位置。

- A. CISC, CISC, CISC
- B. RISC, RISC, CISC
- C. RISC, CISC, RISC
- D. CISC, RISC, RISC

**E1**

10. 下面有三组对于指令集的描述，它们分别符合 ①\_\_\_\_\_，②\_\_\_\_\_，③\_\_\_\_\_ 的特点。

- ① 某指令集中，只有两条指令能够访问内存。
- ② 某指令集中，指令的长度都是 4 字节。
- ③ 某指令集中，可以只利用一条指令完成字符串的复制，也可以只利用一条指令查找字符串中第一次出现字母 K 的位置。

- A. CISC, CISC, CISC
- B. RISC, RISC, CISC
- C. RISC, CISC, RISC
- D. CISC, RISC, RISC

**【答】B。** ①的访存模式单一，更加符合 RISC 的特点；②的指令长度固定，更加符合 RISC 的特点；③的指令功能丰富而复杂，更加符合 CISC 的特点。

## E2

9. 请比较 RISC 和 CISC 的特点，回答下述问题：

假设编译技术处于发展初期，程序员更愿意使用汇编语言编程来解决实际问题，那么程序员会更倾向于选用 \_\_\_\_\_ ISA。

假设你设计的处理器速度非常快，但存储系统设计使得取指令的速度非常慢（也许是处理单元的十分之一）。这时你会更倾向于选用 \_\_\_\_\_ ISA。

- A) RISC、RISC
- B) CISC、CISC
- C) RISC、CISC
- D) CISC、RISC

## E2

9. 请比较 RISC 和 CISC 的特点，回答下述问题：

假设编译技术处于发展初期，程序员更愿意使用汇编语言编程来解决实际问题，那么程序员会更倾向于选用 \_\_\_\_\_ ISA。

假设你设计的处理器速度非常快，但存储系统设计使得取指令的速度非常慢（也许是处理单元的十分之一）。这时你会更倾向于选用 \_\_\_\_\_ ISA。

- A) RISC、RISC
- B) CISC、CISC
- C) RISC、CISC
- D) CISC、RISC

答案：B

//知识点 1：CISC 有更多的指令，有些更接近高级语言

//知识点 2：CISC 的指令功能更复杂，指令执行需要更多的周期，一定程度上可以平衡处理速度与指令访存的速度差异。不过，通常处理器设计中，主要通过多层次的存储体系结构来弥补两者之的速度差异。

## E3

9. 下面关于 RISC 和 CISC 的描述中，正确的是：
- A. CISC 和早期 RISC 在寻址方式上相似，通常只有基址和偏移量寻址
  - B. CISC 指令集可以对内存和寄存器操作数进行算术和逻辑运算，而 RISC 只能对寄存器操作数进行算术和逻辑运算
  - C. CISC 和早期的 RISC 指令集都有条件码，用于条件分支检测
  - D. CISC 机器中的寄存器数目较少，函数参数必须通过栈来进行传递；RISC 机器中的寄存器数目较多，只需要通过寄存器来传递参数，避免了不必要的存储访问

## E3

9. 下面关于 RISC 和 CISC 的描述中，正确的是：
- A. CISC 和早期 RISC 在寻址方式上相似，通常只有基址和偏移量寻址
  - B. CISC 指令集可以对内存和寄存器操作数进行算术和逻辑运算，而 RISC 只能对寄存器操作数进行算术和逻辑运算
  - C. CISC 和早期的 RISC 指令集都有条件码，用于条件分支检测
  - D. CISC 机器中的寄存器数目较少，函数参数必须通过栈来进行传递；RISC 机器中的寄存器数目较多，只需要通过寄存器来传递参数，避免了不必要的存储访问

**答案： B**

## E4

8. 下面对指令系统的描述中，错误的是：（ ）

- A. CISC 指令系统中的指令数目较多，有些指令的执行周期很长；而 RISC 指令系统中通常指令数目较少，指令的执行周期都较短。
- B. CISC 指令系统中的指令编码长度不固定；RISC 指令系统中的指令编码长度固定，这样使得 CISC 机器可以获得了更短的代码长度。
- C. CISC 指令系统支持多种寻址方式，RISC 指令系统支持的寻址方式较少。
- D. CISC 机器中的寄存器数目较少，函数参数必须通过栈来进行传递；RISC 机器中的寄存器数目较多，只需要通过寄存器来传递参数，避免了不必要的存储访问。

## E4

8. 下面对指令系统的描述中，错误的是：（ ）

- A. CISC 指令系统中的指令数目较多，有些指令的执行周期很长；而 RISC 指令系统中通常指令数目较少，指令的执行周期都较短。
- B. CISC 指令系统中的指令编码长度不固定；RISC 指令系统中的指令编码长度固定，这样使得 CISC 机器可以获得了更短的代码长度。
- C. CISC 指令系统支持多种寻址方式，RISC 指令系统支持的寻址方式较少。
- D. CISC 机器中的寄存器数目较少，函数参数必须通过栈来进行传递；RISC 机器中的寄存器数目较多，只需要通过寄存器来传递参数，避免了不必要的存储访问。

**答案： D**

## E5

11. 下面有关指令系统设计的描述正确的是：

- A. 采用 CISC 指令比 RISC 指令代码更长。
- B. 采用 CISC 指令比 RISC 指令运行时间更短
- C. 采用 CISC 指令比 RISC 指令译码电路更加复杂
- D. 采用 CISC 指令比 RISC 指令的流水线吞吐更高

## E5

11. 下面有关指令系统设计的描述正确的是：

- A. 采用 CISC 指令比 RISC 指令代码更长。
- B. 采用 CISC 指令比 RISC 指令运行时间更短
- C. 采用 CISC 指令比 RISC 指令译码电路更加复杂
- D. 采用 CISC 指令比 RISC 指令的流水线吞吐更高

答案：C，CISC 的比 RISC 代码复杂（如不定长），因此译码也更复杂，优点是代码更短。运行时间和吞吐都谁更高要依据实际应用。

## E6

11、关于 RISC 和 CISC 的描述，正确的是：

- A. CISC 指令系统的指令编码可以很短，例如最短的指令可能只有一个字节，因此 CISC 的取指部件设计会比 RISC 更为简单。
- B. CISC 指令系统中的指令数目较多，因此程序代码通常会比较长；而 RISC 指令系统中通常指令数目较少，因此程序代码通常会比较短。
- C. CISC 指令系统支持的寻址方式较多，RISC 指令系统支持的寻址方式较少，因此用 CISC 在程序中实现访存的功能更容易。
- D. CISC 机器中的寄存器数目较少，函数参数必须通过栈来进行传递；RISC 机器中的寄存器数目较多，只需要通过寄存器来传递参数。

## E6

11、关于 RISC 和 CISC 的描述，正确的是：

- A. CISC 指令系统的指令编码可以很短，例如最短的指令可能只有一个字节，因此 CISC 的取指部件设计会比 RISC 更为简单。
- B. CISC 指令系统中的指令数目较多，因此程序代码通常会比较长；而 RISC 指令系统中通常指令数目较少，因此程序代码通常会比较短。
- C. CISC 指令系统支持的寻址方式较多，RISC 指令系统支持的寻址方式较少，因此用 CISC 在程序中实现访存的功能更容易。
- D. CISC 机器中的寄存器数目较少，函数参数必须通过栈来进行传递；RISC 机器中的寄存器数目较多，只需要通过寄存器来传递参数。

答案：C

考查对 CISC 和 RISC 基本特点的描述，A 和 B 都是描述反了，D 则是太绝对，RISC 也有可能用栈来传递参数。

**E7**

4. 下述关于 RISC 和 CISC 的讨论，哪个是错误的
- A. RISC 指令集包含的指令数量通常比 CISC 的少
  - B. RISC 的寻址方式通常比 CISC 的寻址方式少
  - C. RISC 的指令长度通常短于 CISC 的指令长度
  - D. 手机处理器通常采用 RISC，而 PC 采用 CISC

## E7

4. 下述关于 RISC 和 CISC 的讨论，哪个是错误的
- A. RISC 指令集包含的指令数量通常比 CISC 的少
  - B. RISC 的寻址方式通常比 CISC 的寻址方式少
  - C. RISC 的指令长度通常短于 CISC 的指令长度
  - D. 手机处理器通常采用 RISC，而 PC 采用 CISC
4. C。CISC 变长指令，有不少指令的长度是要比 RISC 短的。当代手机常常采用 ARM 架构，这是 RISC；实际上对能耗要求高或者结构简单的设备一般都用 RISC。

## E8

5. 下面对指令系统的描述中，错误的是：（ ）
- A. 通常 CISC 指令集中的指令数目较多，有些指令的执行周期很长；而 RISC 指令集中指令数目较少，指令的执行周期较短。
  - B. 通常 CISC 指令集中的指令长度不固定；RISC 指令集中的指令长度固定。
  - C. 通常 CISC 指令集支持多种寻址方式，RISC 指令集支持的寻址方式较少。
  - D. 通常 CISC 指令集处理器的寄存器数目较多，RISC 指令集处理器的寄存器数目较少。

## E8

5. 下面对指令系统的描述中，错误的是：（ ）
- A. 通常 CISC 指令集中的指令数目较多，有些指令的执行周期很长；而 RISC 指令集中指令数目较少，指令的执行周期较短。
  - B. 通常 CISC 指令集中的指令长度不固定；RISC 指令集中的指令长度固定。
  - C. 通常 CISC 指令集支持多种寻址方式，RISC 指令集支持的寻址方式较少。
  - D. 通常 CISC 指令集处理器的寄存器数目较多，RISC 指令集处理器的寄存器数目较少。
5. D。送分题，RISC 中寄存器一般更多。

# E9

1. 下列描述更符合（早期）RISC 还是 CISC？

	描述
(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 add R1,R2,R3 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 add S,#ZR,D 进行操作（#ZR 是一个恒为 0 的寄存器），而没有类似于 mov 的指令。
(12)	已知某个体系结构提供了 xlat 指令，它以一个固定的寄存器 A 为基址，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。

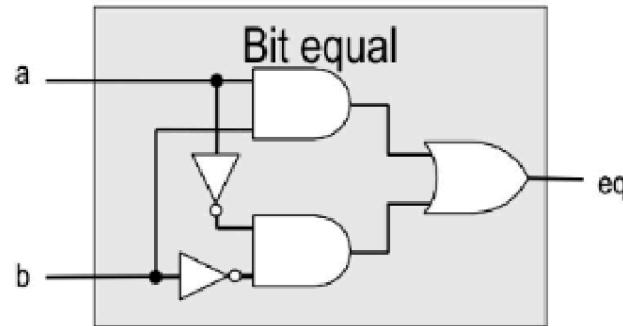
# E9

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 add R1, R2, R3 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 add S, #ZR, D 进行操作（#ZR 是一个恒为 0 的寄存器），而没有类似于 mov 的指令。	✓	
(12)	已知某个体系结构提供了 xlat 指令，它以一个固定的寄存器 A 为基址，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。		✓

# E10

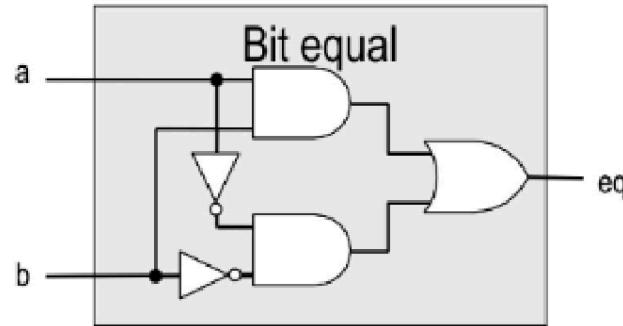
10. 对应下述组合电路的正确 HCL 表达式为：



- A. Bool eq = ( a or b ) and ( !a or !b )
- B. Bool eq = ( a and b ) or ( !a and !b )
- C. Bool eq = ( a or !b ) and ( !a or b )
- D. Bool eq = ( a and !b ) or ( !a and b )

# E10

10. 对应下述组合电路的正确 HCL 表达式为：

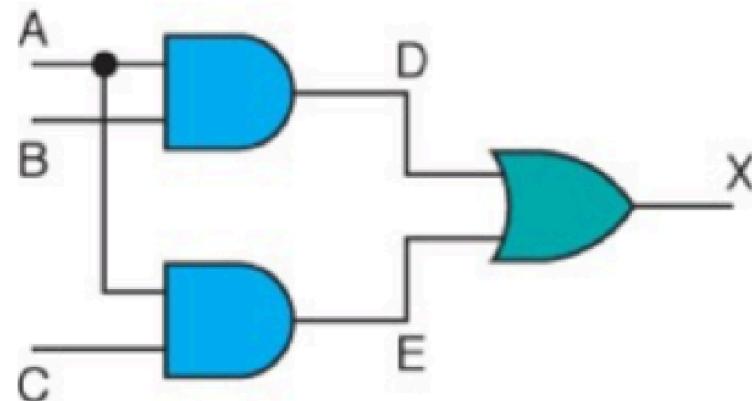


- A. Bool eq = ( a or b ) and ( !a or !b )
- B. Bool eq = ( a and b ) or ( !a and !b )
- C. Bool eq = ( a or !b ) and ( !a or b )
- D. Bool eq = ( a and !b ) or ( !a and b )

答案： B

# E11

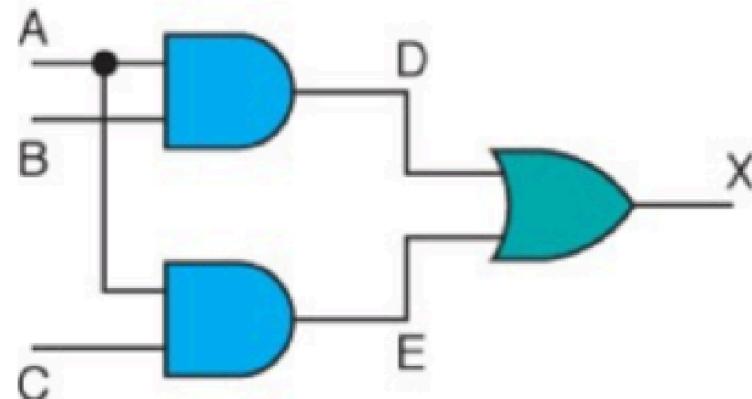
9、对应下述组合电路的正确 HCL 表达式为



- A. Bool X = (A || B) && (A || C)
- B. Bool X = A || (B && C)
- C. Bool X = A && (B || C)
- D. Bool X = A || B || C

**E11**

9、对应下述组合电路的正确 HCL 表达式为

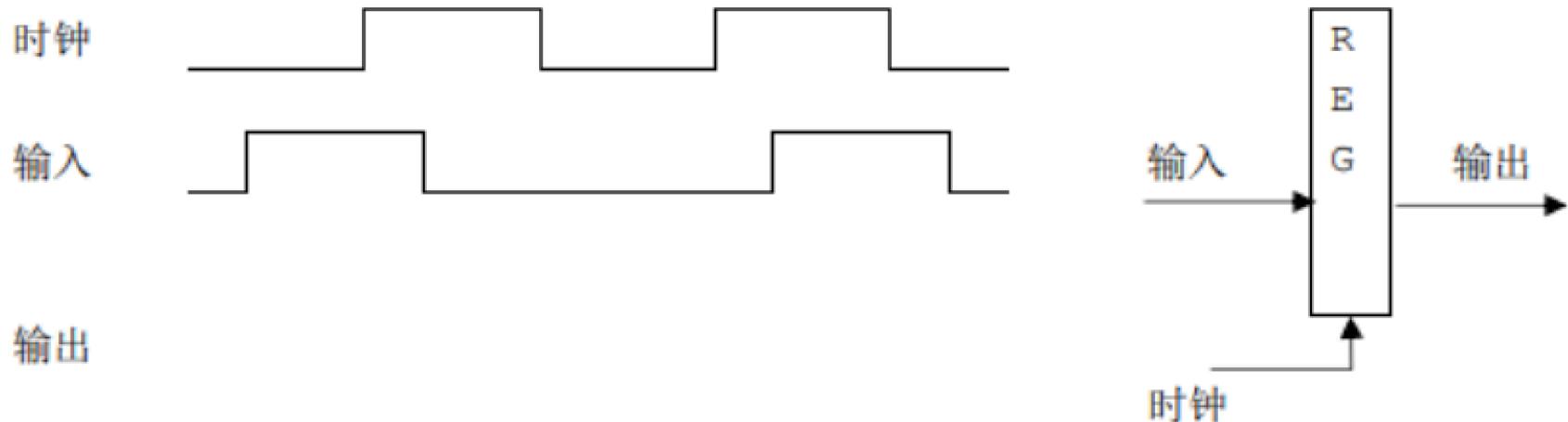


- A. Bool X = (A || B) && (A || C)
- B. Bool X = A || (B && C)
- C. Bool X = A && (B || C)
- D. Bool X = A || B || C

**答案: C**

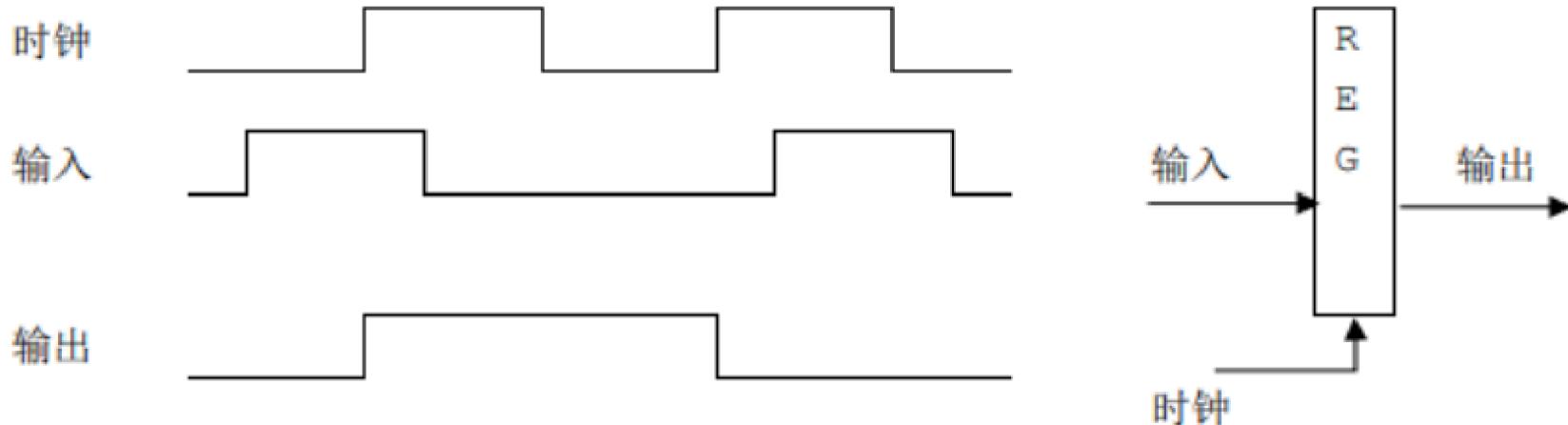
## E12

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）



## E12

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）



# E13

## 第二题 (20 分)

1. 在 64 位机器上，判断下列等式是否恒成立

```
/* random_int() 函数返回一个随机的 int 类型值 */
int x = random_int();
int y = random_int();
int z = random_int();
unsigned ux = (unsigned)x;
long lx = (long)x; /* long 为 64 位 */
long ly = (long)y;
double dx = (double)x;
double dy = (double)y;
double dz = (double)z;
```

Expression	Always True?
$(x \geq 0) \    \ (3*x < 0)$	Y N
$(x \geq 0) \    \ (x < ux)$	Y N
$((x \gg 1) \ll 1) \leq x$	Y N
$((x-y)<3) + (x>>1) - y == 8*x - 9*y + x/2$	Y N
$(x - y > 0) == ((y+x+1)>>31 == 1)$	Y N
$dx + dy == (double)(y+x)$	Y N
$dx + dy + dz == dz + dy + dx$	Y N
$(int)((lx+ly)>>1) == ((x&y) + ((x^y)>>1))$	Y N

2. 假设 C 语言中新定义了一种数据类型 T，该类型为 12-bit 长的浮点数，此浮点数遵循 IEEE 浮点数格式，其字段划分如下：

符号位 (s): 1-bit; 阶码字段 (exp): 6-bit; 小数字段 (frac): 5-bit。

- 1) 若将该格式下能表示的所有正规格数从小到大依次排列，则  
相邻两数之间差值的最小值为 \_\_\_\_\_，最大值为 \_\_\_\_\_

- 2) 现定义了如下变量：

```
T a = -15.875;
T b = (1<<28) + (1<<24) + (1<<22);
T c = a*b;
```

请写出各变量的二进制表示：

变量	二进制表示
a	
b	
c	

# E13

## 第二题 (20 分)

1. 在 64 位机器上，判断下列等式是否恒成立

```
/* random_int() 函数返回一个随机的 int 类型值 */
int x = random_int();
int y = random_int();
int z = random_int();
unsigned ux = (unsigned)x;
long lx = (long)x; /* long 为 64 位 */
long ly = (long)y;
double dx = (double)x;
double dy = (double)y;
double dz = (double)z;
```

Expression	Always True?
(x >= 0)    (3*x < 0)	Y N
(x >= 0)    (x < ux)	Y N
((x >> 1) << 1) <= x	Y N
((x-y)<3) + (x>>1) - y == 8*x - 9*y + x/2	Y N
(x - y > 0) == ((y+x+1)>>31 == 1)	Y N
dx + dy == (double)(y+x)	Y N
dx + dy + dz == dz + dy + dx	Y N
(int)((lx+ly)>>1) == ((x&y) + ((x^y)>>1))	Y N

2. 假设 C 语言中新定义了一种数据类型 T，该类型为 12-bit 长的浮点数，此浮点数遵循 IEEE 浮点数格式，其字段划分如下：

符号位(s): 1-bit; 阶码字段(exp): 6-bit; 小数字段(frac): 5-bit。

1) 若将该格式下能表示的所有正规格数从小到大依次排列，则

相邻两数之间差值的最小值为\_\_\_\_\_，最大值为\_\_\_\_\_

2) 现定义了如下变量：

```
T a = -15.875;
T b = (1<<28) + (1<<24) + (1<<22);
T c = a*b;
```

请写出各变量的二进制表示：

变量	二进制表示
a	
b	
c	

### 答案

N 考虑  $x = (1<<31)>>1$

N 考虑  $x = -1$

Y

N 考虑  $x = -1 \ y = 0$

N 考虑  $x-y = T_{min}$

N 考虑  $x+y$  溢出

Y 恒成立，每一个 int 型都可以由一个 double 型精确表示

Y 恒成立

### 答案：

1)

$2^{-35}$

$2^{26}$

2)

1 100011 00000 (发生进位)

0 111011 00010 (发生舍入)

1 111111 00000 (溢出,  $+\infty$ )

# E14

## 第三题 (20 分)

(1) 观察下面C语言函数和它相应的x86-64汇编代码

```
int foo(int x, int i)
{
    switch(i)
    {
        case 1:
            x -= 10;
        case 2:
            x *= 8;
            break;
        case 3:
            x += 5;
        case 5:
            x /= 2;
            break;
        case 0:
            x *= 1;
            default:
                x += i;
    }
    return x;
}

00000000004004a8 <foo>:
4004a8: mov %edi,%edx
4004aa: cmp $0x5,%esi
4004ad: ja 4004d4 <foo+0x2c>
4004af: mov %esi,%eax
4004b1: jmpq *0x400690(%rax,8)
4004b8: sub $0xa,%edx
4004bb: shr $0x3,%edx
4004be: jmp 4004d6 <foo+0xe>
4004c0: add $0x5,%edx
4004c3: mov %edx,%eax
4004c5: shr $0x1f,%eax
4004c8: lea (%rdx,%rax,1),%eax
4004cb: mov %eax,%edx
4004cd: sar %edx
4004cf: jmp 4004d6 <foo+0xe>
4004d1: and $0x1,%edx
4004d4: add %esi,%edx
4004d6: mov %edx,%eax
4004d8: retq
```

调用gdb命令`x/kg $rsp`将会检查从`rsp`中的地址开始的k个8字节字，请填写下面gdb命令的输出（每空一分）。

>(gdb) x/6g 0x400690

0x400690: 0x\_\_\_\_\_ 0x\_\_\_\_\_  
0x4006a0: 0x\_\_\_\_\_ 0x\_\_\_\_\_  
0x4006b0: 0x\_\_\_\_\_ 0x\_\_\_\_\_

(2) 右边的汇编代码是由左边程序中的`m`函数编译而成。回答如下问题。

<pre>typedef struct _list {     struct _list* next;     int value; } list;  int m(list* p) {     int r = 100;     while (①) {         r = ②;         p = ③;     }     if (p != 0) r = ④;     return r; }</pre>	<pre>m:     testq %rdi, %rdi     je .L6     movq (%rdi), %rdx     movl \$100, %eax     testq %rdx, %rdx     jne .L4     jmp .L3  .L14:     movq (%rdi), %rdx     testq %rdx, %rdx     je .L3     movl 8(%rdx), %r8d     ⑤ 8(%rdi), %r8d     ⑥ %r8d, %eax     movq (%rdi), %rdi     testq %rdi, %rdi     jne .L14     ret .L3:     ⑦ 8(%rdi), %eax     ret .L6:     movl \$100, %eax     ret</pre>
--	---

已知访存延迟 1 个指令周期。不访存的时候 `addl` 延迟 4 个指令周期, `imull` 延迟 8 个指令周期, 其他指令延迟 1 个指令周期。指令访存时延迟的执行周期为不访存的时候延迟的指令周期和访存延迟的周期之和。函数`m`中的循环在处理链表`p`的时候 CPE 为 4 个指令周期。⑤处的指令为`movl`, `addl`, `imull`中的一个。请填写①~⑦处的代码。

① \_\_\_\_\_

② \_\_\_\_\_

③ \_\_\_\_\_

④ \_\_\_\_\_

⑤ \_\_\_\_\_

# E14

答案：

(1) 每空 1 分

0x400690:	0x00000000004004d1	0x00000000004004b8
0x4006a0:	0x00000000004004bb	0x00000000004004c0
0x4006b0:	0x00000000004004d4	0x00000000004004c3

(2) ⑤空 2 分, 其他 3 分

①  $p \neq 0 \ \&\ p->next \neq 0$

②  $r * (p->value * p->next->value)$  注意：答案中运算可以交换，但不可结合

③  $p->next->next$

④  $r * p->value$

⑤ imull (因为每次处理两个元素, 所以关键路径的时间周期为 8。关键路径要么为两次访存和两次 mov, 要么为⑤处的运算。前者不可能达到 8, 所以只能是 imull。)

给分说明：如果②④的答案和⑤一致，但是⑤答错了就②④各给 2 分。

# E15

## 第三题 (15 分)

分析下面C语言程序和相应的x86-64汇编程序。其中缺失部分代码（被遮挡），请在对应的横线上填写缺失的内容。

```
#include <stdio.h>
#include "string.h"

void myprint(char *str)
{
    char buffer[16];
    [REDACTED](buffer,str);
    printf("%s \n",buffer);
}

void alert(void)
{
    printf("[REDACTED] \n");
}

int main(int argc,char *argv[])
{
    myprint("123456712345671234567\xaa\x84\x04\x08");
    return 0;
}
*****.section .rodata
.LCO:
    .string "[REDACTED]"
    .text
    .globl myprint
    .type myprint, @function
myprint:
.LFBO:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq %rsp, %rbp
    .cfi_def_cfa_register 6
    subq $48, %rbp
    movq %rdi, -40(%rbp)
    movq %fs:40, [REDACTED]
    movq %rax, -8(%rbp)
    xorl %eax, %eax
    movq %rax, %rdx
    leaq -32(%rbp), %rax
    movq %rdx, [REDACTED]
    [REDACTED]
    call strcpy
    [REDACTED] -32(%rbp), %rax
    movq %rax, %rsi
    movi $LCO, %edi
    movi $0, %eax
    call printf
```

```
nop [REDACTED], %rax
    [REDACTED]
    xorq [REDACTED]
    je [REDACTED]
    call _stack_chk_fail
.L2:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFBO:
    .size myprint, .-myprint
    .section .rodata
.LC1:
    .string "Where am I?"
    .text
    .globl alert
    .type alert, @function
alert:
.LFBI:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq %rsi, %rbp
    .cfi_def_cfa_register 6
    movl $.LC1, %edi
    call puts
    nop
    popq %rbp
    .cfi_def_cfa 7, 8
    [REDACTED]
    .cfi_endproc
.LFE1:
    .size alert, .-alert
    .section .rodata
    .align 8
.LC2:
    .string "1234567123456712345671234567\252[REDACTED]\004\b"
    .text
    .globl main
    .type main, @function
main:
.LFBS2:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq %rbp, %rbp
    .cfi_def_cfa_register 6
    subq $16, %rsp
    movl %edi, -4(%rbp)
    movq %rsi, -16(%rbp)
    movl $.LC2, %edi
    [REDACTED]
    movl $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE2:
    .size main, .-main
```

# E15

第三题 (15 分)

分析下面C语言程序和相应的x86-64汇编程序。其中缺失部分代码（被遮挡），请在对应的横线上填写缺失的内容。（注：每格1分）

```
#include <stdio.h>
#include "string.h"

void myprint(char *str)
{
    char buffer[16];
    [REDACTED] (buffer,str);          @strcpy _____
    printf("%s \n",buffer);
}

void alert(void)
{
    printf("[REDACTED] \n");          @Where am I?
}

int main(int argc,char *argv[])
{
    myprint("123456712345671234567\xaa\x84\x04\x08");
    return 0;
}
*****+.section .rodata
.LCO:
.string "[REDACTED]"          @%s \n _____
.text
.global myprint
.type myprint, @function
myprint:
.LFBO:
.cfi_startproc
pushq %rbp
.cfi_offset %rbp, -16
movq %rbp, %rbp
.cfi_def_cfa_register 6
subq $48, %rbp
movq %rdi, -40(%rbp)
movq %fs:40, [REDACTED]          @%rsp _____
movq %rax, -8(%rbp)
xorl %eax, %eax
movq [REDACTED], %rdx
leaq -32(%rbp), %rax
movq %rdx, [REDACTED]          @-40(%rbp) _____
call strcpy
[REDACTED] -32(%rbp), %rax
movq %rax, %rsi
movl $.LC0, %edi
movl $.S, %eax
call printf
```

```
nop [REDACTED], %rax
xord [REDACTED]
je [REDACTED] _stack_chk_fail
.L2:
Leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size myprint, .-myprint
.section .rodata
.LCI:
.string "Where am I?"
.text
.globl alert
.type alert, @function
alert:
.LFB1:
.cfi_startproc
pushq %rbp
.cfi_offset %rbp, -16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $.LC1, %edi
call puts
nop
popq %rbp
.cfi_def_cfa 7, 8
[REDACTED]          @ret _____
.cfi_endproc
.LFE1:
.size alert, .-alert
.section .rodata
.align 8
.LC2:
.string "123456712345671234567\x25[REDACTED]\x004\b"  @204 _____
.text
.globl main
.type main, @function
main:
.LFB2:
.cfi_startproc
pushq %rbp
.cfi_offset %rbp, -16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $16, %rbp
movl %edi, -4(%rbp)
movq %rsi, -16(%rbp)
movl $.LC2, %edi
[REDACTED]          @call myprint _____
movl $.S, %eax
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE2:
.size main, .-main
```

注：下面这种回答也对：

```
@movq %fs:40
@-8(%rbp), %rax
```

# Processor Arch: SEQ/PIPE

# E1

10. Y86 指令 `popl rA` 的 SEQ 实现如下图所示，其中①和②分别为：

Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{ra:rb} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \textcircled{1}$
Decode	$\text{valA} \leftarrow R[\%esp]$ $\text{valB} \leftarrow R[\%esp]$
Execute	$\text{valE} \leftarrow \textcircled{2}$
Memory	$\text{valM} \leftarrow M_4[\text{valA}]$
Write Back	$R[\%esp] \leftarrow \text{valE}$ $R[\text{ra}] \leftarrow \text{valM}$
PC Update	$\text{PC} \leftarrow \text{valP}$

- A)  $\text{PC} + 4 \quad \text{valA} + 4$
- B)  $\text{PC} + 4 \quad \text{valA} + (-4)$
- C)  $\text{PC} + 2 \quad \text{valB} + 4$
- D)  $\text{PC} + 2 \quad \text{valB} + (-4)$

# E1

答案: C

//知识点: popl 弹栈指令, 栈结构

- 1) popl 是双字节指令, 下一条指令位置 PC+2
- 2) Y86 是 32 位体系结构, popl 弹出 4 字节, 弹栈栈指针增加, valA + 4

## E2

10. 在 Y86 的 SEQ 实现中，对仅考虑 IRMMOVQ, ICALL, IPOPQ, IRET 指令，对 mem\_addr 的 HCL 描述正确的是：

```
word mem_addr = [
    icode in { (1), (2) } : valE;
    icode in { (3), (4) } : valA;
];
```

- A. (1) IRMMOVQ      (2) IPOPQ      (3) IRET      (4) ICALL
- B. (1) IRMMOVQ      (2) IRET      (3) IPOPQ      (4) ICALL
- C. (1) ICALL      (2) IPOPQ      (3) IRMMOVQ      (4) IRET
- D. (1) IRMMOVQ      (2) ICALL      (3) IPOPQ      (4) IRET

## E2

10. 在 Y86 的 SEQ 实现中，对仅考虑 IRMMOVQ, ICALL, IPOPQ, IRET 指令，对 mem\_addr 的 HCL 描述正确的是：

```
word mem_addr = [
    icode in { (1), (2) } : valE;
    icode in { (3), (4) } : valA;
];
```

- A. (1) IRMMOVQ      (2) IPOPQ      (3) IRET      (4) ICALL
- B. (1) IRMMOVQ      (2) IRET      (3) IPOPQ      (4) ICALL
- C. (1) ICALL      (2) IPOPQ      (3) IRMMOVQ      (4) IRET
- D. (1) IRMMOVQ      (2) ICALL      (3) IPOPQ      (4) IRET

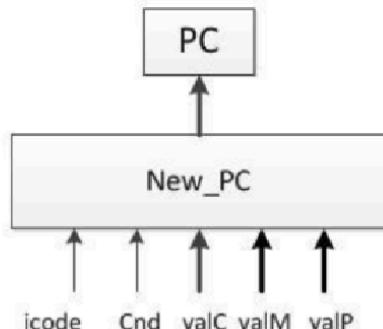
答案： D

# E3

14. 在 Y86 的 SEQ 实现中，PC（Program Counter，程序计数器）更新的逻辑

结构如下图所示，请根据 HCL 描述为①②③④选择正确的数据来源。

```
Int new_pc = [
    # Call.
    Icode = ICALL : ①;
    # Taken branch.
    Icode = IJXX && Cnd : ②;
    # Completion of RET instruction.
    Icode = IRET : ③;
    # Default.
    1 : ④;
```



其中：Icode 为指令类型，Cnd 为条件是否成立，valC 表示指令中的常数值，  
valM 表示来自返回栈的数据，valP 表示 PC 自增。

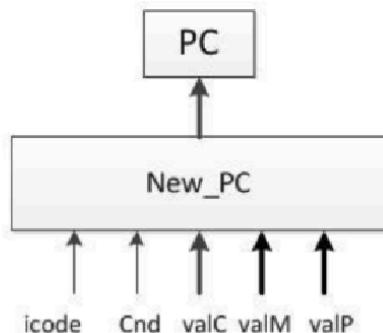
- A. valC, valM, valP, valP
- B. valC, valC, valP, valP
- C. valC, valC, valM, valP
- D. valM, valC, valC, valP

# E3

14. 在 Y86 的 SEQ 实现中，PC（Program Counter，程序计数器）更新的逻辑

结构如下图所示，请根据 HCL 描述为①②③④选择正确的数据来源。

```
Int new_pc = [
    # Call.
    Icode = ICALL : ①;
    # Taken branch.
    Icode = IJXX && Cnd : ②;
    # Completion of RET instruction.
    Icode = IRET : ③;
    # Default.
    1 : ④;
```



其中：Icode 为指令类型，Cnd 为条件是否成立，valC 表示指令中的常数值，  
valM 表示来自返回栈的数据，valP 表示 PC 自增。

- A. valC, valM, valP, valP
- B. valC, valC, valP, valP
- C. valC, valC, valM, valP
- D. valM, valC, valC, valP

答：C

## E4

11. 关于流水线技术的描述，错误的是：

- A. 流水线技术能够提高执行指令的吞吐率，但也同时增加单条指令的执行时间
- B. 增加流水线级数，不一定能获得总体性能的提升
- C. 指令间数据相关引发的数据冒险，不一定可以通过暂停流水线来解决。
- D. 流水级划分应尽量均衡，吞吐率会受到最慢的流水级影响，均衡的流水线能提高吞吐量。

## E4

11. 关于流水线技术的描述，错误的是：

- A. 流水线技术能够提高执行指令的吞吐率，但也同时增加单条指令的执行时间
- B. 增加流水线级数，不一定能获得总体性能的提升
- C. 指令间数据相关引发的数据冒险，不一定可以通过暂停流水线来解决。
- D. 流水级划分应尽量均衡，吞吐率会受到最慢的流水级影响，均衡的流水线能提高吞吐量。

**答案：C**

## E5

13. 关于流水线技术的描述，错误的是：

- A. 流水线技术能够提高执行指令的吞吐率，但也同时增加单条指令的执行时间。
- B. 减少流水线的级数，能够减少数据冒险发生的几率。
- C. 指令间数据相关引发的数据冒险，都可以通过 data forwarding 来解决。
- D. 现代处理器支持一个时钟内取指、执行多条指令，会增加控制冒险的开销。

## E5

13. 关于流水线技术的描述，错误的是：

- A. 流水线技术能够提高执行指令的吞吐率，但也同时增加单条指令的执行时间。
- B. 减少流水线的级数，能够减少数据冒险发生的几率。
- C. 指令间数据相关引发的数据冒险，都可以通过 data forwarding 来解决。
- D. 现代处理器支持一个时钟内取指、执行多条指令，会增加控制冒险的开销。

答：C，load-use 冒险 不能通过 data forwarding 来解决

## E6

12、关于流水线技术的描述，正确的是：

- A. 指令间数据相关引发的数据冒险，一定可以通过暂停流水线来解决。
- B. 流水线技术不仅能够提高执行指令的吞吐率，还能减少单条指令的执行时间。
- C. 增加流水线的级数，一定能获得性能上的提升。
- D. 流水级划分应尽量均衡，不均衡的流水线会增加控制冒险。

答：( )

## E6

12、关于流水线技术的描述，正确的是：

- A. 指令间数据相关引发的数据冒险，一定可以通过暂停流水线来解决。
- B. 流水线技术不仅能够提高执行指令的吞吐率，还能减少单条指令的执行时间。
- C. 增加流水线的级数，一定能获得性能上的提升。
- D. 流水级划分应尽量均衡，不均衡的流水线会增加控制冒险。

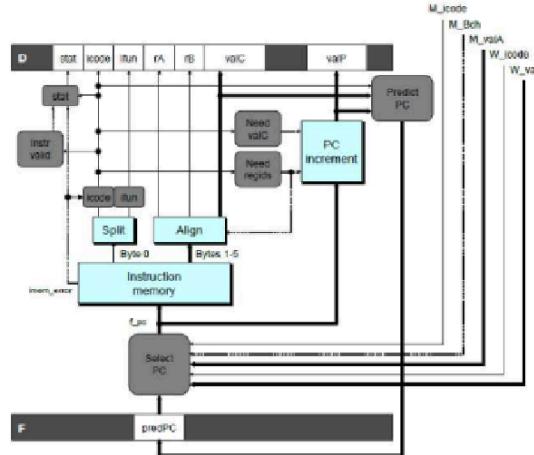
答：( )

**答案：A**

**说明：B 会增加单条指令的执行时间，C 可能会降低性能，D 和控制冒险没有关系**

# E7

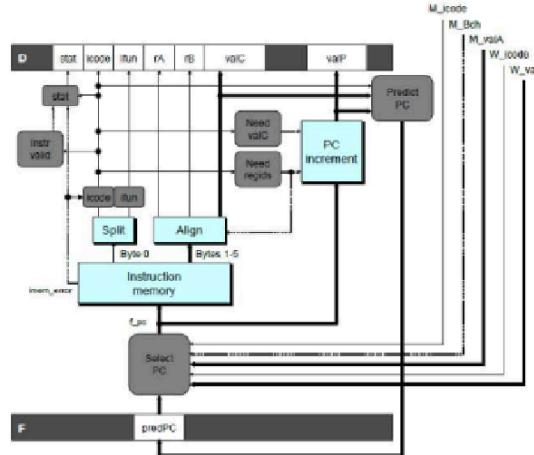
11. 流水线数据通路中的转移预测策略为总是预测跳转。如果转移预测错误，需要恢复流水线，并从正确的目标地址开始取值。其中，用来判断转移预测是否正确的信号是\_①\_和\_②\_，用来获得正确的目标地址的信号是\_③\_。



- A. ① M\_icode ② M\_Bch ③ M\_valA
- B. ① W\_icode ② M\_Bch ③ M\_valA
- C. ① W\_icode ② M\_Bch ③ W\_valM
- D. ① M\_icode ② M\_Bch ③ W\_valM

# E7

11. 流水线数据通路中的转移预测策略为总是预测跳转。如果转移预测错误，需要恢复流水线，并从正确的目标地址开始取值。其中，用来判断转移预测是否正确的信号是\_①\_和\_②\_，用来获得正确的目标地址的信号是\_③\_。



- A. ① M\_icode ② M\_Bch ③ M\_valA
- B. ① W\_icode ② M\_Bch ③ M\_valA
- C. ① W\_icode ② M\_Bch ③ W\_valM
- D. ① M\_icode ② M\_Bch ③ W\_valM

# E8

## 第二题 (20 分)

1. 在 64 位机器上，判断下列等式是否恒成立

```
/* random_int() 函数返回一个随机的 int 类型值 */
int x = random_int();
int y = random_int();
int z = random_int();
unsigned ux = (unsigned)x;
long lx = (long)x; /* long 为 64 位 */
long ly = (long)y;
double dx = (double)x;
double dy = (double)y;
double dz = (double)z;
```

Expression	Always True?
$(x \geq 0) \    \ (3*x < 0)$	Y N
$(x \geq 0) \    \ (x < ux)$	Y N
$((x \gg 1) \ll 1) \leq x$	Y N
$((x-y)<3) + (x>>1) - y == 8*x - 9*y + x/2$	Y N
$(x - y > 0) == ((y+x+1)>>31 == 1)$	Y N
$dx + dy == (double)(y+x)$	Y N
$dx + dy + dz == dz + dy + dx$	Y N
$(int)((lx+ly)>>1) == ((x&y) + ((x^y)>>1))$	Y N

2. 假设 C 语言中新定义了一种数据类型 T，该类型为 12-bit 长的浮点数，此浮点数遵循 IEEE 浮点数格式，其字段划分如下：

符号位 (s): 1-bit; 阶码字段 (exp): 6-bit; 小数字段 (frac): 5-bit。

- 1) 若将该格式下能表示的所有正规格数从小到大依次排列，则  
相邻两数之间差值的最小值为 \_\_\_\_\_，最大值为 \_\_\_\_\_

- 2) 现定义了如下变量：

```
T a = -15.875;
T b = (1<<28) + (1<<24) + (1<<22);
T c = a*b;
```

请写出各变量的二进制表示：

变量	二进制表示
a	
b	
c	

# E8

## 第二题 (20 分)

1. 在 64 位机器上，判断下列等式是否恒成立

```
/* random_int() 函数返回一个随机的 int 类型值 */
int x = random_int();
int y = random_int();
int z = random_int();
unsigned ux = (unsigned)x;
long lx = (long)x; /* long 为 64 位 */
long ly = (long)y;
double dx = (double)x;
double dy = (double)y;
double dz = (double)z;
```

Expression	Always True?
$(x \geq 0) \    \ (3*x < 0)$	Y N
$(x \geq 0) \    \ (x < ux)$	Y N
$((x \gg 1) \ll 1) \leq x$	Y N
$((x-y)<3) + (x>>1) - y == 8*x - 9*y + x/2$	Y N
$(x - y > 0) == ((y+x+1)>>31 == 1)$	Y N
$dx + dy == (double)(y+x)$	Y N
$dx + dy + dz == dz + dy + dx$	Y N
$(int)((lx+ly)>>1) == ((x&y) + ((x^y)>>1))$	Y N

2. 假设 C 语言中新定义了一种数据类型 T，该类型为 12-bit 长的浮点数，此浮点数遵循 IEEE 浮点数格式，其字段划分如下：

符号位 (s): 1-bit; 阶码字段 (exp): 6-bit; 小数字段 (frac): 5-bit。

1) 若将该格式下能表示的所有正规格数从小到大依次排列，则

相邻两数之间差值的最小值为 \_\_\_\_\_，最大值为 \_\_\_\_\_

2) 现定义了如下变量：

```
T a = -15.875;
T b = (1<<28) + (1<<24) + (1<<22);
T c = a*b;
```

请写出各变量的二进制表示：

变量	二进制表示
a	
b	
c	

答案： A, B, B, B

答案: A,A,A,B

答案: A,A,B,C

## E9

12. 若处理器实现了三级流水线，每一级流水线实际需要的运行时间为 1ns、2ns 和 3ns，则此处理器不停顿地执行完毕 10 条指令需要的时间为：  
A. 21 ns   B. 12 ns   C. 24 ns   D. 36 ns

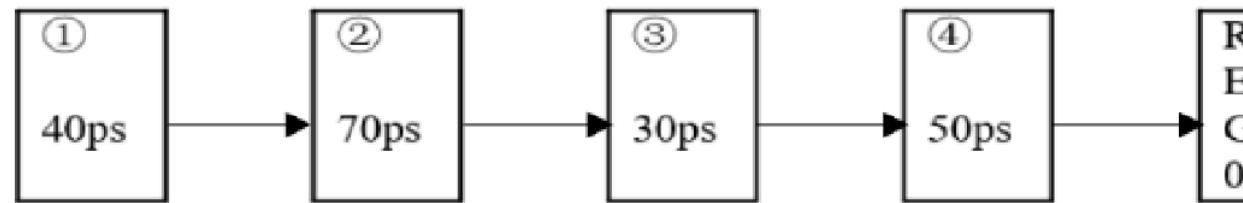
## E9

12. 若处理器实现了三级流水线，每一级流水线实际需要的运行时间为 1ns、2ns 和 3ns，则此处理器不停顿地执行完毕 10 条指令需要的时间为：  
A. 21 ns   B. 12 ns   C. 24 ns   D. 36 ns

答案 D  $3+3+10*3=36\text{ns}$

# E10

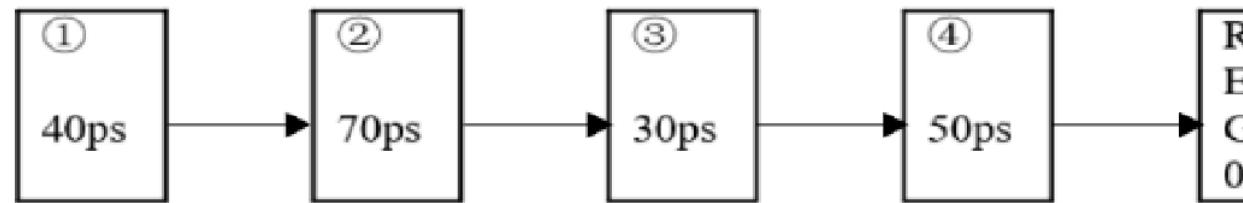
11. 如下图所示，①~④为四个组合逻辑单元，对应的延迟已在图上标出，REG0 为一寄存器，延迟为 20ps。通过插入额外的 2 个流水线寄存器 REG1、REG2（延迟均为 20ps），可以对其进行流水化改造。改造后的流水线的吞吐率最大为 \_\_\_\_\_ GIPS。



- A. 7.69      B. 8.33      C. 10.00      D. 11.11

# E10

11. 如下图所示，①~④为四个组合逻辑单元，对应的延迟已在图上标出，REG0 为一寄存器，延迟为 20ps。通过插入额外的 2 个流水线寄存器 REG1、REG2（延迟均为 20ps），可以对其进行流水化改造。改造后的流水线的吞吐率最大为 \_\_\_\_\_ GIPS。



- A. 7.69      B. 8.33      C. 10.00      D. 11.11

**【答】C。** 额外的 2 个寄存器应当插入①②之间、②③之间，最慢的一级延迟为  $30+50+20=100\text{ps}$ ，吞吐率为  $1000/100=10\text{GIPS}$

# E11

12. 一个功能模块包含组合逻辑和寄存器，组合逻辑单元的总延迟是 100ps，单个寄存器的延时是 20ps，该功能模块执行一次并保存执行结果，理论上能达到的最短延时和最大吞吐分别是多少？
- A. 20ns, 50GIPS
  - B. 120ns, 50GIPS
  - C. 120ns, 10GIPS
  - D. 20ps, 10GIPS

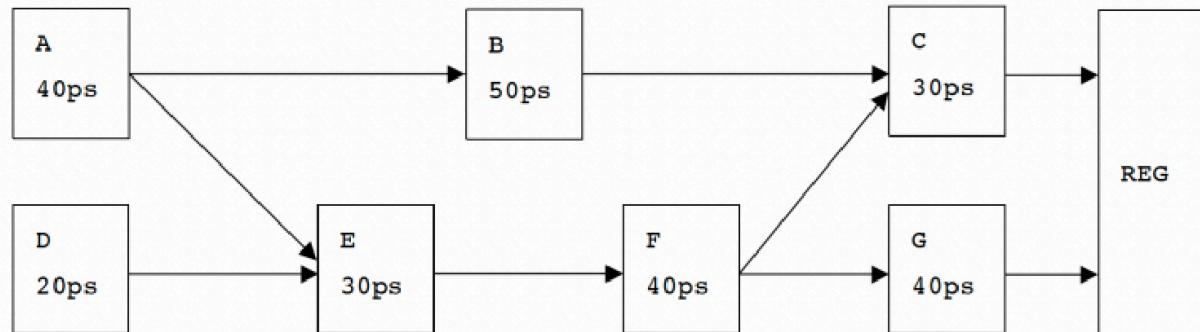
# E11

12. 一个功能模块包含组合逻辑和寄存器，组合逻辑单元的总延迟是 100ps，单个寄存器的延时是 20ps，该功能模块执行一次并保存执行结果，理论上能达到的最短延时和最大吞吐分别是多少？
- A. 20ns, 50GIPS
  - B. 120ns, 50GIPS
  - C. 120ns, 10GIPS
  - D. 20ps, 10GIPS

答案：B，主要考察延时和吞吐的区别，延时最小是 logic + 1 个 register，吞吐最高时是把 logic 划分成无穷多份，每一级需要 20ps， $1000/20ps = 50GIPS$

# E12

8. A-G 为 7 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所有的依赖关系。寄存器的延迟均为 20ps，在图中以 REG 符号表示。假设流水线寄存器只能添加在有直接依赖关系的基本逻辑单元之间，而不能在 C 或 G 与 REG 之间。以下说法正确的是：



- A. 原电路的吞吐量 (throughput) 舍入后大约是  $1000/150=6.667$  GIPS。
- B. 将该电路改造成 2 级流水线有 8 种方法
- C. 如果将该电路改造成 3 级流水线，延迟最小可以到 80 ps。
- D. 不论实现该电路时遇到怎样的数据冒险和控制冒险，一定可以对流水线寄存器使用暂停 (stalling) 解决。

## E12

- A. 错。 $1000/170 = 5.882$  而非  $1000/150 = 6.667$ 。不要漏掉寄存器。
- B. 错。考虑 D-E-F-G 这条路，要么插 D-E，要么 E-F，要么 F-G。如果是 D-E，那么为了使每条极大路径上都恰有一个新插入的寄存器，考虑 D-E-F-C，那么 F-C 之间也不能插入了。但是 A-E-F-G 上又必须有一个，所以只能插 A-E。这样之后不管是插在 A-B 还是 B-C 所得方案都是合法的。对称地，如果是 F-G，结果也是如此。最后考虑 E-F 的情况，此时 A-E，D-E，F-C，F-G 之间均不能再插入，但 A-B 和 B-C 任选一个插入得到的方案都合法。因此一共有  $2+2+2=6$  种。
- C. 错。3 阶段里的最优为 90ps。
- D. 对。最朴素的情况每条指令 stall 足够多次，电路回到 SEQ 的状态。

# E13

## 第四题 (15 分)

请分析 Y86-64 ISA 中加入的一族间接跳转指令: `jxx *rB`, 其格式如下:

C	Fn	F	rB
---	----	---	----

该指令的功能是跳转到寄存器 `R[rB]` 所存放的地址。类似于直接跳转指令，间接跳转指令也包括无条件跳转和条件跳转，通过不同的功能码 Fn 来指示。为了和直接跳转区别，icode 为 IJREGXX。时钟周期适当进行延长，在不修改原有的硬件线路和信号设置的前提下，只增加和新指令有关的逻辑，回答以下问题。

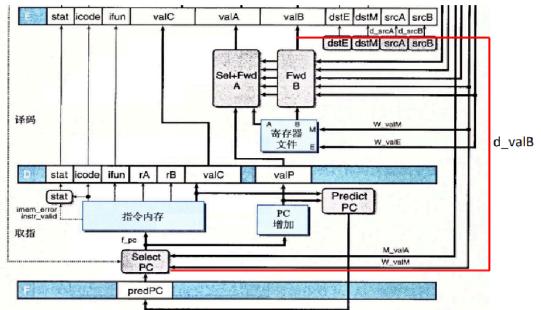
1. 在教材中的 SEQ 处理器上实现该指令，请补全下表中每个阶段的操作。需要说明的信号可能有: icode, ifun, rA, rB, valA, valB, valC, valE, valP, Cnd, R[], M[], PC, CC

Stage	<code>jxx *rB</code>
Fetch	<code>icode : ifun ← M1[PC]</code>
Decode	
Execute	
Memory	
Write Back	
Update PC	<code>PC ← Cnd ? valE : valP</code>

在 foo 函数运行过程中，计算以下情况 foo 函数的执行周期数。（周期数计算从执行 foo 第一条指令开始，直到其返回指令 ret 完全通过流水线为止。另外假设 foo 函数开始的若干条指令不会和 foo 函数体外的指令形成冒险。）

`n=-1: _____; n=0: _____; n=2: _____`

2. 考虑在教材中的 Pipeline 处理器上实现该指令，采用总是选择分支（预测下一条指令时使用跳转地址 `R[rB]`）的预测策略。



由于 `R[rB]` 需要到译码阶段才能得到，需要增加一条从 `Fwd B` 输出信号 `d_valB` 到 `Select PC` 的旁路通路，增加线路如图所示。增加旁路后，为了预测 `jxx *rB` 的下一条 PC，\_\_\_\_\_（需要/不需要）在该指令和下一条指令间插入气泡。

`Select PC` 的 HCL 代码如下图所示：

```
word f_pc = [
    ①
    (M_icode == IJXX || M_icode == IJREGXX) && !M_cnd : M_valA;
    ②
    W_icode == IRET : W_valM;
    ③
    1 : F_predPC;
    ④
]
```

为了预测下一条 PC，需要修改 `Select PC` 的 HCL 代码，增加一行 \_\_\_\_\_，增加的位置可以是 \_\_\_\_\_（写出所有可能的位置，错填不得分，漏填可得部分分）

3. 请将该指令预测错误的触发条件，以及此时流水线的控制逻辑补充完整。

触发条件：（如果有多种可能请任写一种）

== IJREGXX &&

控制逻辑：（如果有多种可能请任写一种）

F	D	E	M	W

4. 基于改造后的 Y86-64 PIPE 考虑如下代码片段，回答问题。

```
# Array of 3 elements
array:
.quad return
.quad L1
.quad L2

# void foo(long n, long *arr)
# n in %rdi, arr in %rsi
foo:
    rmovq %rdi, %rdx          # line 1
    addq %rdx, %rdx            # line 2
    addq %rdx, %rdx            # line 3
    addq %rdx, %rdx            # line 4
    irmovq array, %rcx         # line 5
    addq %rcx, %rcx             # line 6
    andq %rdi, %rdi            # line 7
    jge *%rcx                  # line 8
    return:                   #
    ret                         # line 9
L2: #
    rmovq 16(%rsi), %rcx      # line 10
    rmovq %rcx, 8(%rsi)        # line 11
L1: #
    rmovq 8(%rsi), %rcx        # line 12
    rmovq %rcx, (%rsi)         # line 13
    jmp return                 # line 14
```

# E13

1.

Stage	jxx *rB
Fetch	icode : ifun $\leftarrow M_1[PC]$ xA : rB $\leftarrow M_1[PC+1]$ valP $\leftarrow PC+2$
Decode	valB $\leftarrow R[rB]$
Execute	valE $\leftarrow valB + 0$ Cnd = Cond(CC, ifun)
Memory	/
Write Back	/
Update PC	PC $\leftarrow Cnd ? valE : valP$

2. 不需要。间接跳转在 decode 阶段时 SelectPC 正好利用最新转发的 valB 的值作为预测地址访问指令内存。

1 2 3 处都可以插入。插入的内容见下。④是无效的插入位置。注意，原来的 M\_XXX 条件和 W\_XXX 条件互斥，所以其顺序任意，但它们都不会和新加入的条件冲突。例如 mispredict 发生时，Decode 阶段的指令已经被清空，所以最终不会导致多个条件成立。

```
word f_pc = [
    // D_icode == IREGXX: d_valB;
    (M_icode == IJXX || M_icode == IREGXX) && !M_cnd : M_valA;
    // D_icode == IREGXX: d_valB;
    W_icode == IRET : W_valM;
    D_icode == IREGXX: d_valB;
    1 : F_predPC;
    ④
]
```

3. E\_icode == IJREGXX && !e\_Cnd

F	D	E	M	W
normal/stall	bubble	bubble	normal	normal

分析方法同 IJXX。注意，触发条件是在 E 阶段，因为 E 阶段结束、M 阶段开始时就要控制流水线寄存器。

4. 15 13 20

n == -1: line 8 分支预测错误，惩罚 2 个周期。一共 9 + 2 + 4 (trailing cycles for ret) = 15。  
n == 0: 一共 9 + 4 (trailing cycles for ret) = 13。  
n == 1: line 12 和 13 发生 load/use hazard (不考虑加载转发)，惩罚 1 个周期。一共 12 + 1 + 4 (trailing cycles for ret) = 17。  
n == 2: line 12 和 line 13, 以及 line 10 和 line 11 各发生一次 load/use hazard, 共惩罚 2 个周期。一共 14 + 2 + 4 (trailing cycles for ret) = 20。

# E14

【综合】

20. 以下提供了一段代码的 C 语言、汇编语言以及运行到某一时刻栈的情况

汇编:

```
000000000400596 <func>:
400596: sub    $0x28,%rsp
400598: mov    %fs:0x28,%rax
4005a3: mov    %rax,0x18(%rsp)
4005a8: xor    %eax,%eax
4005aa: mov    (%rdi),%rax
4005ad: mov    0x8(%rdi),%rdx
4005b1: cmp    %rdx,%rax
4005b4: jge    .(1)
4005b6: mov    %rdx,(%rdi)
4005b9: mov    %rax,0x8(%rdi)
4005bd: mov    0x8(%rdi),%rax
4005c1: test   %rax,%rax
4005c4: jne    4005cb <func+0x35>
4005c6: mov    (%rdi),%rax
4005c9: jmp    .(2)
4005cb: mov    (%rdi),%rdx
4005ce: sub    %rax,%rdx
4005d1: mov    %rdx,(%rsp)
4005d5: mov    %rax,0x8(%rsp)
4005da: mov    .(3),%rdi
4005dd: callq  400596 <func>
4005e2: mov    0x18(%rsp),%rcx
4005e7: xor    .(4),%rcx
4005f0: .(5) 4005f7 <func+0x61>
4005f2: callq  400460 <_stack_chk_fail@plt>
4005f7: add    .(6),%rsp
4005fb: retq

0000000004005fc <main>:
4005fc: sub    $0x28,%rsp
400600: mov    %fs:0x28,%rax
400609: mov    %rax,0x18(%rsp)
40060e: xor    %eax,%eax
400610: movq   0x69,%rsp
400618: movq   0xfc,0x8(%rsp)
400621: mov    %rsp,%rdi
400624: callq  400596 <func>
400629: mov    %rax,%rsi
```

```
40062c: mov    $0x4006e4,%edi
400631: mov    $0x0,%eax
400636: callq  400470 <printf@plt>
40063b: mov    0x18(%rsp),%rdx
400640: xor    .(4),%rdx
400649: .(5) 400650 <main+0x54>
40064b: callq  400460 <_stack_chk_fail@plt>
400650: mov    $0x0,%eax
400655: add    .(6),%rsp
400659: retq
```

C 语言与堆栈:

```
typedef struct{
    long a;
    long b;
} pair_type;
long func(pair_type *p) {
    if (p -> p < p -> b) {
        long temp = p -> a;
        p -> a = p -> b;
        p -> b = temp;
    }
    if ((7) ) {
        return p -> a;
    }
    pair_type np;
    np.a = .(8);
    np.b = .(9);
    return func(&np);
}
int main(int argc, char* argv[]) {
    pair_type np;
    np.a = .(10);
    np.b = .(11);
    printf("%ld", func(&np));
    return 0;
}
```

.....
0x0000000000000000
0xc76d5add7bbeaa00
0x00007fffffdff60
(a)
(b)
0x0000000000400629
(c)
(d)
0x0000000000000001
0x0000000000000069
0x0000000000000093
(e)
(f)
0x00000000ff000000
(g)
(h)
(i)
(j)
(k)
0x0000000000000000
(栈顶) [低地址]

一些可能用到的字符的 ASCII 码表:

换行	空格	"	%	(	)	,	0	A	a
0x0a	0x20	0x22	0x25	0x28	0x29	0x2c	0x30	0x41	0x61

回答问题:

I. gdb 下使用命令 x/4b 0x4006e4 后（即查看 0x4006e4 开始的 4 个字节，用 16 进制表示）得到的输出结果是

0x4006e4: 0x\_\_\_\_\_ 0x\_\_\_\_\_ 0x\_\_\_\_\_ 0x\_\_\_\_\_

II. 互相翻译 C 语言代码和汇编代码，补充缺失的空格（标号相同的为同一格）。

(1) \_\_\_\_\_ <func+\_\_\_\_\_ >

(2) \_\_\_\_\_ <func+\_\_\_\_\_ >

(3) \_\_\_\_\_

(4) \_\_\_\_\_

(5) \_\_\_\_\_

(6) \_\_\_\_\_

(7) \_\_\_\_\_

(8) \_\_\_\_\_

(9) \_\_\_\_\_

(10) \_\_\_\_\_

(11) \_\_\_\_\_

III. 补充栈的内容。使用 16 进制，可以不写前导多余的 0；对于给定已知条件后仍无法确定的值，填写“不确定”；已知程序运行过程中寄存器%rs 的值没有改变。

(a) \_\_\_\_\_

(b) \_\_\_\_\_

(c) \_\_\_\_\_

(d) \_\_\_\_\_

(e) \_\_\_\_\_

(f) \_\_\_\_\_

(g) \_\_\_\_\_

(h) \_\_\_\_\_

(i) \_\_\_\_\_

(j) \_\_\_\_\_

(k) \_\_\_\_\_

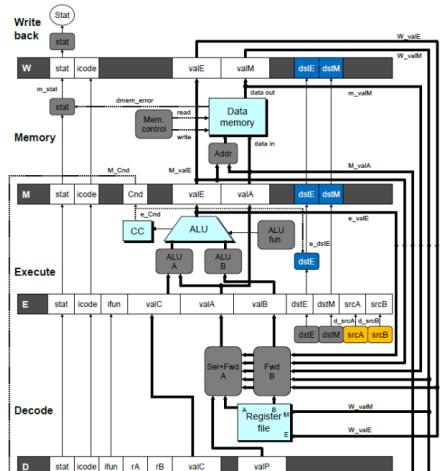
IV. 程序运行结果为\_\_\_\_\_。

# E14

得分

第四题 (15 分)

这是一款 Y86-32 流水线处理器的结构图 (局部), 请以此为基础, 依次回答下列问题。



1、该处理器设计采用了前递 (forwarding) 技术, 一定程度上解决了数据相关的问题, 在上图中体现在 Sel+FwdA 和 FwdB 部件上。前者输出的信号会存到流水线寄存器 E 的 valA 域 (即 E\_valA 信号), 请补全该信号的 HCL 语言描述。

```
int E_valA = [
    D_icode in { ICALL, IJXX } : _____ ; # ① 答案: D_valP
    d_srcA == e_dstE : _____ ;# ② 答案: e_valE
    d_srcA == M_dstM : _____ ;# ③ 答案: m_valM
```

```
d_srcA == M_dstE : M_valE
d_srcA == W_dstM : W_valM
...
];
```

2、如果在该处理器上运行下面的程序, 每条指令在不同时钟周期所处的流水线阶段如下表所示, 在这种情况下, 哪条指令的执行结果会有错误? 写出该指令的地址:

0x010 - (1 分)

	1	2	3	4	5	6	7	8	9	10	11	12
F	D	E	M	W								
F	D	E	M	W								
	F	D	E	M	W							
F	D	E	M	W								
F	D	E	M	W								
	F	D	E	M	W							
	F	D	E	M	W							
	F	D	E	M	W							
	F	D	E	M	W							
	F	D	E	M	W							

3、如需检测出这个情况, 需要增加逻辑电路, 用 HCL 语言表达如下:

E\_icode in {IMMOVL, IPOPL} && \_\_\_\_\_ in { \_\_\_\_\_ }

答案: E\_icode in {IMMOVL, IPOPL} && E\_dstM in { d\_srcA, d\_srcB }, 2 分,  
全对才得分

4、当新增的电路检测出这个情况后, 应对各流水线寄存器进行不同的设置, 以便在尽可能少影响性能的前提下解决该问题。请填写下表, 可选的设置包括 normal/bubble/stall 三种。

F	D	E	M	W

答案: stall, stall, bubble, normal, normal. 3 分, 全对才得分

5、如果遇到下面程序代码所示的情况, 该处理器运行时仍然存在问题。因此, 还需要新增检测电路。当新增的电路检测出这个情况后, 应对各流水线寄存器进行不同的设置, 以便在尽可能少影响性能的前提下解决该问题。请填写下表, 可选的设置包括 normal/bubble/stall 三种。

```
demo2, ys
...
0x018: rmovl %ecx, 0(%edx)
0x01e: irmovl $10, %ebx
0x024: popl %esp
0x026: ret
```

F	D	E	M	W

答案: stall, stall, bubble, normal, normal. 3 分, 全对才得分

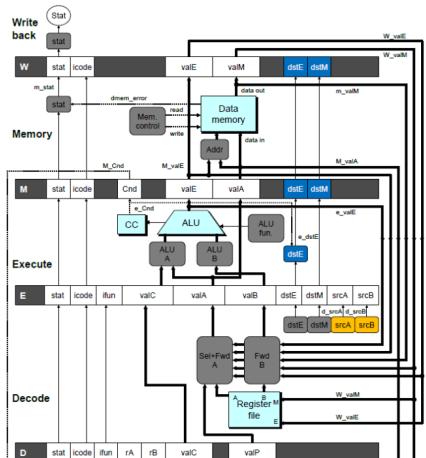
# E15

得分	
----	--

## 【2018年期中】

第四题 (15分)

这是一款 Y86-32 流水线处理器的结构图 (局部), 请以此为基础, 依次回答下列问题。



1. 该处理器设计采用了前送 (Forwarding) 技术, 一定程度上解决了数据相关的问题, 在上图中体现在 Sel+FwdA 和 FwdB 部件上。前者输出的信号会存到流水线寄存器 E 的 valA 域 (即 E\_valA 信号), 请补全该信号的 HCL 语言描述。

int E\_valA = [

    D\_code in { ICALL, IJXX } : \_\_\_\_\_ ; # ①

    d\_srcA == e\_dstE : \_\_\_\_\_ ;# ②

    d\_srcA == M\_dstM : \_\_\_\_\_ ;# ③

```
d_srcA == M_dstE : M_valE      ;
d_srcA == W_dstM : W_valM      ;
...
```

];

2. 如果在该处理器上运行下面的程序, 每条指令在不同时钟周期所处的流水线阶段如下表所示。在这种情况下, 哪条指令的执行结果会有错误? 写出该指令的地址: \_\_\_\_\_。

```
demo1.lys
0x000: irmovl $128, %edx
0x006: irmovl $3, %ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10, %ebx
0x018: irmovl 0(%edx), %eax
0x01e: addl %ebx, %eax
0x020: halt
```

I	2	3	4	5	6	7	8	9	10	11	12
F	D	E	M	W							
F	D	E	M	W							
F	D	E	M	W							
F	D	E	M	W							
F	D	E	M	W							

3. 如需检测出这个情况, 需要增加逻辑电路, 用 HCL 语言表达如下:  
E\_icode in { IMRMVOL, IPOPOL } && \_\_\_\_\_ in { \_\_\_\_\_ }

4. 当新增的电路检测出这个情况后, 应对各流水线寄存器进行不同的设置, 以便在尽可能少影响性能的前提下解决该问题。请填写下表, 可选的设置包括 normal/bubble/stall 三种。

F	D	E	M	W

5. 如果遇到下面程序代码所展示的情况, 该处理器运行时仍然存在问题。因此, 还需要新增检测电路。当新增的电路检测出这个情况后, 应对各流水线寄存器进行不同的设置, 以便在尽可能少影响性能的前提下解决该问题。请填写下表, 可选的设置包括 normal/bubble/stall 三种。

demo2.lys

```
...
0x018: rmmovl %ecx, 0(%edx)
0x01e: irmovl $10, %ebx
0x024: popl %esp
0x026: ret
```

F	D	E	M	W

4.57 在我们的 PIPE 设计中, 只要一条指令执行了 Load 操作, 从内存读一个值到寄存器, 并且下一条指令要用这个寄存器作为源操作数, 就会产生一个暂停。如果要在执行阶段中使用这个源操作数, 暂停是不可避免的唯一方法。对于第二条指令将源操作数存储到内存的情况, 例如 rmovq 或 pushq 指令, 是不需要这样的暂停的。考虑下面这段代码示例:

```
1    rmovq %rcx,%rdx    # Load 1
2    pushq %rdx           # Store 1
3    nop
4    popq %rdx            # Load 2
5    rmovq %rax,0(%rdx)   # Store 2
```

在第 1 行和第 2 行, rmovq 指令从内存读一个值到 %rdx, 然后 pushq 指令将这个值压入栈中。我们的 PIPE 设计会让 pushq 指令暂停, 以避免加载/使用冒险。不过, 可以看到, pushq 指令要到访存阶段才会需要 %rdx 的值。我们可以再添加一条旁路通路, 如图 4-70 所示, 将内存输出 (信号 m\_valM) 转发到流水线寄存器 M 中的 valA 字段。在下一个时钟周期, 被传送的值就能写入内存了。这种技术称为加载转发 (load forwarding)。

注意, 上述代码序列中的第二个例子 (第 4 行和第 5 行) 不能利用加载转发, popq 指令加载的值是作为下一条指令地址计算的一部分的, 而在执行阶段而非访存阶段就需要这个值了。A. 写出描述发现加载/使用冒险条件的逻辑公式, 类似于图 4-64 所示, 除了能用加载转发时不会导致暂停以外。

B. 文件 pipe1.hcl 包含一个 PIPE 控制逻辑的修改版。它含有信号 e\_valA 的定义, 用来实现图 4-70 中标号为 “Fwd A” 的块, 它还将流水线控制逻辑中的加载/使用冒险的条件设置为 0, 因此流水线控制逻辑将不会发现任何形式的加载/使用冒险。修改这个 HCL 描述以实现加载转发。可以参考实验资料获得如何为你的解答生成模拟器以及如何测试模拟器的指导。

# E15

回答问题：

1. gdb 下使用命令 `x/4b 0x4006e4` 后（即查看 `0x4006e4` 开始的 4 个字节，用 16 进制表示）得到的输出结果是？

`0x4006e4: 0x25 0x6c 0x64 0x00`

2. 互相翻译 C 语言代码和汇编代码，补充缺失的空格（标号相同的为同一格）。

- (1) `4005bd <func+0x27>`
- (2) `4005e2 <func+0x4c>`
- (3) `%rsp`
- (4) `%fs:0x28`
- (5) `je`
- (6) `$0x28`
- (7) `p -> b == 0`
- (8) `p -> a - p -> b`
- (9) `p -> b`
- (10) `105`
- (11) `252`

3. 补充栈的内容。使用 16 进制，可以不写前导多余的 0；对于给定已知条件后仍无法确定的值，填写“不确定”；已知程序运行过程中寄存器`%fs` 的值没有改变。

- (a) `0x0000000000000009`
- (b) `0x00000000000000fc`
- (c) `不确定`
- (d) `0xc76d5add7bbeaa00`
- (e) `0x00000000004005e2`
- (f) `0xc76d5add7bbeaa00`
- (g) `0x000000000000002a`
- (h) `0x0000000000000069`
- (i) `0x00000000004005e2`
- (j) `0xc76d5add7bbeaa00`
- (k) `不确定`

4. 程序输出结果是？

21

**【注意】**

- 1 最后一空为 `0x00`，因为要用`\0`作字符串结尾
- 2 (1) (2) 较难
- 2 (10) (11) 不能颠倒，因为汇编如此写
- 3 (a) (b) 两空不能颠倒，因为 func 函数会修改内存的值  
程序的作用是用（类似于）更相减损术求最大公约数

# E16

得分

## 【2022年期中】

第四题(15分) 请分析 Y86-64 ISA 中新加入的一组条件 POP 指令: cpopqXX。 cpopqXX 和 popq 指令的机器码格式如下。

cpopqXX	B	fun	rA	F
popq	B	0	rA	F

类似cmovXX，该组cpopqXX指令在条件码(Cnd)满足时，会将栈顶的8字节数据读入到寄存器rA中，并且使栈指针增加8；如果条件不满足，则不执行该指令，即对rA、栈指针都无影响。

1. 在教材所描述的SEQ处理器上实现该指令，请按下表补全每个阶段的操作。需说明的信号可能会包括: icode, ifun, rA, rB, valA, valB, valC, valE, valP, Cnd: the register file R[], data memory M[], Program counter PC, condition codes CC。其中对存储器的引用必须标明字节数。如果在某一阶段没有任何操作，请填写none指明。

Stage	cpopqXX rA
Fetch	<code>icode:ifun ← M<sub>1</sub>[PC]</code> <code>rA:rB ← M<sub>1</sub>[PC+1]</code> <code>valP ← PC+2</code>
Decode	_____
Execute	_____
Memory	_____
Write back	_____
PC update	<code>PC ← valP</code>

2. 考虑在教材中的 Pipeline 处理器上实现该指令，需要改进教材所描述的 PIPE 处理器在执行 (E:Execute) 阶段的处理逻辑，使其能够在条件不满足时不修改 rA 和栈指针的值。请依据改进后的处理器，补全执行阶段的部分 HCL 代码：

```
word e_dstE = [
    (E_icode in {IRRMOVQ, ① } && ② ) : RNONE;
    1 : E_dstE;
];

word e_dstM = [
    (E_icode == ① && ② ) : RNONE;
    1 : E_dstM;
]
```

① \_\_\_\_\_ ② \_\_\_\_\_

3. 和 popq 与 mrmovq 相同，在执行该指令的过程中有可能会引发加载-使用冒险 (load-use hazard)。请将在引入了 cpopqXX 指令后的 Y86-64 PIPE 处理器中，加载-使用冒险的触发条件以及处理冒险的控制逻辑补充完整。如有多种可能，任写一种即可：

触发条件: E\_icode in {IMRMOVQ, IPOPQ} && e\_dstM in {\_\_\_\_\_, \_\_\_\_\_}  
控制逻辑: (选填 stall, bubble 和 normal)

F	D	E	M	W
			normal	normal

4. 基于改造后的 Y86-64 PIPE 考虑如下代码片段，回答问题。

```
# long foo(long n)
# n in %rdi
foo:
    xor    %rax, %rax          # Line 1
    pushq  %rsp               # Line 2
    irmovq $1, %rsi            # Line 3
L1:
    subq    %rsi, %rdi          # Line 4
    andq    %rdi, %rdi          # Line 5
    cpopqe %rsi                # Line 6
```

addq    %rsi, %rax          # Line 7
andq    %rdi, %rdi          # Line 8
jne     L1                  # Line 9
ret                 # Line 10

假设条件分支总是预测跳转，计算在输入的n值取1和3时，函数foo执行的周期数。(周期数计算从执行foo第一条指令开始，直到其返回指令ret完全通过流水线为止。另外在调用foo之前，处理器已经执行过足够多的nop指令。)  
n为1时候需要执行 \_\_\_\_\_ 个周期；n为3时候需要执行 \_\_\_\_\_ 个周期。

# E16

答案：

1.

Stage	copopqXX rA
Fetch	<u>icode:ifun</u> $\leftarrow M_1[PC]$ <u>rA:rB</u> $\leftarrow M_1[PC+1]$ <u>valP</u> $\leftarrow PC+2$
Decode	<u>valB</u> $\leftarrow R[\%rsp]$ <u>valA</u> $\leftarrow R[\%rsp]$
Execute	<u>valE</u> $\leftarrow valB + 8$ <u>Cnd</u> $\leftarrow Cond(CC, ifun)$
Memory	<u>valM</u> $\leftarrow M_8[valA]$
Write back	<u>if (Cnd) R[\%rsp] <math>\leftarrow valE</math></u> <u>if (Cnd) R[rA] <math>\leftarrow valM</math></u>
PC update	<u>PC <math>\leftarrow valP</math></u>

(每空1分，共7分)

2. ①IPOPQ ②!e\_Cnd (每空1分，共2分)，第1空填ICPOPOQXX也算对

3.

触发条件：E\_icode in {IMRMOVQ, IPOPQ} && e\_dstM in {d\_srcA, d\_srcB}

控制逻辑：(选填 stall, bubble 和 normal)

F	D	E	M	W
Stall	Stall	Bubble	normal	normal

(共3分，其中 e\_dstM in {d\_srcA, d\_srcB} 1分，全对才得分；控制逻

辑2分，全对才得分)

4.

解析：循环过程为Line 4~9，共6条

n=3 : 4(填充流水线)+3(Line 1~3)+6\*3(共三次循环)+1(Line 7~8:load-use hazard)+2(Line 9:branch misprediction)+1(ret指令)=29

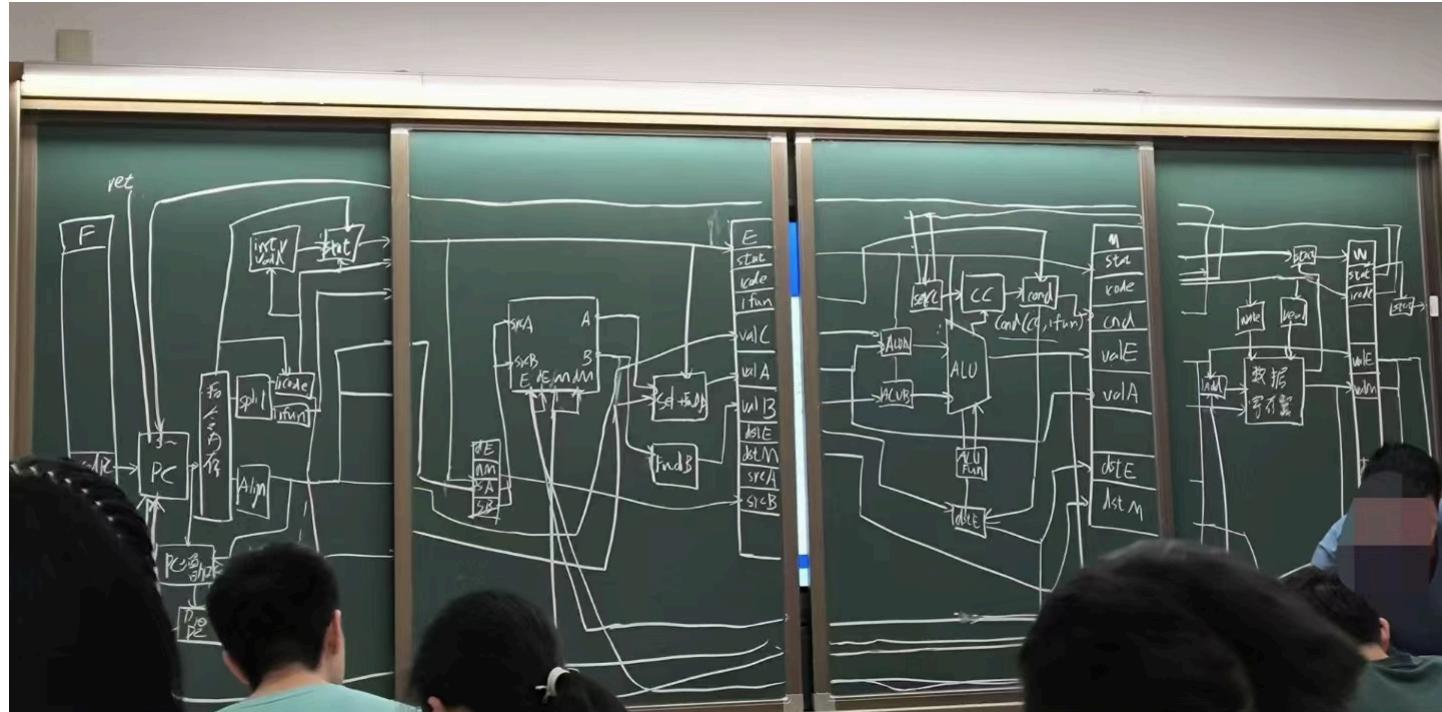
n=1 : 4(填充流水线)+3(Line 1~3)+6 (第一次循环)+1(Line 7~8:load-use hazard)+2(Line 9:branch misprediction)+1(ret指令)=17

(对任意一空得2分，两空全对得3分)

# Notices

# 记忆

ICS 去年小班同学默写盛况



# 时间轴

周次	大班	节次	主题 (放假安排以学校通知为准)	小班	节次	小班对应大班	LAB日程(参考)	回课同学
一	9月9日	1	Overview	9月11日	1	大班1		
	9月12日	2	Bits and Bytes/Integers				L1 (datalab) out	
二	9月14日	3	Floating Point	9月18日	2	大班2/3		A/B
	9月19日	4	Machine Prog: Basics					
三	9月23日	5	Machine Prog: Control	9月25日	3	大班4/5	L2 (bomblab) out	C/D
	9月26日	6	Machine Prog: Procedures					
四	9月30日	7	Machine Prog: Data					
	10月3日		国庆节快乐!					
五	10月7日		国庆节快乐!	10月9日	4	大班6/7	L3(attacklab) out	E/F
	10月10日	8	Machine Prog: Advanced					
六	10月14日	9	Processor Arch: ISA&Logic	10月16日	5	大班8/9	L4 (archlab) out	G/A
	10月17日	10	Processor Arch: Sequential					
七	10月21日	11	Processor Arch: Pipelined	10月23日	6	大班10/11		B/C
	10月24日	12	The Memory Hierarchy				L5 (cachelab) out	
八	10月28日	13	Cache Memories	10月30日	7	大班12/13/期中复习		D/E/F
	10月31日	14	Program optimization					
九	11月4日	15	期中考试	11月6日	8	期中讲解		
	11月7日	16	专题讲座					
十	11月11日	17	Linking I	11月13日	9	大班14/17		G/A
	11月14日	18	Linking II					
十一	11月18日	19	EOF: Exceptions & Processes	11月20日	10	大班18/19	L6 (tshlab) out	B/C
	11月21日	20	EOF: Signals & Nonlocal Jumps					
十二	11月25日	21	System Level I/O	11月27日	11	大班20/21		D/E
	11月28日	22	Virtual Memory: Concepts				L7(malloclab) out	
十三	12月2日	23	Virtual Memory: Systems	12月4日	12	大班22/23		F/G
	12月5日	24	Dynamic Memory Allocation					
十四	12月9日	25	Network Programming I	12月11日	13	大班24/25	L8 (proxylab) out	A/B
	12月12日	26	Network Programming II					
十五	12月16日	27	Concurrent Programming	12月18日	14	大班26/27		C/D
	12月19日	28	Synchronization: Basic					
十六	12月23日	29	Synchronization: Advanced	12月25日	15	大班28/29/期末复习	期末核查	E/F/G
	12月26日	30	期末复习					

# 补充资料

- HCL语言: [HCL Descriptions of Y86-64 Processors.pdf](#)

CS:APP3e Web Aside ARCH:HCL:  
HCL Descriptions of Y86-64 Processors\*

Randal E. Bryant  
David R. O'Hallaron

December 29, 2014

- 我的Y86-64学习笔记: [Y86-64 Note.html](#)

- Y86-64 Note  
 - Y86-64指令集体系结构  
 - Y86-64程序实现  
 - Y86-64硬件结构  
 - SEQ  
 - SEQ+  
 - PIPE-  
 - PIPE

## Y86-64 Note

### Y86-64指令集体系结构

#### Y86-64指令

- YAS: Y86-64 assembler ( $ys \rightarrow yo$ )
- YIS: Y86-64 simulator (run  $yo$  programs)

- 我的回课: [Pipelined Review.pdf](#) \\ 往年补充资料: [pipelined res.pdf](#)

# THANKS

Made by WalkerCH

[changxinhai@stu.pku.edu.cn](mailto:changxinhai@stu.pku.edu.cn)

Reference: [Weicheng Lin]'s presentation.

