

# Processor Arch-ISA

元培數科 王恩博

Let's get started →

2025/10/22



# Knowledge Review

# 缓冲区溢出

buffer overflow

**缓冲区溢出**：指当程序试图将数据写入超出其分配的内存区域时发生的一种错误。

前置条件：

- 未正确检查输入数据的边界或者长度，导致数据溢出到相邻的内存区域。
- 局部变量和状态信息（如备份的被调用者保存寄存器）都存放在栈中，可能与缓冲区（数组）相邻。

此时，越界写操作会破坏存储在栈中的状态信息。

当程序使用这个被破坏的状态，试图重新加载寄存器或执行 `ret` 指令时，就会出现很严重的错误。

尤其是一些字符串输入的函数容易出现溢出，如 `strcpy` `sprintf` `scanf` `gets` 等。

# ROP 攻击

return-oriented programming attack

ROP 攻击是一种利用程序中已有的指令片段 (gadget) 来构造出特定指令序列的技术。

通过精心构造的指令序列，可以实现对程序的控制，绕过安全机制，执行任意代码。

1. 找到许多以 `ret` (`0xc3`) 结尾的小代码段 (gadget)
2. 把他们的地址逐一放以某个栈上返回地址结尾的一段内存中
3. 这些小代码段被正常的过程返回机制逐一执行

在 Attacklab 中，你会亲手实现 ROP 攻击！

# ROP 攻击

return-oriented programming attack

```
/* Echo Line */
void echo() {
    /* Way too small! */
    char buf[4];
    gets(buf);
}
```

这段代码分配了一个大小为 4 字节的缓冲区，然后调用 `gets` 函数从标准输入读取一行数据并存储在缓冲区中。

```
echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...
```

- `subq $24, %rsp`  
将栈指针向下移动 24 字节
- `movq %rsp, %rdi`  
将栈指针的值存储到 `%rdi` 寄存器中，从而准备好 `gets` 函数的第一个参数。

*Before call to gets*

Stack Frame  
for `call_echo`

Return Address  
(8 bytes)

20 bytes unused

[3] [2] [1] [0]

`buf` ←— `%rsp`

# ROP 攻击

return-oriented programming attack

输入：

```
01234567890123456789012\0 // 24 个字符
```

此时，发生了缓冲区溢出，但是没造成严重后果。

尤其注意，`gets` 函数会在缓冲区末尾自动添加一个空字符 `\0`。

绿色框圈出的预期的安全输入缓冲区范围。

*After call to gets*

Stack Frame  
for `call_echo`

00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

`buf ← %rsp`

# ROP 攻击

return-oriented programming attack

输入：

```
012345678901234567890123\0 // 25 个字符
```

此时，不仅发生了缓冲区溢出，还造成了严重后果：

因为溢出到了存放返回后下一条指令的（调用者的栈帧内）区域，导致程序 `ret` 后，会跳转到错误的位置执行。

按照这个思路，继续溢出直至恰好将整个返回地址覆盖，就可以跳转到我们想要执行的代码位置。

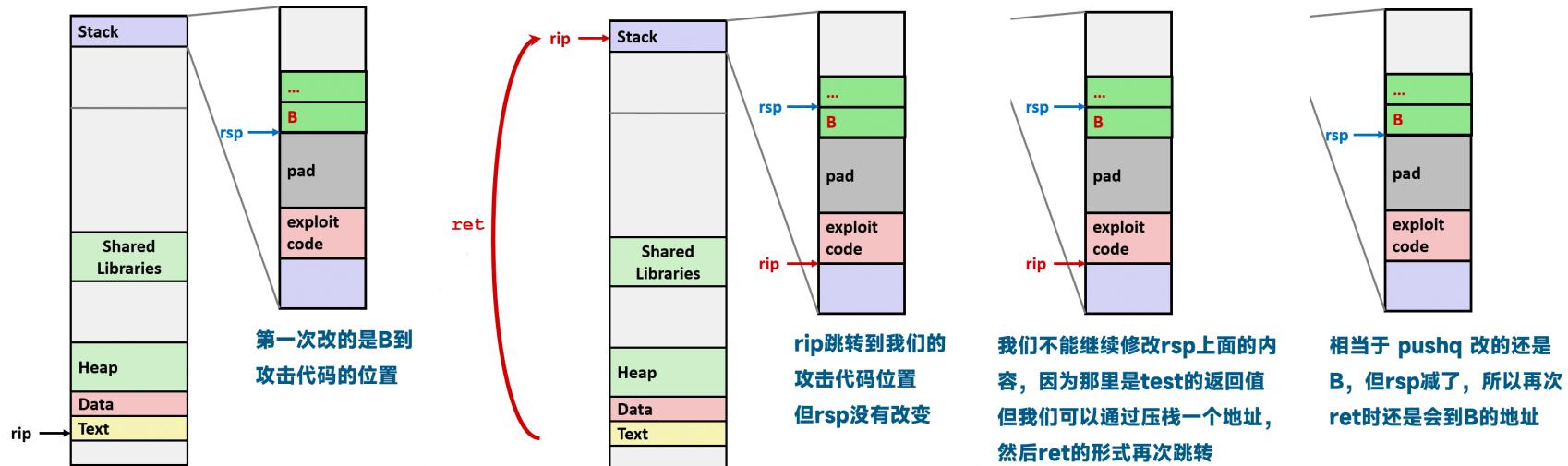
*After call to gets*

Stack Frame for <code>call_echo</code>			
00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

`buf` ← %rsp

# ROP 攻击

return-oriented programming attack



# 避免缓冲区溢出攻击

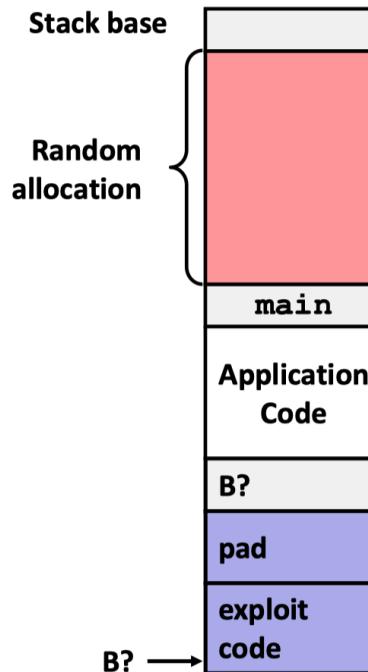
avoid buffer overflow attack

- **使用安全的函数编写程序:** `fgets` `strncpy` `snprintf`，指定每次读取的字节数
- **地址随机化 (Address Space Layout Randomization, ASLR)**：随机化程序的内存布局，使得攻击者无法事先确定数据地址
- **限制可执行代码区域**：将可执行代码区域限制在特定的内存区域，使用页表进行限制
- **设置金丝雀值 (canary) 进行栈破坏检测**：在栈帧中复制一处不可修改的地方的值过来，当程序试图覆盖返回地址时，必然会破坏金丝雀值，从而在返回时可以检测到栈溢出

# 地址随机化

address space layout randomization

- 随机化栈偏移：**在程序启动时，系统会在栈上分配一个随机大小的空间。从而每次程序执行时，栈的布局都会有所不同。
- 栈地址的偏移：**由于栈的随机分配，整个程序的栈地址都会发生变化。这种变化使得攻击者难以预测插入代码的起始位置。从而即使插入了恶意代码，也无法准确执行。



# 限制可执行代码区域

restrict the executable code region

给予内存区域一个 **标记**，来标志其内的字节是否可以作为代码执行。

后续章节中会学到，这会通过页表的权限位来实现。

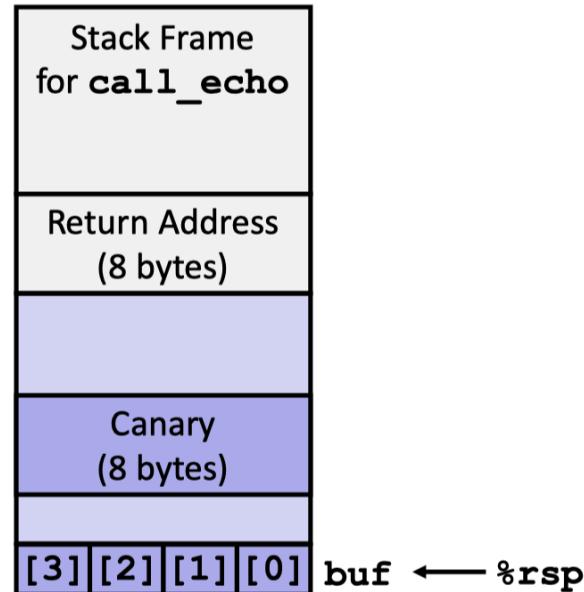
类似的，还会有 **只读** 权限位。

# 金丝雀值

canary value

在栈帧中，除了返回地址外，还会在栈帧的末尾添加一个金丝雀值。

当程序试图覆盖返回地址时，必然会破坏金丝雀值，从而在返回时可以检测到栈溢出。



# 金丝雀值

canary value

```
echo:  
    sub $0x18,%rsp  
    mov %fs:0x28,%rax  
    mov %rax,0x8(%rsp)  
    xor %eax,%eax  
    mov %rsp,%rdi  
    callq 4006e0 <gets>  
    mov %rsp,%rdi  
    callq 400570 <puts@plt>  
    mov 0x8(%rsp),%rax  
    xor %fs:0x28,%rax  
    je 400768 <echo+0x39>  
    callq 400580 <__stack_chk_fail@plt>  
    add $0x18,%rsp  
    retq
```

- 从特定的段寄存器 `%fs` 的偏移地址 `0x28` 加载金丝雀，并将金丝雀值存储到栈中分配的空间中，然后清除中间用过的寄存器。
- 在函数返回前，检查金丝雀值是否被破坏，如果被破坏，则调用 `__stack_chk_fail` 函数终止程序。

# 什么是 ISA?

Instruction Set Architecture

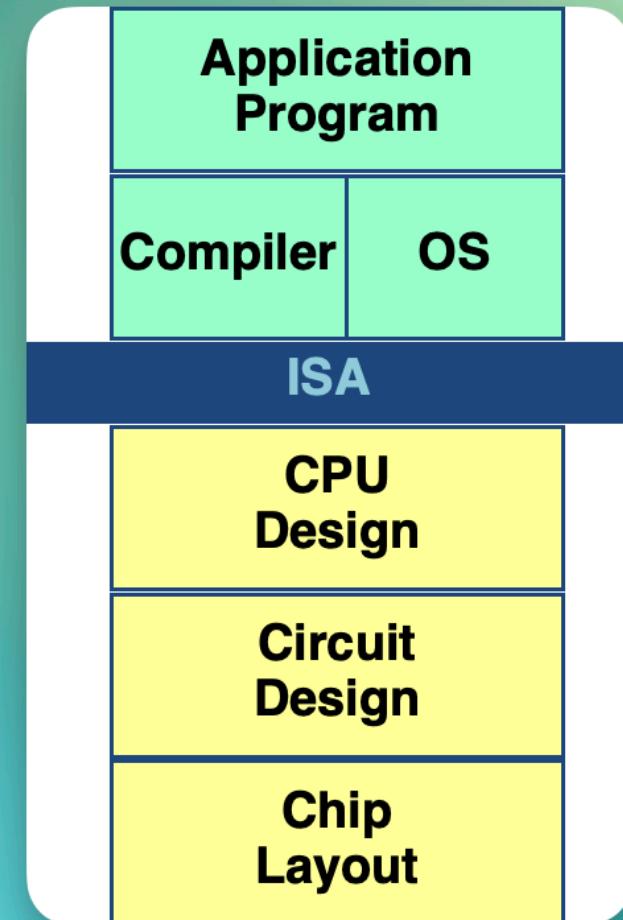
直译：指令集体系结构

如果非要强行解释... 

- “汇编语言”转换到“机器码”（相当于一个翻译过程）
- CPU 执行机器码的晶体管和逻辑电路的集合

Y86-64: 一种精简的 ISA

- 
1. CPU 指令集 (Instruction Set Architecture, ISA) / Zhihu ↪



# 程序员可见状态

programmer visible state

缩写	全称	描述	包括
RF	Register File	程序寄存器	%rax ~ %r14
CC	Condition Code	条件码	ZF <sub>zero</sub> , OF <sub>overflow</sub> , SF <sub>symbol</sub>
Stat	Status	程序状态	-
PC	Program Counter	程序计数器	-
DMEM	Data Memory	内存	-

# Y86-64 ISA

一个 X86-64 的子集

```
* halt # 停机
* nop # 空操作，可以用于对齐字节
* cmovXX rA, rB # 如果条件码满足，则将寄存器 A 的值移动到 B
* rrmovq rA, rB # 将寄存器 A 的值移动到寄存器 B
* irmovq V, rB # 将立即数 V 移动到寄存器 B
* rmmovq rA, D(rB) # 将寄存器 A 的值移动到内存地址 rB +
* mrmovq D(rB), rA # 将内存地址 rB + D 的值移动到寄存器 A
* OPq rA, rB # 将寄存器 A 和寄存器 B 的值进行运算，结果存入 rA
* jXX Dest # 如果条件码满足，跳转到 Dest
* call Dest # 跳转到 Dest，同时将下一条指令的地址压入栈
* ret # 从栈中弹出地址，跳转到该地址
* pushq rA # 将寄存器 A 的值压入栈
* popq rA # 从栈中弹出值，存入寄存器 A
```

- ◀  ▶
- 第一个字节为 **代码**，其高 4 位为操作类型，低 4 位为操作类型 (fn) 的具体操作 (或 0)
- F: 0xF, 为 Y86-64 中“不存在的寄存器”
- 所有数值 (立即数、内存地址) 均以 hex 表示，为 8 字节

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	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
call Dest	8	0								Dest
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

# Y86-64 ISA

## 一个 X86-64 的子集

```

* halt # 停机
* nop # 空操作, 可以用于对齐字节
* cmovXX rA, rB # 如果条件码满足, 则将寄存器 A 的值移动到 B
* rrmovq rA, rB # 将寄存器 A 的值移动到寄存器 B
* irmovq V, rB # 将立即数 V 移动到寄存器 B
* rmmovq rA, D(rB) # 将寄存器 A 的值移动到内存地址 rB +
* mrmovq D(rB), rA # 将内存地址 rB + D 的值移动到寄存器 A
* OPq rA, rB # 将寄存器 A 和寄存器 B 的值进行运算, 结果存入 rA
* jXX Dest # 如果条件码满足, 跳转到 Dest
* call Dest # 跳转到 Dest, 同时将下一条指令的地址压入栈
* ret # 从栈中弹出地址, 跳转到该地址
* pushq rA # 将寄存器 A 的值压入栈
* popq rA # 从栈中弹出值, 存入寄存器 A

```



- i(immediate): 立即数
- r(register): 寄存器
- m(memory): 内存地址
- d(displacement): 偏移量
- dest(destination): 目标地址
- v(value): 数值

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	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
call Dest	8	0								Dest
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

# Fn

## Jmp Fn

jmp	7	0
jle	7	1
jl	7	2
je	7	3
jne	7	4
jge	7	5
jg	7	6

## Mov Fn

rrmovq	2	0
cmovele	2	1
cmovl	2	2
cmove	2	3
cmovne	2	4
cmovge	2	5
cmovg	2	6

## OP Fn

addq	6	0
subq	6	1
andq	6	2
xorq	6	3

# 寄存器

## Register

- \* 0x0 %rax
- \* 0x1 %rcx
- \* 0x2 %rdx
- \* 0x3 %rbx
- \* 0x4 %rsp
- \* 0x5 %rbp
- \* 0x6 %rsi
- \* 0x7 %rdi
- \* 0x8 %r8
- \* 0x9 %r9
- \* 0xA %r10
- \* 0xB %r11
- \* 0xC %r12
- \* 0xD %r13
- \* 0xE %r14
- \* 0xF F / No Register

- a,c,d,b + x # AcFun 倒 (D) 了, 然后 Bilibili 兴起了
- 栈指针 (包括栈顶%rsp和栈底%rbp)
- 前两个参数指针
- 按序的 %r8 ~ %r14



%rax	0
%rcx	1
%rdx	2
%rbx	3
%rsp	4
%rbp	5
%rsi	6
%rdi	7

%r8	8
%r9	9
%r10	A
%r11	B
%r12	C
%r13	D
%r14	E
No Register	F

# 汇编代码翻译

translate assembly code to machine code

以下习题节选自书 P248, 练习题 4.1 / 4.2

## Step 1

---

0x200      a0 6f | 80 0c 02 00 00 00 00 00 00 | 00 | 30 f 3 0a 00 00 00 00 00 00 00 00 | 90

---

loop      rmmovq, rcx, rbx, -3

---

## Step 2

```
0x200:
    pushq %rbp
    call 0x20c
    halt
0x20c:
    irmovq $10 %rbx
    ret
```

40 1 3 ff ff ff fd

# Y86-64 vs x86-64, CISC vs RISC

Complex Instruction Set Computer & Reduced Instruction Set Computer

- Y86-64 是 X86-64 的子集
- X86-64 更复杂，但是更强大
- Y86-64 更简单，复杂指令由简单指令组合而成
- CISC：复杂指令集计算机
- RISC：精简指令集计算机
- 设计趋势是融合的

如 Y86-64 的算数指令（`OPq`）只能操作寄存器，而 X86-64 可以操作内存

所以 Y86-64 需要额外的指令（`mrmovq`、`rmmovq`）来先加载内存中的值到寄存器，再进行运算



# Y86-64 状态

status

值	名字	含义	全称
1	AOK	正常操作	All OK
2	HLT	遇到器执行 halt 指令遇到非法地址	Halt
3	ADR	遇到非法地址，如向非法地址读/写	Address Error
4	INS	遇到非法指令，如遇到一个 ff	Invalid Instruction

除非状态值是 AOK，否则程序会停止执行。

# Y86-64 栈

stack

```
Pushq rA F / 0xA0 rA F
```

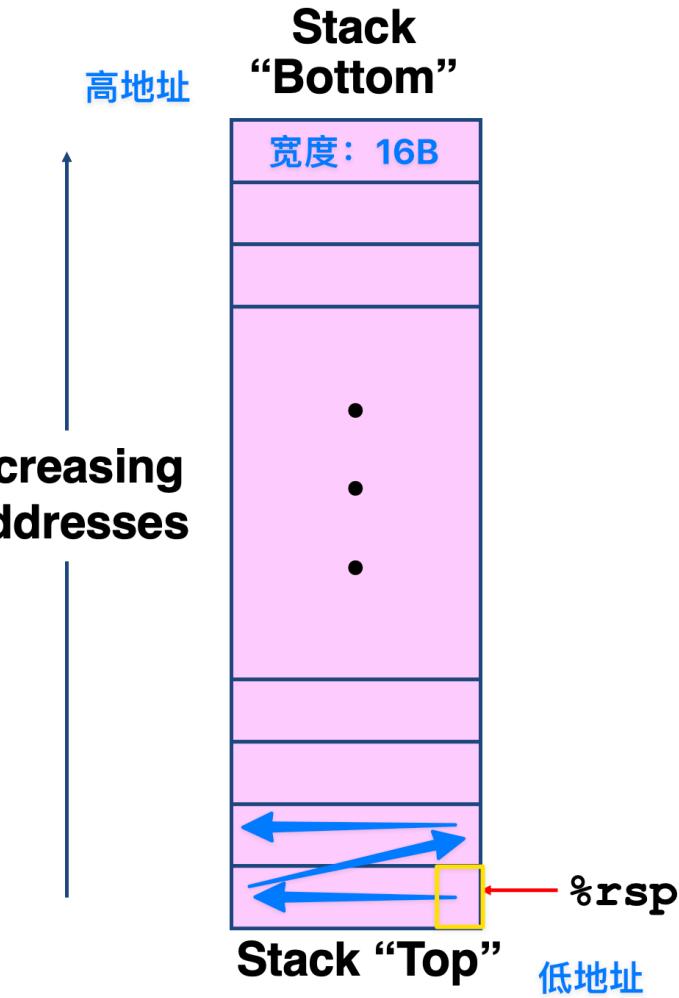
压栈指令

- 将 `%rsp` 减去8
- 将字从 `rA` 存储到 `%rsp` 的内存中

```
Popq rA F / 0xB0 rA F
```

弹栈指令

- 将字从 `%rsp` 的内存中取出
- 将 `%rsp` 加上8
- 将字存储到 `rA` 中



# X86-64 程序调用

call & ret

Call Dest / 0x80 Dest

调用指令

- 将下一条指令的地址 pushq 到栈上 ( %rsp 减 8、地址存入栈中)
- 从目标处开始执行指令

---

Ret / 0x90

返回指令

- 从栈上 popq 出地址，用作下一条指令的地址 ( %rsp 加 8、地址从栈中取出，存入 %rip )

# Y86-64 终止与对齐

Halt / 0x00

## 终止指令

- 停止执行
- 停止模拟器
- 在遇到初始化为 0 的内存地址时，也会终止
- 记忆：没有事情做了 ➔ 停止

Nop / 0x10

## 空操作

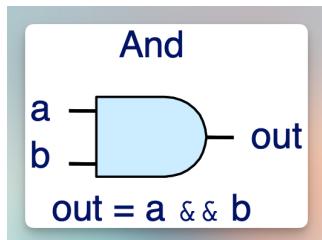
- 什么都不做（但是 PC Program Counter + 1），可以用于对齐字节
- 记忆：扣 1 真的没有用

# 逻辑设计和硬件控制语言 HCL

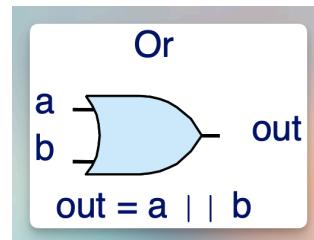
hardware control language

- 计算机底层是 0 (低电压) 和 1 (高电压) 的世界
- HCL (硬件 控制 语言) 是一种硬件 描述 语言 (HDL)，用于描述硬件的逻辑电路
- HCL 是 HDL 的子集

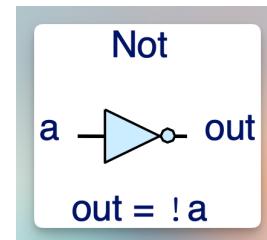
与门 And



或门 Or



非门 Not



$out = a \&\& b$

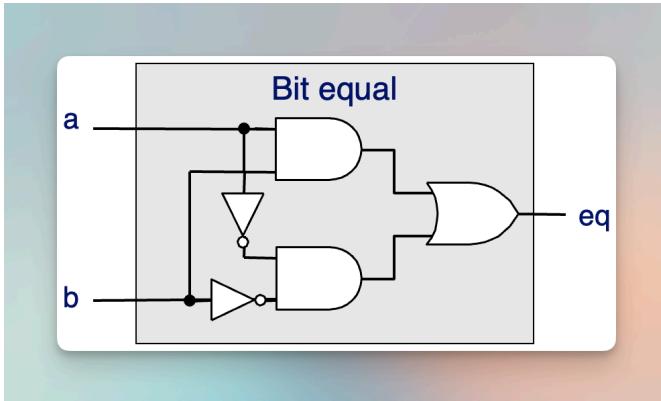
$out = a || b$

$out = !a$

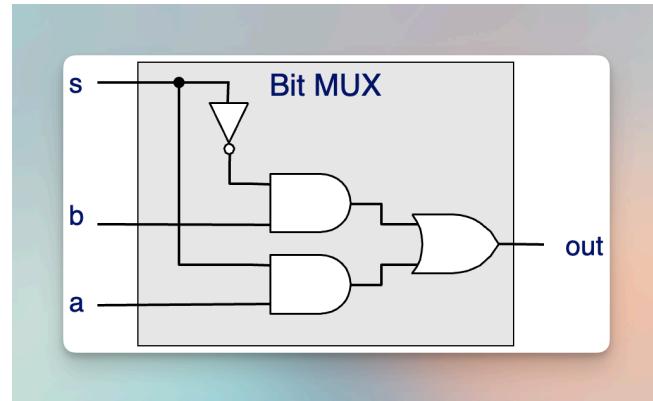
记忆：方形的更严格→与；圆形的更宽松→或

# 组合电路 / 高级逻辑设计

```
bool eq = (a && b) || (!a && !b);
```



```
bool out = (s && a) || (!s && b);
```

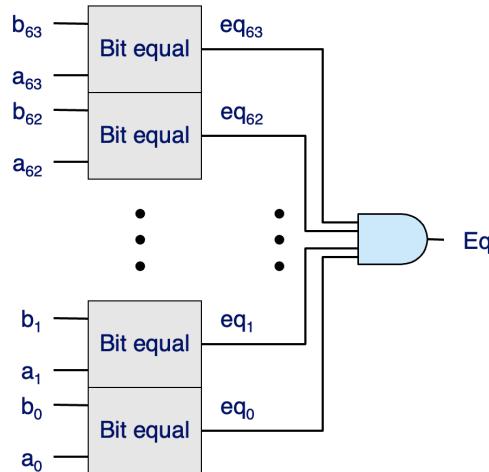


- 组合电路是 **响应式** 的：在输入改变时，输出经过一个很短的时间会立即改变
- 没有短路求值特性：`a && b` 不会在 `a` 为 `false` 时就不计算 `b`

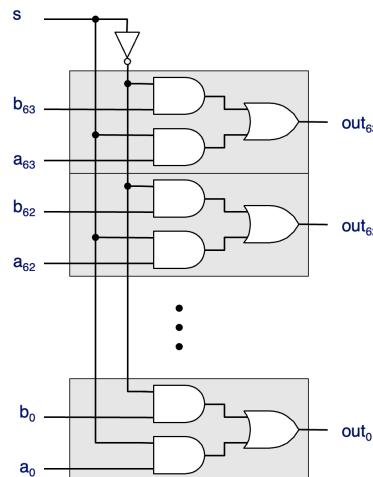
- Mux: Multiplexer / 多路复用器，用一个 `s` 信号来选择 `a` 或 `b`

# 组合电路 / 高级逻辑设计

```
bool Eq = (A == B)
```

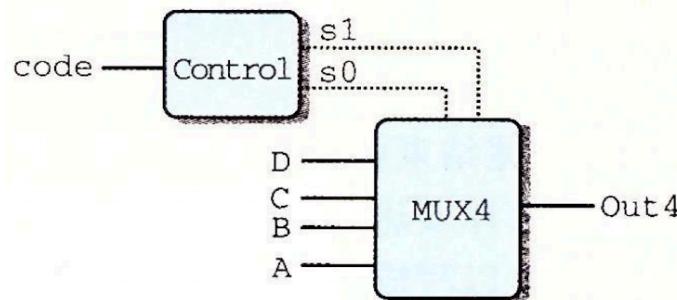


```
int Out = [
    s : A; # select: expr
    1 : B;
];
```



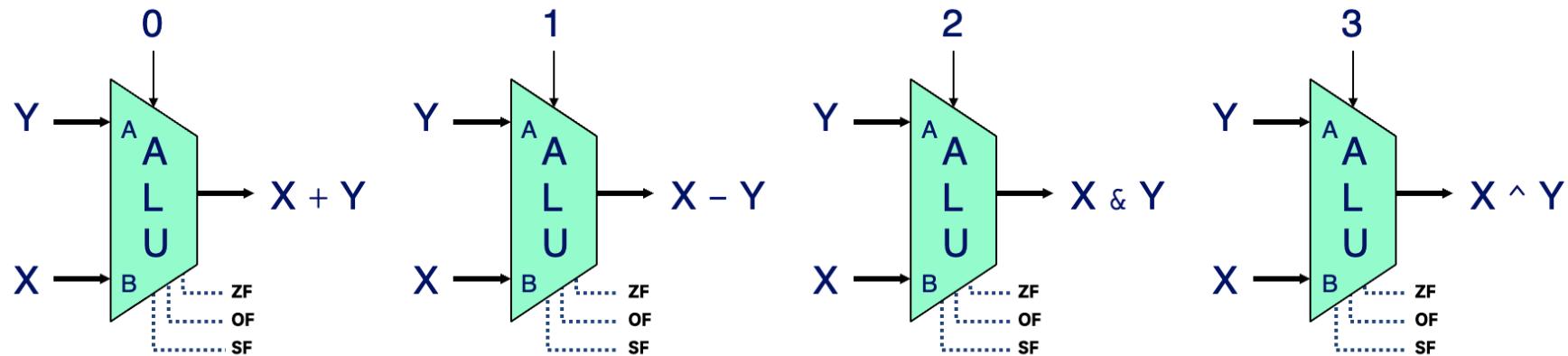
# 组合电路 / 集合关系

```
int Out4 = [  
    bool s1 = code in {2, 3}; # 10, 11  
    bool s2 = code in {1, 3}; # 01, 11  
];
```



# 组合电路 / 算数逻辑单元 ALU

Arithmetic Logic Unit



- 组合逻辑
- 持续响应输入
- 控制信号选择计算的功能
- 对应于 Y86-64 中的 4 个算术 / 逻辑操作
- 计算条件码的值
- 注意 Sub 是被减的数在后面, 即输入 B 减去输入 A, 等于 subq A, B

# 存储器和时钟

响了十二秒的电话我没有接，只想要在烟花下闭着眼~

组合电路：不存储任何信息，只是一个 **输入** 到 **输出** 的映射（有一定的延迟）

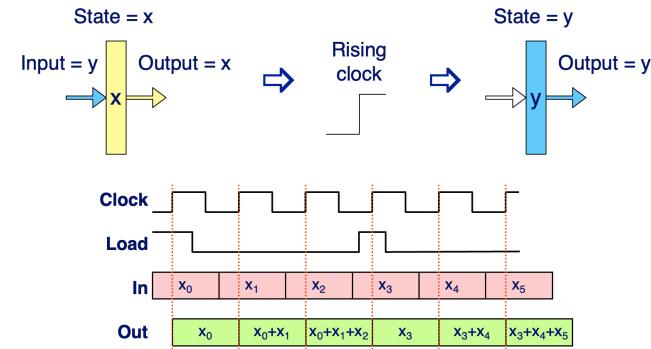
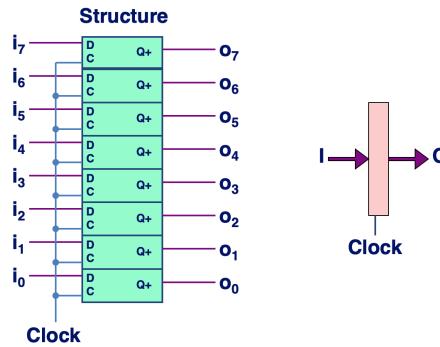
时序电路：有**状态**，并基于此进行计算

# 时钟寄存器 / 寄存器 / 硬件寄存器

register

存储单个位或者字

- 以时钟信号控制寄存器加载输入值
- 直接将它的输入和输出线连接到电路的其他部分



在 Clock 信号的上升沿，寄存器将输入的值采样并加载到输出端，其他时间输出端保持不变

# 随机访问存储器 / 内存

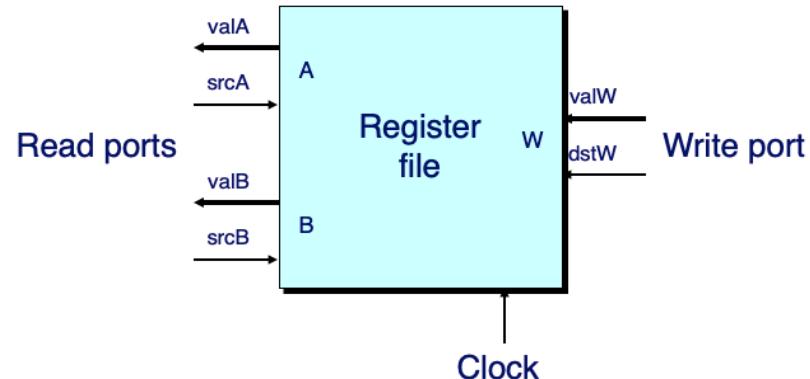
memory

以 地址 选择读写

包括：

- 虚拟内存系统，寻址范围很大
- 寄存器文件 / 程序寄存器，个数有限，在 Y86-64 中为 15 个程序寄存器（`%rax ~ %r14`）

可以在一个周期内读取和 / 或写入多个字词



# 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	

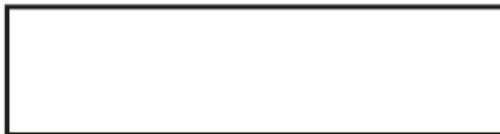
CC:  
Condition  
codes



Stat: Program status



DMEM: Memory



PC



# 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					
cmoveq 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 $S$	$R[%rsp] \leftarrow R[%rsp] - 8;$ $M[R[%rsp]] \leftarrow S$	Push quad word
popq $D$	$D \leftarrow M[R[%rsp]];$ $R[%rsp] \leftarrow R[%rsp] + 8$	Pop quad word

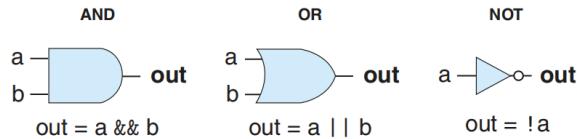
Figure 3.8 Push and pop instructions.

# 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

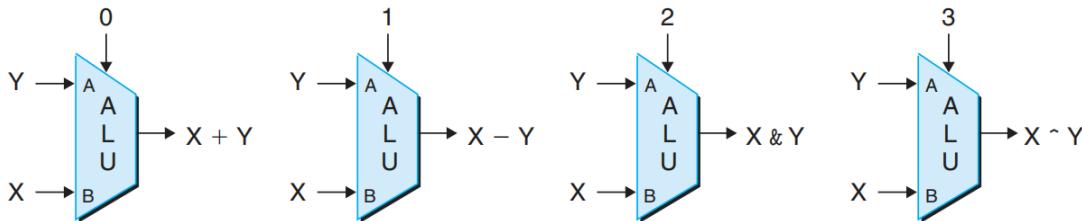
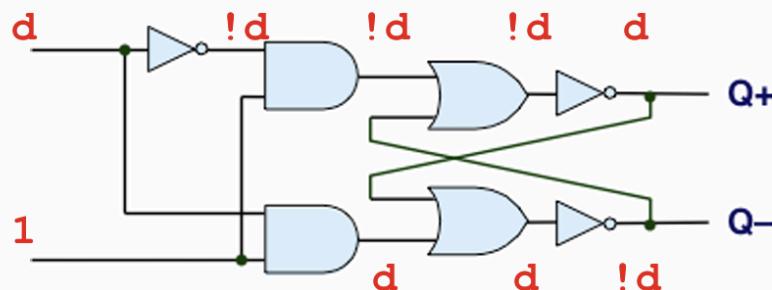


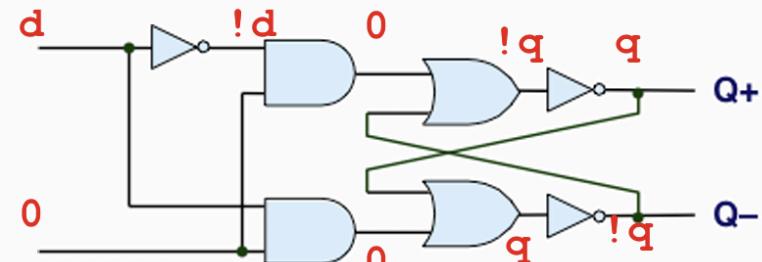
Figure 4.15 Arithmetic/logic unit (ALU). Depending on the setting of the function input, the circuit will perform one of four different arithmetic and logical operations.

# 寄存器原理概述

## Latching



## Storing



# Homework Review

# 冒泡排序

## C 指针实现

```
void bubble_sort(long *data, long count)
{
    long *i, *last;
    for(last = data + count - 1; last > data; last--) {
        for(i = data; i < last; i++) {
            if(*(i + 1) < *i) {
                long t = *(i + 1);
                *(i + 1) = *i;
                *i = t;
            }
        }
    }
}
```

# 冒泡排序

```
.pos 0
irmovq stack,%rsp
call main
halt

.align 8
array:
.quad 0xbca
.quad 0xcba
.quad 0xacb
.quad 0xcbab
.quad 0xabc
.quad 0xba

bubble_a:
pushq %rbx
pushq %r13
pushq %r14
irmovq $1, %r10
irmovq $8, %r9
rrmovq %rsi, %rax
subq %r10, %rax
jle L1
```

```
rrmovq %rdi, %r8
rrmovq %rax, %r11
addq %r11, %r11
addq %r11, %r11
addq %r11, %r11
addq %r11, %r8
jmp L6
L4:
addq %r9,%rax
rrmovq %rax,%rbx
subq %rsi, %rbx
je L3
L5:
mrmovq (%rax),%rcx
mrmovq 8(%rax),%rdx
rrmovq %rdx,%r13
subq %rcx,%r13
#####
jge L4
rmmovq %rcx,8(%rax)
rmmovq %rdx,(%rax)
#####
jmp L4
```

# 冒泡排序

```
L3:  
    subq %r9,%r8  
    rrmovq %rdi,%rbx  
    subq %r8,%rbx  
    je L1
```

```
L6:  
    rrmovq %r8,%rsi  
    rrmovq %rdi,%rbx  
    subq %rsi,%rbx  
    je L3  
    rrmovq %rdi,%rax  
    jmp L5
```

```
L1:  
    popq %r13  
    popq %rbx  
    ret
```

```
main:  
    irmovq array, %rdi  
    irmovq $6, %rsi  
    call bubble_a  
    ret
```

```
.pos 0x200  
stack:
```

# 冒泡排序

## 三次条件传送

```
L5:  
    mrmovq (%rax),%rcx  
    mrmovq 8(%rax),%rdx  
    rrmovq %rdx,%r13  
    subq %rcx,%r13  
    #####  
    cmovl %rcx,%rbx  
    cmovl %rdx,%rcx  
    cmovl %rbx,%rdx  
    rmovq %rcx,(%rax)  
    rmovq %rdx,8(%rax)  
    #####  
    jmp L4
```

# 冒泡排序

## 一次条件传送

L5:

```
mrmovq (%rax),%rcx
mrmovq 8(%rax),%rdx
#####
rrmovq %rdx,%r13
rrmovq %rdx,%r14
xorq %rcx,%rdx
subq %rcx,%r13
cmovge %r14,%rcx
xorq %rdx,%rcx
xorq %rcx,%rdx
rmovq %rcx,(%rax)
rmovq %rdx,8(%rax)
#####
jmp L4
```

# Exercises

**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。

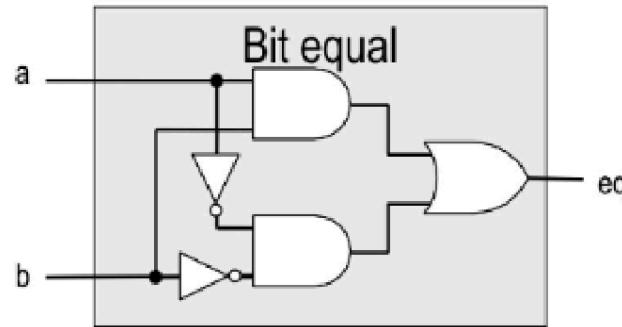
# E9

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

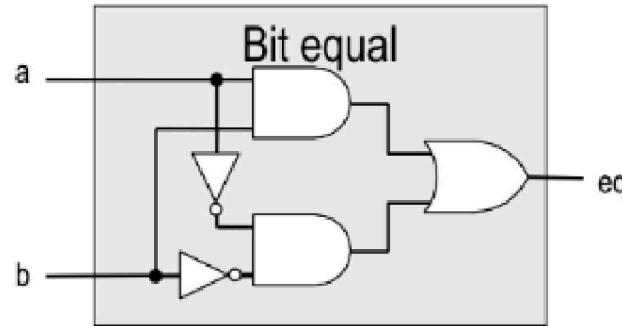
10. 对应下述组合电路的正确 HCL 表达式为：



- 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：

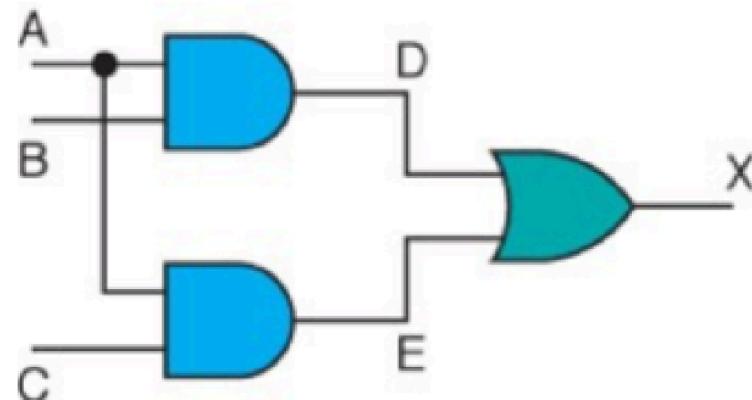


- A. Bool eq = ( a or b ) and ( !a or !b )
- B. Bool eq = ( a and b ) or ( !a and !b )
- C. Bool eq = ( a or !b ) and ( !a or b )
- D. Bool eq = ( a and !b ) or ( !a and b )

答案： B

**E11**

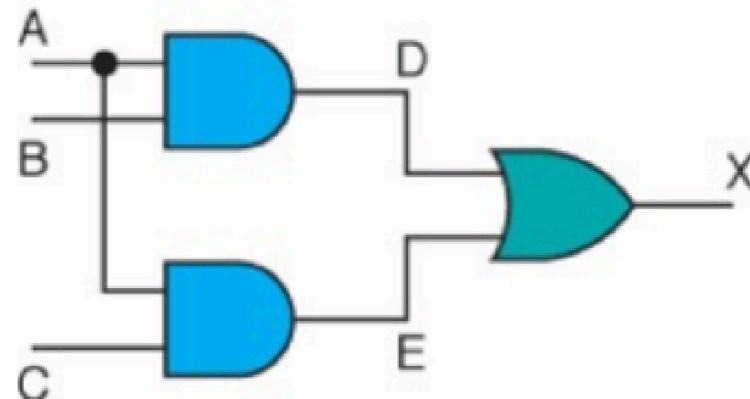
9、对应下述组合电路的正确 HCL 表达式为



- 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

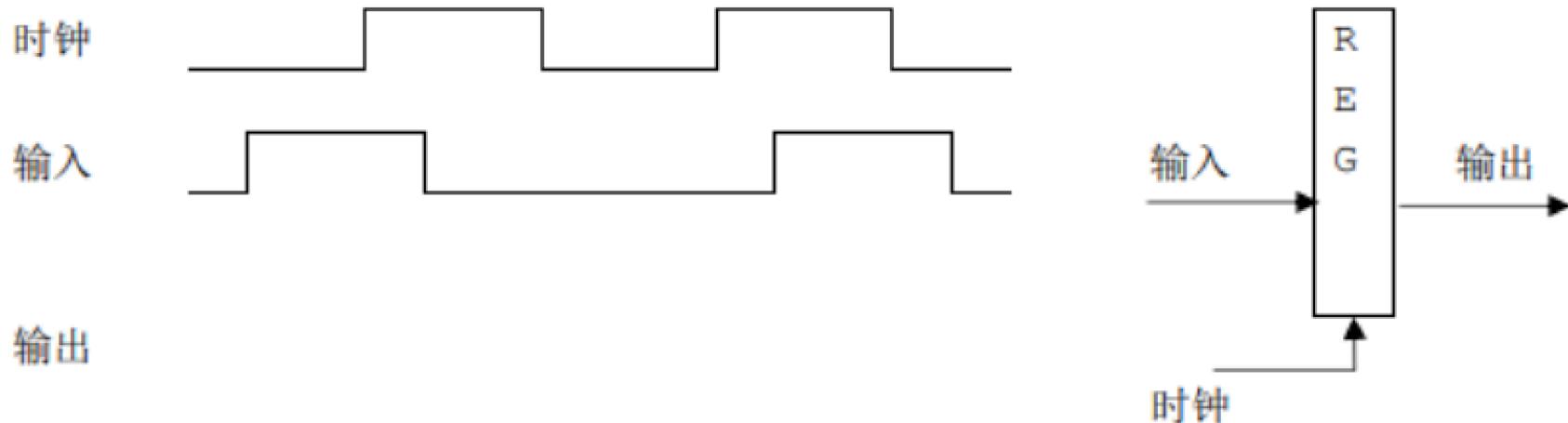


- A. Bool X = (A || B) && (A || C)
- B. Bool X = A || (B && C)
- C. Bool X = A && (B || C)
- D. Bool X = A || B || C

**答案: C**

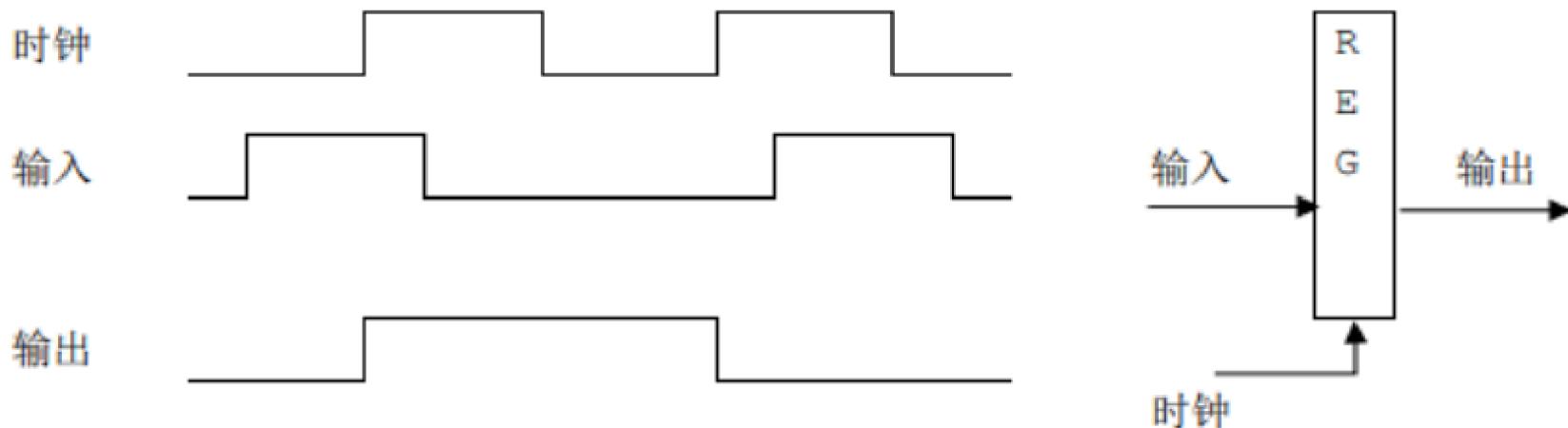
## E12

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）



# 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

- Y86-64学习笔记: [Y86-64 Note.html](#)

- Y86-64 Note
  - Y86-64指令集体系结构
  - Y86-64顺序实现
  - 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 → .yo)
- YIS: Y86-64 simulator (run .yo programs)

# THANKS

Made by WEB-05

webrun@stu.pku.edu.cn

Reference: [WalkerCH]'s and [Arthals]'s presentations.



扫一扫上面的二维码图案，加我为朋友。