

Processor Arch: ISA & Logic

Instruction-Set Architecture (ISA)

ISA

- Instruction-Set Architecture (ISA)

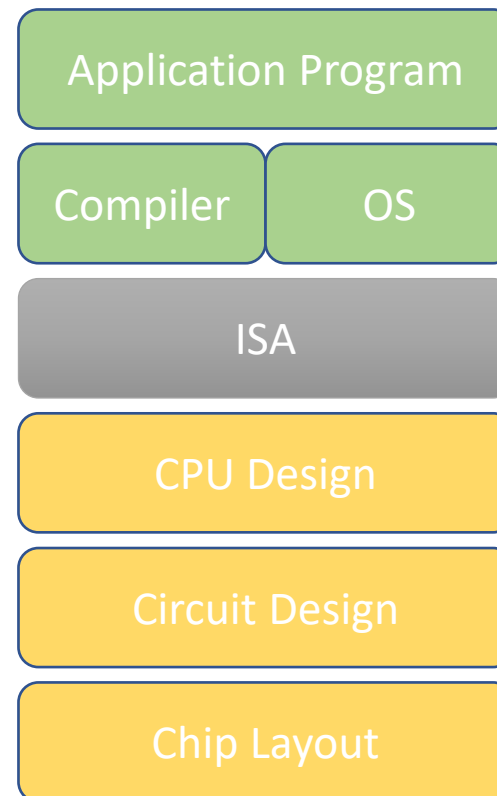
旧译**指令集**，现在更正式的翻译是**指令系统**

也常用 Architecture 指代 ISA

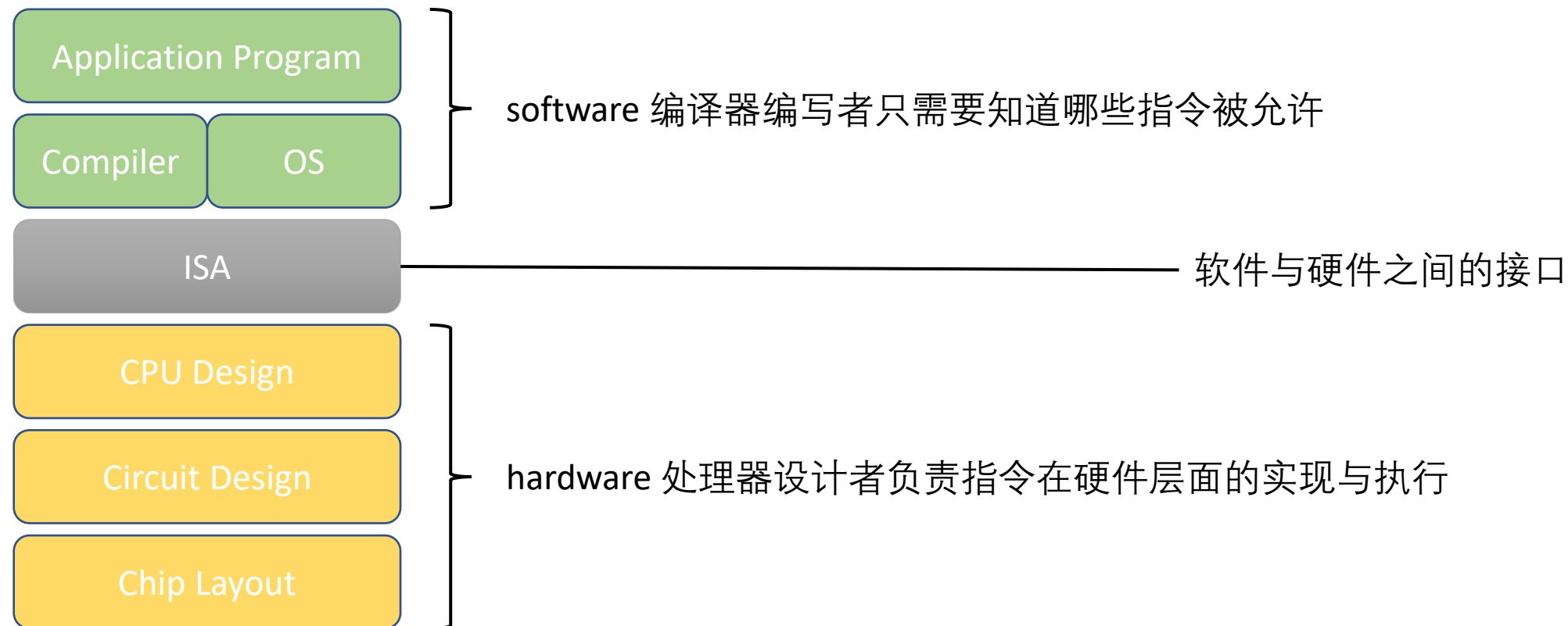
- 一个**处理器**支持的**指令和指令的字节级编码**
- 不同处理器“家族”有着不同的ISA，同一处理器家族的不同型号处理器在 ISA 层面保持兼容
- 一旦程序被编译为特定指令集下的汇编程序，就不能在其他指令集机器上运行
- Intel IA32 x86-64

IBM/Freescale Power

ARM



ISA



Introduction

Concepts

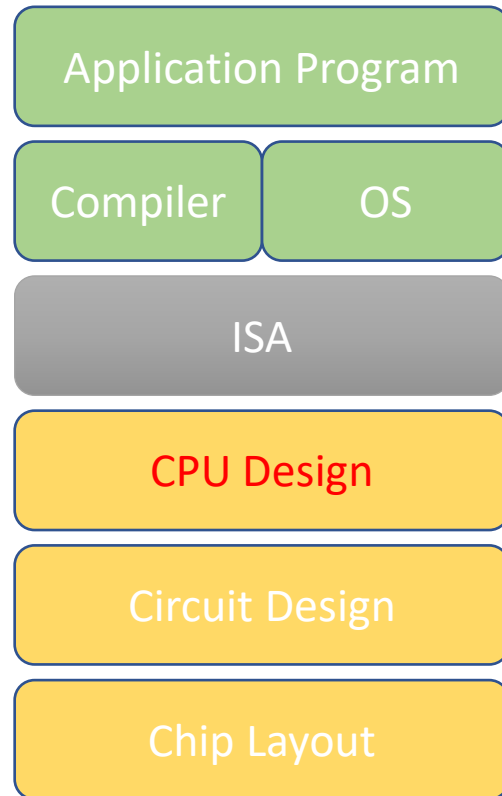
Components

RISC&CISC

Y86-64 ISA

Complement

Microarchitecture



x86-64 ISA

Intel: Nehalem, Sandy Bridge, Ivy Bridge, Haswell, Skylake...

AMD: K8, Bulldozer, Piledriver, Zen, Zen2

- 微架构 (Microarchitecture), 与 ISA (Architecture) 不同
 - 关注处理器 (CPU) 内部的设计和实现
 - 描述了处理器如何执行特定体系结构中的指令集, 以实现高性能和效率; 涉及到处理器的内部组件、数据通路、控制逻辑、缓存、流水线结构等
 - 即使 ISA 相同, Microarchitecture 也可不同
- 不同的 CPU 制造商和产品线通常有不同的微架构, 即使它们共享相同的体系结构。不同的微架构可以采用不同的设计方法和技术, 以实现不同性能、功耗和特性

Introduction

Concepts

Components

RISC&CISC

Y86-64 ISA

Complement

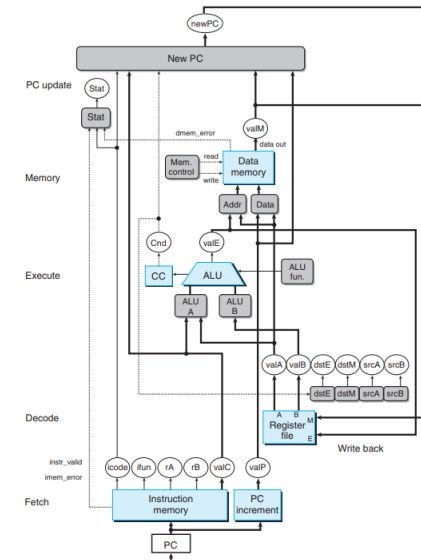
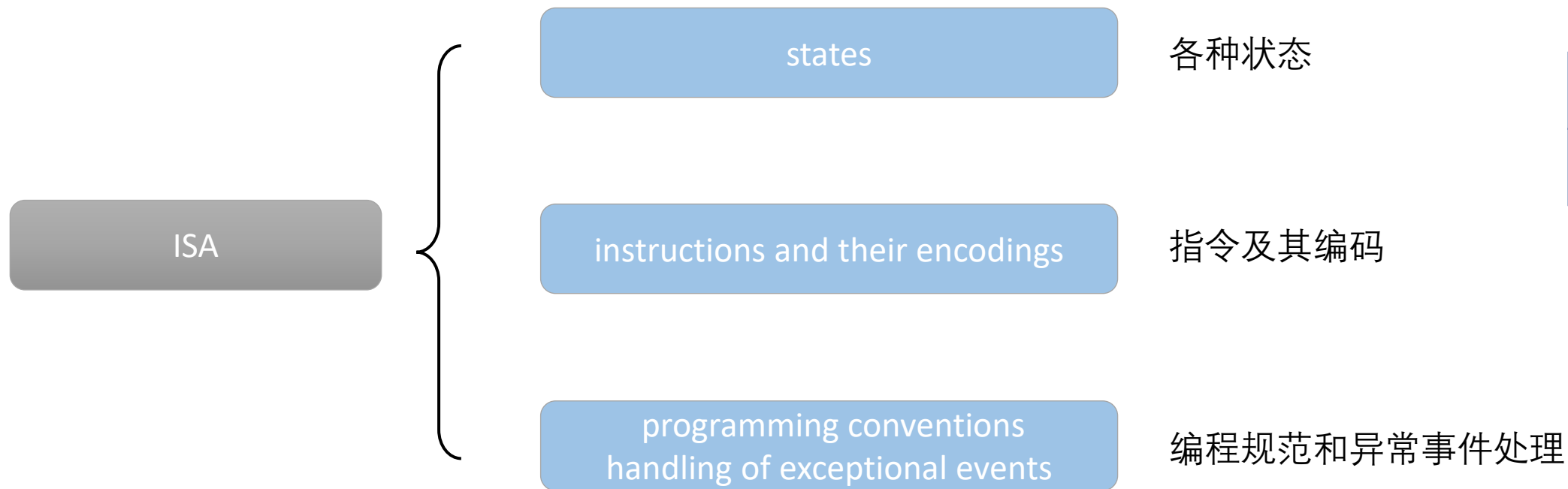


Figure 4.23 Hardware structure of SEQ, a sequential implementation. Some of the control signals, as well as the register and control word connections, are not shown.

ISA



Introduction

Concepts

Components

RISC&CISC

Y86-64 ISA

Complement

ISA

以下概念是否由ISA规定？

1. registers
2. nop instruction
3. memory address width
4. condition codes, status flags

ISA

以下概念是否由ISA规定？

- | | |
|----------------------------------|---|
| 1. registers | Y |
| 2. nop instruction | Y |
| 3. memory address width | Y |
| 4. condition codes, status flags | Y |

Introduction

Concepts

Components

RISC&CISC

Y86-64 ISA

Complement

RISC & CISC

RISC: Reduced Instruction-Set Computer (ARM, RISC-V, MIPS)

CISC: Complex Instruction-Set Computer (x86: IA32, x86-64; AVX)

CISC	早期的 RISC
指令数量很多。Intel 描述全套指令的文档[51]有 1200 多页。	指令数量少得多。通常少于 100 个。
有些指令的延迟很长。包括将一个整块从内存的一个部分复制到另一部分的指令，以及其他一些将多个寄存器的值复制到内存或从内存复制到多个寄存器的指令。	没有较长延迟的指令。有些早期的 RISC 机器甚至没有整数乘法指令，要求编译器通过一系列加法来实现乘法。
编码是可变长度的。x86-64 的指令长度可以是 1~15 个字节。	编码是固定长度的。通常所有的指令都编码为 4 个字节。
指定操作数的方式很多样。在 x86-64 中，内存操作数指示符可以有许多不同的组合，这些组合由偏移量、基址和变址寄存器以及伸缩因子组成。	简单寻址方式。通常只有基址和偏移量寻址。
可以对内存和寄存器操作数进行算术和逻辑运算。	只能对寄存器操作数进行算术和逻辑运算。允许使用内存引用的只有 load 和 store 指令，load 是从内存读到寄存器，store 是从寄存器写到内存。这种方法被称为 load/store 体系结构。
对机器级程序来说实现细节是不可见的。ISA 提供了程序和如何执行程序之间的清晰的抽象。	对机器级程序来说实现细节是可见的。有些 RISC 机器禁止某些特殊的指令序列，而有些跳转要到下一条指令执行完了以后才会生效。编译器必须在这些约束条件下进行性能优化。
有条件码。作为指令执行的副产品，设置了一些特殊的标志位，可以用于条件分支检测。	没有条件码。相反，对条件检测来说，要用明确的测试指令，这些指令会将测试结果放在一个普通的寄存器中。
栈密集的过程链接。栈被用来存取过程参数和返回地址。	寄存器密集的过程链接。寄存器被用来存取过程参数和返回地址。因此有些过程能完全避免内存引用。通常处理器有更多的(最多的有 32 个)寄存器。

RISC

- 出现晚于CISC
- 可以用很少的硬件实现，能以高效的流水线结构组织起来
- load/store体系结构
- 无条件码

RISC & CISC

CISC	(早期) RISC
指令类型多，功能丰富；部分指令延迟高	指令类型少，功能简单，无延迟高的指令
指令字节编码长度可变	指令字节编码长度不可变，通常为4字节
简单/复杂寻址模式	简单寻址模式
算术逻辑操作可以访问内存，访存模式多样	load/store体系结构
机器级程序不可见实现细节	机器级程序可见实现细节
有条件码	无条件码
栈密集型的过程链接	寄存器密集型的过程链接

RISC: MIPS

32*32bit registers

\$0	\$0	Constant 0	\$16	\$s0	
\$1	\$at	Reserved Temp.	\$17	\$s1	
\$2	\$v0	Return Values	\$18	\$s2	
\$3	\$v1		\$19	\$s3	
\$4	\$a0		\$20	\$s4	
\$5	\$a1	Procedure arguments	\$21	\$s5	
\$6	\$a2		\$22	\$s6	
\$7	\$a3		\$23	\$s7	
\$8	\$t0		\$24	\$t8	
\$9	\$t1		\$25	\$t9	
\$10	\$t2	Caller Save Temporaries: May be overwritten by called procedures	\$26	\$k0	
\$11	\$t3		\$27	\$k1	
\$12	\$t4		\$28	\$gp	
\$13	\$t5		\$29	\$sp	
\$14	\$t6		\$30	\$s8	
\$15	\$t7		\$31	\$ra	

**Callee Save Temporaries:
May not be overwritten by
called procedures**

Caller Save Temp

**Reserved for
Operating Sys**

Global Pointer

Stack Pointer

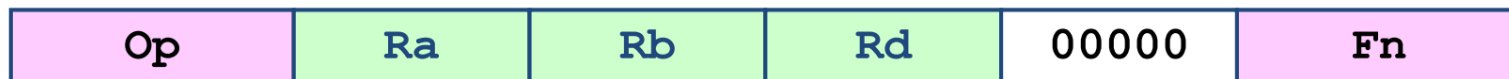
Callee Save Temp

Return Address

RISC: MIPS

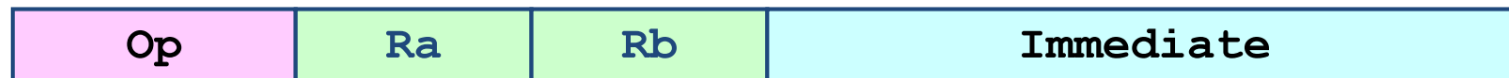
4-byte instructions

R-R



addu `$3,$2,$1` # Register add: $\$3 = \$2 + \$1$

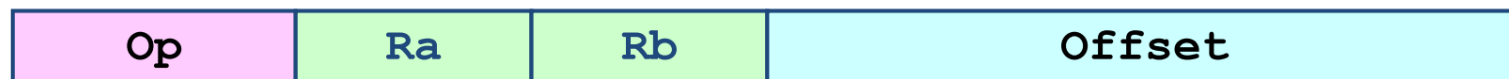
R-I



addu `$3,$2, 3145` # Immediate add: $\$3 = \$2 + 3145$

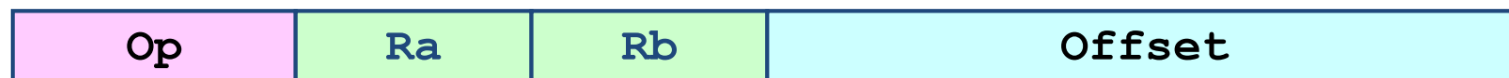
sll `$3,$2,2` # Shift left: $\$3 = \$2 \ll 2$

Branch



beq `$3,$2,dest` # Branch when $\$3 = \2

Load/Store



lw `$3,16($2)` # Load Word: $\$3 = M[\$2 + 16]$

sw `$3,16($2)` # Store Word: $M[\$2 + 16] = \3

RISC or CISC?

Text Segment			Address
pt	Address	Code	
	0x00400000	0x3c011001 lui \$1, 0x00	0x00400000
	0x00400004	0x34300000 ori \$16, \$1,	0x00400004
	0x00400008	0x3c011001 lui \$1, 0x00	
	0x0040000c	0x3435004c ori \$21, \$1,	0x00400008
	0x00400010	0x8eb50000 lw \$21, 0x00	
	0x00400014	0x3c011001 lui \$1, 0x00	0x0040000c
	0x00400018	0x34240050 ori \$4, \$1, 0	
	0x0040001c	0x24020004 addiu \$2, \$0	0x00400010
	0x00400020	0x0000000c syscall	
	0x00400024	0x24020005 addiu \$2, \$0	0x00400014
	0x00400028	0x0000000c syscall	
	0x0040002c	0x02a2082a slt \$1, \$21, \$2	
	0x00400030	0x1420fff8 bne \$1, \$0, 0xffffffff8	
	0x00400034	0x0040082a slt \$1, \$2, \$0	
	0x00400038	0x1420fff6 bne \$1, \$0, 0xffffffff6	
	0x0040003c	0x0040a820 add \$21, \$2, \$0	

RISC or CISC?

```
lock addl $0x12345678, %es:0x11223344(%eax, %ecx, 8)
```

```
26 66 67 F0 81 84 C8 44 33 22 11 78 56 34 12
```

Introduction

Concepts

Components

RISC&CISC

Y86-64 ISA

Complement

RISC or CISC?

```
lock addl $0x12345678, %es:0x11223344(%eax, %ecx, 8)
```

26 66 67 F0 81 84 C8 44 33 22 11 78 56 34 12

x86-32中最复杂的指令，15字节编码

RISC or CISC?

1. 下列描述更符合（早期）RISC 还是 CISC?

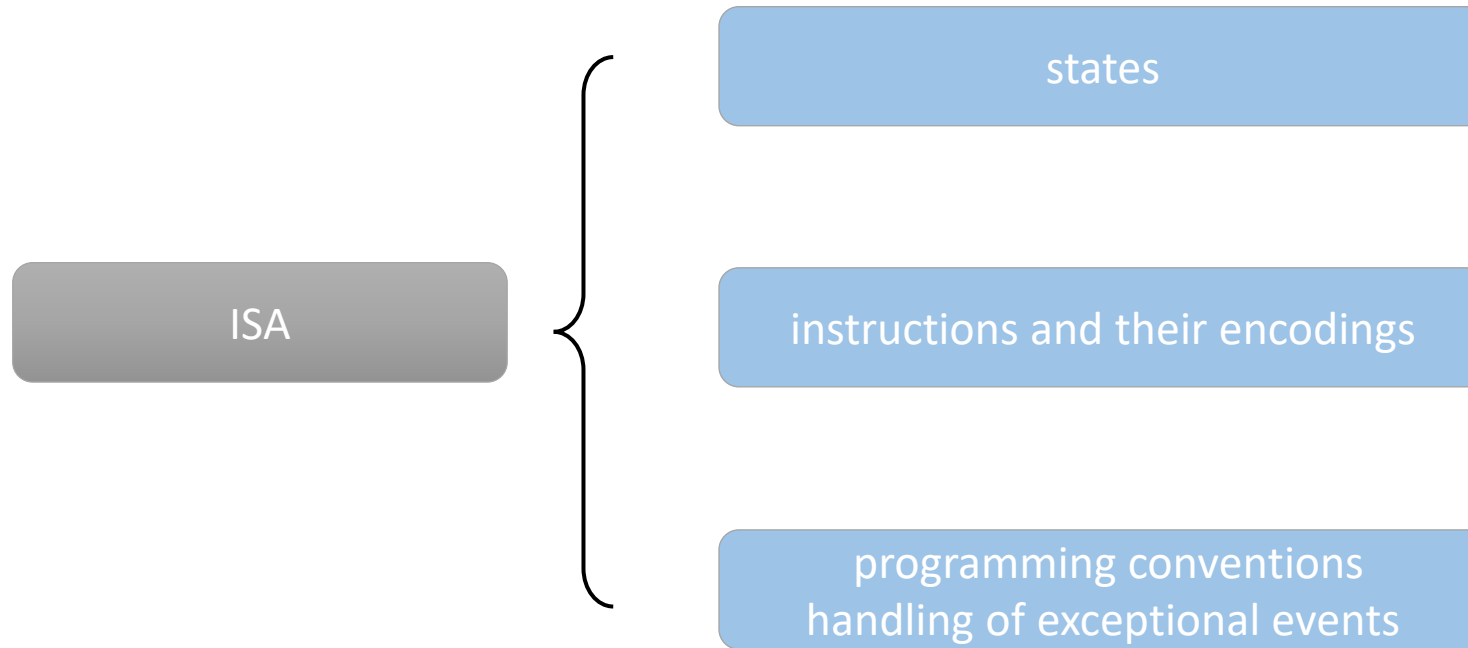
	描述
(1)	指令机器码长度固定
(2)	指令类型多、功能丰富
(3)	不采用条件码
(4)	实现同一功能，需要的汇编代码较多
(5)	译码电路复杂
(6)	访存模式多样
(7)	参数、返回地址都使用寄存器进行保存
(8)	x86-64
(9)	MIPS
(10)	广泛用于嵌入式系统
(11)	已知某个体系结构使用 <code>add R1,R2,R3</code> 来完成加法运算。当要将数据从寄存器 <code>S</code> 移动至寄存器 <code>D</code> 时，使用 <code>add S,#ZR,D</code> 进行操作（ <code>#ZR</code> 是一个恒为 0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

RISC or CISC?

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 <code>S</code> 移动至寄存器 <code>D</code> 时，使用 <code>add S, #ZR, D</code> 进行操作（ <code>#ZR</code> 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 <code>A</code> 为基地址，以另一个固定的寄存器 <code>B</code> 为偏移量，在 <code>A</code> 对应的数组中取出下标为 <code>B</code> 的项的内容，放回寄存器 <code>A</code> 中。		✓

Y86-64 ISA



Introduction

Y86-64 ISA

Programmer-Visible States

Instructions

Exceptions

Complement

Programmer-Visible States

states

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

RF: 程序寄存器

PC: 程序计数器

ZF	SF	OF
----	----	----

CC: 条件码

Stat: 程序状态

DMEM: 内存 用虚拟地址引用内存位置

Instructions

instructions and their encodings

halt	0	0			
nop	1	0			
rrmovq rA, rB	2	0	rA	rB	
cmovXX rA, rB	2	fn	rA	rB	
irmovq V, rB	3	0	F	rB	V
rmmovq rA, D(rB)	4	0	rA	rB	D
mrmmovq 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	



Introduction

Y86-64 ISA

Programmer-Visible States

Instructions

Exceptions

Complement

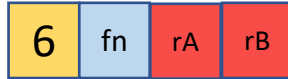
Instructions

instructions and their encodings

cmovXX rA, rB



OPq rA, rB



jXX Dest



icode



ifun



register ID



constant

fn	cmovXX	OPq	jXX
0	rrmovq	addq	jmp
1	cmovle	subq	jle
2	cmovl	andq	jl
3	cmove	xorq	je
4	cmovne		jne
5	cmovge		jge
6	cmovg		jg

Introduction

Y86-64 ISA

Programmer-Visible States

Instructions

Exceptions

Complement

Register ID

rA	rB
----	----


0:%rax	4:%rsp	8:%r8	C:%r12
1:%rcx	5:%rbp	9:%r9	D:%r13
2:%rdx	6:%rsi	A:%r10	E:%r14
3:%rbx	7:%rdi	B:%r11	F

 register ID

- register file: 寄存器堆（寄存器文件）
 - 存储在CPU中的随机访问存储器
 - 以register ID作为地址
- register ID: 寄存器标识符, 4 bit

Constant Word



 constant

- 均为8字节 (word)
- 在内存中为**小端法**编码，因此在指令对应的字节编码中应当从低位字节到高位字节（二进制目标代码存储在内存中）

401d9f: 69 c0 **72 d7 00 00** imul \$0xd772,%eax,%eax

Details

pushq %rsp



irmovq \$8,%rbx

irmovq \$8,%rbx

subq %rbx,%rsp

rmmovq %rsp,-8(%rsp)

rmmovq %rsp,(%rsp)

subq %rbx,%rsp

popq %rsp



irmovq \$8,%rax

irmovq \$8,%rax

addq %rax,%rsp

mrmovq (%rsp),%rsp

mrmovq -8(%rsp),%rsp

addq %rax,%rsp

```

1  .text
2  .globl pushtest
3  pushtest:
4      movq    %rsp, %rax    Copy stack pointer
5      pushq   %rsp          Push stack pointer
6      popq    %rdx          Pop it back
7      subq    %rdx, %rax    Return 0 or 4
8      ret

```

```

1  .text
2  .globl poptest
3  poptest:
4      movq    %rsp, %rdi    Save stack pointer
5      pushq   $0xabcd      Push test value
6      popq    %rsp          Pop to stack pointer
7      movq    %rsp, %rax    Set popped value as return value
8      movq    %rdi, %rsp    Restore stack pointer
9      ret

```


Exceptions

programming conventions
handling of exceptional events



Stat: 程序状态

值	名字	含义
1	AOK	正常操作
2	HLT	遇到器执行 halt 指令
3	ADR	遇到非法地址
4	INS	遇到非法指令

HLT: 处理器执行了一条halt指令

ADR: 处理器试图从（向）非法内存地址读（写） 取指令/读写数据

INS: 遇到非法指令

停止处理器执行指令/调用异常处理程序

Y86-64 <-> x86-64

- 不支持第二变址寄存器和任何寄存器值的伸缩，只支持简单寻址模式 D(rB)
- 不允许将立即数传送到内存 **immovq**
- Opq指令只对寄存器数据操作
- 没有pc相对寻址方式，使用绝对地址，因此代码在内存中不能被迁移
- 允许程序使用halt指令
- 常数值只能被编码为8个字节
- 无leaq指令
- 指令集数据类型、指令和寻址方式更少
- 指令的字节级编码简单，机器代码没有那么紧凑
- CPU译码逻辑更简单

.....

总的来看，Y86-64是在x86-64的基础上利用RISC的思路进行简化后得到的ISA

写一个简单的Y86-64汇编程序

```
1  #初始化
2  |      .pos 0                #程序从地址0x0开始
3  |      irmovq stack,%rsp     #初始化栈指针
4  |      call main
5  |      halt
6
7  #声明全局变量
8  |      .align 8              #8字节对齐
9  array:
10 |      ...
11
12 #main函数
13 main:
14 |      ...
15 |      ret
16
17 #各种过程 (函数)
18 f1:
19 |      ...
20 |      ret
21 f2:
22 |      ...
23 |      ret
24 ...
25
26 #运行时栈
27 |      .pos 0x200             #栈从0x200处开始, 向低地址增长
28 stack:
```

- 设置栈指针
- 初始化数据
- 指定程序在内存中的起始位置
- 指定栈在内存中的起始位置
- 汇编器伪指令不能缺省

Y86-64工具链

```
unix> yas len.yas
```

```

0x054: | len:
0x054: 30f80100000000000000 | irmovq $1, %r8      # Constant 1
0x05e: 30f90800000000000000 | irmovq $8, %r9      # Constant 8
0x068: 30f00000000000000000 | irmovq $0, %rax     # len = 0
0x072: 50270000000000000000 | mrmovq (%rdi), %rdx  # val = *a
0x07c: 6222 | andq %rdx, %rdx     # Test val
0x07e: 73a000000000000000 | je Done             # If zero, goto Done
0x087: | Loop:
0x087: 6080 | addq %r8, %rax      # len++
0x089: 6097 | addq %r9, %rdi      # a++
0x08b: 502700000000000000 | mrmovq (%rdi), %rdx  # val = *a
0x095: 6222 | andq %rdx, %rdx     # Test val
0x097: 748700000000000000 | jne Loop            # If !0, goto Loop
0x0a0: | Done:
0x0a0: 90 | ret

```

- 运行汇编器yas，汇编我们手写的汇编代码
- 可读机器级代码->二进制目标代码
- 输出结果是ASCII码格式，方便阅读（类似objdump反汇编得到的结果）

```
unix> yis len.yo
```

```

Stopped in 33 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax: 0x0000000000000000 0x0000000000000004
%rsp: 0x0000000000000000 0x0000000000000100
%rdi: 0x0000000000000000 0x0000000000000038
%r8: 0x0000000000000000 0x0000000000000001
%r9: 0x0000000000000000 0x0000000000000008

Changes to memory:
0x00f0: 0x0000000000000000 0x0000000000000053
0x00f8: 0x0000000000000000 0x0000000000000013

```

- 运行指令集模拟器yis，模拟Y86-64可执行程序执行
- 不模拟处理器层面的实现
- 输出在模拟过程中被改变了的寄存器或内存部分，显示原始值和最终值

The diagram illustrates the layers of computer architecture and their associated concepts. On the left, a vertical stack of six boxes represents the layers, from top to bottom: Application Program (green), Compiler (green) and OS (green), ISA (grey), CPU Design (yellow), Circuit Design (yellow), and Chip Layout (yellow). A large curly bracket on the right groups the layers from Compiler/OS down to Chip Layout. To the right of this bracket are three blue boxes representing concepts: states, instructions and their encodings, and programming conventions handling of exceptional events.

Layer	Concepts
Application Program	states instructions and their encodings programming conventions handling of exceptional events
Compiler	
OS	
ISA	
CPU Design	
Circuit Design	
Chip Layout	

RF: 程序寄存器

Stat: 程序状态

DMEM: 内存 用虚拟地址引用内存位置



HLT: 处理器执行了一条halt指令

ADR: 处理器试图从(向)非法内存地址读(写) 取指令/读写数据

INS: 遇到非法指令

Logic and Hardware Unit Design

HCL

- 硬件控制语言 (Hardware Control Language)
- 语法与C中逻辑表达式很相似, 但也有不同
 - HCL不存在部分求值规则 (短路表达式)
- 我们只用HCL表达硬件设计的**控制部分 (组合逻辑)**
- 不关注大部分硬件单元的具体实现, 把它们当作黑箱
- 逻辑合成程序根据HCL可以生成有效的电路设计

HCL

Units

Combination

Hardware

$(a \& \& !a) \& \& \text{func}(b, c)$

硬件单元

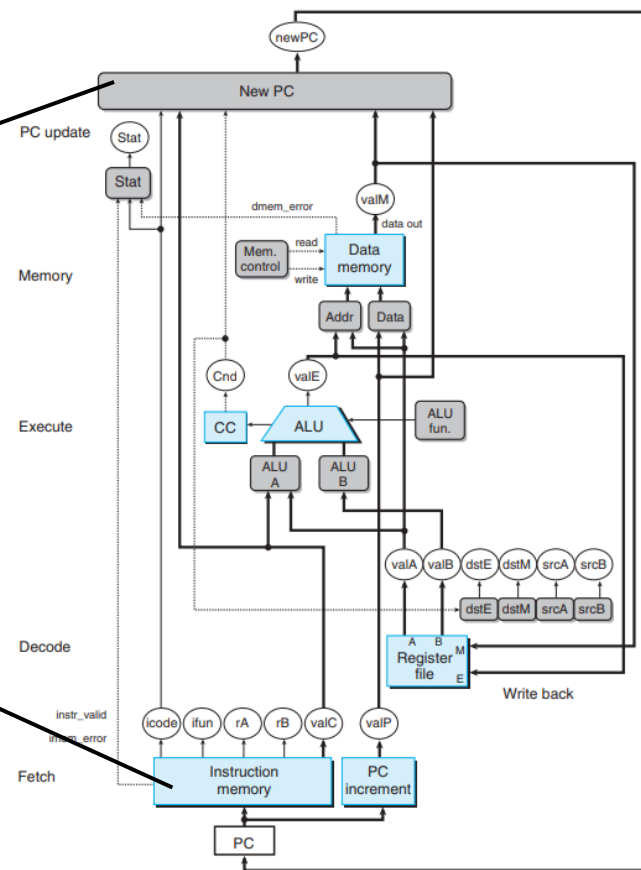
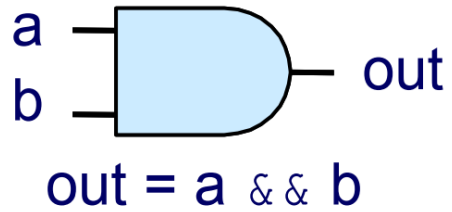


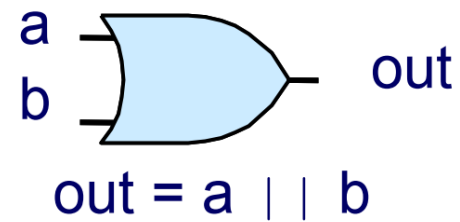
Figure 4.23 Hardware structure of SEQ, a sequential implementation. Some of the control signals, as well as the register and control word connections, are not shown.

Units

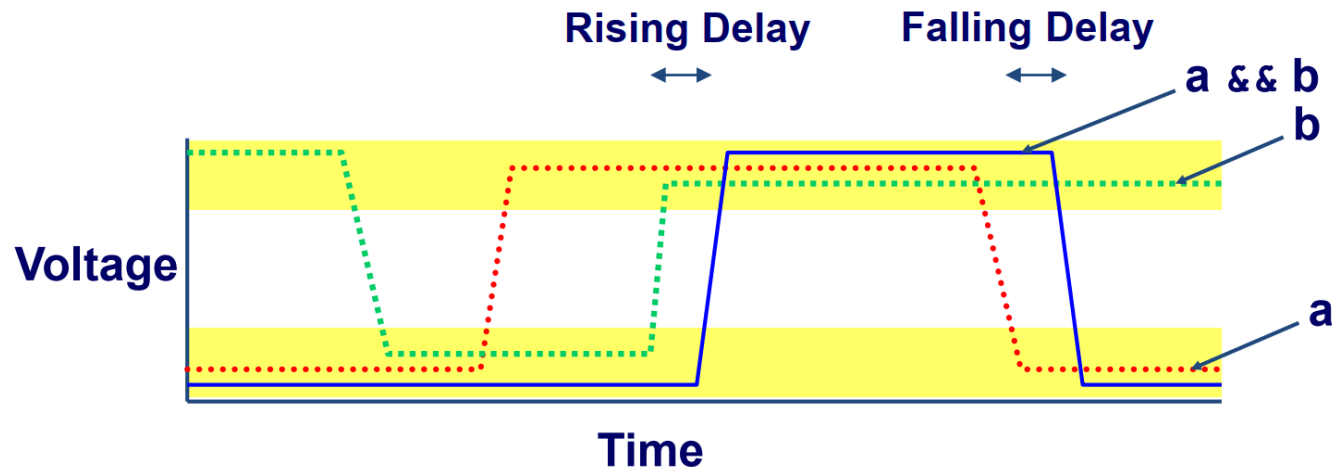
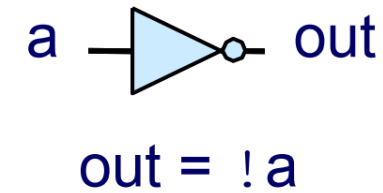
And



Or



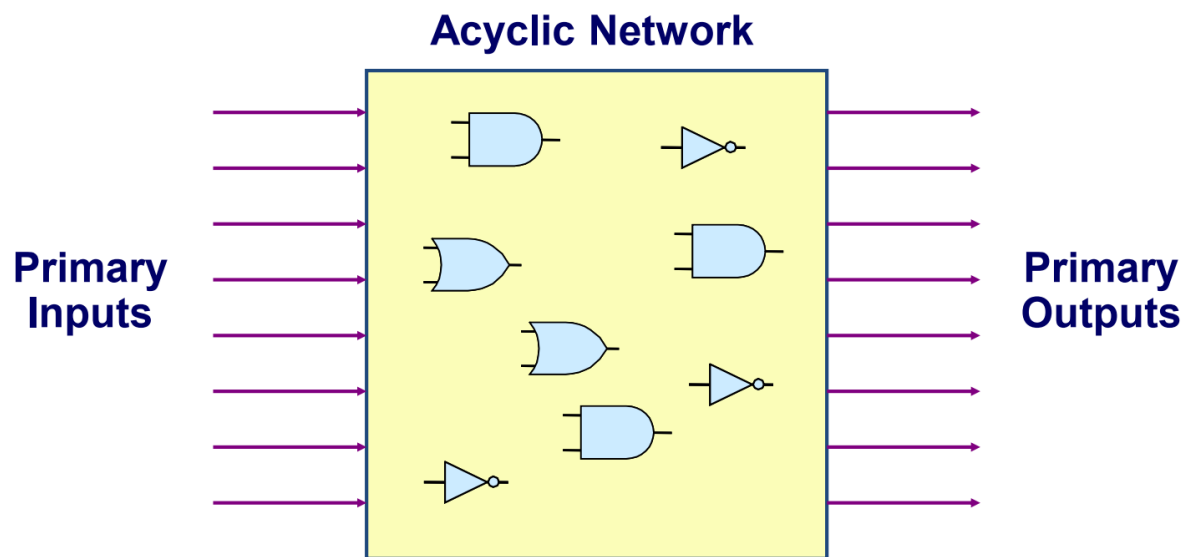
Not



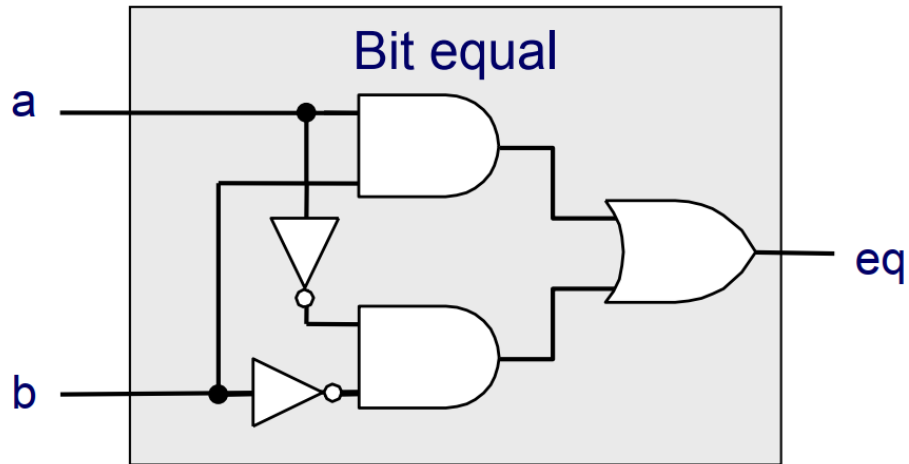
- 输出与输入几乎同步变化
- 有微小的延迟
- 单个逻辑门没有存储功能

Combinational Circuits

- 组合电路/组合逻辑
- 由很多逻辑门组合起来的计算块（computational block）网络
 - 三类输入
 - ✓ 系统输入（主输入）
 - ✓ 某个存储器单元的输出
 - ✓ 某个逻辑门的输出
 - 逻辑门的输出不能连接在一起：矛盾/短路
 - 网络中无回路，只能计算布尔函数
 - 没有存储功能，实现与或非的逻辑运算



Bit Equality

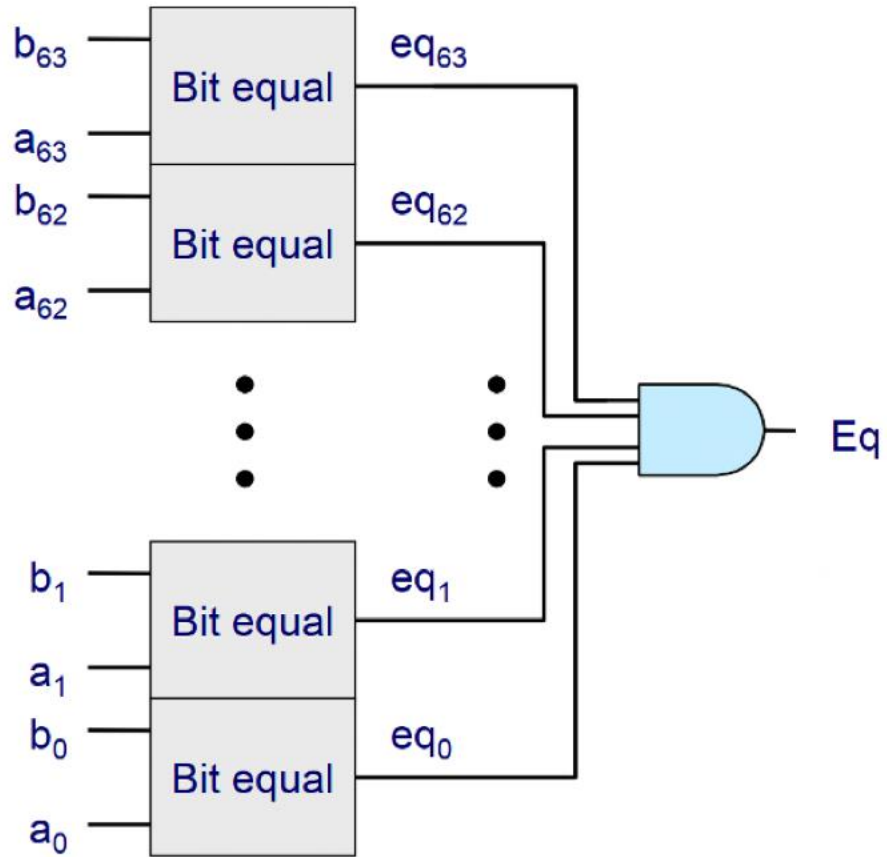


HCL Expression

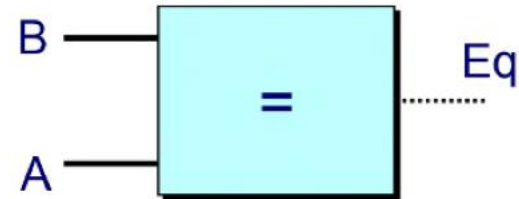
```
bool eq = (a&&b) || (!a&&!b)
```

- bool定义位级信号
- 根据电路直接翻译即可得到HCL表达式

Word Equality



Word-Level Representation



HCL Representation

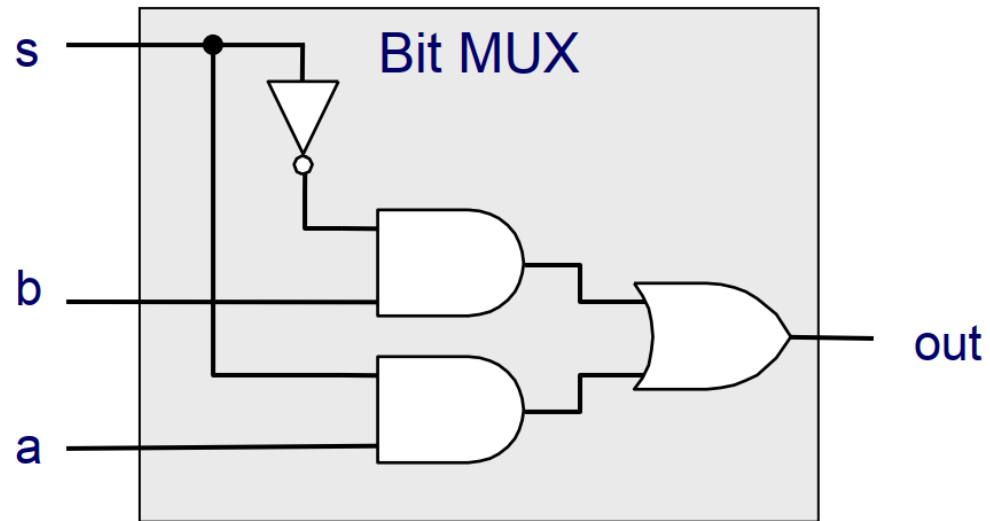
```
bool Eq = (A == B)
```

```
int A,B;
```

HCL中将所有字级信号都声明为int类型，不区分字长

Bit Multiplexor

位多路复用器



HCL Expression

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

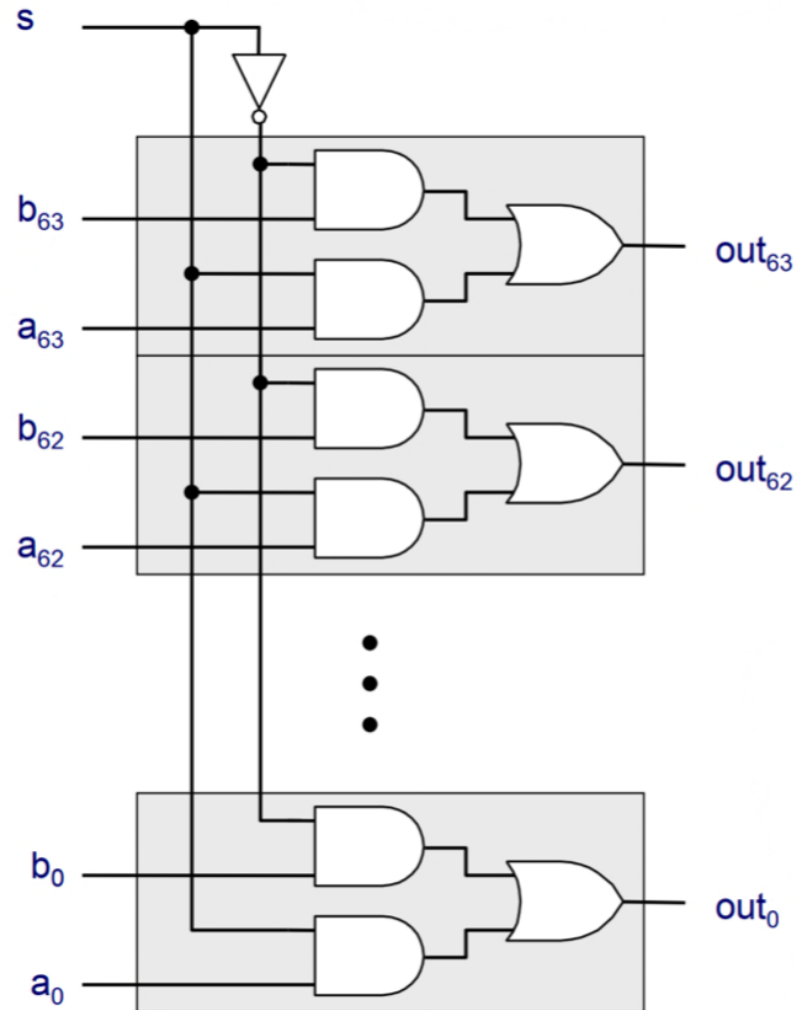
Word Multiplexor

字级多路复用器

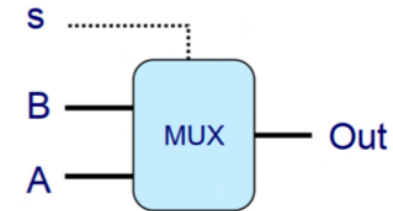
HCL 情况表达式

```
word out = [
    select1 : expr1;
    select2 : expr2;
    ...
    selectk : exprk;
];
```

顺序求值，选择第一个求
值为1的情况



Word-Level Representation



HCL Representation

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

Word Multiplexor

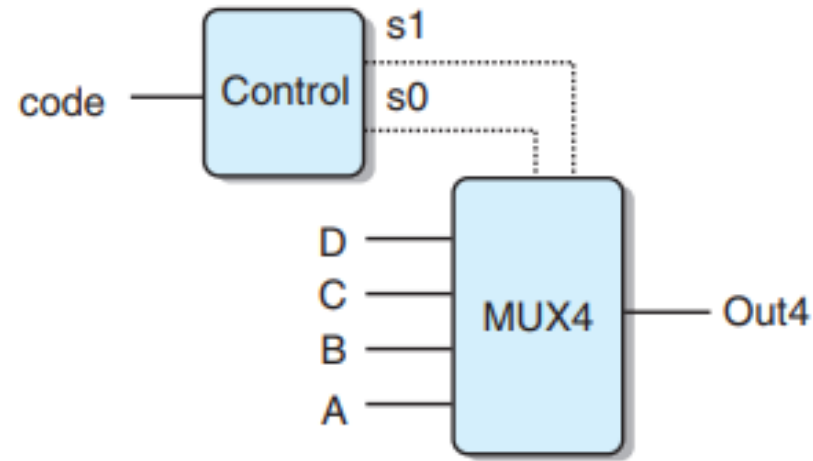
字级四路复用器

HCL 集合关系

```
bool s = code in { item1, item2, ... };
```

```
bool s1 = code in {2,3};
```

```
bool s0 = code in {1,3};
```



```
word Out4=[  
    code == 1 : A;  
    code == 2 : B;  
    code == 3 : C;  
    code == 4 : D;  
];
```

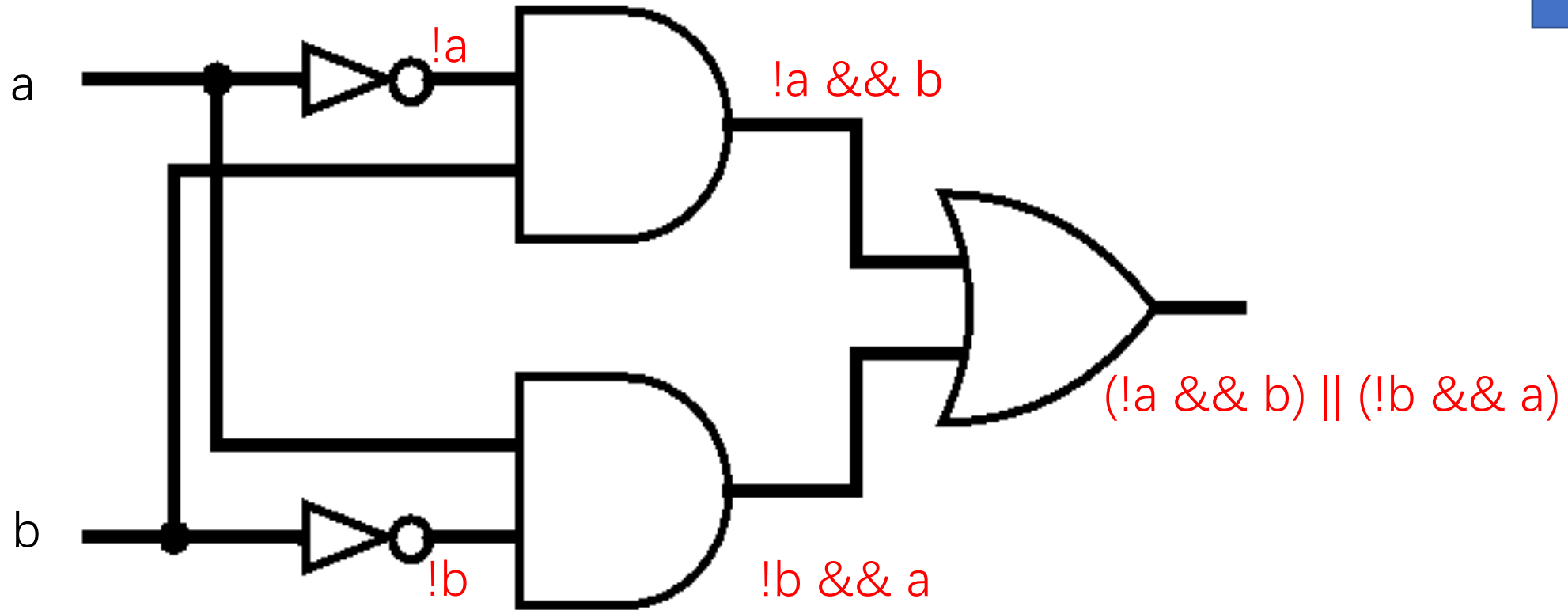
Combinational Circuits

HCL

Units

Combination

Hardware



Hardware Units

- 硬件单元
- 一般不关注其硬件实现，当作黑箱即可
- 介绍三种硬件单元的大致原理（显然都不是组合逻辑）
 - ALU
 - 时钟寄存器
 - 随机访问存储器
 - ✓ 寄存器文件
 - ✓ 虚拟内存系统

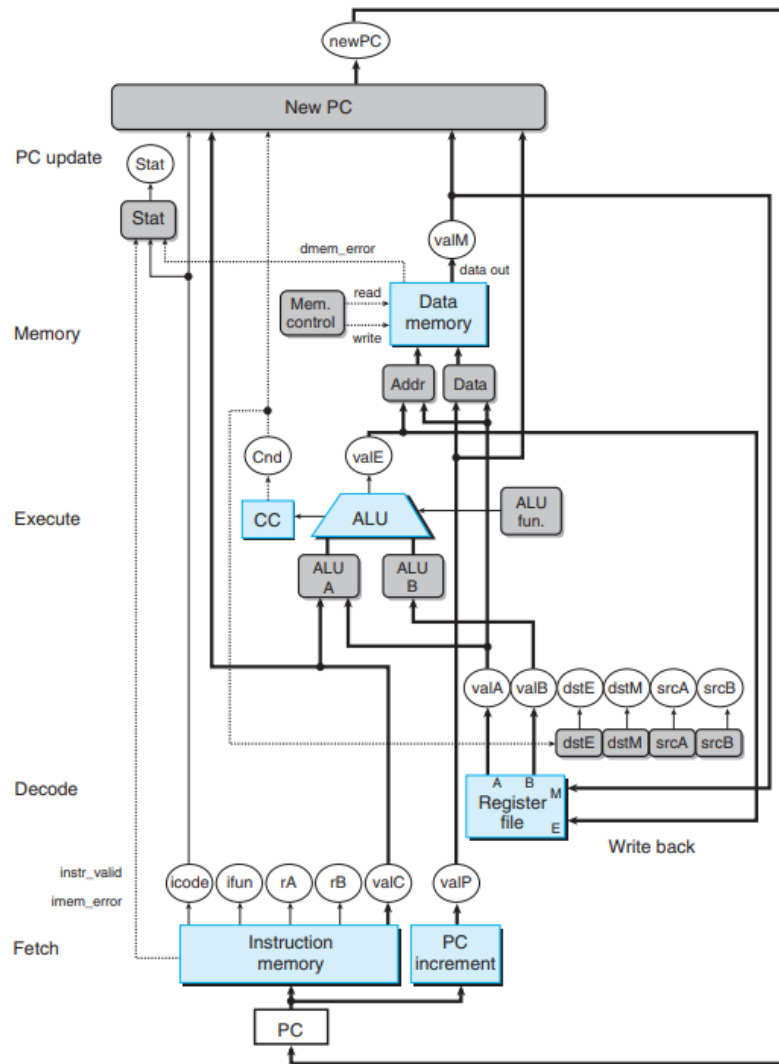


Figure 4.23 Hardware structure of SEQ, a sequential implementation. Some of the control signals, as well as the register and control word connections, are not shown.

ALU

HCL

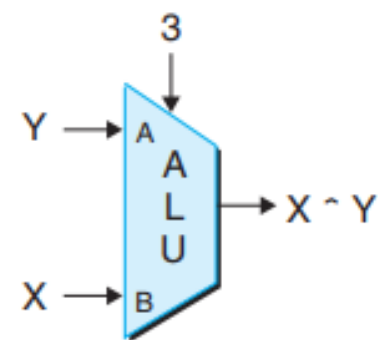
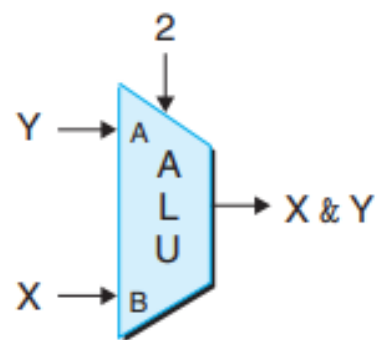
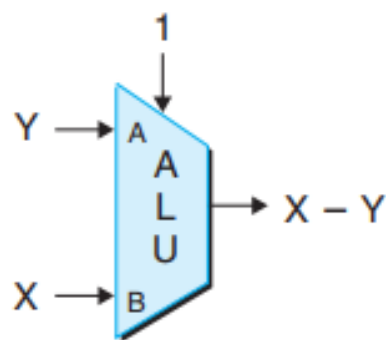
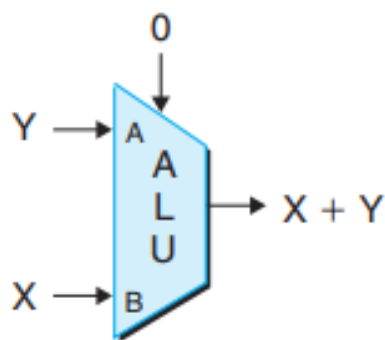
Units

Combination

Hardware

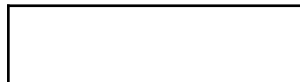
fn	OPq
0	addq
1	subq
2	andq
3	xorq

ALU也是组合逻辑，但我们不关注其具体实现



Clocked Register

- 时钟寄存器
- 存储单个位/字
- 受周期性的时钟信号控制
- 不同于汇编中的寄存器，更像是一种电路实现的原理
- 作为输入与输出之间的屏障，有短暂的存储功能



PC: 程序计数器

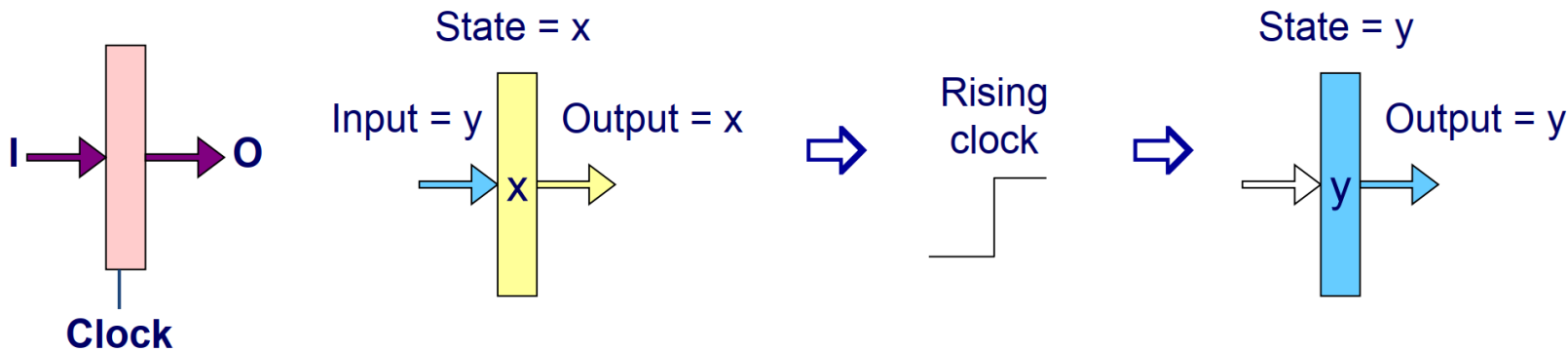
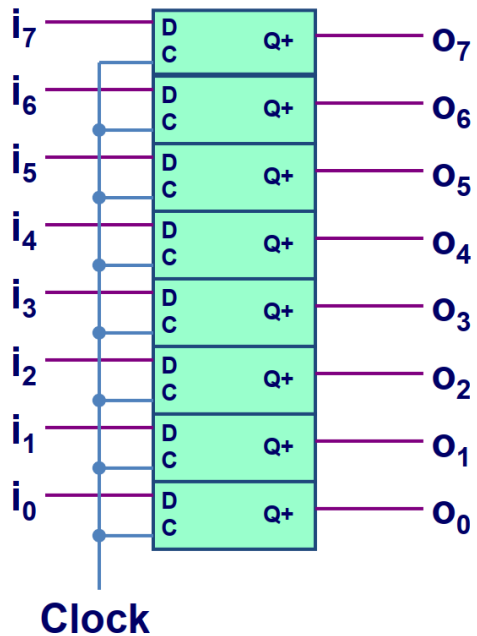


CC: 条件码

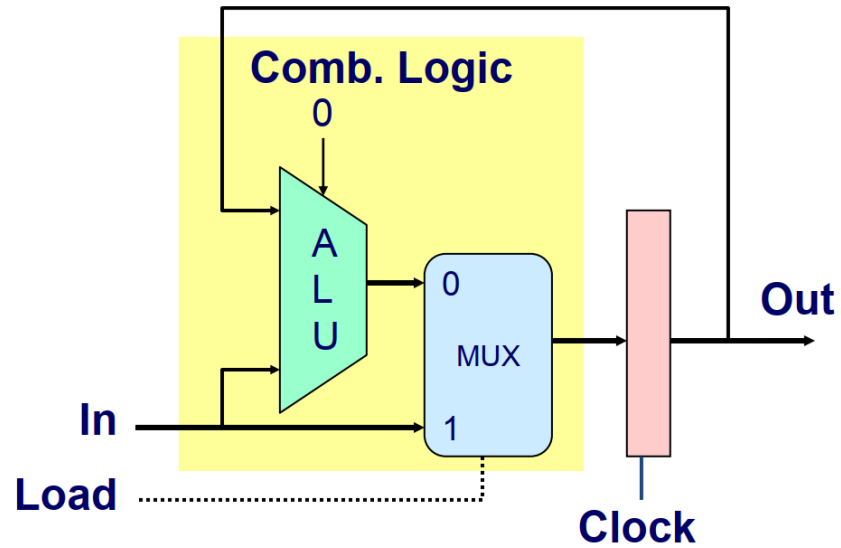


Stat: 程序状态

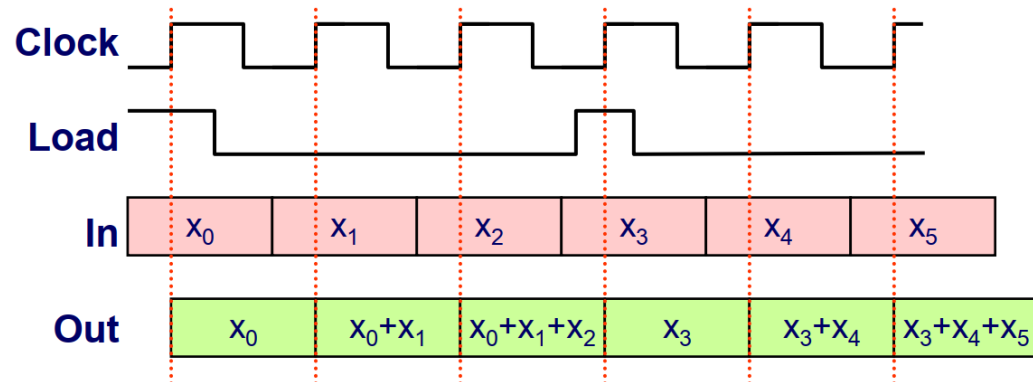
Structure



Clocked Register

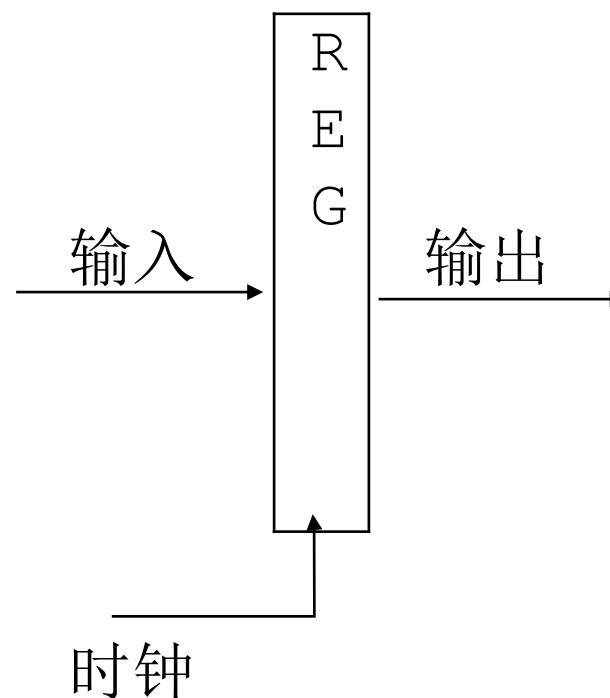
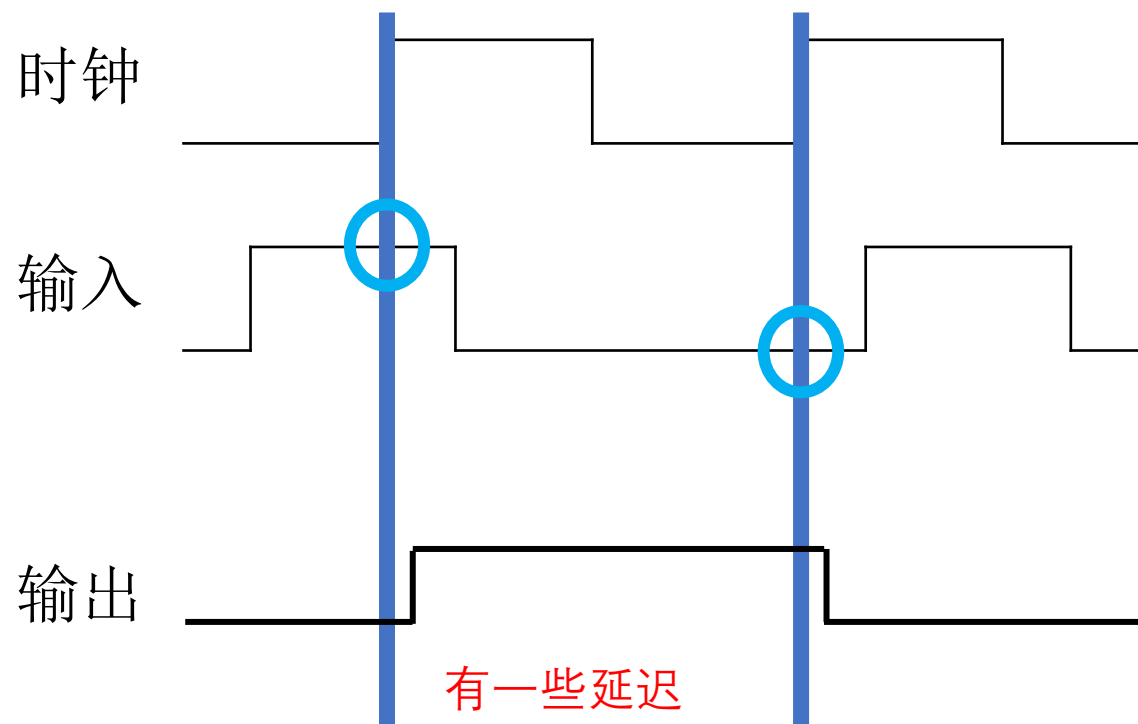


- Accumulator circuit
- Load or accumulate on each cycle

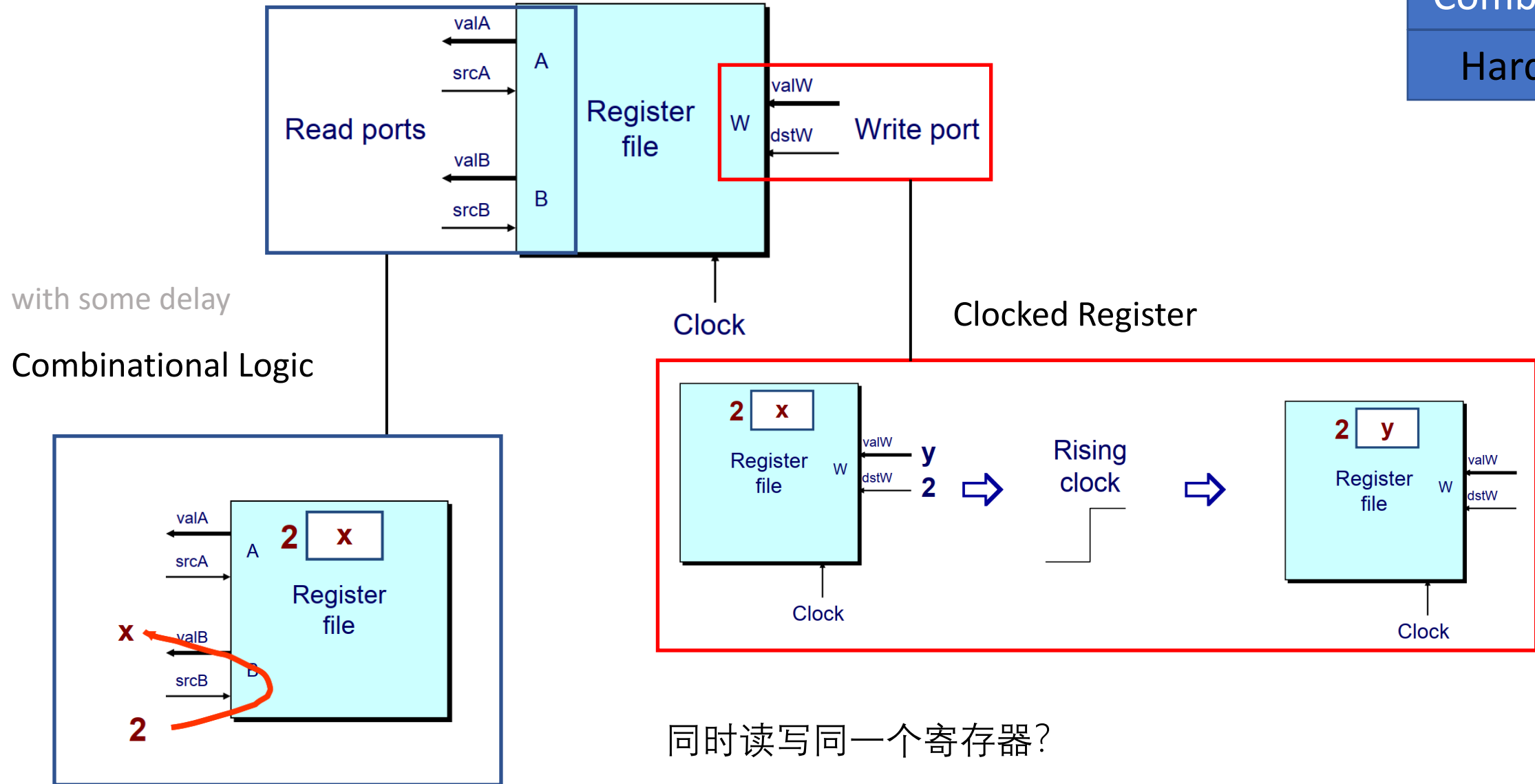


Clocked Register

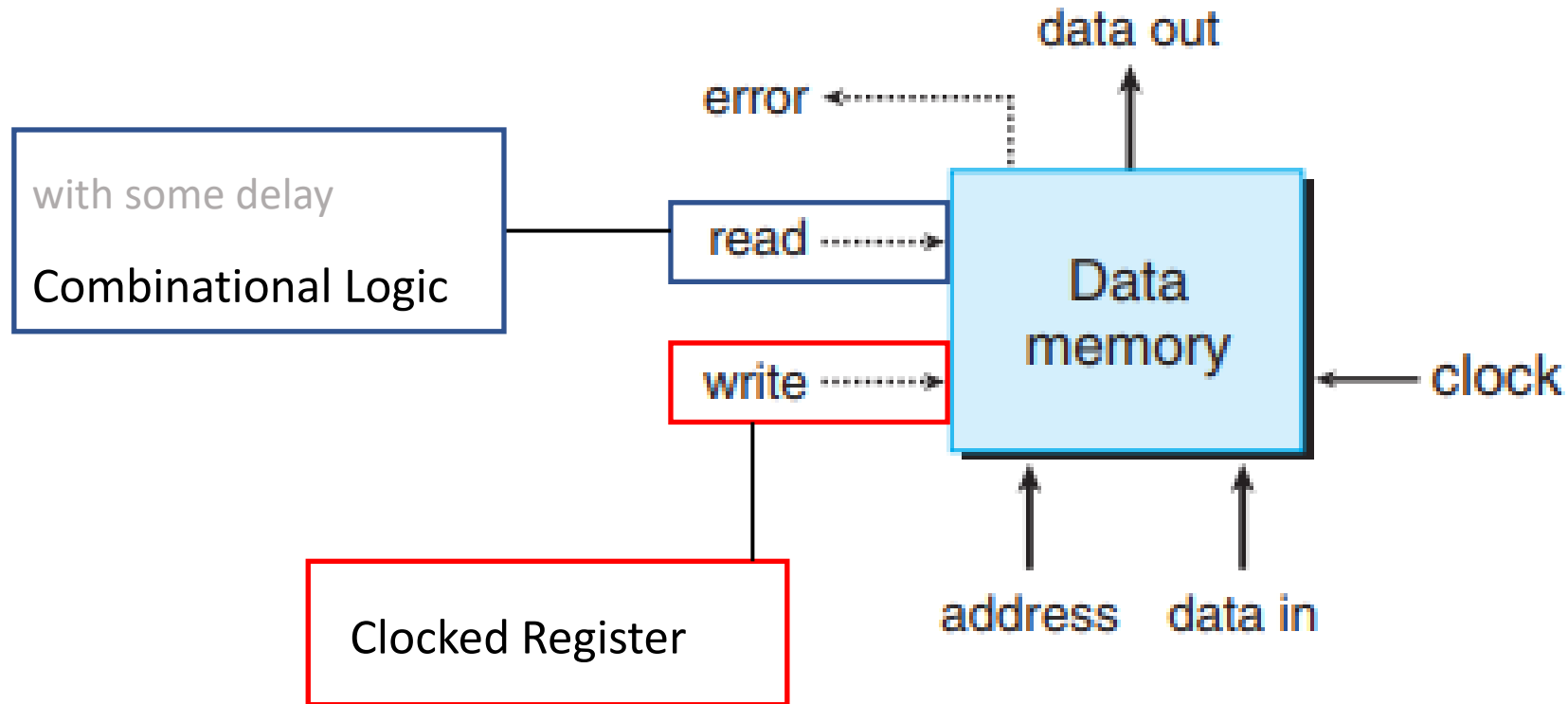
在时钟上升沿采样。假设一开始输出为0



Random Access Memories: Register File



Random Access Memories: Memory



Thanks for watching!