中国科学院大学
University of Chinese Academy of Sciences

CS101

# 系统思维-2

## 控制抽象
## 模块化

zxu@ict.ac.cn

# 提纲



- 系统思维概述
- 抽象化
  - 控制抽象（含模块抽象）
- 模块化
  - 模块、信息隐藏原理
  - 组合电路
  - 时序电路
- 无缝衔接

课件中可能包含素材引用，特此致谢！

程序
进程
**指令**
**冯氏结构**
**指令流水线**
**时序电路**
**组合电路**

# 1. 控制抽象

- 控制抽象指明计算过程的操作步骤的执行顺序
  - 主要有五类：运算符优先级、顺序、条件跳转、循环、子程序调用
    - 子程序调用（函数调用，包含递归调用）提供了模块抽象
- 表达式中的运算符优先级（Precedence）
  - 表达式5*3<20/5<<2+3的求值结果是true
    - 等同于 (5*3)<(((20/5)<<2)+3)，即15<((4<<2)+3)；15<19成立
  - 表达式5*3<20/(5<<2)+3的求值结果是false
    - 等同于15<20/20+3，即15<(20/20+3)；15<4不成立

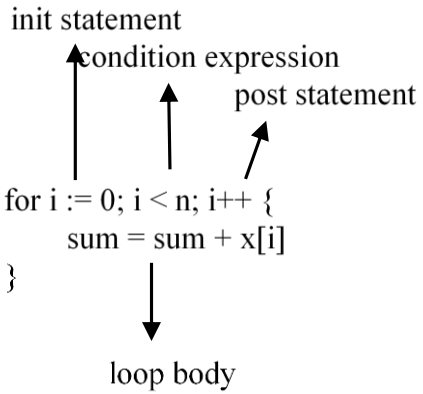
| 优先级 | 运算符 | 运算符含义 |
|---|---|---|
| 6 最高 | - ! | 负号，逻辑非 |
| 5 | * / % << >> & | 乘、除、求余、左移、右移，按位与 |
| 4 | + - \| ^ | 加、减、按位或、按位异或 |
| 3 | == != < <= > >= | 等于、不等于、小于、小于或等于、大于、大于或等 |
| 2 | && | 逻辑与 |
| 1最低 | \|\| | 逻辑或 |

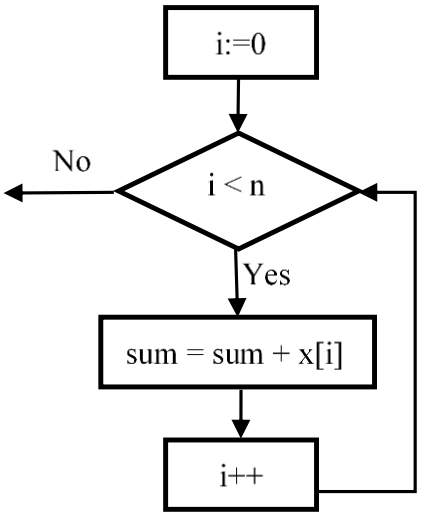
# 顺序、条件、循环、函数

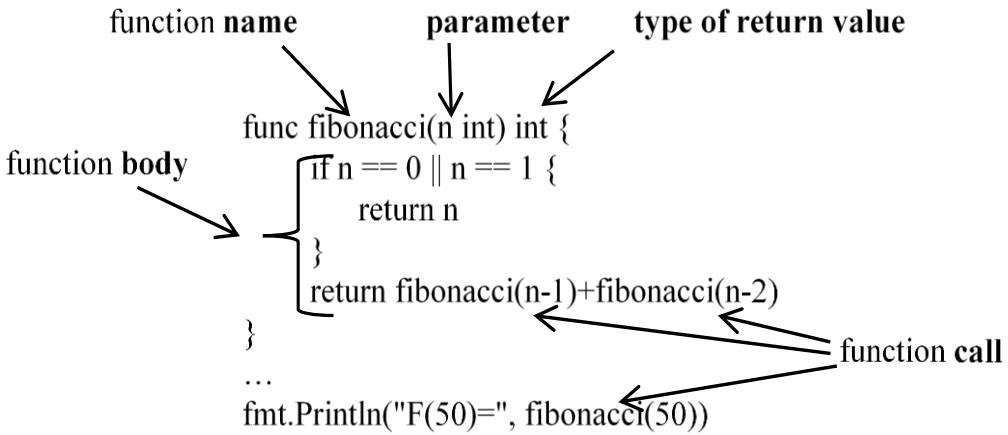
- 顺序（sequence）
  - 按代码语法顺序执行语句
  - 缺省控制流抽象
- 条件语句、选择语句（selection，conditional）
  - if-then-else语句

```
if i<7 {
    fmt.Println(i)
}
```

- 循环语句（for loop）
  - 重复迭代执行一段代码
- 函数抽象（function）
  - 声明一次，可多次调用的代码块

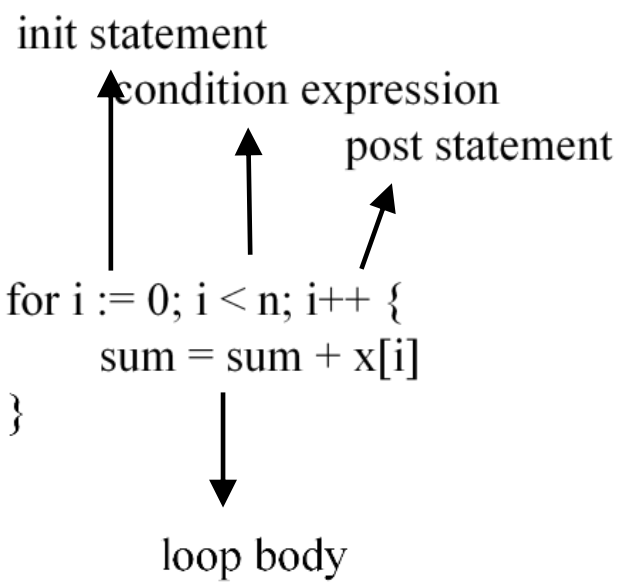


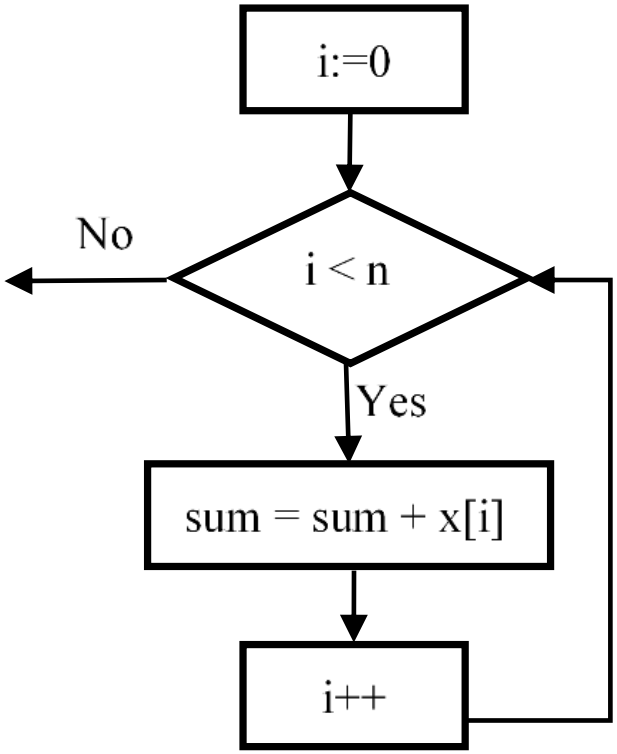



# 循环

- 重点是掌握程序流图（flow chart）
- 注意事项
  - i<n是表达式，不是语句；i++是语句，不是表达式
  - 常见错误：无穷循环




初始语句

条件表达式

循环体

后语句

# 下述代码分别输出什么？

- 完整代码

```
var A [500000000]int    //数组A有5亿个元素
// 将5亿个家庭用电量放入数组A的代码
TotalKWH := 0
for   i := 0;   i<50000000;    i++   {
   TotalKWH = TotalKWH + A[i]
}
fmt.Println(TotalKWH / 500000000)
```

- 去掉初始语句**i := 0**

```
var A [500000000]int    //数组A有5亿个元素
// 将5亿个家庭用电量放入数组A的代码
TotalKWH := 0
for ;   i<50000000;    i++   {
   TotalKWH = TotalKWH + A[i]
}
fmt.Println(TotalKWH / 500000000)
```

- 去掉条件表达式**i<50000000**

```
var A [500000000]int    //数组A有5亿个元素
// 将5亿个家庭用电量放入数组A的代码
TotalKWH := 0
for   i := 0; ;    i++   {
   TotalKWH = TotalKWH + A[i]
}
fmt.Println(TotalKWH / 500000000)
```

- 去掉后语句**i++**

```
var A [500000000]int    //数组A有5亿个元素
// 将5亿个家庭用电量放入数组A的代码
TotalKWH := 0
for   i := 0;   i<50000000; {
   TotalKWH = TotalKWH + A[i]
}
fmt.Println(TotalKWH / 500000000)
```

# 下述代码分别输出什么？

- 完整代码

```
var A [500000000]int    //数组A有5亿个元素
// 将5亿个家庭用电量放入数组A的代码
TotalKWH := 0
for   i := 0;   i<50000000;    i++   {
   TotalKWH = TotalKWH + A[i]
}
fmt.Println(TotalKWH / 500000000)
```

- 去掉初始语句**i := 0**
  - 编译错误

```
var A [500000000]int    //数组A有5亿个元素
// 将5亿个家庭用电量放入数组A的代码
TotalKWH := 0
for ;   i<50000000;    i++   {
   TotalKWH = TotalKWH + A[i]
}
fmt.Println(TotalKWH / 500000000)
```

- 去掉条件表达式**i<50000000**
  - 运行时错误，数组越界

```
var A [500000000]int    //数组A有5亿个元素
// 将5亿个家庭用电量放入数组A的代码
TotalKWH := 0
for   i := 0; ;    i++   {
   TotalKWH = TotalKWH + A[i]
}
fmt.Println(TotalKWH / 500000000)
```

- 去掉后语句**i++**
  - 无穷循环

```
var A [500000000]int    //数组A有5亿个元素
// 将5亿个家庭用电量放入数组A的代码
TotalKWH := 0
for   i := 0;   i<50000000; {
   TotalKWH = TotalKWH + A[i]
}
fmt.Println(TotalKWH / 500000000)
```

# 下述代码分别输出什么？

- 完整代码
  - 五亿家庭的平均用电量

```
var A [500000000]int    //数组A有5亿个元素
// 将5亿个家庭用电量放入数组A的代码
TotalKWH := 0
for   i := 0;   i<50000000;    i++   {
   TotalKWH = TotalKWH + A[i]
}
fmt.Println(TotalKWH / 500000000)
```

- 去掉
  - 初始语句**i := 0，**
  - 条件表达式**i<50000000**
  - 后语句**i++**

```
var A [500000000]int    //数组A有5亿个元素
// 将5亿个家庭用电量放入数组A的代码
TotalKWH := 0
for {
   TotalKWH = TotalKWH + A[i]
}
fmt.Println(TotalKWH / 500000000)
```

# 下述代码分别输出什么？

- 完整代码
  - 五亿家庭的平均用电量

```
var A [500000000]int    //数组A有5亿个元素
// 将5亿个家庭用电量放入数组A的代码
TotalKWH := 0
for   i := 0;   i<50000000;    i++   {
   TotalKWH = TotalKWH + A[i]
}
fmt.Println(TotalKWH / 500000000)
```

- 去掉
  - 初始语句**i := 0，**
  - 条件表达式**i<50000000**
  - 后语句**i++**

```
var A [500000000]int    //数组A有5亿个元素
// 将5亿个家庭用电量放入数组A的代码
TotalKWH := 0
i := 0
for {
   TotalKWH = TotalKWH + A[i]
}
fmt.Println(TotalKWH / 500000000)
```

  - **Go语言的for循环很灵活**
    - **可实现其他类循环（如while）**

# 下述代码分别输出什么？

- 完整代码
  - 五亿家庭的平均用电量

```
var A [500000000]int    //数组A有5亿个元素
// 将5亿个家庭用电量放入数组A的代码
TotalKWH := 0
for   i := 0;   i<50000000;    i++   {
   TotalKWH = TotalKWH + A[i]
}
fmt.Println(TotalKWH / 500000000)
```

- 去掉
  - 初始语句**i := 0，**
  - 条件表达式**i<50000000**
  - 后语句**i++**

  - 例如，完整代码的另一种写法

```
var A [500000000]int    //数组A有5亿个元素
// 将5亿个家庭用电量放入数组A的代码
TotalKWH := 0
i := 0
for {
   if i>=50000000 { break }
   TotalKWH = TotalKWH + A[i]
   i++
}
fmt.Println(TotalKWH / 500000000)
```

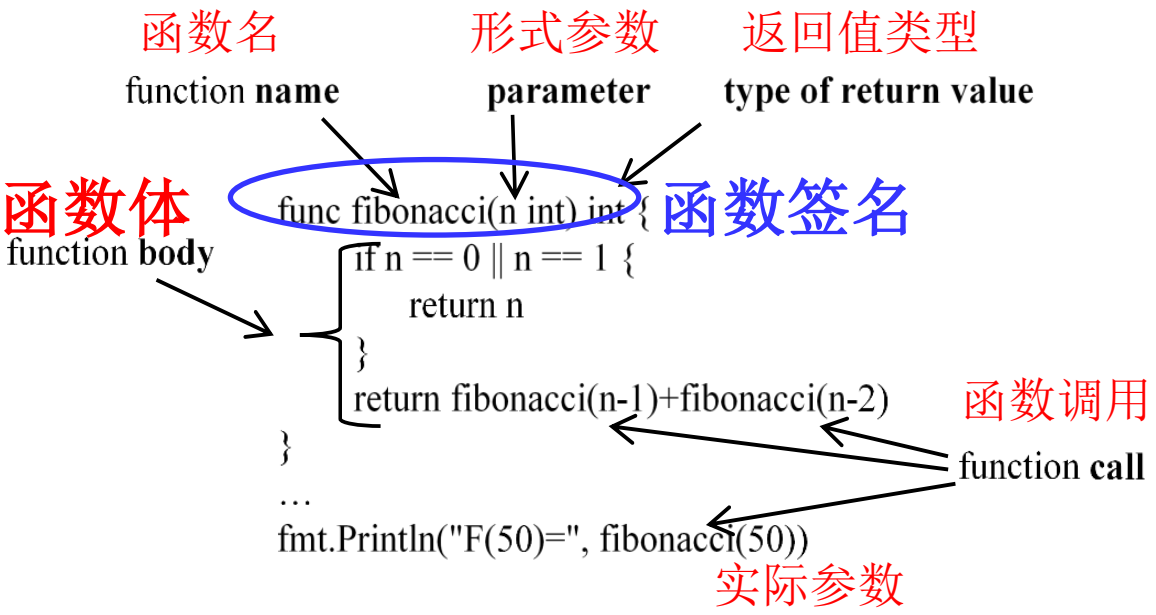
# 函数抽象

- 声明一次，可多次多处调用的代码块
  - 代码示例中，函数fibonacci在三处被调用
  - 运行时，函数fibonacci被调用了$O(2^{50})$次

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

- 函数声明包括函数签名与函数体

- 函数可被递归调用

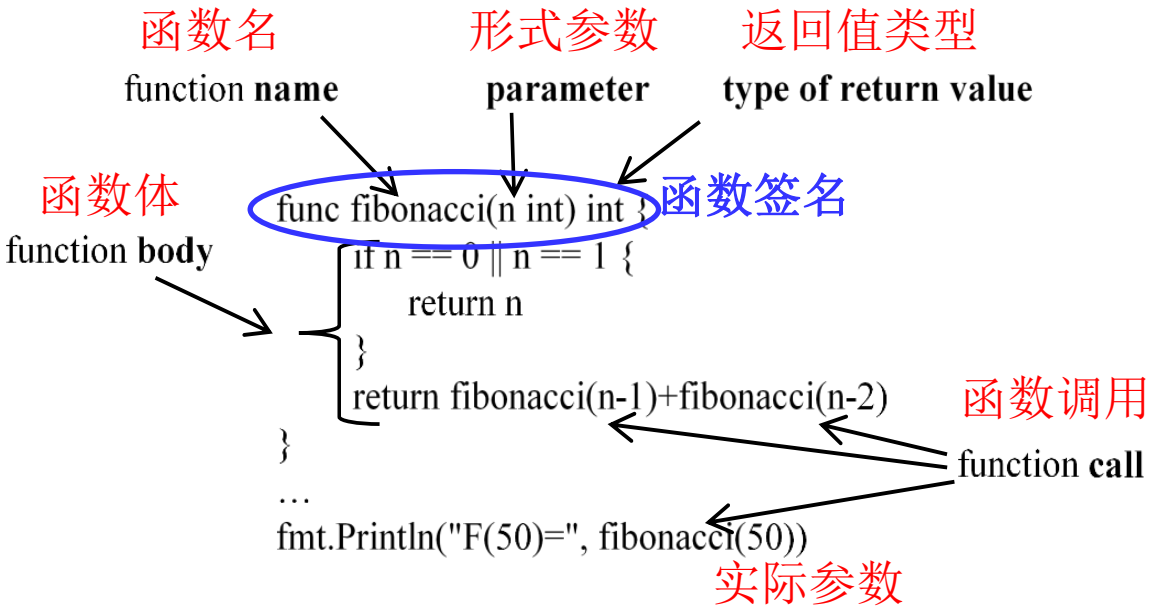


# 函数抽象

- 声明一次，可多次多处调用的代码块
  - 代码示例中，函数fibonacci在三处被调用
  - 运行时，函数fibonacci被调用了$O(2^{50})$次

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

- 函数声明包括函数签名与函数体
- 函数可被递归调用
- 注意事项
  - 可能出现指数复杂度
  - 递归调用必需停止
- 常见错误
  - 基线缺失、循环调用



# 函数抽象

- 声明一次，可多次多处调用的代码块
  - 代码示例中，函数fibonacci在三处被调用
  - 运行时，函数fibonacci被调用了$O(2^N)$次

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

```
package main
import "fmt“
const N = 50
func fibonacci(n int) int {
   if n == 0 || n == 1 {
      return n
   }
   return fibonacci(n-1)+fibonacci(n-2)
}
func main() {
   fmt.Println("F(N)=", fibonacci(N))
}
```

```
package main  // 基线缺失
import "fmt“
const N = 50
func fibonacci(n int) int {
   if n == 0 {
      return n
   }
   return fibonacci(n-1)+fibonacci(n-2)
}
func main() {
   fmt.Println("F(N)=", fibonacci(N))
}
```

# 函数抽象

- 声明一次，可多次多处调用的代码块
  - 代码示例中，函数fibonacci在三处被调用
  - 运行时，函数fibonacci被调用了$O(2^N)$次

$$F(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

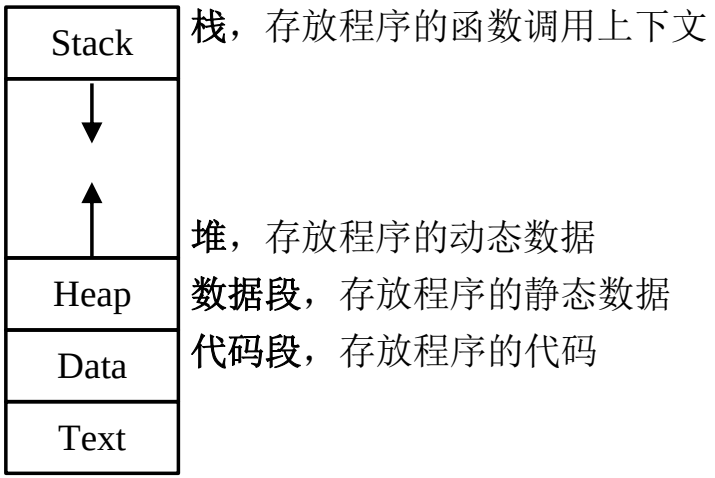
```
package main
import "fmt“
const N = 50
func fibonacci(n int) int {
   if n == 0 || n == 1 {
      return n
   }
   return fibonacci(n-1)+fibonacci(n-2)
}
func main() {
   fmt.Println("F(N)=", fibonacci(N))
}
```

```
package main   // 循环调用
import "fmt“
const N = 50
func fibonacci(n int) int {
 if n == 0 || n == 1 {
      return n
   }
   return fibonacci(n-1)+fibonacci(n+1)
}
func main() {
   fmt.Println("F(N)=", fibonacci(N))
}
```

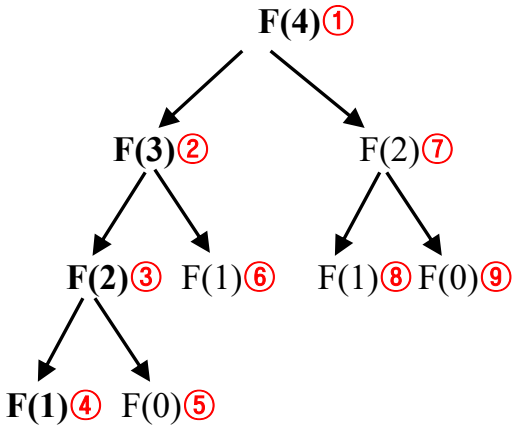
# Segments of Text, Data, Heap and Stack

- Segments：内存段
- 递归调用可能需要较大内存
- 求F(50)需要多大内存?
- 求F(5000000)需要多大内存?
  - 500字节？500万字节？ $2^{5000000}$字节？

```
package main
import "fmt“
const N = 50
func fibonacci(n int) int {
   if n == 0 || n == 1 {
      return n
   }
   return fibonacci(n-1)+fibonacci(n-2)
}
func main() {
   fmt.Println("F(N)=", fibonacci(N))
}
```



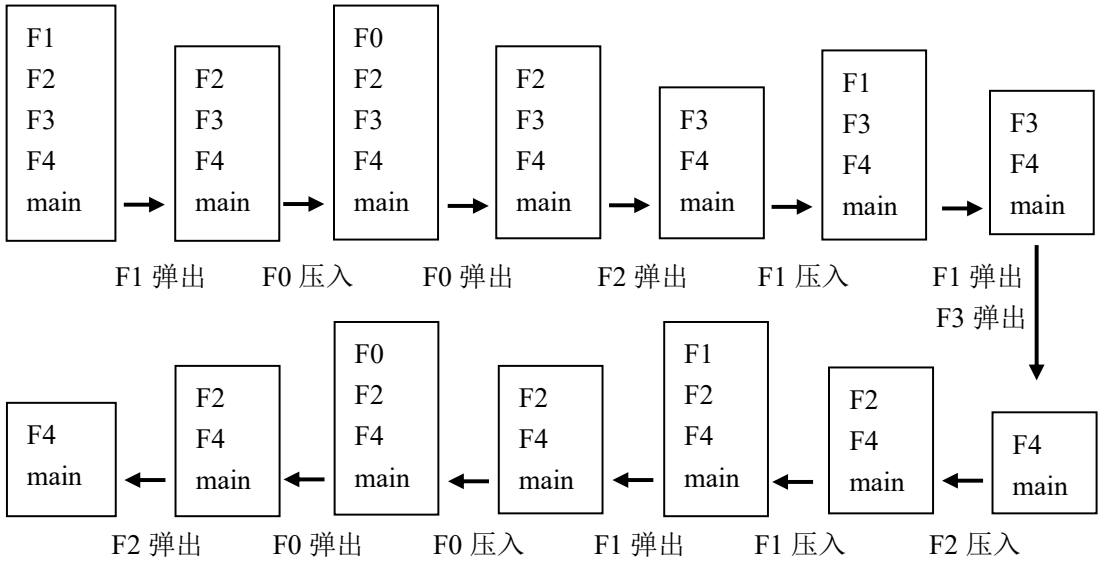
# 递归调用F(N)为什么需要O(N)内存？




- 求斐波那契数F(4)的函数调用序列
  - main, F(4), F(3), F(2), F(1)；F(0)；F(1)；F(2), F(1)；F(0)
- 栈帧（stack frame）
  - 函数调用的上下文，包含参数n=4和返回地址
  - 每次调用先将栈帧压入调用栈，执行完毕该栈帧弹出

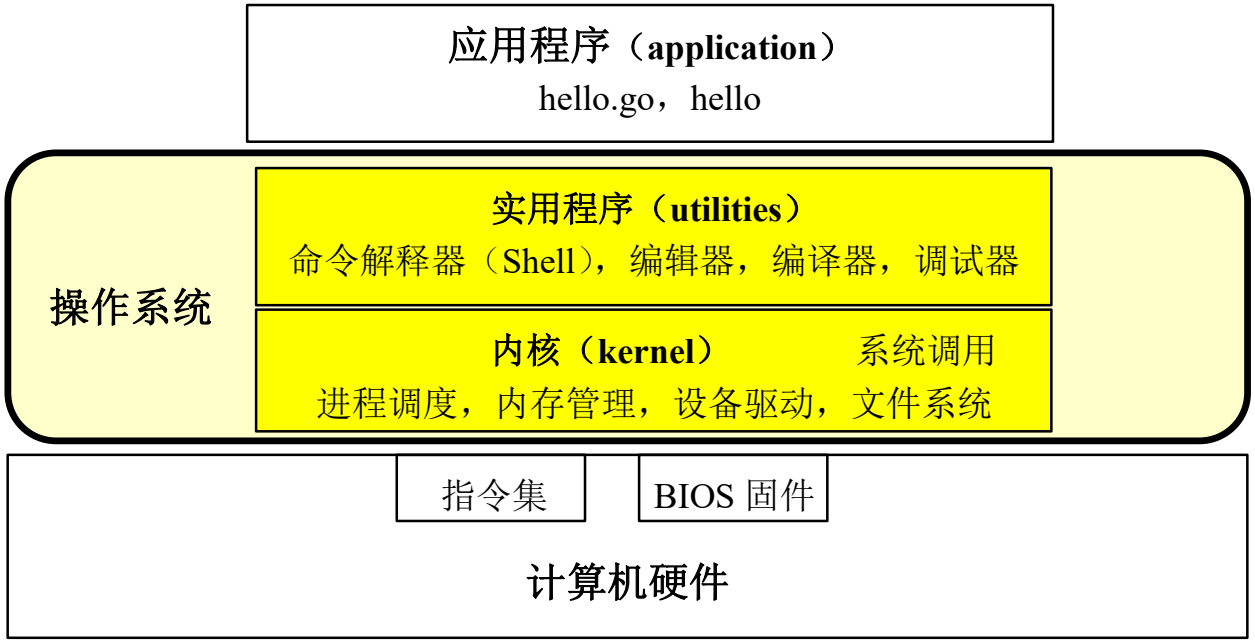
```
package main
import "fmt“
const N = 4
func F(n int) int {
   if n == 0 || n == 1 {
      return n
   }
   return F(n-1)+F(n-2)
}
func main() {
   fmt.Println("F(N)=", F(N))
}
```

调用栈状态变化



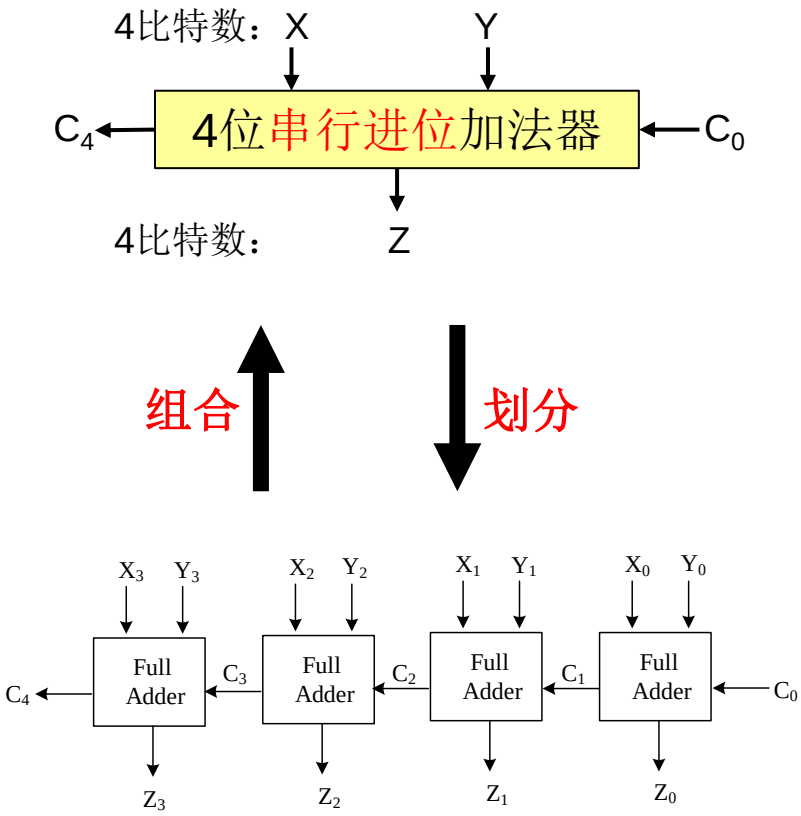
# 2. 操作系统简介——从动手动脑角度理解

- 我们已经动手动脑使用到了哪些操作系统概念？
  - 包括两大部分：实用程序（utilities）和内核（kernel）
    - 开发了多个Go语言应用程序，它们在运行时是进程
    - 在命令行界面使用VS Code编辑器和Go编译器
    - 内存管理：进程的代码段、数据段、栈、堆的管理
    - 信息隐藏实验使用了文件系统；通过Go库函数使用系统调用
    - StdIn/StdOut/StdErr标准输入输出设备



# 3. 模块化与模块

- 模块化像算法思维的分治方法
  - 将系统划分成模块
    - 很多情况下，系统中的两个模块可以连接，但不重叠
  - 将模块组合成系统
    - 该系统成为一个更高级的抽象
    - 例如，从N个全加器构建N位加法器
    - 是否忽略进位？
  - 划分与组合是一个整体
    - 现实中往往同时采用



# 3. 划分：从系统到模块

## 比特精准地将抽象映射到最底层硬件

```
fib[0] = 0
fib[1] = 1
for i := 2; i < 51; i++ {
   fib[i] = fib[i-1] + fib[i-2]
}
```

**万千应用**

程序
进程
指令
冯氏结构
指令流水线
时序电路
组合电路

# 比特精准地将抽象映射到最底层硬件

**万千应用**

程序
进程
指令
冯氏结构
指令流水线
时序电路
组合电路

```
MOV 0, R1
MOV R1, M[R0]
MOV 1, R1
MOV R1, M[R0+8]
MOV 2, R2    // i:=2
MOV 0, R1    // label Loop
ADD M[R0+R2*8-16], R1
ADD M[R0+R2*8-8], R1
MOV R1, M[R0+R2*8-0]
INC R2          // i++
CMP 51, R2  // i < 51?
JL Loop   // if '<' goto Loop
```

```
fib[0] = 0
fib[1] = 1
for i := 2; i < 51; i++ {
   fib[i] = fib[i-1] + fib[i-2]
}
```

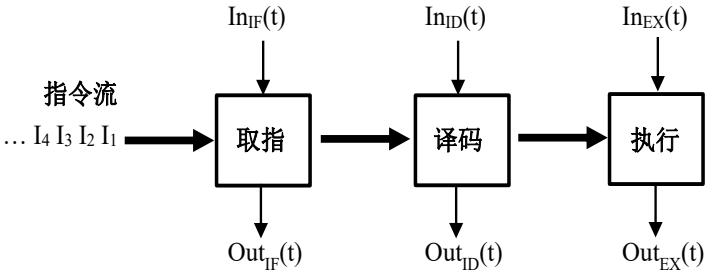
# 比特精准地将抽象映射到最底层硬件

**万千应用**

程序
进程
指令
冯氏结构
**指令流水线**
**时序电路**
组合电路

```
MOV 0, R1
MOV R1, M[R0]
MOV 1, R1
MOV R1, M[R0+8]
MOV 2, R2    // i:=2
MOV 0, R1    // label Loop
ADD M[R0+R2*8-16], R1
ADD M[R0+R2*8-8], R1
MOV R1, M[R0+R2*8-0]
INC R2       // i++
CMP 51, R2   // i < 51?
JL Loop   // if '<' goto Loop
```

```
fib[0] = 0
fib[1] = 1
for i := 2; i < 51; i++ {
   fib[i] = fib[i-1] + fib[i-2]
}
```



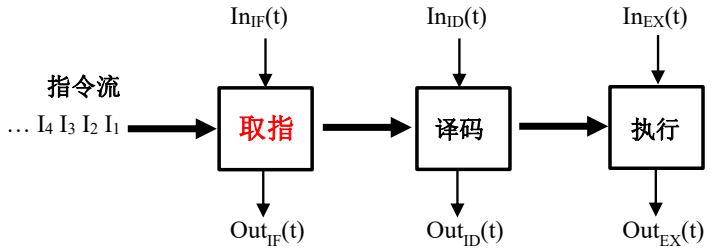
# 比特精准地将抽象映射到最底层硬件

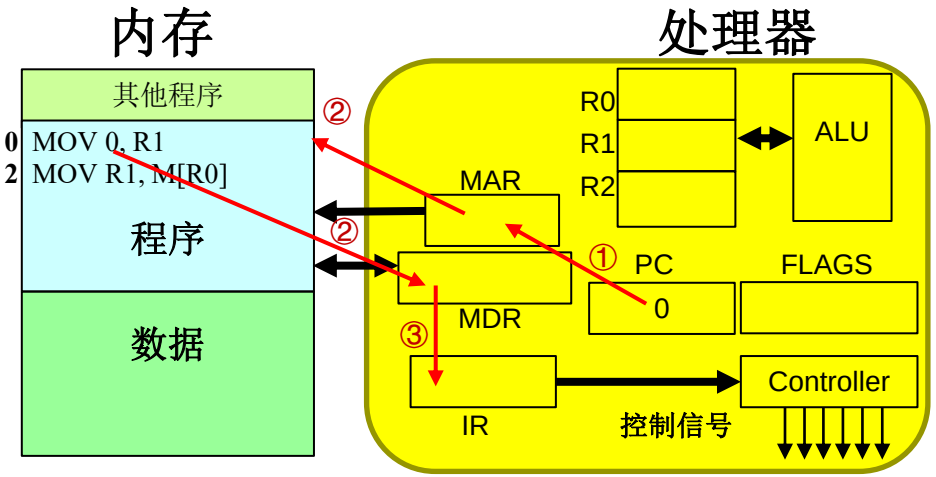
```
fib[0] = 0
fib[1] = 1
for i := 2; i < 51; i++ {
   fib[i] = fib[i-1] + fib[i-2]
}
```

**万千应用**

程序
进程
指令
冯氏结构
指令流水线
时序电路
组合电路

```
MOV 0, R1
MOV R1, M[R0]
MOV 1, R1
MOV R1, M[R0+8]
MOV 2, R2    // i:=2
MOV 0, R1    // label Loop
ADD M[R0+R2*8-16], R1
ADD M[R0+R2*8-8], R1
MOV R1, M[R0+R2*8-0]
INC R2       // i++
CMP 51, R2   // i < 51?
JL Loop   // if '<' goto Loop
```



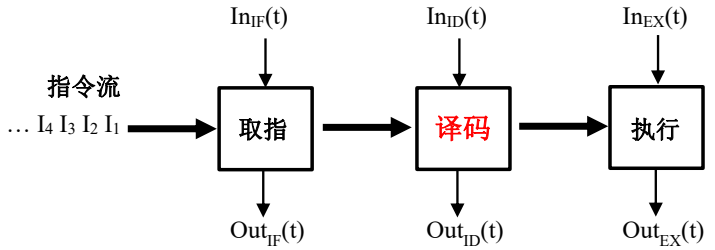
# 比特精准地将抽象映射到最底层硬件

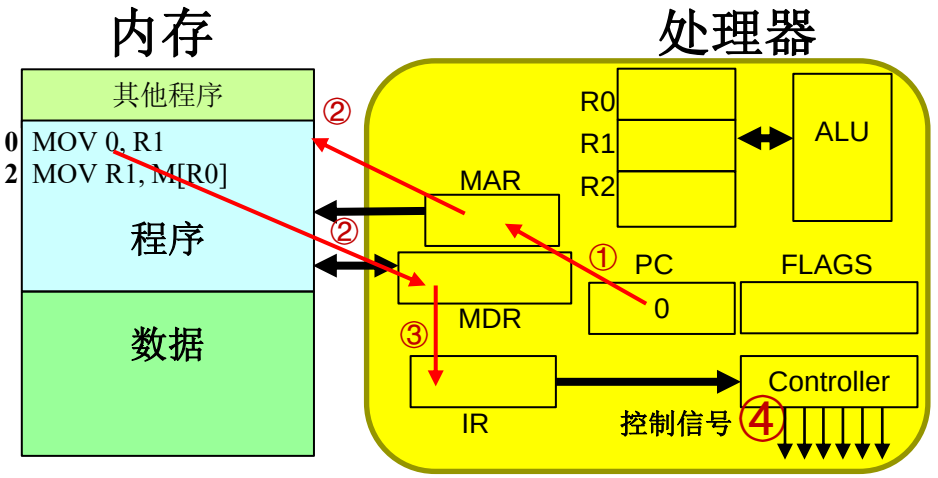
```
fib[0] = 0
fib[1] = 1
for i := 2; i < 51; i++ {
   fib[i] = fib[i-1] + fib[i-2]
}
```

**万千应用**

程序
进程
指令
冯氏结构
指令流水线
时序电路
组合电路

```
MOV 0, R1
MOV R1, M[R0]
MOV 1, R1
MOV R1, M[R0+8]
MOV 2, R2    // i:=2
MOV 0, R1    // label Loop
ADD M[R0+R2*8-16], R1
ADD M[R0+R2*8-8], R1
MOV R1, M[R0+R2*8-0]
INC R2       // i++
CMP 51, R2   // i < 51?
JL Loop   // if '<' goto Loop
```



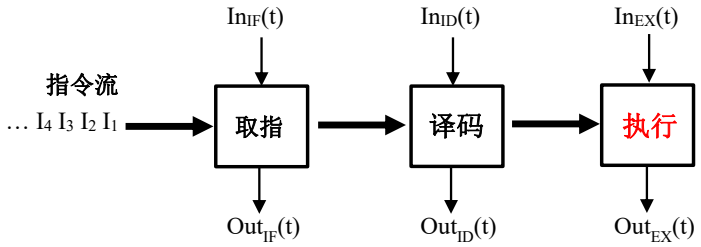
# 比特精准地将抽象映射到最底层硬件

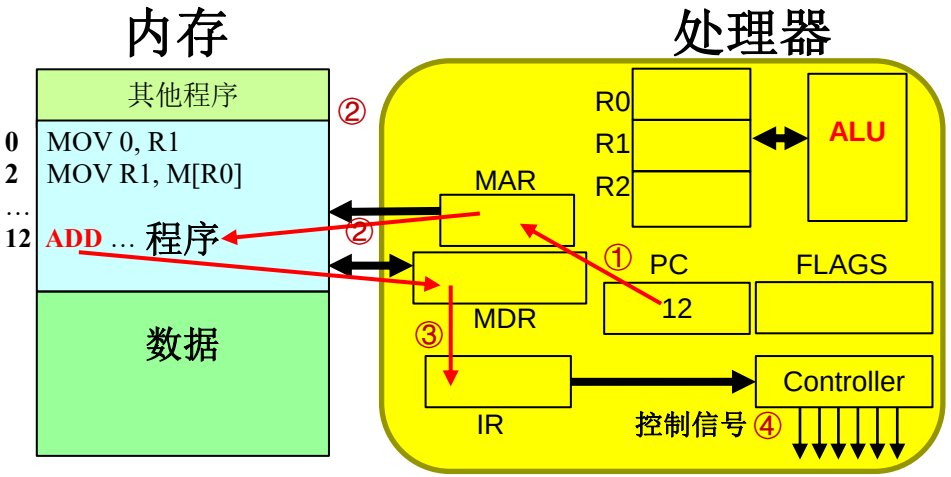
```
fib[0] = 0
fib[1] = 1
for i := 2; i < 51; i++ {
   fib[i] = fib[i-1] + fib[i-2]
}
```

**万千应用**

程序
进程
指令
冯氏结构
指令流水线
时序电路
组合电路

```
MOV 0, R1
MOV R1, M[R0]
MOV 1, R1
MOV R1, M[R0+8]
MOV 2, R2    // i:=2
MOV 0, R1    // label Loop
ADD M[R0+R2*8-16], R1
ADD M[R0+R2*8-8], R1
MOV R1, M[R0+R2*8-0]
INC R2       // i++
CMP 51, R2   // i < 51?
JL Loop   // if '<' goto Loop
```



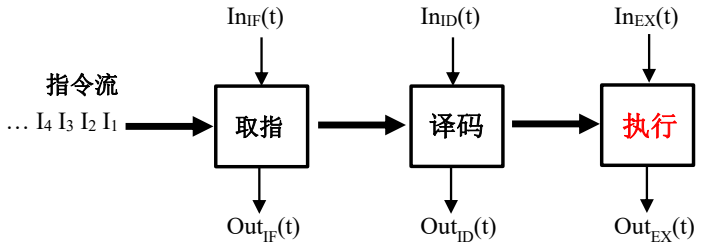
# 比特精准地将抽象映射到最底层硬件

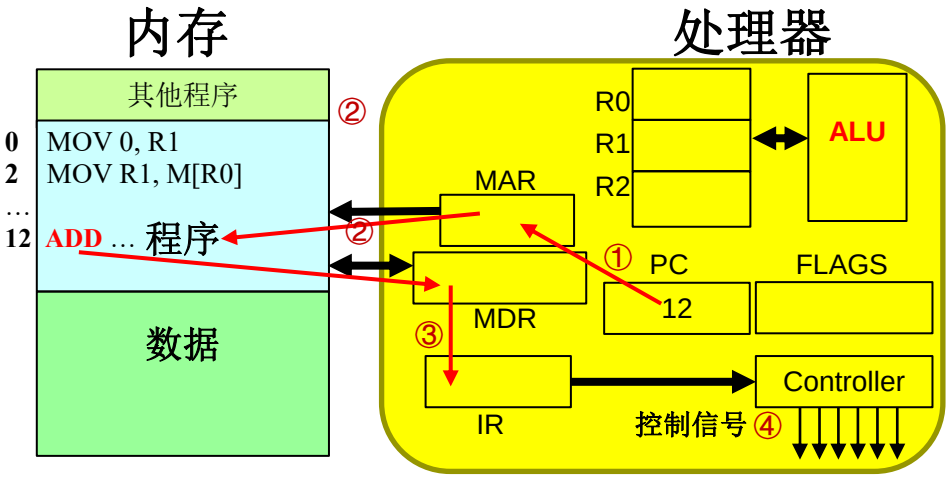
```
fib[0] = 0
fib[1] = 1
for i := 2; i < 51; i++ {
   fib[i] = fib[i-1] + fib[i-2]
}
```

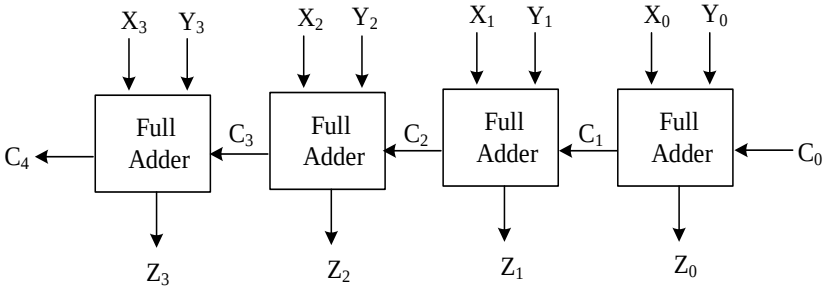
**万千应用**

程序
进程
指令
冯氏结构
指令流水线
时序电路
组合电路

```
MOV 0, R1
MOV R1, M[R0]
MOV 1, R1
MOV R1, M[R0+8]
MOV 2, R2    // i:=2
MOV 0, R1    // label Loop
ADD M[R0+R2*8-16], R1
ADD M[R0+R2*8-8], R1
MOV R1, M[R0+R2*8-0]
INC R2       // i++
CMP 51, R2   // i < 51?
JL Loop   // if '<' goto Loop
```



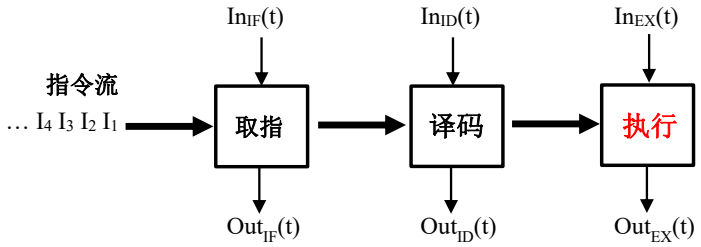
# 比特精准地将抽象映射到最底层硬件

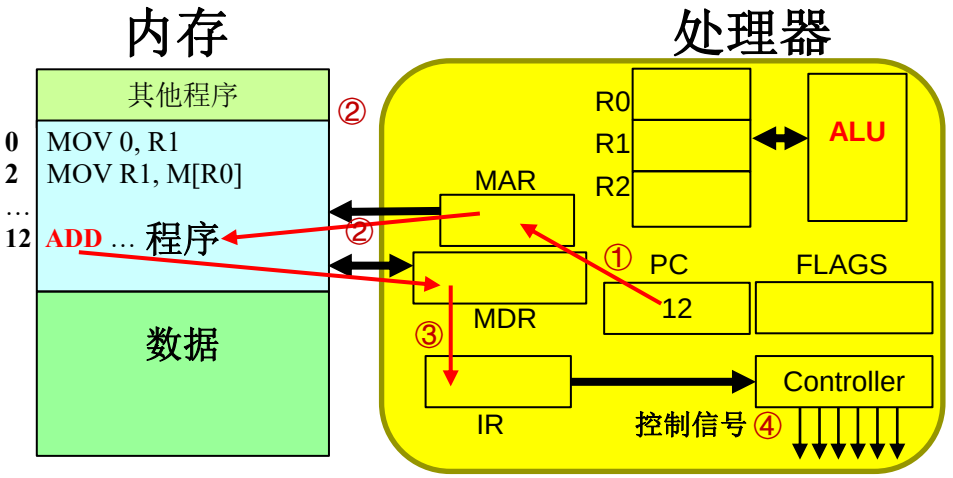
```
fib[0] = 0
fib[1] = 1
for i := 2; i < 51; i++ {
   fib[i] = fib[i-1] + fib[i-2]
}
```

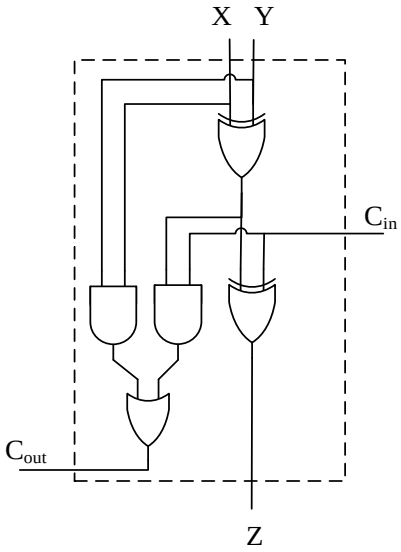
**万千应用**

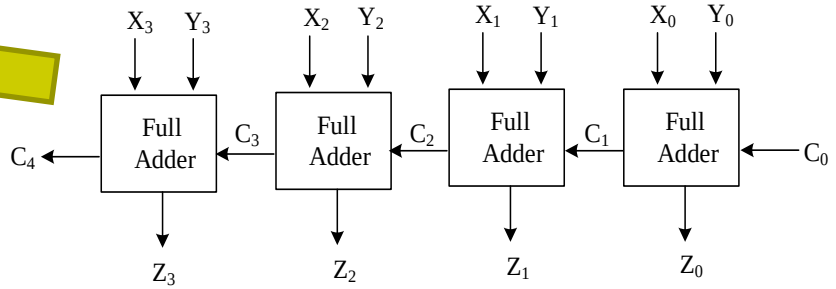
程序
进程
指令
冯氏结构
指令流水线
时序电路
组合电路

```
MOV 0, R1
MOV R1, M[R0]
MOV 1, R1
MOV R1, M[R0+8]
MOV 2, R2    // i:=2
MOV 0, R1    // label Loop
ADD M[R0+R2*8-16], R1
ADD M[R0+R2*8-8], R1
MOV R1, M[R0+R2*8-0]
INC R2       // i++
CMP 51, R2   // i < 51?
JL Loop   // if '<' goto Loop
```



# 举一反三

**假如改成减法fib[i-1] - fib[i-2]，会怎样？**

# 万千应用

```
fib[0] = 0
fib[1] = 1
for i := 2; i < 51; i++ {
   fib[i] = fib[i-1] - fib[i-2]
}
```

程序
进程
指令
冯氏结构
指令流水线
时序电路
组合电路

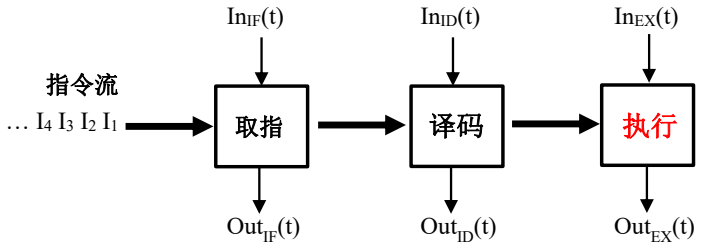
# 哪些地方要改？如何改？

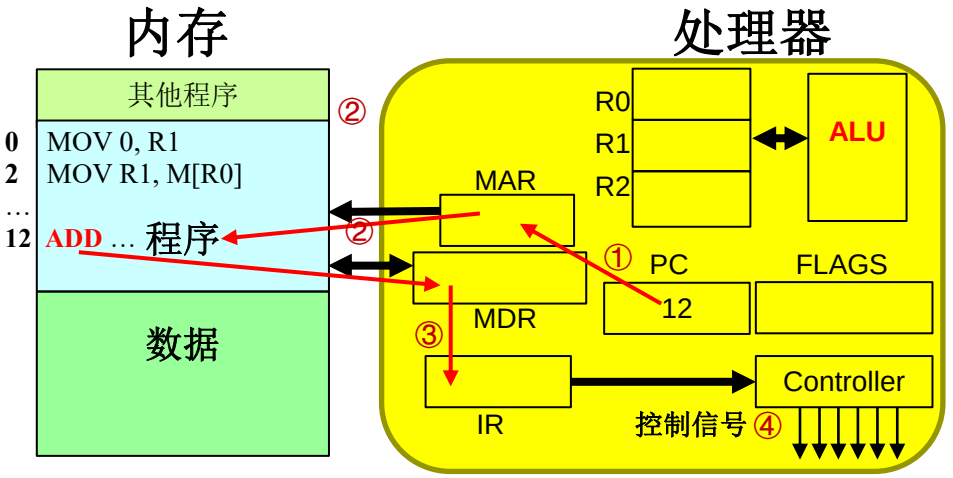
```
fib[0] = 0
fib[1] = 1
for i := 2; i < 51; i++ {
   fib[i] = fib[i-1] + fib[i-2]
}
```

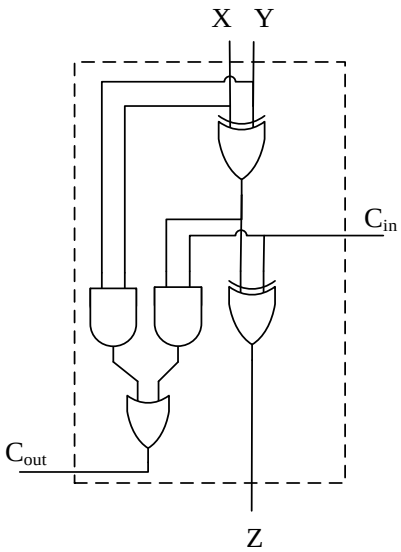
**万千应用**

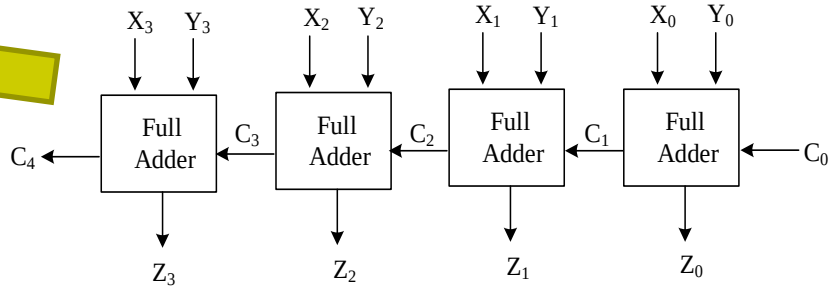
程序
进程
指令
冯氏结构
指令流水线
时序电路
组合电路

```
MOV 0, R1
MOV R1, M[R0]
MOV 1, R1
MOV R1, M[R0+8]
MOV 2, R2    // i:=2
MOV 0, R1    // label Loop
ADD M[R0+R2*8-16], R1
ADD M[R0+R2*8-8], R1
MOV R1, M[R0+R2*8-0]
INC R2       // i++
CMP 51, R2   // i < 51?
JL Loop   // if '<' goto Loop
```



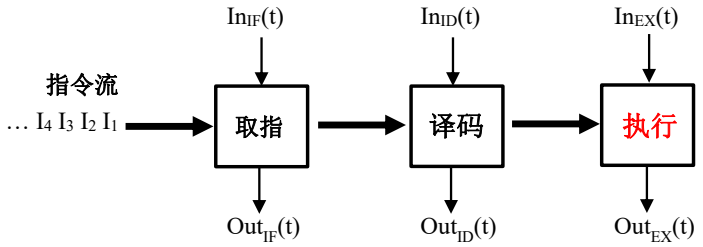
# 哪些地方要改？如何改？

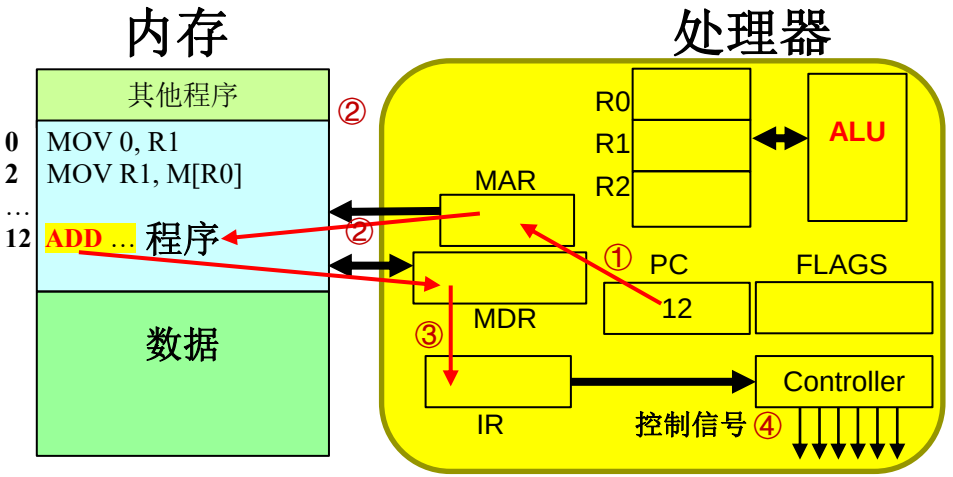
```
fib[0] = 0
fib[1] = 1
for i := 2; i < 51; i++ {
   fib[i] = fib[i-1] + fib[i-2]
}
```

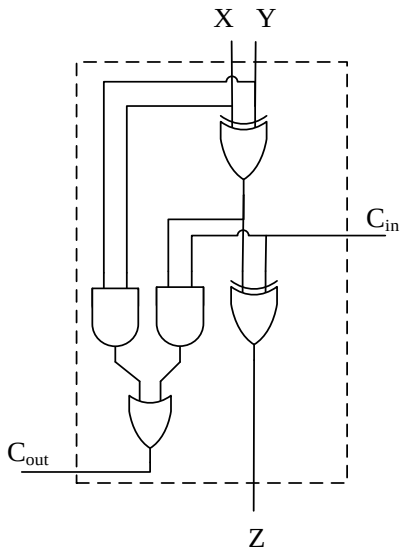
**万千应用**

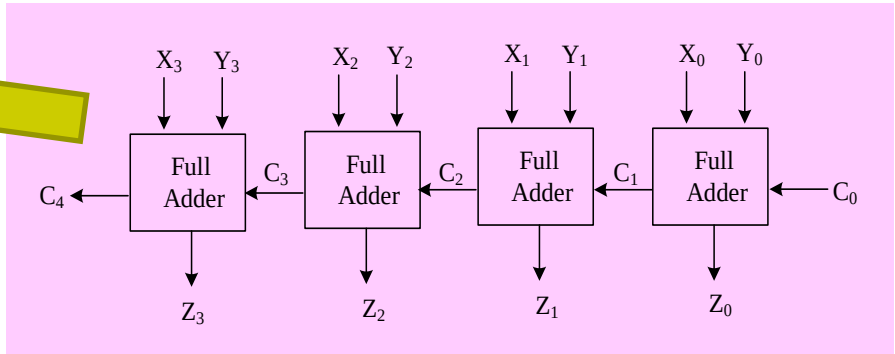
程序
进程
指令
冯氏结构
指令流水线
时序电路
组合电路

```
MOV 0, R1
MOV R1, M[R0]
MOV 1, R1
MOV R1, M[R0+8]
MOV 2, R2    // i:=2
MOV 0, R1    // label Loop
ADD M[R0+R2*8-16], R1
ADD M[R0+R2*8-8], R1
MOV R1, M[R0+R2*8-0]
INC R2       // i++
CMP 51, R2   // i < 51?
JL Loop   // if '<' goto Loop
```



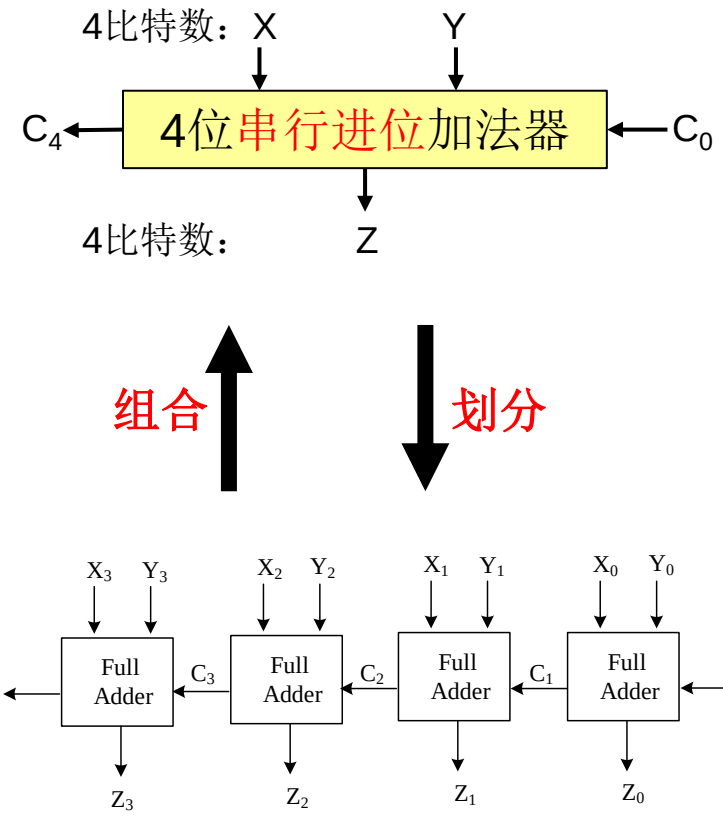



# 4. 模块化与模块

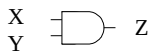
- 模块化像算法思维的分治方法
  - 将系统划分成模块
    - 很多情况下，系统中的两个模块可以连接，但不重叠
  - 将模块组合成系统
    - 该系统成为一个更高级的抽象
    - 例如，从N个全加器构建N位加法器
    - 是否忽略进位？
- 什么是模块
  - 模块是采用了信息隐藏原理的抽象
- 模块化是艺术，需要想象力和创造力
- 从逻辑门到指令流水线的旅行
  - 硬件为主：组合电路、时序电路



# 4.1 逻辑门与组合电路

- 什么是门？实现布尔运算的电路

| X | Y | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



AND

与门

$Z = X \cdot Y$

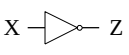
| X | Y | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

OR

或门

$Z = X + Y$

| X | Z |
|---|---|
| 0 | 1 |
| 1 | 0 |



NOT

非门

$Z = \overline{X}$

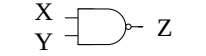
| X | Y | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

XOR

异或门

$Z = X \oplus Y$

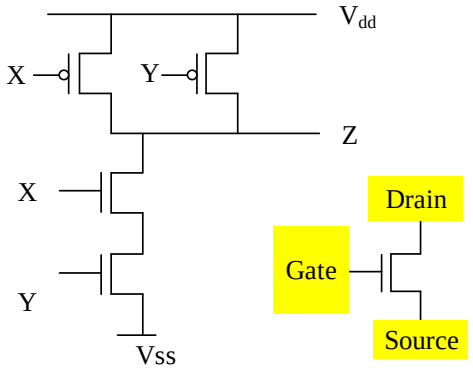
| X | Y | Z |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



NAND

与非门

$Z = \overline{X \cdot Y}$




CMOS circuit for NAND

与非门的CMOS电路

每个逻辑门都有半导体电路实现

只需知道与非门的CMOS电路实现

作为例子

**晶体管行为**

Gate 为1（高电平）：导通

Gate为0（低电平）：断开

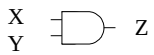
**Vss: ground (0)**

**Vdd: high voltage (1)**

**X,Y都是高电平（1）时，下两个晶体管联通，上两个晶体管断开，Z连接到地（0）**

# 逻辑门与组合电路

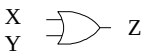
| X | Y | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |




AND

$Z = X \cdot Y$

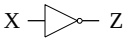
| X | Y | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |




OR

$Z = X + Y$

| X | Z |
|---|---|
| 0 | 1 |
| 1 | 0 |




NOT

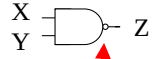
$Z = \overline{X}$

| X | Y | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |


XOR

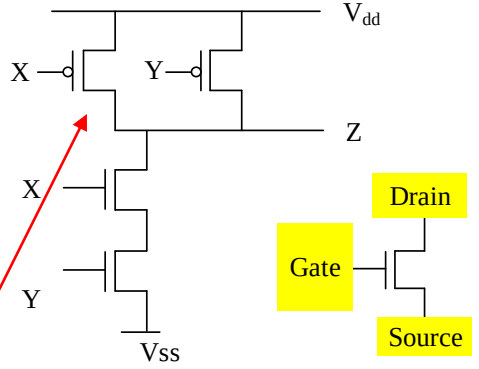
$Z = X \oplus Y$

| X | Y | Z |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |




NAND

$Z = \overline{X \cdot Y}$

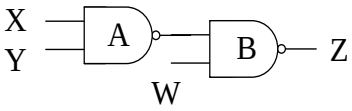


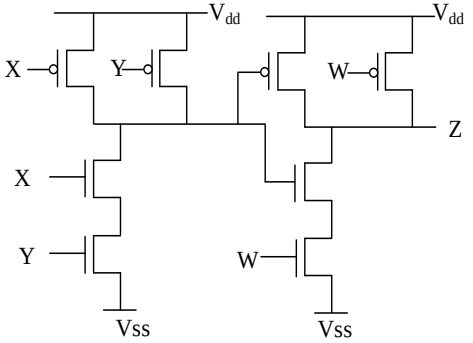

CMOS circuit for NAND

小圈表示**NOT**（非）

**circle means NOT** 高电平变成低电平，低电平变成高电平

$Z = \overline{(\overline{X \cdot Y}) \cdot W}$

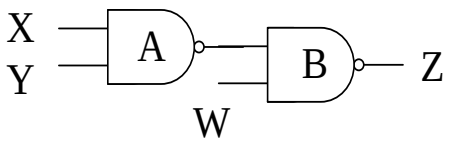


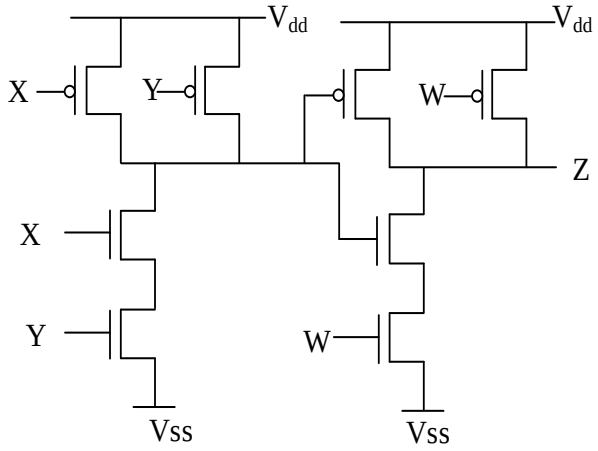



# 逻辑门与组合电路

- 组合电路由逻辑门连接而成，没有反馈连线
- 组合电路实现布尔表达式
- 使用逻辑线路图（logic diagram）表示组合电路
- 不用更加繁杂的CMOS电路

$$Z = \overline{(\overline{X \cdot Y}) \cdot W}$$

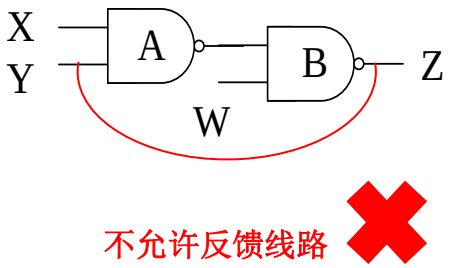


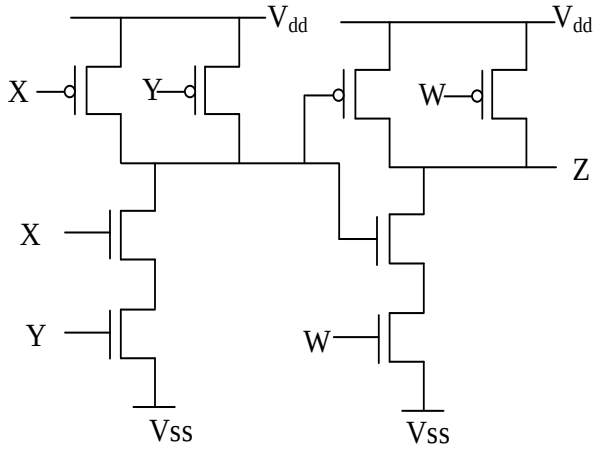



# 逻辑门与组合电路

- 组合电路由逻辑门连接而成，没有反馈连线
- 组合电路实现布尔表达式
- 使用逻辑线路图（logic diagram）表示组合电路
- 不用更加繁杂的CMOS电路

$$Z = \overline{(\overline{X \cdot Y}) \cdot W}$$



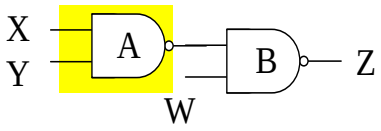



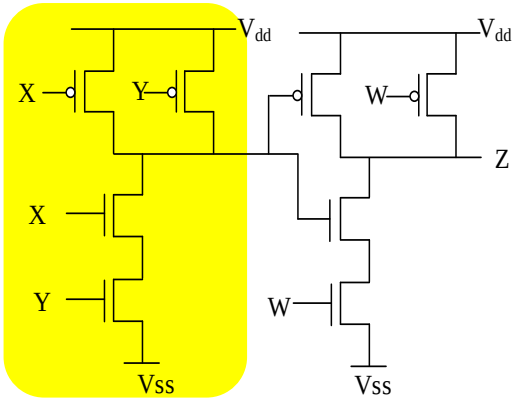
# 4.2 信息隐藏原理

- 模块仅暴露接口和可见行为，隐藏内部细节和内部行为
- 这是一个通用原理，广泛应用于硬件和软件

- 例子
  - 一个组合电路的三种表示，它们逻辑等价（有相同真值表）
    - Boolean expression　布尔表达式
    - Logic diagram　逻辑线路图
    - CMOS circuit　CMOS电路
  - 布尔表达式、逻辑线路图
    - 更简洁
    - 允许不同实现
  - 三个黄色区域表示同样的2-输入-1-输出与非门
  - CMOS电路最复杂
    - 暴露了内部
    - 固定了实现

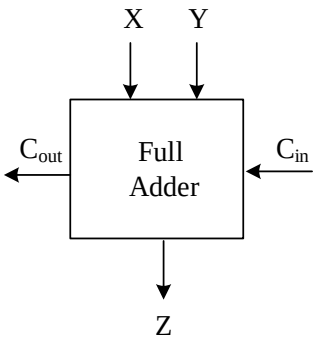
$$Z = \overline{(\overline{X \cdot Y}) \cdot W}$$



# 4.3 从N位加法器看组合电路的四种设计方法

- 从本位输入X与Y产生结果Z；其中X，Y，Z都是N位无符号数
  - 要考虑进位输入比特$C_{in}$与进位输出比特$C_{out}$
  - 采用通常的加法方法
- 当N=1时，设计1位加法器，即全加器（full adder）
  - “全”：考虑进位输入（ $C_{in}$ ）与进位输出（ $C_{out}$ ），而不只是本位输入（X,Y）与本位输出（Z）




Full adder symbol

# 从N位加法器看组合电路的四种设计方法

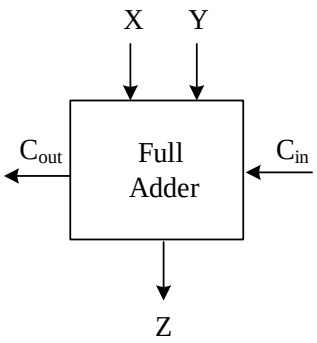
- 从本位输入X与Y产生结果Z；其中X，Y，Z都是N位无符号数
  - 要考虑进位输入比特$C_{in}$与进位输出比特$C_{out}$
  - 采用通常的加法方法
- 当N=1时，设计1位加法器，即全加器（full adder）
  - “全”：考虑进位输入（ $C_{in}$ ）与进位输出（ $C_{out}$ ），而不只是本位输入（X,Y）与本位输出（Z）

- **课堂测验**

给定X，Y，$C_{in}$, 分别写出全加器的Z与$C_{out}$的布尔表达式

$Z =$

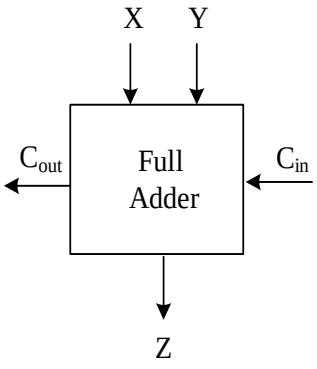
$C_{out} =$




Full adder symbol

# Full adder 全加器
## 代表了设计方法1：直接连接逻辑门

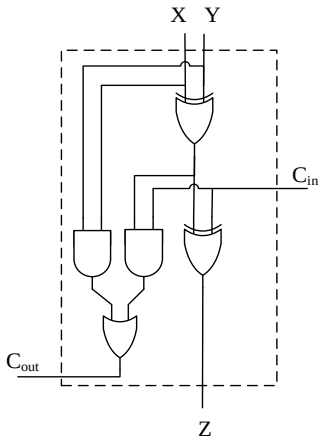
- 从本位输入X与Y产生结果Z；其中X，Y，Z都是N位无符号数
  - 要考虑进位输入比特$C_{in}$与进位输出比特$C_{out}$
  - 采用通常的加法方法
- 当N=1时，设计1位加法器，即全加器（full adder）
  - “全”：考虑进位输入（$C_{in}$）与进位输出（$C_{out}$），而不只是本位输入（X,Y）与本位输出（Z）
- 采用二进制加法原理导出真值表，继而导出输出Z和$C_{out}$的布尔表达式
  - $Z = X \oplus Y \oplus C_{in}$
  - $C_{out} = (X \cdot Y) + (X \oplus Y) \cdot C_{in}$




Full adder symbol

| Cin | X | Y | Z | Cout |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Truth table



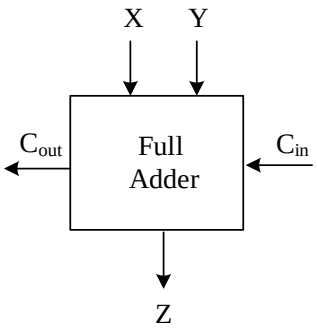

Implementation by gates

全加器是系统

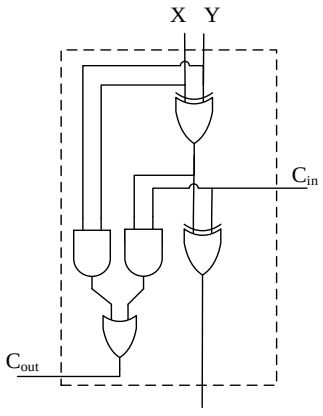
逻辑门是模块

# Ripple-carry adder 波纹进位加法器
## 代表了设计方法2：串行连接多个模块

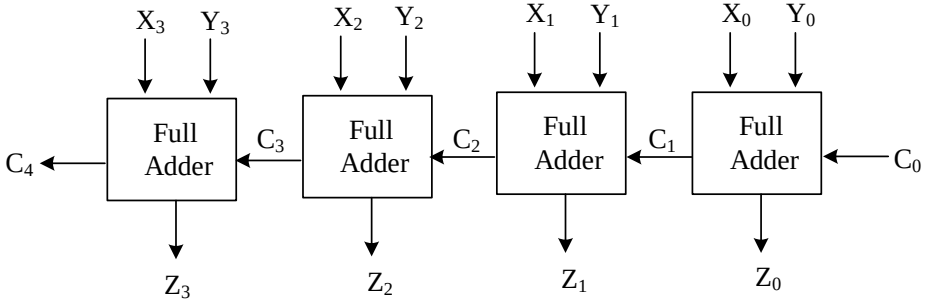
- 从本位输入X与Y产生结果Z；其中X，Y，Z都是N位无符号数
  - 要考虑进位输入比特$C_{in}$与进位输出比特$C_{out}$
  - 采用通常的加法方法
- 串联4个全加器（**模块**），实现一个4位波纹进位加法器（**系统**）
  - 用X+Y=1011+1001 = 10100验证




Full adder symbol
全加器是**模块**




Its implementation by gates

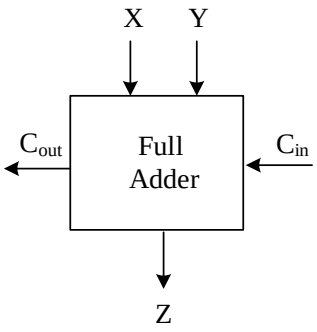



A 4-bit ripple-carry adder
4位波纹进位加法器是**系统**

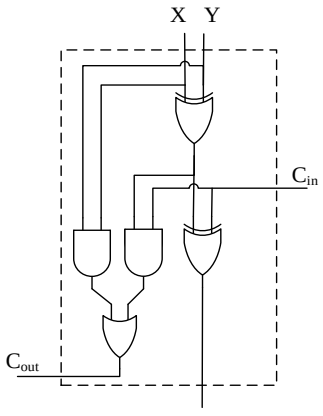
# Ripple-carry adder 波纹进位加法器
## 代表了设计方法2：串行连接多个模块

- 从本位输入X与Y产生结果Z；其中X，Y，Z都是N位无符号数
  - 要考虑进位输入比特$C_{in}$与进位输出比特$C_{out}$
  - 采用通常的加法方法
- 串联4个全加器（**模块**），实现一个4位波纹进位加法器（**系统**）
  - 用X+Y=1011+1001 = 10100验证

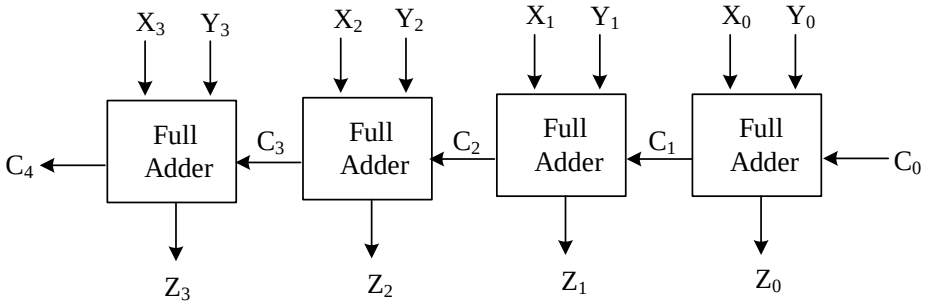
- **将模块组合成系统的抽象过程可能会丢失信息**
  - 4位波纹进位加法器产生多少级门延迟？n位呢？




Full adder symbol
全加器是**模块**



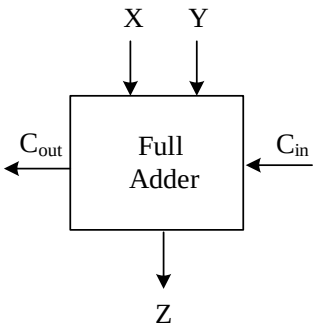

Its implementation by gates



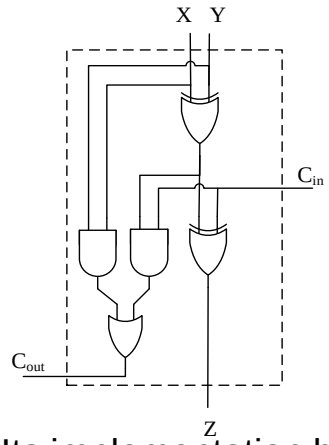

A 4-bit ripple-carry adder
4位波纹进位加法器是**系统**

# 波纹进位加法器

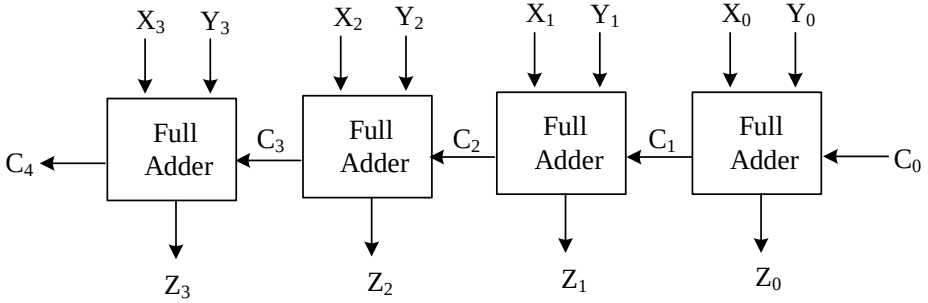
- 将模块组合成系统的抽象过程可能会丢失信息
  - 问题：n位波纹进位加法器产生多少级门延迟？
  - 答案1： 3n
  - 4位波纹进位加法器产生4*3=12级门延迟，因为每个全加器产生3级门延迟
  - 此答案正确吗？




Full adder symbol



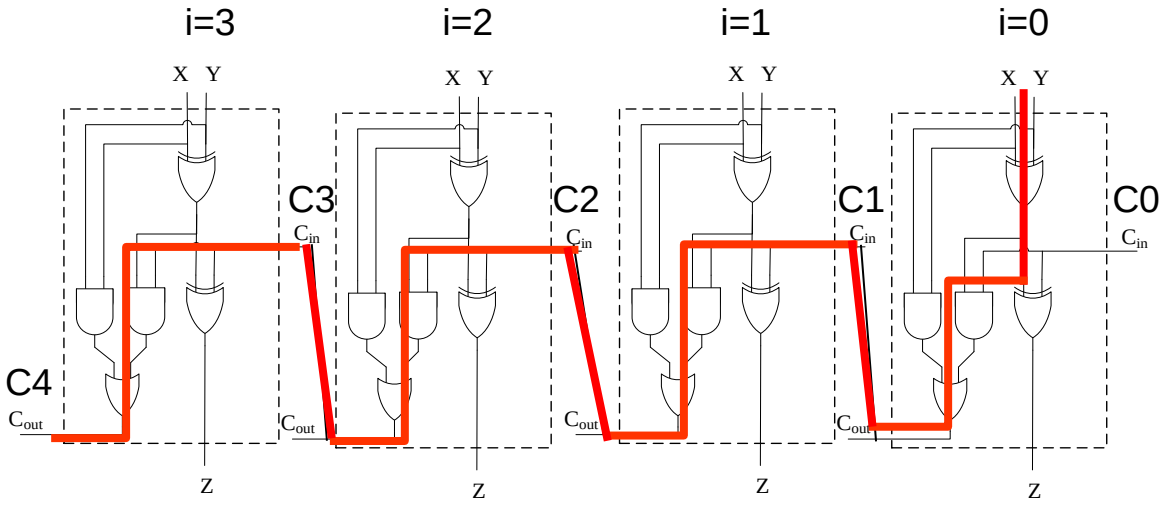

Its implementation by gates




A 4-bit ripple-carry adder

# 波纹进位加法器

- 将模块组合成系统的抽象过程可能会丢失信息
  - 问题：n位波纹进位加法器产生多少级门延迟？
  - 答案2：2n+1
    4位波纹进位加法器产生2*4+1=9级门延迟
    因为当算出$C_1$时，$X_i \oplus Y_i$ 也已经被算出来了，$1 \le i < n$
    - 红线展示了最长路径

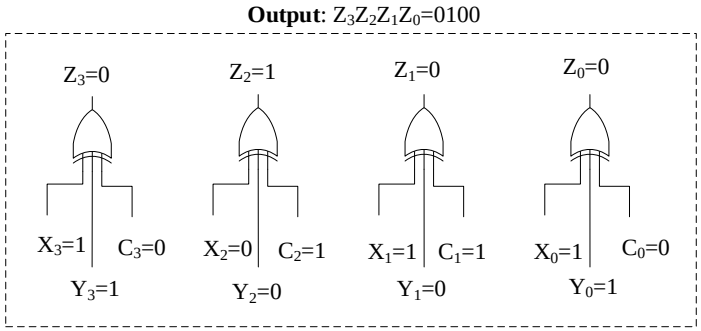


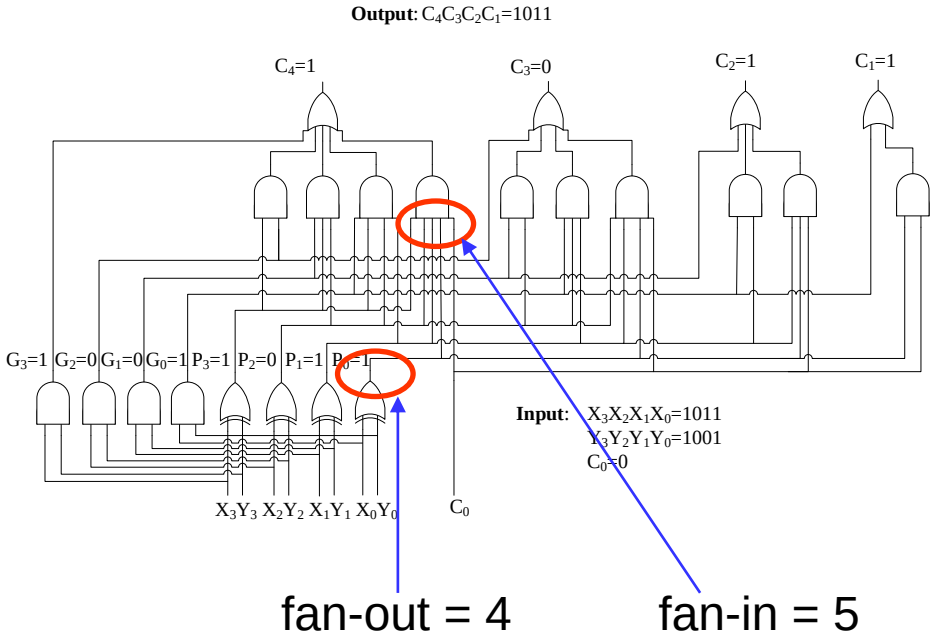

Expand to show the gate-level details

# 并行进位加法器
## 代表了设计方法3：并行连接，使用中间模块Gi和Pi

- **先在一步并行算出全部进位比特值**
  - 需要三级门延迟
  - 为什么是3?
- 其后一步并行算出Z的*n*个比特
  - 需要一级门延迟
  - 为什么是1?
- 总共只需4级门延迟　　4 vs. 2n-1
  - 波纹进位加法器需要2*n*-1级门延迟

- 实际电路有扇入（fan-in）和扇出（fan-out）限制
  - 扇入：一个门电路支持的输入数
  - 扇出：一个门电路输出驱动的线路数



# 加减器

## 代表了设计方法4：使用多路复用器和控制信号（本例中的选择输入S）

- 采用多路复用器（multiplexer，MUX）的加减器
  - 控制信号S选择做加法(S=0)还是做减法(S=1)
  - 采用二进制补码，减法等价于加负数，5-5 = 5+(-5)
    - 因为-X刚好是X的二进制补码
  - 假设X = Y = 5。即，$X_3X_2X_1X_0 = Y_3Y_2Y_1Y_0 = 0101$
    - Then, X – Y = 5 + (-5) = 5 + 5的补码

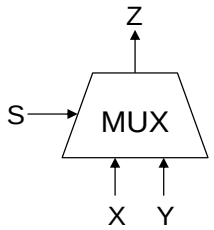
$$\rightarrow 0101 + (\overline{0}\,\overline{1}\,\overline{0}\,\overline{1} + 0001)$$

$$= \quad 0101 + \quad 1011$$

$$= 10000 = C_4\,Z_3\,Z_2\,Z_1\,Z_0$$

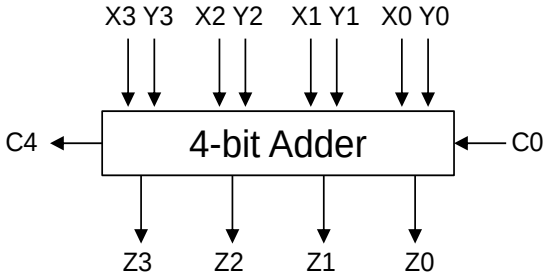
$$Z = \begin{cases} X & \text{if } S = 0 \\ Y & \text{if } S = 1 \end{cases}$$

| S | X | Y | Z |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

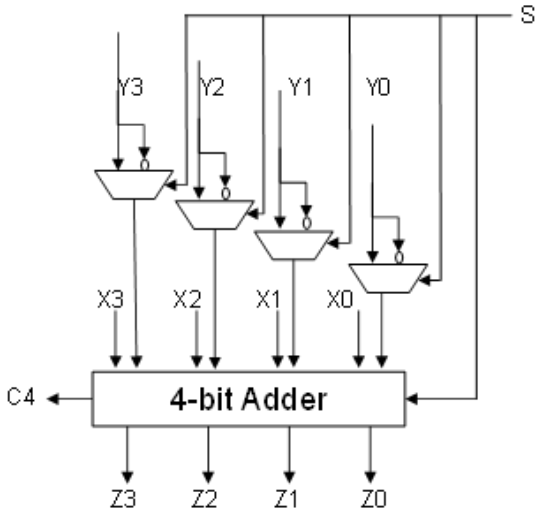



| S | Z |
|---|---|
| 0 | X |
| 1 | Y |

A multiplexer 多路复用器

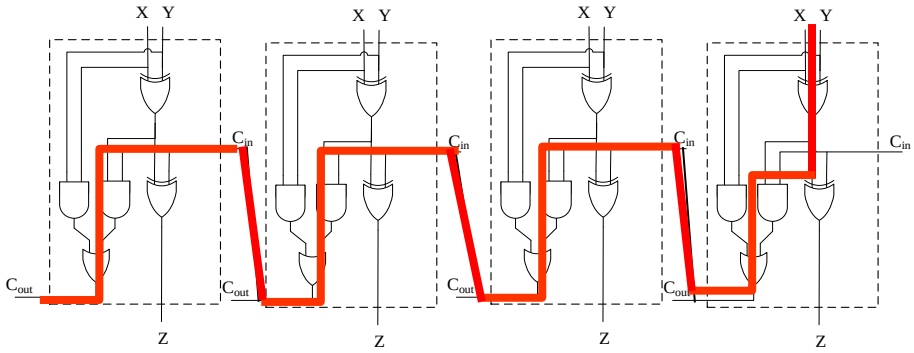


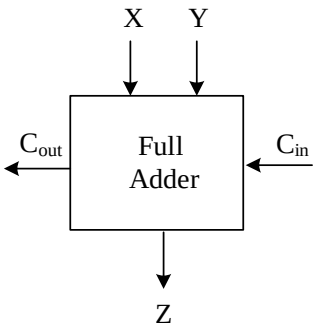

A two's complement 4-bit adder and an adder-subtractor



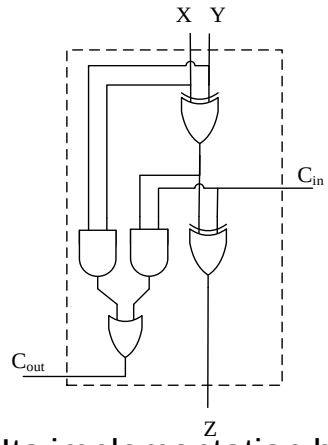
# 通过实例理解模块化的一般原理

- 模块是采用信息隐藏原理的抽象
  - 全加器模块是一个组合电路抽象，隐藏了虚框内的逻辑电路细节
- 系统中的两个模块可以连接，但不重叠
  - 波纹进位加法器中的两个全加器可连接，但两个全加器的内部电路不重叠



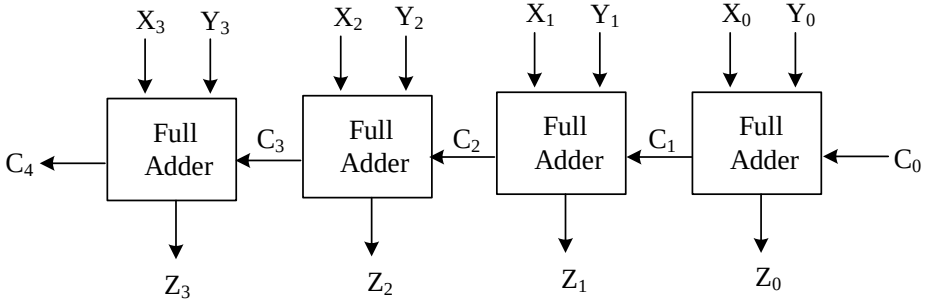




Full adder symbol




Its implementation by gates

全加器是模块




A 4-bit ripple-carry adder

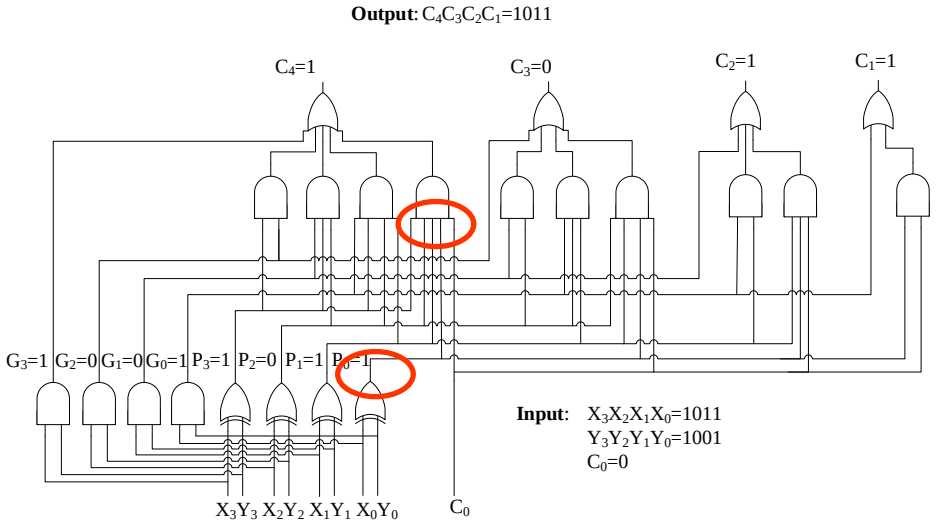
4位波纹进位加法器是系统

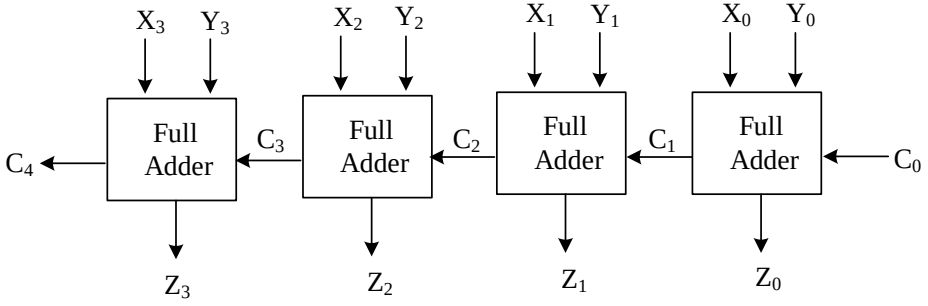
# 通过实例理解模块化的一般原理

- 模块化是艺术，需要人的创造力
  - 波纹进位加法器比较直接简单
  - 并行进位加法器则需要艺术：
    体现在中间表示P，G（留作练习）
- 利用了加法的领域知识
  - 设下列表达式
    - $G_i = X_i \cdot Y_i$，$P_i = X_i \oplus Y_i$
      - $G_0 = X_0 \cdot Y_0, G_1 = X_1 \cdot Y_1$
      - $P_0 = X_0 \oplus Y_0, P_1 = X_1 \oplus Y_1$
  - 有
    - $C_{i+1} = G_i + P_i \cdot C_i$
  - 当$i = 1$时，有
    - $C_2 = G_1 + P_1 \cdot C_1$
  - 展开，则有

$$C_2 = G_1 + P_1 \cdot \left(G_0 + P_0 \cdot C_0\right)$$

$$C_2 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$



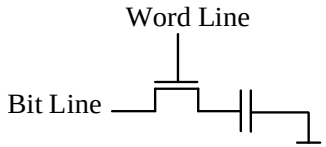



# 4.4 时序电路

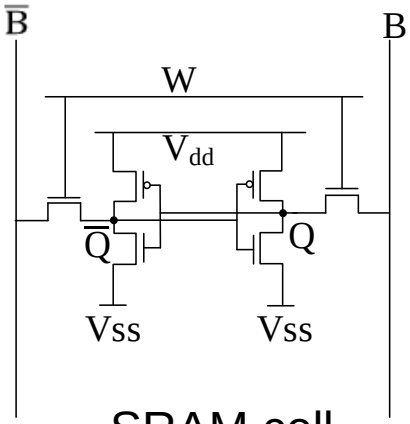
- 时序电路 = 组合电路 + 状态电路
- 有了状态，可以实现多步计算过程
  - 每一步从两类值
    - 当前输入
    - 当前状态
  - 计算出两类值
    - 当前输出
    - 下一状态
- 逻辑电路与逻辑思维之对应
  - 组合电路实现布尔表达式；时序电路实现自动机
- 状态电路有两种实现方法：存储单元与触发器
- 触发器：带反馈回路的组合电路
  - 只需理解D触发器

# 4.4.1 四类存储器

- 非易失存储器（Non-volatile memory，NVM）
  - 计算机下电后，非易失存储器中的内容不会丢失
  - **ROM** (read-only memory) 只读存储器　　例子：开机后读取数据与指令
  - Read-write **NVM** 可读可写的非易失存储器　　例子：U盘
- 易失存储器（Volatile memory）
  - 计算机下电后，易失存储器中的内容会丢失
  - **DRAM** 动态随机访问存储器
    - 成本低，但需要刷新（每7.8-128 μs刷新一次），速度较慢
    - 因为电容会漏电；因此，称为动态
  - **SRAM** 静态随机访问存储器
    - 成本高，但避免了刷新开销




DRAM cell
只需一个晶体管



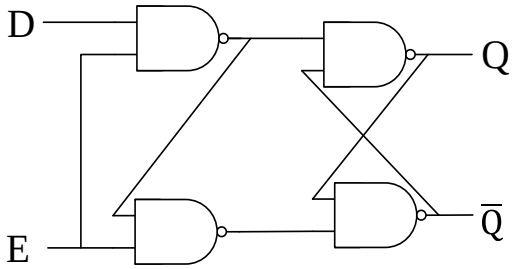

SRAM cell
需要6个晶体管

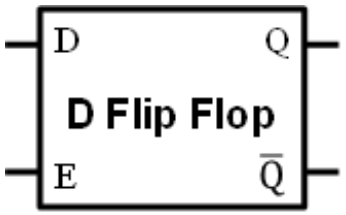
| Type | Latency | Price $/GB |
|---|---|---|
| Register | 100s ps | N/A |
| SRAM | 100s ps ~ 10 ns | $100's ~1000s /GB |
| DRAM | 10s ~ 100 ns | $2~4 /GB |
| NVM: Main Memory | 100 ns ~ 10 μs | $6 / GB |
| NVM: SSD | 10 μs ~ 1 ms | $0.1~0.2 / GB |
| Hard Disk: HDD | > A few ms | $0.02 / GB |

# 4.4.2 D flip-flop D触发器

- 触发器是含反馈线路的逻辑门电路
- 一类常用触发器是D触发器（delay flip-flop）
  - 2-输入-2-输出；一个输入是数据，另一个是时钟；两个输出互为其非
  - 功能
    - When E=0, Q remains the same; when E=1, the next Q will be the current D
    - 状态Q值是数据输入D一个时钟周期前的值即D延迟一周期后成为Q

| E | D | Q | $Q_{next}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

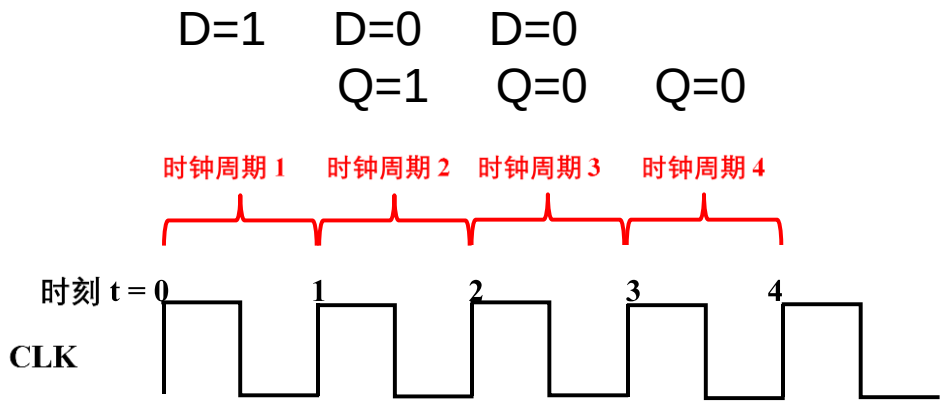


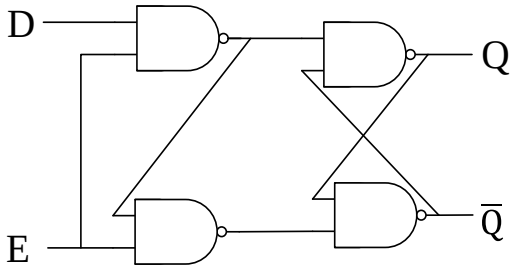

D flip-flop: Truth table　　Internal logic diagram　　Symbol

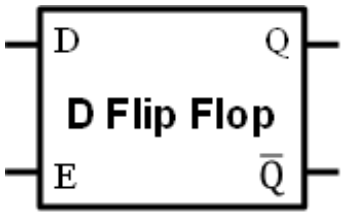
# D触发器

- D是数据输入，E常常是时钟信号CLK
- 状态Q值是数据输入D一个时钟周期前的值
  - 即D延迟一周期后变成Q




| E | D | Q | $Q_{next}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |



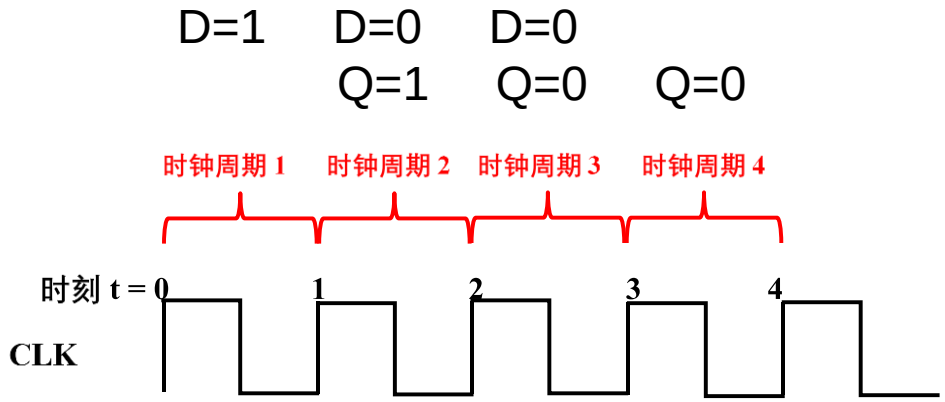




D flip-flop: Truth table　　Internal logic diagram　　Symbol

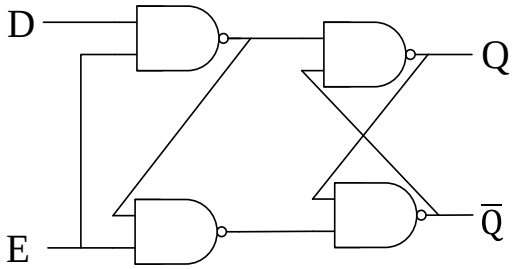
# D触发器

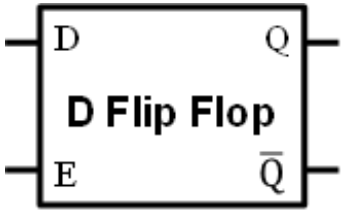
- D是数据输入，E常常是时钟信号CLK
- 状态Q值是数据输入D一个时钟周期前的值
  - 即D延迟一周期后变成Q

- **CPU主频是CPU时钟周期的倒数**
  - 假设时钟周期=1纳秒
  - 主频 = 1/时钟周期 = 1 GHz




| E | D | Q | $Q_{next}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

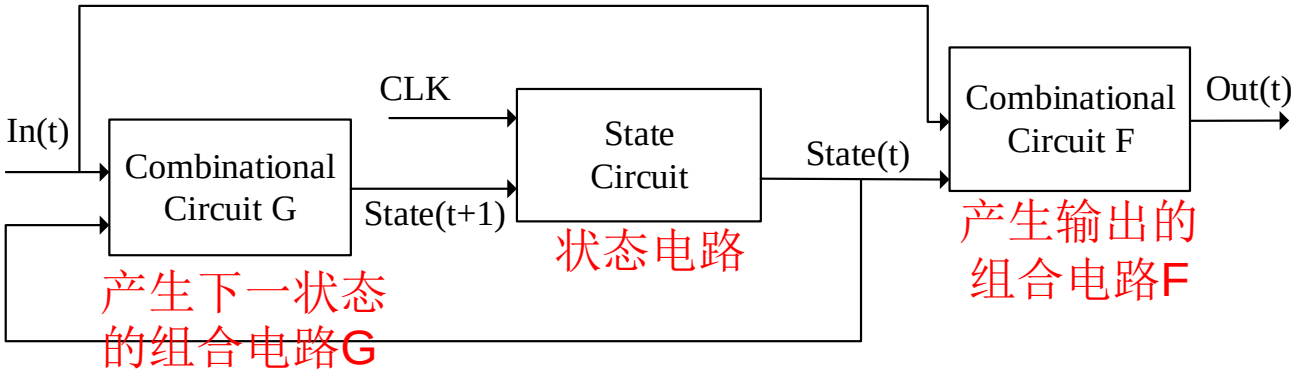



D flip-flop: Truth table　　Internal logic diagram　　Symbol

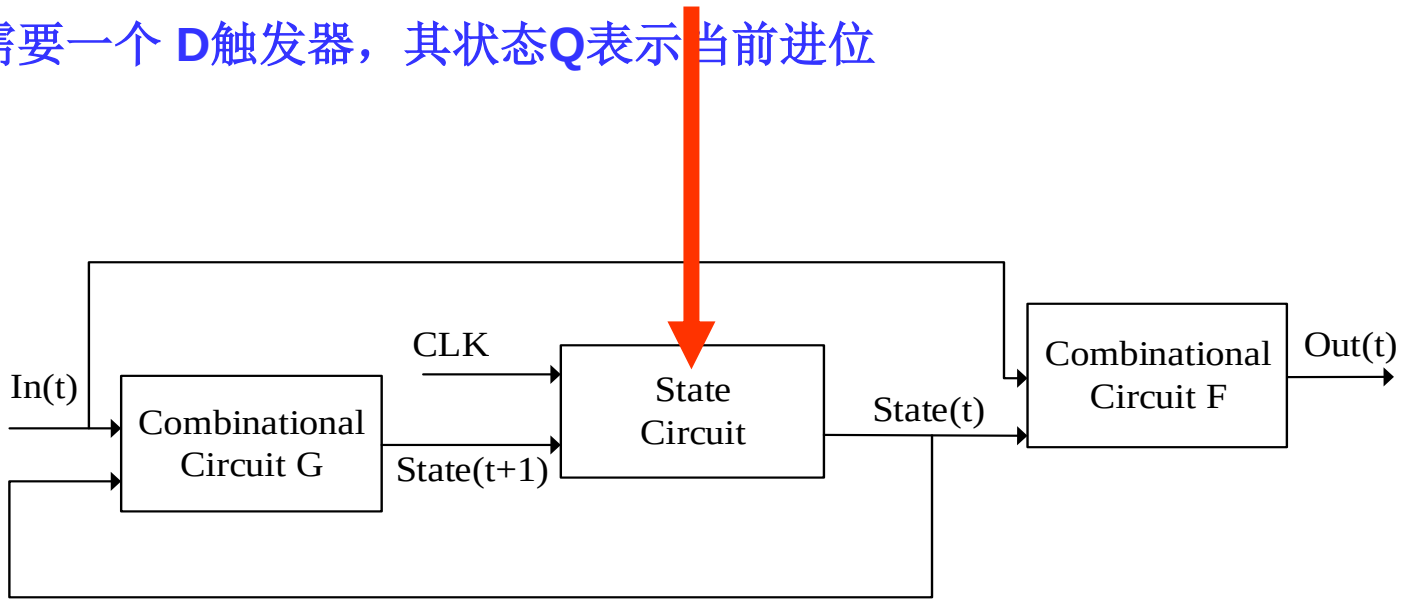
# 4.4.3 时序电路的一般组成

- 两个组合电路，一个状态电路；时钟信号驱动
- 状态电路由一个或多个D触发器组成
- 两个组合电路是
  - 输出电路F：Out(t) = F(In(t), State(t))
  - 下一状态电路G：State(t+1) = G(In(t), State(t))
- 时序电路的逻辑线路图
  - 也称为时钟同步的时序电路（synchronous sequential circuit）



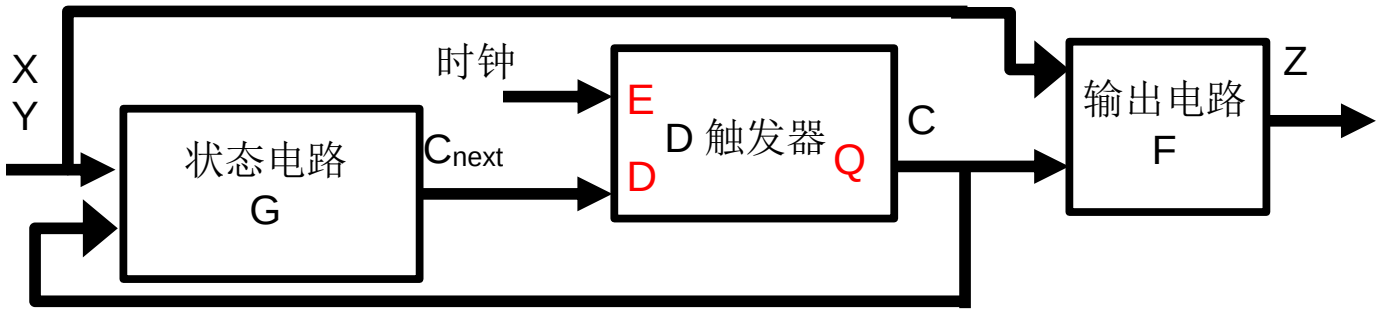
# 4.4.4 设计4位串行加法器

- 在四个时钟周期完成加法 $Z_3Z_2Z_1Z_0 = X_3X_2X_1X_0 + Y_3Y_2Y_1Y_0$
  - 每一周期完成一个全加操作
- 再用一个实例验证
  - $11_{10} + 9_{10} = 1011_2 + 1001_2 = 10100_2 = 20_{10} = 4_{10}$ and overflow
    输入X=1011，Y=1001；则输出Z=0100且产生进位溢出
- 设计过程
  - 第1步：首先确定状态电路，需要多少个D触发器？
  - 只需要一个 D触发器，其状态Q表示当前进位



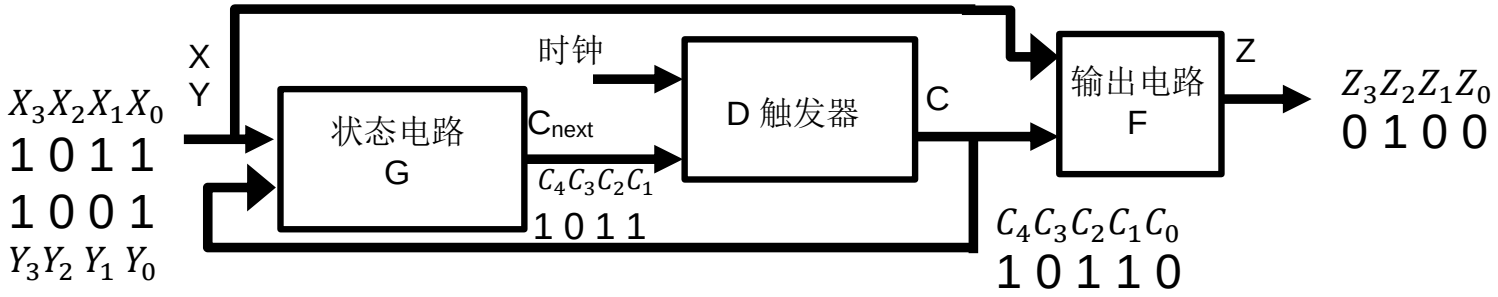
# 4.4.4 设计4位串行加法器

- 设计过程
  - 第1步：首先确定状态电路，需要多少个D触发器？
  - 只需要一个 D触发器，其状态Q表示当前进位C
  - 第2步：套用时序电路一般结构
  - In是输入X，Y；Out是输出Z；Q是进位C；Enable=时钟信号CLK

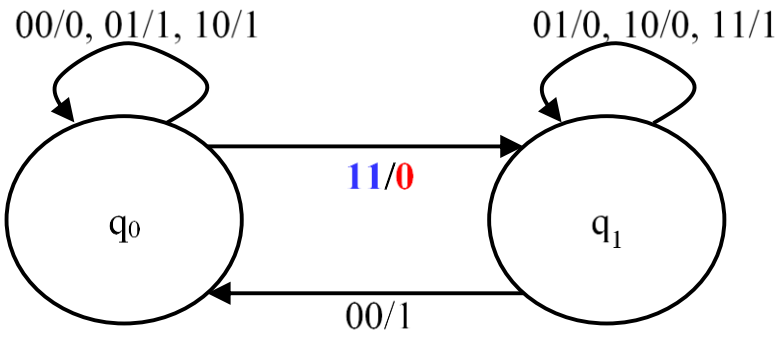


# 设计过程

- **第3步：求输出电路F与状态电路G的布尔表达式**
  - 画出时序电路的状态转换图




XY/Z



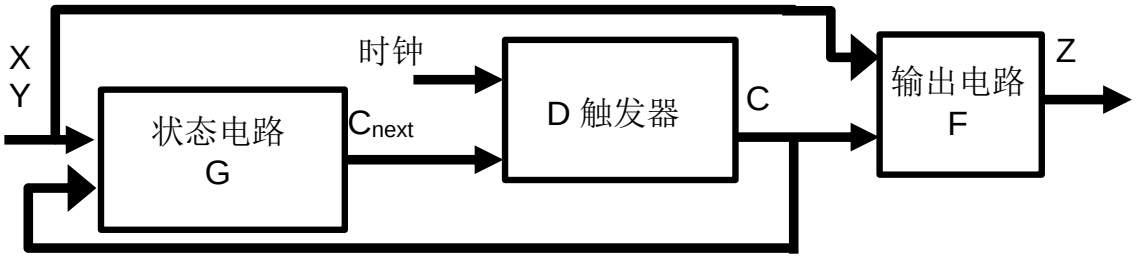

画出时序电路的状态转换图

$q_0$ denotes C=0
$q_1$ denotes C=1

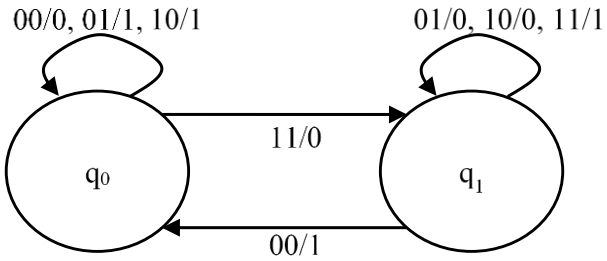
$$\begin{array}{rr} & 1\ 0\ 1\ \mathbf{1} = X \\ + & 1\ 0\ 0\ \mathbf{1} = Y \\ \hline & 1\ 0\ 1\ \mathbf{1}\ \mathbf{0} = C \\ & 0\ 1\ 0\ \mathbf{0} = Z \end{array}$$

# 设计过程

- **第3步：求输出电路F与状态电路G的布尔表达式**
  - 画出时序电路的状态转换图；进而导出真值表




XY/Z



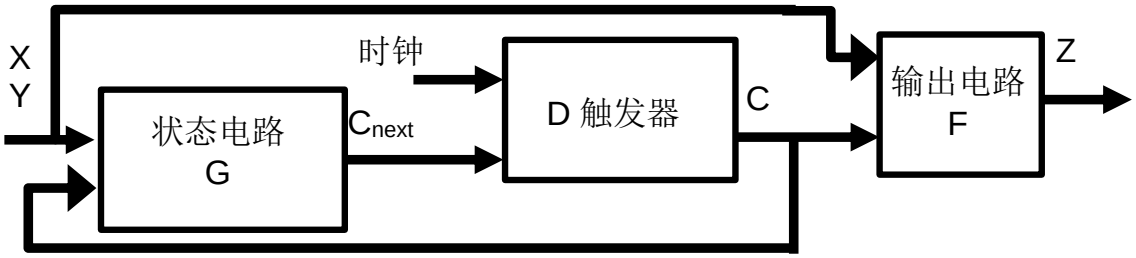

$q_0$ denotes C=0
$q_1$ denotes C=1

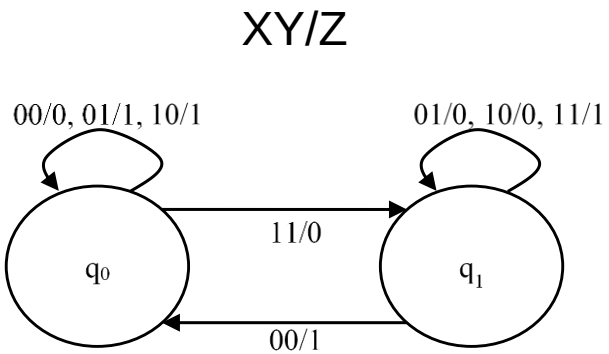
| C | X | Y | Z | $C_{next}$ |
|---|---|---|---|---|
| $q_0$ | 0 | 0 | 0 | $q_0$ |
| $q_0$ | 0 | 1 | 1 | $q_0$ |
| $q_0$ | 1 | 0 | 1 | $q_0$ |
| $q_0$ | 1 | 1 | 0 | $q_1$ |
| $q_1$ | 0 | 0 | 1 | $q_0$ |
| $q_1$ | 0 | 1 | 0 | $q_1$ |
| $q_1$ | 1 | 0 | 0 | $q_1$ |
| $q_1$ | 1 | 1 | 1 | $q_1$ |

| C | X | Y | Z | $C_{next}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

# 设计过程

- **第3步：求输出电路F与状态电路G的布尔表达式**
  - 画出时序电路的状态转换图；进而导出真值表；**进而求出F和G的布尔表达式**




$q_0$ denotes C=0
$q_1$ denotes C=1

| C | X | Y | Z | $C_{next}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

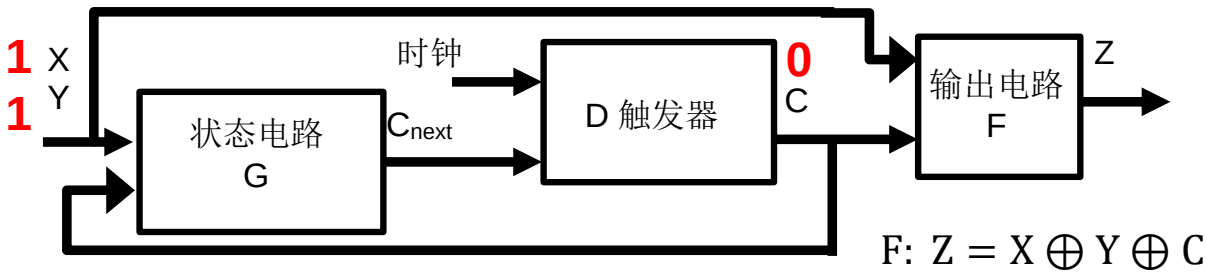
$$Z = F(X, Y, C) = X \oplus Y \oplus C$$

$$C_{next} = G(X, Y, C) = (X \cdot Y) + (X \oplus Y) \cdot C$$

# 设计过程第4步：验算

- 给定
  - （1）设计好的下图时序电路，
  - （2）输入$X_3X_2X_1X_0 = 1011$, $Y_3Y_2Y_1Y_0 = 1001$,与初始进位$C_0 = 0$
  - 正确结果应为$Z_3Z_2Z_1Z_0 = 0100$，且有$C_4 = 1$表示溢出
- 验算步骤
  - 时钟周期0：$Z_0 = X_0 \oplus Y_0 \oplus C_0 = \mathbf{1} \oplus \mathbf{1} \oplus \mathbf{0}$；
  - $C_1 = (X_0 \cdot Y_0) + (X_0 \oplus Y_0) \cdot C_0 = (\mathbf{1} \cdot \mathbf{1}) + (\mathbf{1} \oplus \mathbf{1}) \cdot \mathbf{0}$

$X_3X_2X_1X_0$
1 0 1 **1**
1 0 0 **1**
$Y_3Y_2\ Y_1\ Y_0$



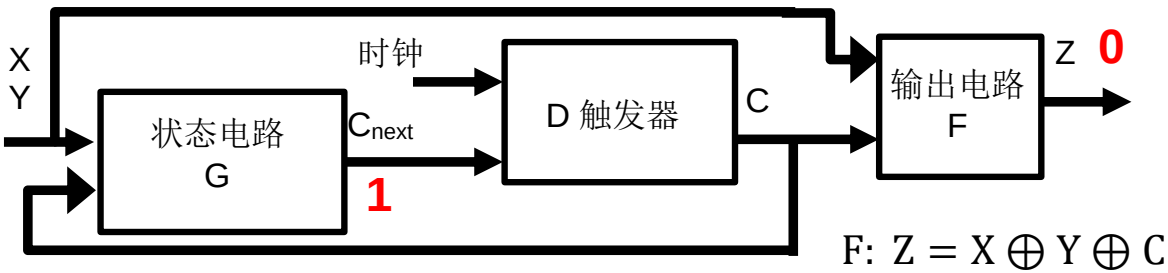

$Z_3Z_2Z_1Z_0$

$C_4C_3C_2C_1C_0$
**0**

# 设计过程第4步：验算

- 给定
  - （1）设计好的下图时序电路，
  - （2）输入$X_3X_2X_1X_0 = 1011$, $Y_3Y_2Y_1Y_0 = 1001$,与初始进位$C_0 = 0$
  - 正确结果应为$Z_3Z_2Z_1Z_0 = 0100$，且有$C_4 = 1$表示溢出
- 验算步骤
  - 时钟周期0：$Z_0 = X_0 \oplus Y_0 \oplus C_0 = 1 \oplus 1 \oplus 0 = \mathbf{0}$；
  - $C_1 = (X_0 \cdot Y_0) + (X_0 \oplus Y_0) \cdot C_0 = (1 \cdot 1) + (1 \oplus 1) \cdot 0 = \mathbf{1}$

$X_3X_2X_1X_0$
1 0 1 **1**
1 0 0 **1**
$Y_3Y_2\ Y_1\ Y_0$



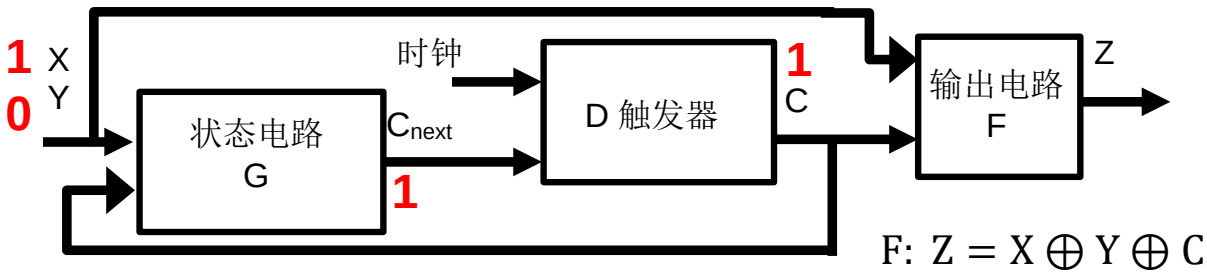

$Z_3Z_2Z_1Z_0$
**0**

$C_4C_3C_2C_1C_0$
**1** 0

# 设计过程第4步：验算

- 给定
  - （1）设计好的下图时序电路，
  - （2）输入$X_3X_2X_1X_0=1011$, $Y_3Y_2Y_1Y_0=1001$,与初始进位$C_0=0$
  - 正确结果应为$Z_3Z_2Z_1Z_0=0100$，且有$C_4=1$表示溢出
- 验算步骤
  - 时钟周期0：$Z_0=X_0\oplus Y_0\oplus C_0=1\oplus1\oplus0=0$；$C_1=(X_0\cdot Y_0)+(X_0\oplus Y_0)\cdot C_0=(1\cdot1)+(1\oplus1)\cdot0=1$
  - **时钟周期1**：$Z_1=X_1\oplus Y_1\oplus C_1=\mathbf{1}\oplus\mathbf{0}\oplus\mathbf{1}$；$C_2=(X_1\cdot Y_1)+(X_1\oplus Y_1)\cdot C_1=(\mathbf{1}\cdot\mathbf{0})+(\mathbf{1}\oplus\mathbf{0})\cdot\mathbf{1}$

$X_3X_2X_1X_0$
1 0 **1** 1
1 0 **0** 1
$Y_3Y_2\ Y_1\ Y_0$



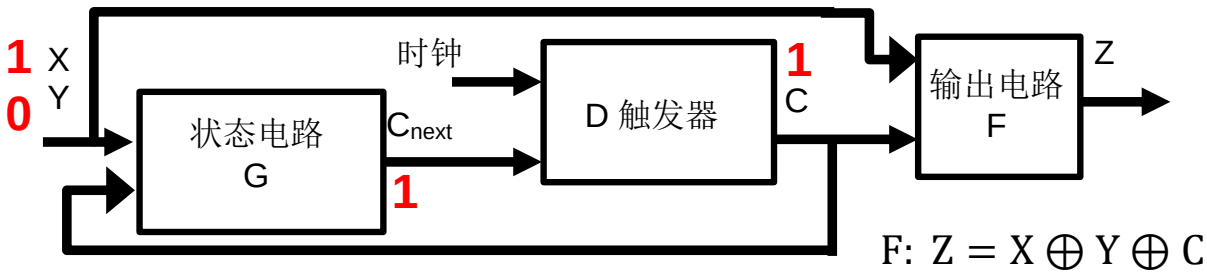

$Z_3Z_2Z_1Z_0$
0

$C_4C_3C_2C_1C_0$
**1** 0

# 设计过程第4步：验算

- 给定
  - （1）设计好的下图时序电路，
  - （2）输入$X_3X_2X_1X_0=1011$, $Y_3Y_2Y_1Y_0=1001$,与初始进位$C_0=0$
  - 正确结果应为$Z_3Z_2Z_1Z_0=0100$，且有$C_4=1$表示溢出
- 验算步骤
  - 时钟周期0：$Z_0=X_0\oplus Y_0\oplus C_0=1\oplus1\oplus0=0$；$C_1=(X_0\cdot Y_0)+(X_0\oplus Y_0)\cdot C_0=(1\cdot1)+(1\oplus1)\cdot0=1$
  - **时钟周期1**：$Z_1=X_1\oplus Y_1\oplus C_1=1\oplus0\oplus1=$ **0**；$C_2=(X_1\cdot Y_1)+(X_1\oplus Y_1)\cdot C_1=(1\cdot0)+(1\oplus0)\cdot1=$ **1**

$X_3X_2X_1X_0$
1 0 **1** 1
1 0 **0** 1
$Y_3Y_2\ Y_1\ Y_0$



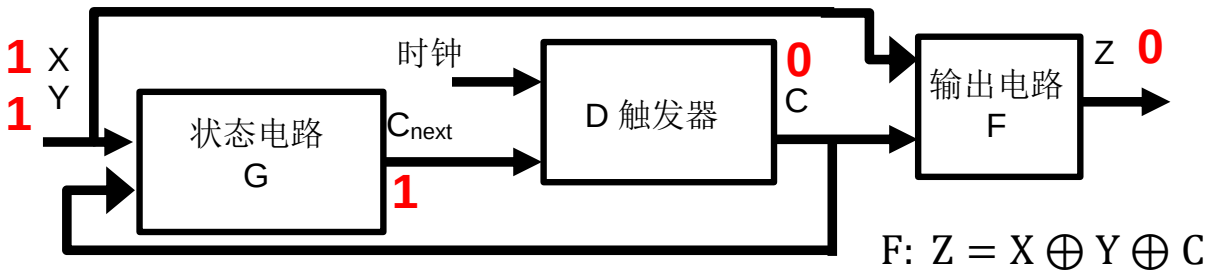

$Z_3Z_2Z_1Z_0$
**0** 0

$C_4C_3C_2C_1C_0$
**1** 1 0

# 设计过程第4步：验算

- 给定
  - （1）设计好的下图时序电路，
  - （2）输入$X_3X_2X_1X_0 = 1011$, $Y_3Y_2Y_1Y_0 = 1001$,与初始进位$C_0 = 0$
  - 正确结果应为$Z_3Z_2Z_1Z_0 = 0100$，且有$C_4 = 1$表示溢出
- 验算步骤
  - 时钟周期0：$Z_0 = X_0 \oplus Y_0 \oplus C_0 = 1 \oplus 1 \oplus 0 = 0$；$C_1 = (X_0 \cdot Y_0) + (X_0 \oplus Y_0) \cdot C_0 = (1 \cdot 1) + (1 \oplus 1) \cdot 0 = 1$
  - 时钟周期1：$Z_1 = X_1 \oplus Y_1 \oplus C_1 = 1 \oplus 0 \oplus 1 = 0$；$C_2 = (X_1 \cdot Y_1) + (X_1 \oplus Y_1) \cdot C_1 = (1 \cdot 0) + (1 \oplus 0) \cdot 1 = 1$
  - 时钟周期2：$Z_2 = X_2 \oplus Y_2 \oplus C_2 = 0 \oplus 0 \oplus 1 = 1$；$C_3 = (X_2 \cdot Y_2) + (X_2 \oplus Y_2) \cdot C_2 = (0 \cdot 0) + (0 \oplus 0) \cdot 1 = 0$
  - 时钟周期3：$Z_3 = X_3 \oplus Y_3 \oplus C_3 = 1 \oplus 1 \oplus 0 = 0$；$C_4 = (X_3 \cdot Y_3) + (X_3 \oplus Y_3) \cdot C_3 = (1 \cdot 1) + (1 \oplus 1) \cdot 0 = 1$

$X_3X_2X_1X_0$
1 0 1 1
1 0 0 1
$Y_3Y_2\ Y_1\ Y_0$



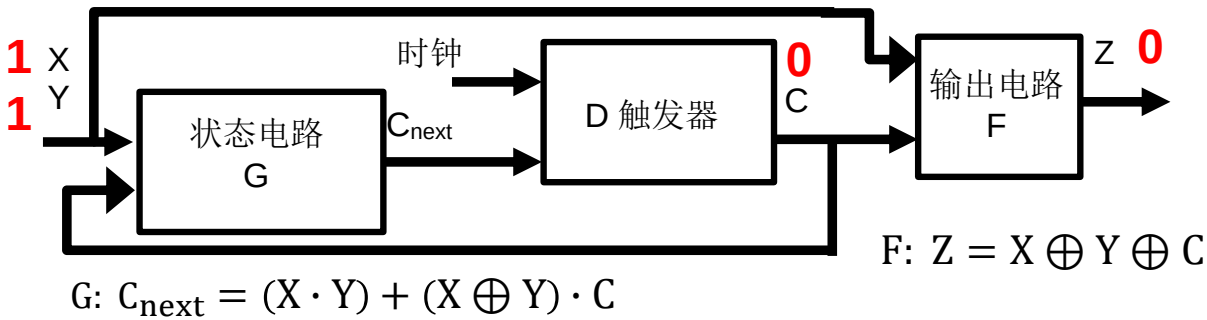

$Z_3Z_2Z_1Z_0$
0 1 0 0

$C_4C_3C_2C_1C_0$
1 0 1 1 0

# 设计过程第4步：验算

- 给定
  - （1）设计好的下图时序电路，
  - （2）输入$X_3X_2X_1X_0 = 1011$, $Y_3Y_2Y_1Y_0 = 1001$,与初始进位$C_0 = 0$
  - 正确结果应为$Z_3Z_2Z_1Z_0 = 0100$，且有$C_4 = 1$表示溢出
- 验算步骤
  - 时钟周期0：$Z_0 = X_0 \oplus Y_0 \oplus C_0 = 1 \oplus 1 \oplus 0 = \mathbf{0}$；$C_1 = (X_0 \cdot Y_0) + (X_0 \oplus Y_0) \cdot C_0 = (1 \cdot 1) + (1 \oplus 1) \cdot 0 = \mathbf{1}$
  - 时钟周期1：$Z_1 = X_1 \oplus Y_1 \oplus C_1 = 1 \oplus 0 \oplus 1 = \mathbf{0}$；$C_2 = (X_1 \cdot Y_1) + (X_1 \oplus Y_1) \cdot C_1 = (1 \cdot 0) + (1 \oplus 0) \cdot 1 = 1$
  - 时钟周期2：$Z_2 = X_2 \oplus Y_2 \oplus C_2 = 0 \oplus 0 \oplus 1 = \mathbf{1}$；$C_3 = (X_2 \cdot Y_2) + (X_2 \oplus Y_2) \cdot C_2 = (0 \cdot 0) + (0 \oplus 0) \cdot 1 = 0$
  - 时钟周期3：$Z_3 = X_3 \oplus Y_3 \oplus C_3 = 1 \oplus 1 \oplus 0 = \mathbf{0}$；$C_4 = (X_3 \cdot Y_3) + (X_3 \oplus Y_3) \cdot C_3 = (1 \cdot 1) + (1 \oplus 1) \cdot 0 = \mathbf{1}$

$X_3X_2X_1X_0$
1 0 1 **1**
1 0 0 **1**
$Y_3Y_2\ Y_1\ Y_0$




$Z_3Z_2Z_1Z_0$
0 1 0 0

$C_4C_3C_2C_1C_0$
1 0 1 1 **0**