北京大学
PEKING UNIVERSITY

智能硬件体系结构

第三讲：电路基础-晶体管与数字电路设计-2

主讲：陶耀宇、李萌

2024年秋季

注意事项

北京大学 PEKING UNIVERSITY

- **课程作业情况**

第1次作业题已经于**9.20日周五**放在课程网站上

作业提交截止日期是国庆节后的**10月8日23:59**

作业内容将涵盖9.18和9.25号的课程内容

提交方式：电子版教学网提交

第1次编程作业：**10月8号~11月8号**

作业说明

北京大学 PEKING UNIVERSITY

1、CMOS逻辑电路（25分） 只能用NAND-2逻辑门

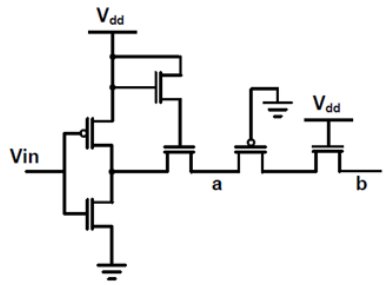
1）最少需要多少个两输入的NAND逻辑门可以构建一个两输入的MUX模块？（3分）

（*MUX模块包含三个输入A、B、SEL，一个输出OUT：当输入SEL = 0，则输出OUT = A；当输入SEL = 1，则输出OUT = B。如果能够使用两输入NAND逻辑门实现MUX，请画出最少门数的MUX电路图；如不能，请说明理由。*）

2）请画出包含PUN和PDN的AOI逻辑门电路（OUT = $\overline{A*B+C}$）（+为OR，*为AND，上划线为INV），并计算相对于输入C的逻辑功效（Logical Effort, g）。（3分）

（*AOI逻辑门是一个独立的逻辑门电路，并不是由独立的AND、OR、INV等组成。*）

3）假设以下电路中，Vdd = 1V，NMOS阈值电压$V_{th,N}$ = 0.3V，PMOS阈值电压$V_{th,P}$ = -0.3V：1）Vin = 0时，Va和Vb分别是多少？2）Vin = 1时，Va和Vb分别是多少？（4分）



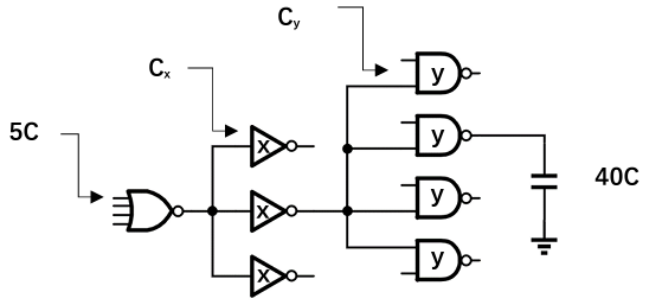
5）逻辑门输入顺序重排：针对上述问题 4）中的电路，分析 A、B、C、D 不同输入顺序导致 F =1->0 的延迟情况。假设：尺寸为 1 的 NMOS 或 PMOS 晶体管其寄生电容为 C，输出 F 上的总电容 C_F、AB 之间电容 C_{AB}、BC 之间电容 C_{BC} 均为相应节点所连晶体管的尺寸总和乘以 C；尺寸为 1 的 NMOS 晶体管等效电阻 = 尺寸为 2 的 PMOS 晶体管 = R。（5 分）

即分析ABCD各自仅有自己变化使得F由一变零时的延迟

比如：对于A的延迟情况，BC输入为1，D输入为0，A由1变0给F带来的延迟

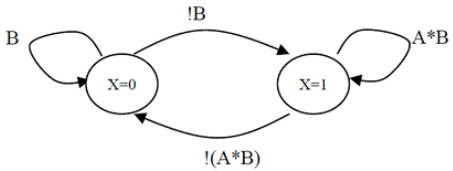
作业说明

1b）假设 r = 3，第一级逻辑门输入电容为 5C，链路负载为 40C，计算该链路的最短延迟 D 和相应的 C_x、C_y。



第2级每个反向器、第3级每个NAND都相同

3、状态机和复杂电路单元（25 分）



画出逻辑门级别的电路图

1）采用两输入的 AND、OR、XOR、反向器、锁存器这 5 种电路模块，完成上述状态机电路的设计。（输入仅为 A、B、Clock，输出仅为 X，*符号为 AND，！符号为 INV）（5 分）

作业说明

北京大学 PEKING UNIVERSITY

3）乘法器设计：利用 Radix-4 Booth 编码，阐明 8bit 补码整型数-30 x 26 的步骤（其中-30 为被乘数、26 为乘数）。A 为当前部分和累加结果、Q 为乘数。（10 分）

3a）将-30 和 26 转化为二进制 bit 串；

3b）按照以下格式完成部分和累加的迭代步骤。

Step i	A = ?	Q = ?	Q-1 = ?	步骤解释

可以不按照该推荐格式完成累加步骤，只需要写清每一步的部分和、累加结果和布斯编码选择即可

4、开放式问题 – 更加复杂的计算单元（15 分）

在课堂中，我们学习了如何设计加法器和乘法器等简单计算电路。对于更复杂的逻辑功能，计算单元的设计变得尤为重要。假设某芯片公司需要设计输入 X 为 8 比特整型数（补码 INT8）、输出 Y 为 16 比特浮点数（FP16 包含 1bit 符号位、5bit 指数位、10bit 尾数位）的复杂计算单元。针对以下两个问题，阐明如何利用数字逻辑电路进行计算单元设计，并利用 Python、C/C++或其他代码进行硬件行为的比特级模拟仿真与验证，研究所设计的复杂计算单元输出结果与电脑软件参考输出的误差，并简单探讨减小计算误差的方法及其可能造成的硬件面积、计算延迟代价。

4a）如何利用数字逻辑电路计算正弦函数 Y = sin(X)?

4b）如何利用数字逻辑电路计算平方根函数 Y = $\sqrt{X}$?

正弦函数的推荐方法：CORDIC、泰勒展开等

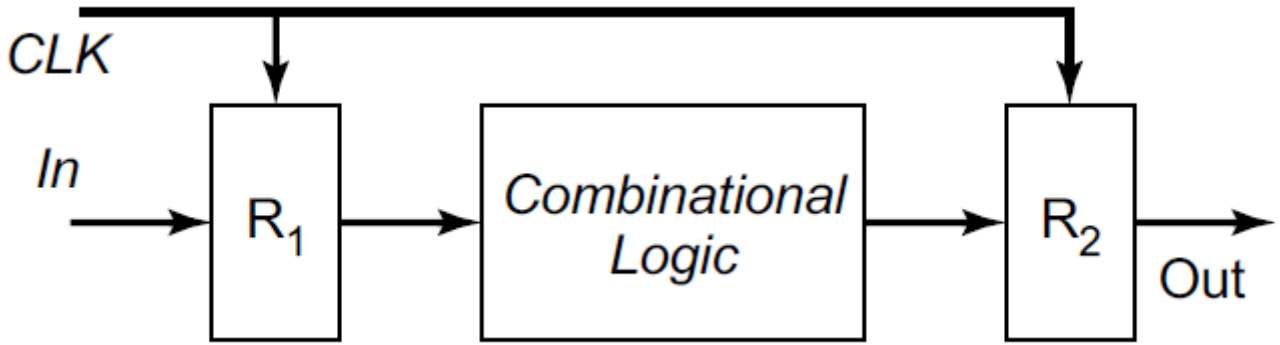
平方根的推荐方法：迭代法、逐次逼近法等

目 录

CONTENTS

电路时序的基本概念

- 同步时序（Synchronous Timing）

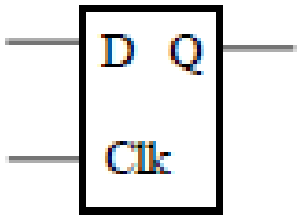


触发器
（Register）

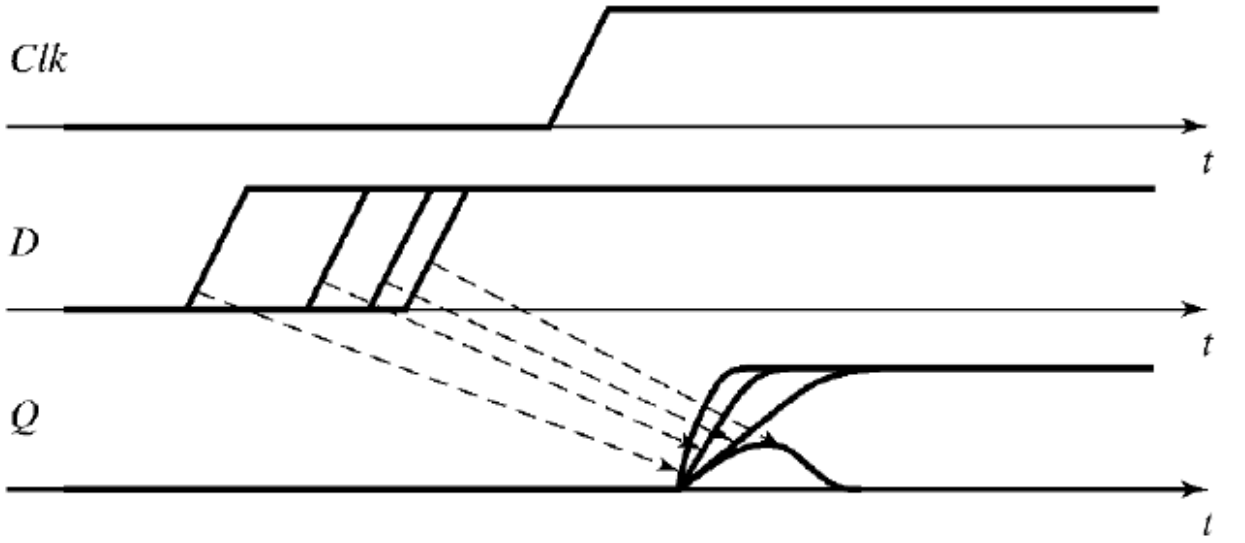
组合逻辑（各种逻辑门电路）

电路时序的基本概念

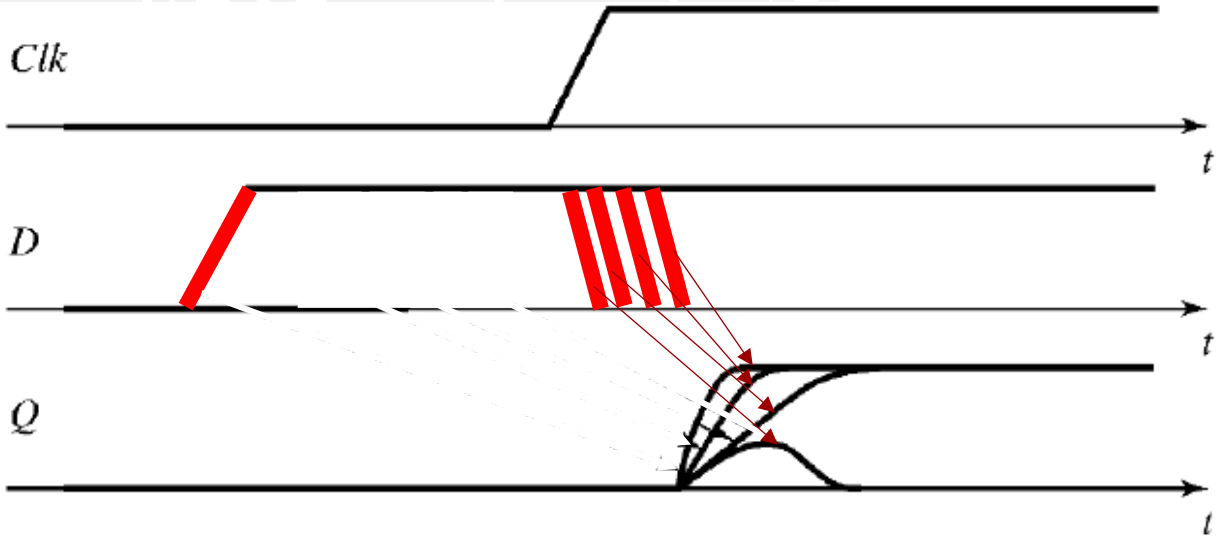
- 触发器的Setup Time和Hold Time



Setup Time

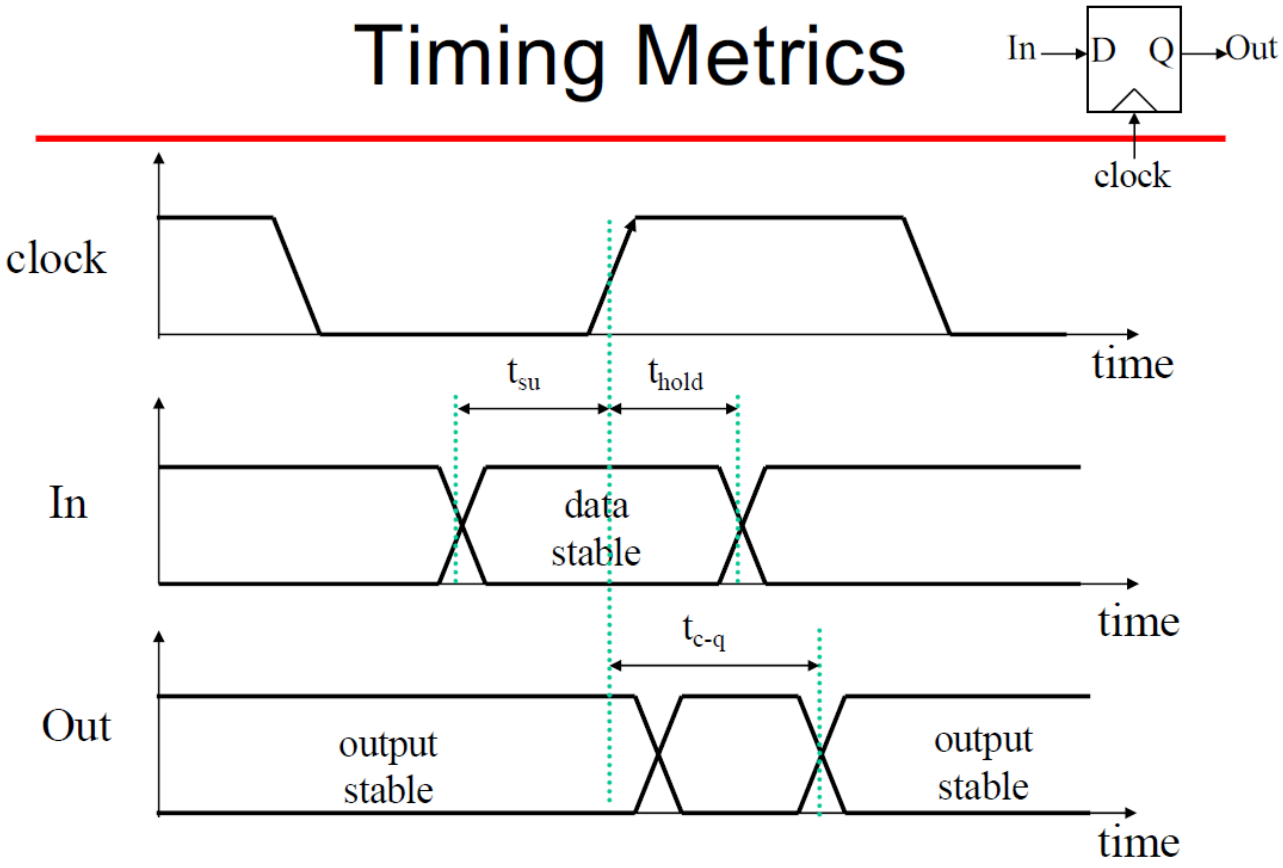


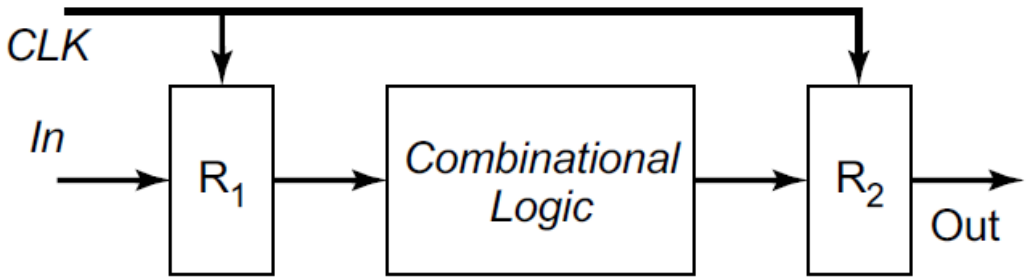
Hold Time



电路时序的基本概念

- 同步时序（Synchronous Timing）





$$T_{c\text{-}q} + t_{plogic,min} \geq t_{hold}$$

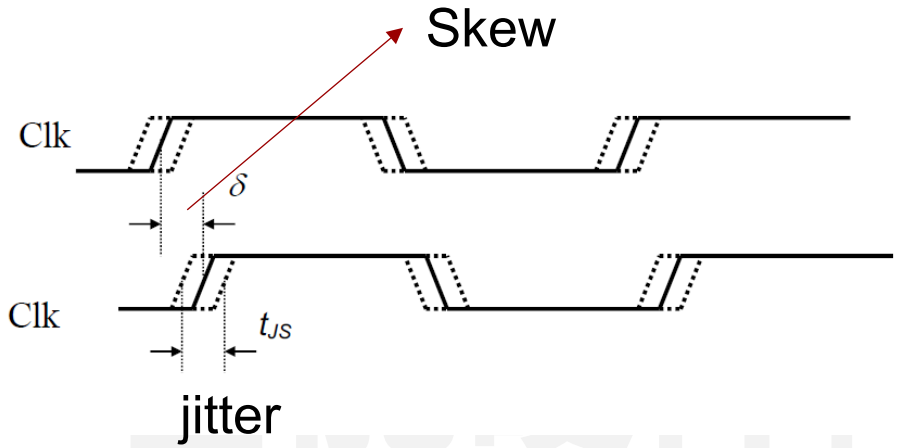
min

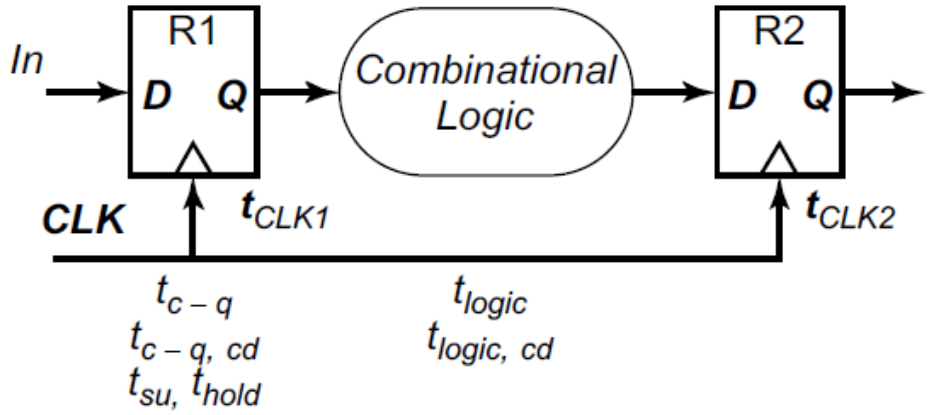
$$T \geq t_{c\text{-}q} + t_{plogic,max} + t_{su}$$

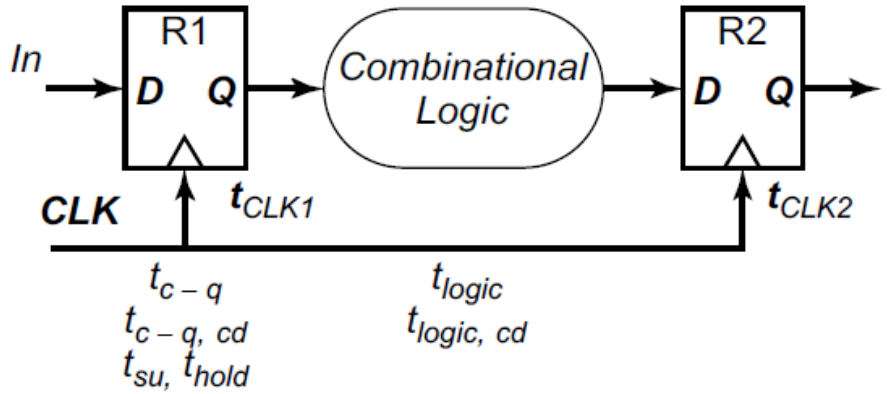
max

电路时序的基本概念

- 时钟的不稳定性







Minimum cycle time:

$T \geq t_{c\text{-}q} + t_{su} + t_{logic} - \delta$

作业题

Hold time constraint:

$t_{(c\text{-}q,\ cd)} + t_{(logic,\ cd)} > t_{hold} + \delta$

Worst case is when receiving edge arrives early (negative δ)

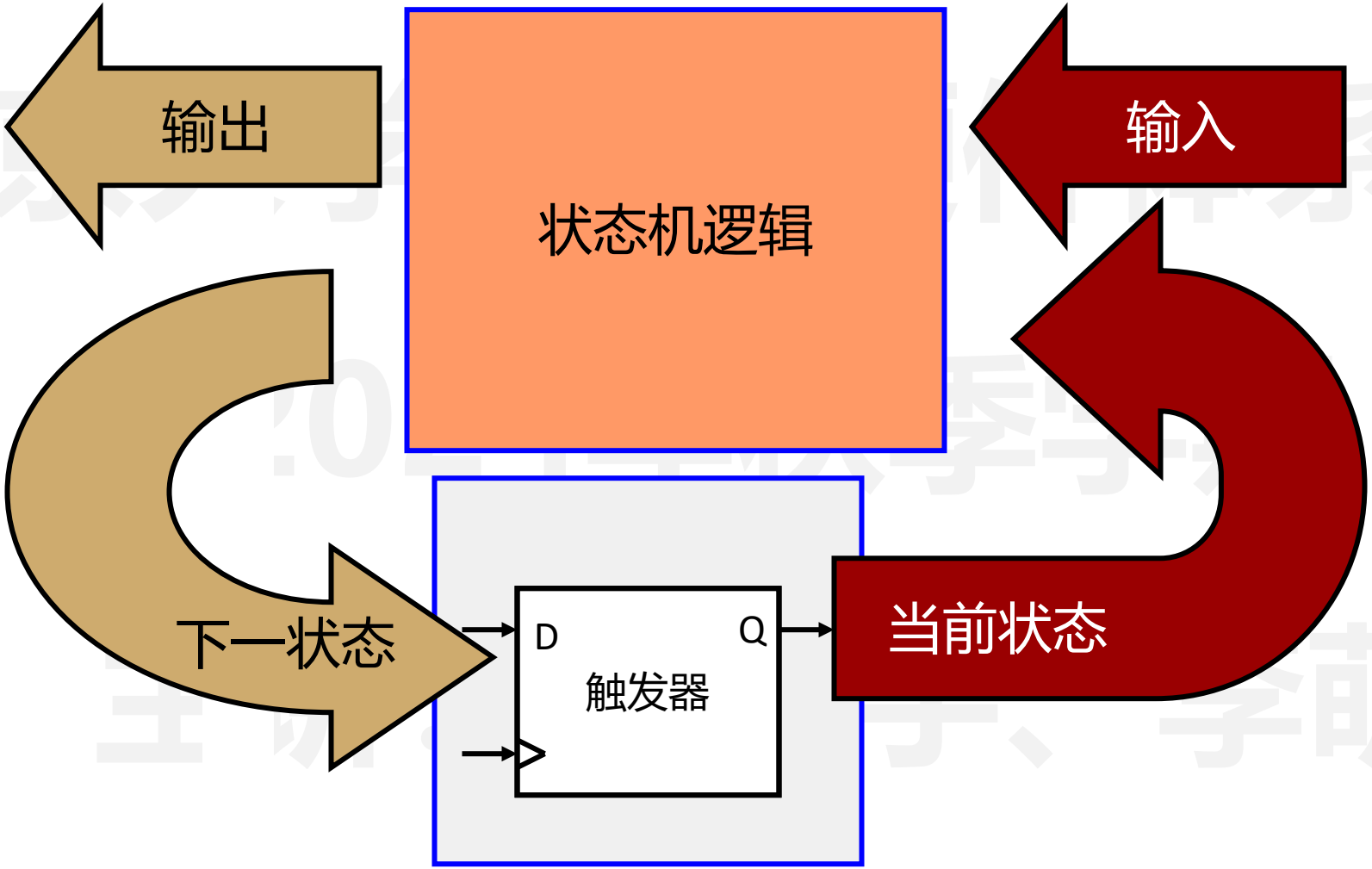
Worst case is when receiving edge arrives late (positive skew)
Race between data and clock
cd: contamination delay (fastest possible delay)

目 录

CONTENTS

为什么需要状态机

- 控制电路的基石



为什么需要状态机

- 控制电路的基石

状态机实例1 – 控制一个红绿灯

- 仅考虑红灯和绿灯，灯转换的速度不快于每次30s（0.033 Hz 时钟）
- 2个输出
  - NSlight：1=南北向为绿灯；0=南北向红灯
  - EWlight：1=东西向为绿灯；0=东西向为红灯
- 2个输入
  - Nscar：1=南北向有车等；0=南北向无车等
  - Ewcar：1=东西向有车等；0=南北向无车等
- 规则
  - 交通灯切换到另一个方向当且仅当另一方向有车等
  - 否则，保持当前交通灯不变

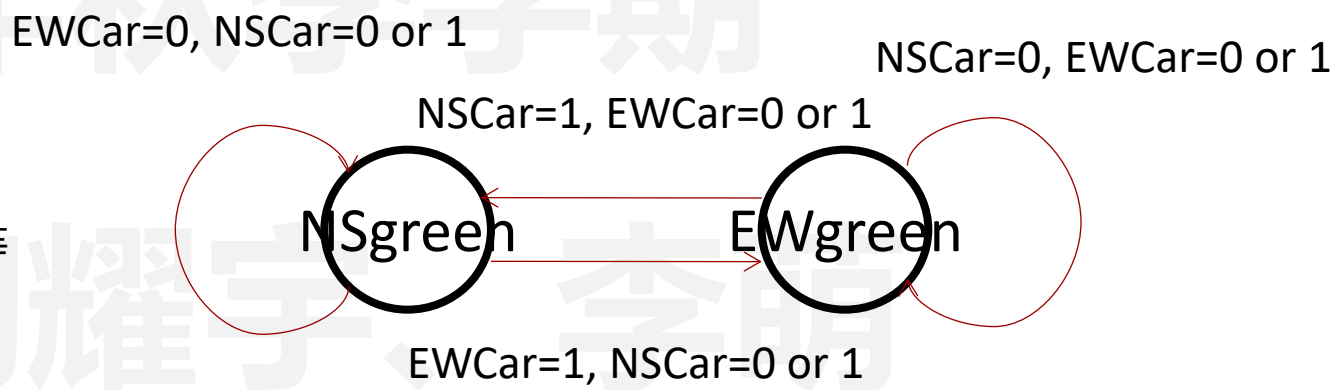
为什么需要状态机

- 控制电路的基石

状态机实例1 – 控制一个红绿灯

- 2个输出
  - NSlight：1=南北向为绿灯；0=南北向红灯
  - EWlight：1=东西向为绿灯；0=东西向为红灯
- 2个输入
  - Nscar：1=南北向有车等；0=南北向无车等
  - Ewcar：1=东西向有车等；0=南北向无车等
- 规则
  - 交通灯切换到另一个方向当且仅当另一方向有车等
  - 否则，保持当前交通灯不变

- 需要2个状态
  - Nsgreen、EWgreen



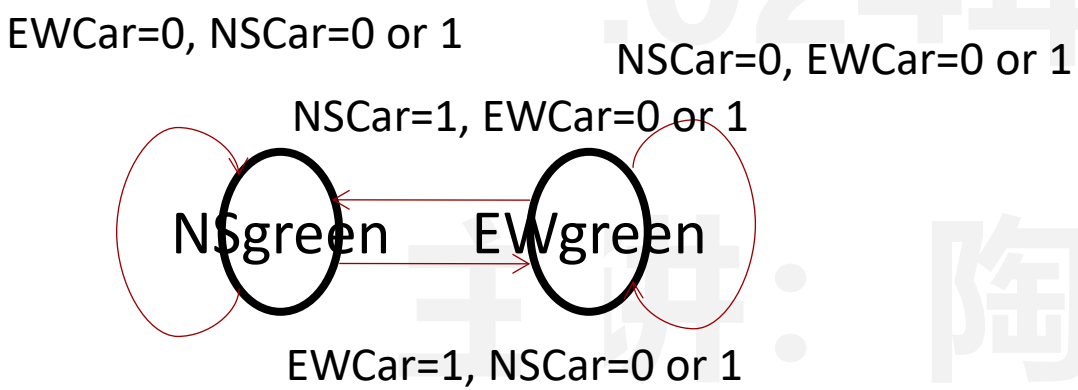
为什么需要状态机

- 控制电路的基石

状态机实例1 – 控制一个红绿灯

- 需要2个状态
  - Nsgreen、EWgreen

Current state	Inputs		Next state
	NScar	EWcar	
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen



Current state	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

$$\text{NSlite} = \overline{\text{CurrentState}}$$

$$\text{EWlite} = \text{CurrentState}$$

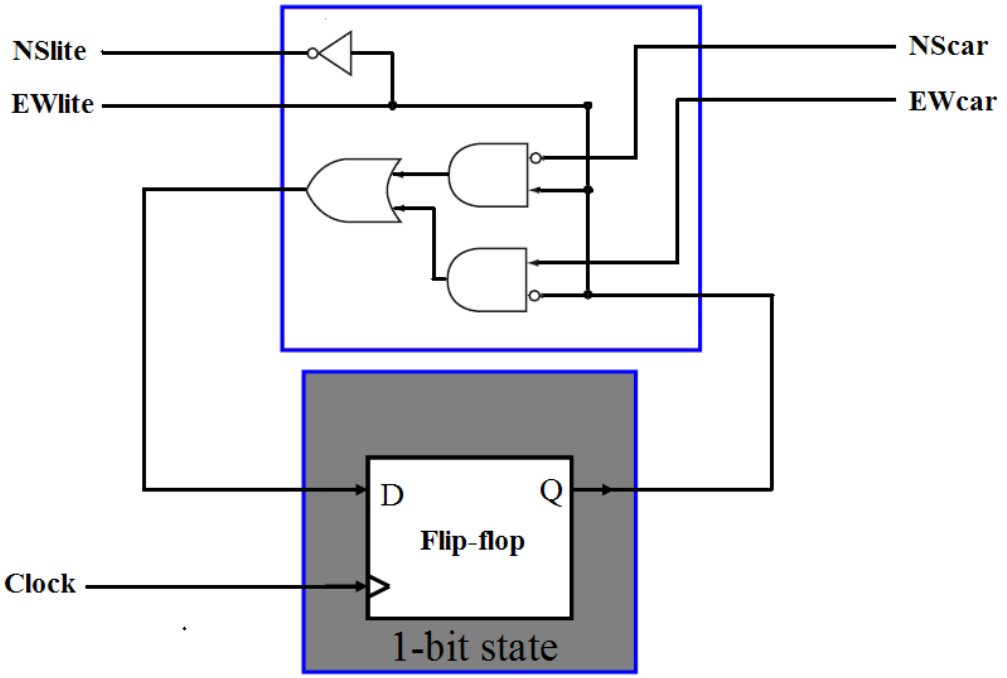
为什么需要状态机

- 控制电路的基石

状态机实例1 – 控制一个红绿灯

Current state	Inputs		Next state
	NScar	EWcar	
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen

Current state	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1



$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

$$\text{NSlite} = \overline{\text{CurrentState}}$$

$$\text{EWlite} = \text{CurrentState}$$

为什么需要状态机

- 控制电路的基石

Step 1 – 定义状态并画出状态转换图

Step 2 – 给每一个状态赋值并更新状态转换图

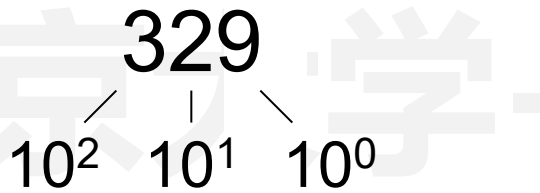
Step 3 – 根据状态转换图写出下一状态和输出的逻辑表达式

Step 4 – 画出实际电路图

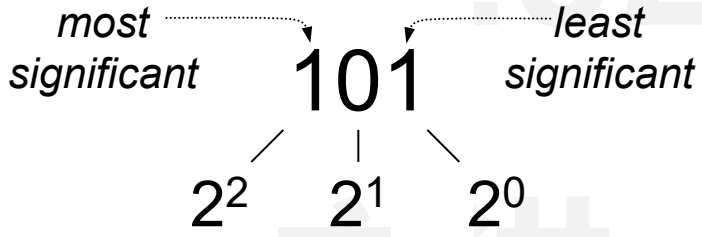
状态将在每一个时钟上升沿更新

量化与数据格式

- 最基础的二进制数据格式 – 原码（无符号数）



3x100 + 2x10 + 9x1 = 329



1x4 + 0x2 + 1x1 = 5

2^2	2^1	2^0	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

An *n*-bit unsigned integer

represents 2^n values:

from 0 to 2^n-1

量化与数据格式

- 最基础的二进制数据格式 – 原码（无符号数）

```
  10010        10010        1111
+  1001      +  1011      +    1
-------      -------      ------
  11011        11101        10000

                10111
              +   111
              -------
                11110
```

carry

量化与数据格式

- **有符号数的表现格式**

With n bits, we have 2^n distinct values.

- **assign about half to positive integers (1 through 2^{n-1}-1) and about half to negative (- (2^{n-1}-1) through -1)**
- **that leaves two values: one for 0, and one extra**

正整数

just like unsigned – zero in most significant bit

00101 = 5

负整数

sign-magnitude – set top bit to show negative,

other bits are the same as unsigned

10101 = -5

量化与数据格式

- **有符号数的表现格式**

With n bits, we have 2^n distinct values.

- **assign about half to positive integers (1 through 2^{n-1}-1) and about half to negative (- (2^{n-1}-1) through -1)**
- **that leaves two values: one for 0, and one extra**

正整数

just like unsigned – zero in most significant bit

00101 = 5

负整数

sign-magnitude – set top bit to show negative,

other bits are the same as unsigned

10101 = -5

量化与数据格式

- **有符号数的表现格式 – 补码**

sign-magnitude有什么问题？

0有两种重复表示 (+0 and –0)

计算电路复杂

Adding a negative number => subtraction

Need to “correct” result to account for borrowing

```
  00101  (5)        01001  (9)
+ 11011  (-5)     + 10111  (-9)
  00000  (0)        00000  (0)
```

Two’s complement representation developed to make circuits easy for arithmetic.

for each positive number (X), assign value to its negative (-X), such that X + (-X) = 0 with “normal” addition, ignoring carry out

量化与数据格式

- 有符号数的表现格式 – 补码

If number is positive or zero,

- normal binary representation, zeroes in upper bit(s)

If number is negative,

- start with positive number
- flip every bit (i.e., take the one’s complement)
- then add one

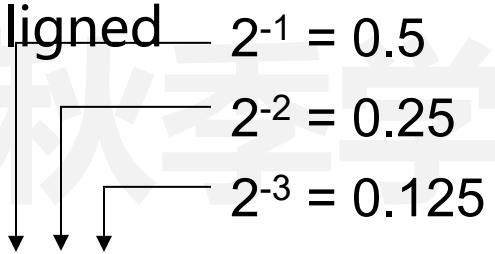
```
  00101  (5)          01001  (9)
  11010  (1's comp)   10110  (1's comp)
+     1              +     1
  11011  (-5)         10111  (-9)
```

北京大学 PEKING UNIVERSITY

- 定点数 – Fixed-point

How can we represent fractions?

- Use a "binary point" to separate positive from negative powers of two -- just like "decimal point."
- 2's comp addition and subtraction still work.
  - if binary points are aligned




```
  00101000.101  (40.625)
+ 11111110.110  (-1.25)
  --------------
  00100111.011  (39.375)
```

No new operations -- same as integer arithmetic.

量化与数据格式

- 特别大和特别小的数：浮点数 – Floating-point

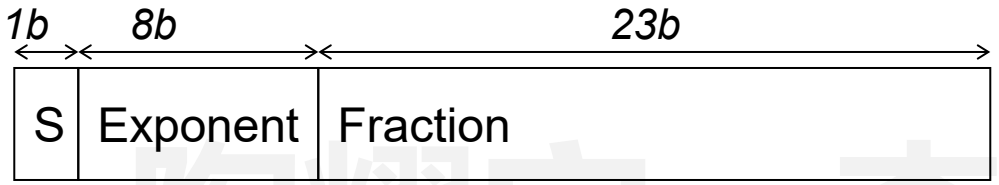
Large values: 6.023×10^{23} -- requires 79 bits

Small values: 6.626×10^{-34} -- requires >110 bits

Use equivalent of “scientific notation”: $F \times 2^{E}$

Need to represent F (*fraction*), E (*exponent*), and sign.

IEEE 754 Floating-Point Standard (32-bits):



$$N = -1^{S} \times 1.\text{fraction} \times 2^{\text{exponent}-127}, \quad 1 \le \text{exponent} \le 254$$

$$N = -1^{S} \times 0.\text{fraction} \times 2^{-126}, \quad \text{exponent} = 0$$

量化与数据格式

- **特别大和特别小的数：浮点数 – Floating-point**

Single-precision IEEE floating point number:

- Sign is 1 – number is negative.
- Exponent field is 01111110 = 126 (decimal).
- Fraction is 0.10000000000… = 0.5 (decimal).

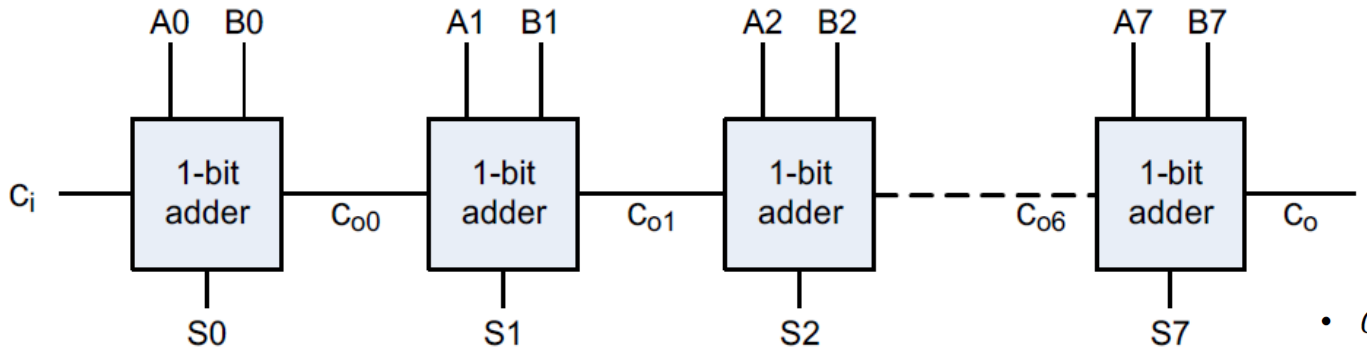
Value = $-1.5 \times 2^{(126-127)} = -1.5 \times 2^{-1} = -0.75$.

目 录

CONTENTS

加法器设计

- Ripple Carry加法器电路



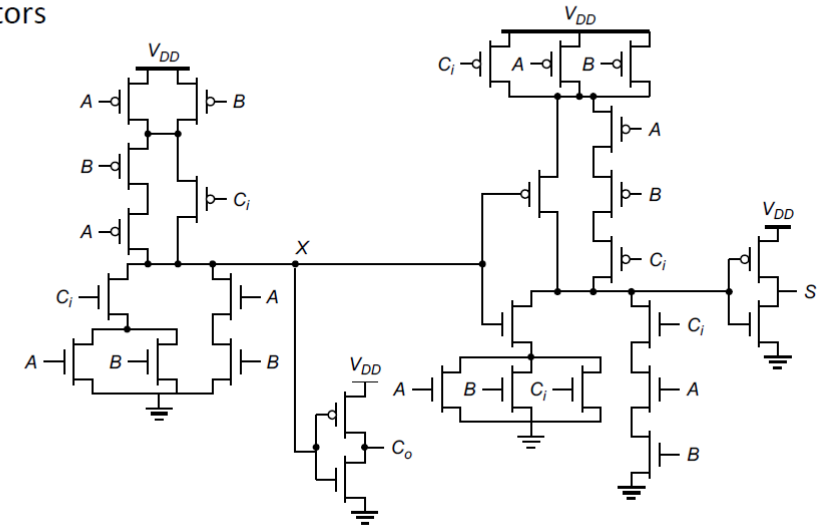
Worst case delay linear with the number of bits

$$t_d = O(N)$$

$$t_{adder} = (N\text{-}1)t_{carry} + t_{sum}$$

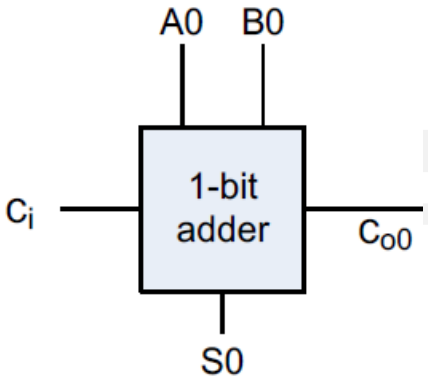
Goal: Make the fastest possible carry path circuit

- $C_o = AB + BC_i + AC_i = AB + (A + B)C_i$
- $S = A \oplus B \oplus C_i = ABC_i + \overline{C_o}(A + B + C_i)$
- 28 transistors



加法器设计

- 基于PGK的加法器设计方法



Generate (G) = AB

Propagate (P) = A ⊕ B

- Generate: C_{out} = 1 independent of C_{in}
  - G = A • B
- Propagate: C_{out} = C_{in}
  - P = A ⊕ B
- Kill: C_{out} = 0 independent of C_{in}
  - K = ~A • ~B

$$C_o(G,P) = G + PC_i$$

$$S(G,P) = P \oplus C_i$$

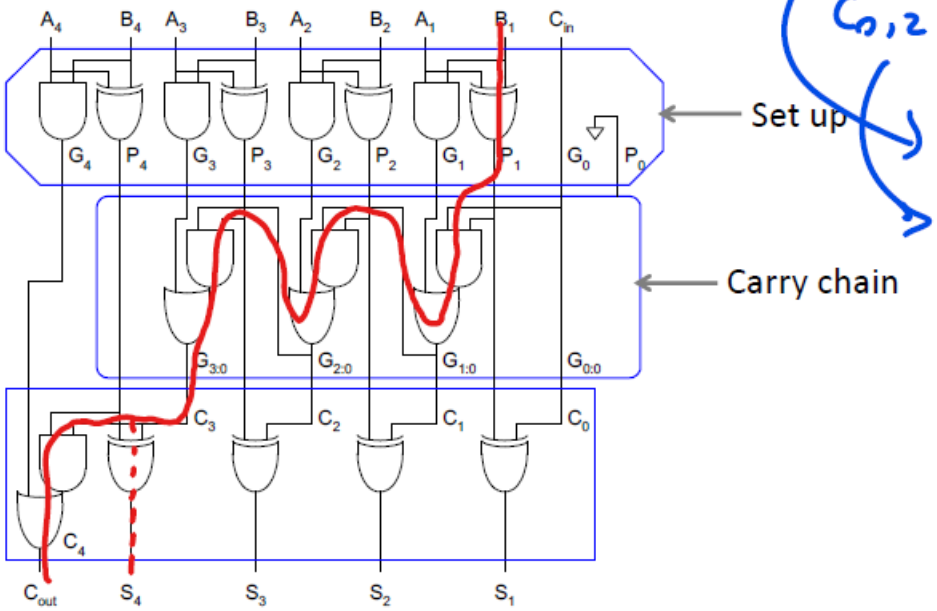
$P = A \oplus B$

$P = A + B$

- 基于PGK的加法器设计方法

Carry-Ripple using P and G

$$C_{i:0} = G_i + P_i \cdot C_{i-1:0}$$
$$G_{0:0} = C_{in}$$
$$P_{0:0} = 0$$
$$C_{out,i} = G_{i:0}$$



$C_{0,1} = G_1 + P_1 C_{in}$

$C_{0,2} = G_2 + P_2 C_{0,1}$

$G_{1:0} = G_1 + P_1 G_0$

$G_{2:0} = G_2 + P_2 G_{1:0}$

$$C_o(G,P) = G + PC_i$$
$$S(G,P) = P \oplus C_i$$

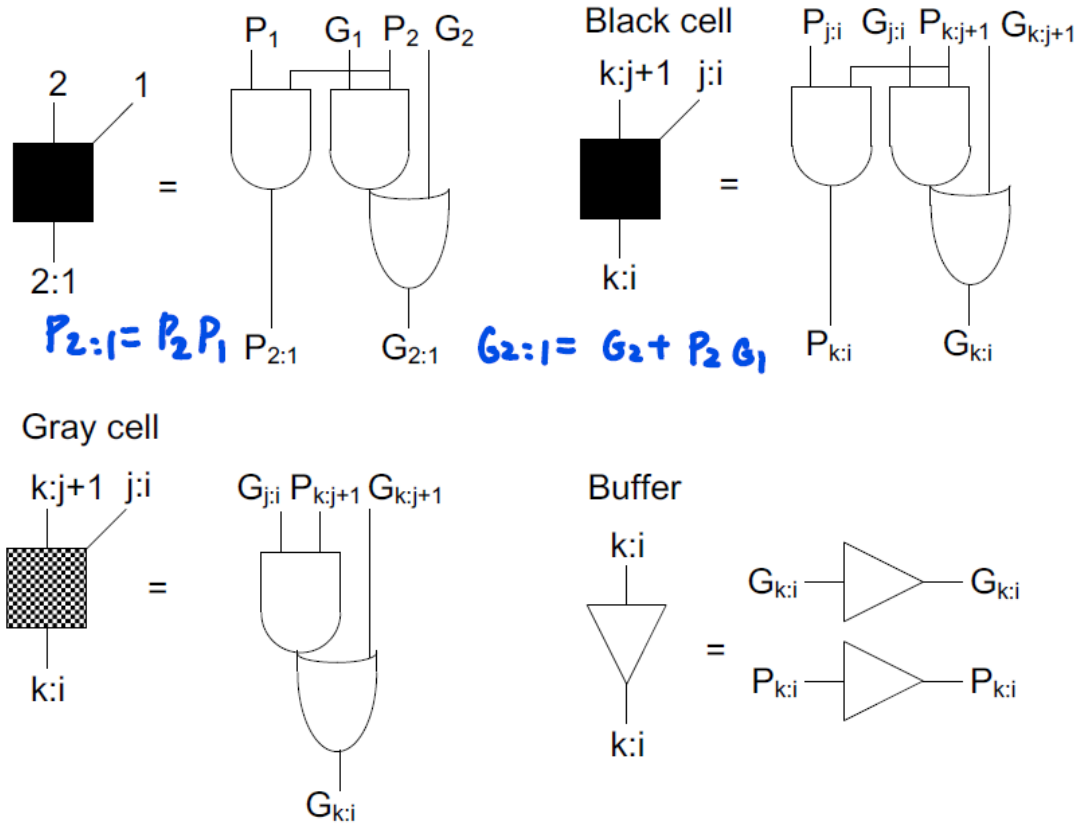
$P = A \oplus B$

$P = A + B$

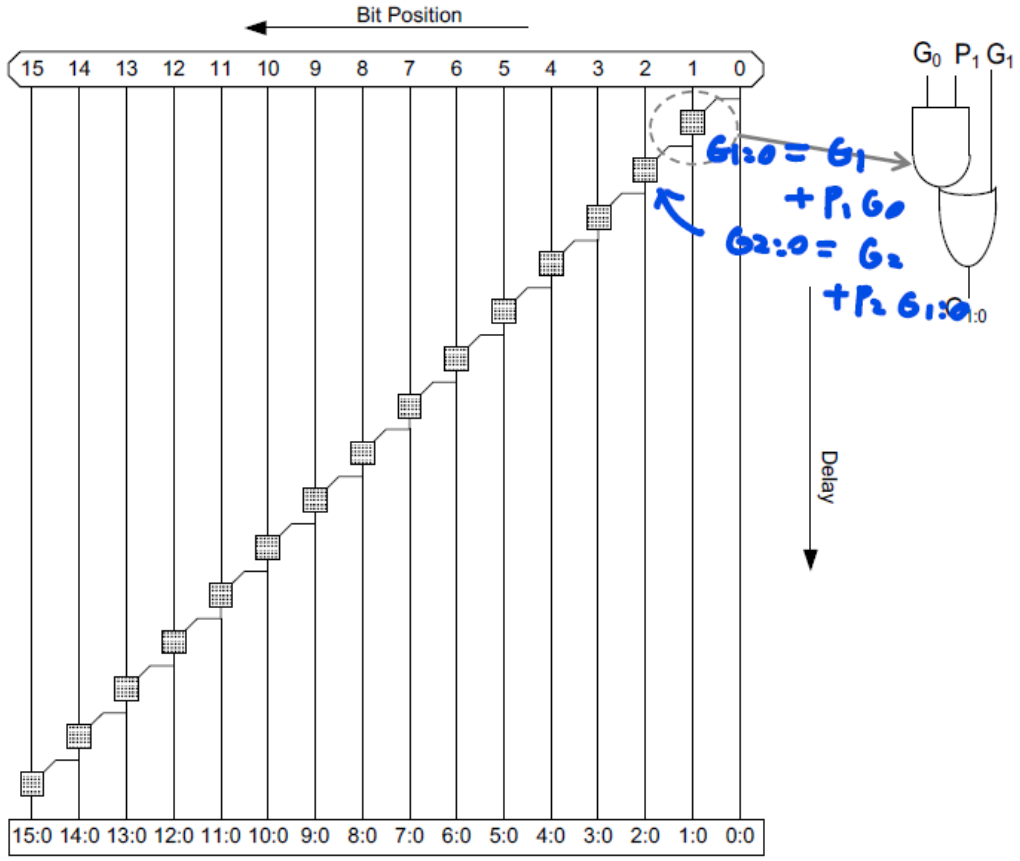
$$t_{adder} = t_{setup} + (N-1)\, t_{carry} + \max(t_{carry}, t_{sum})$$

加法器设计

- 基于PGK的加法器设计方法



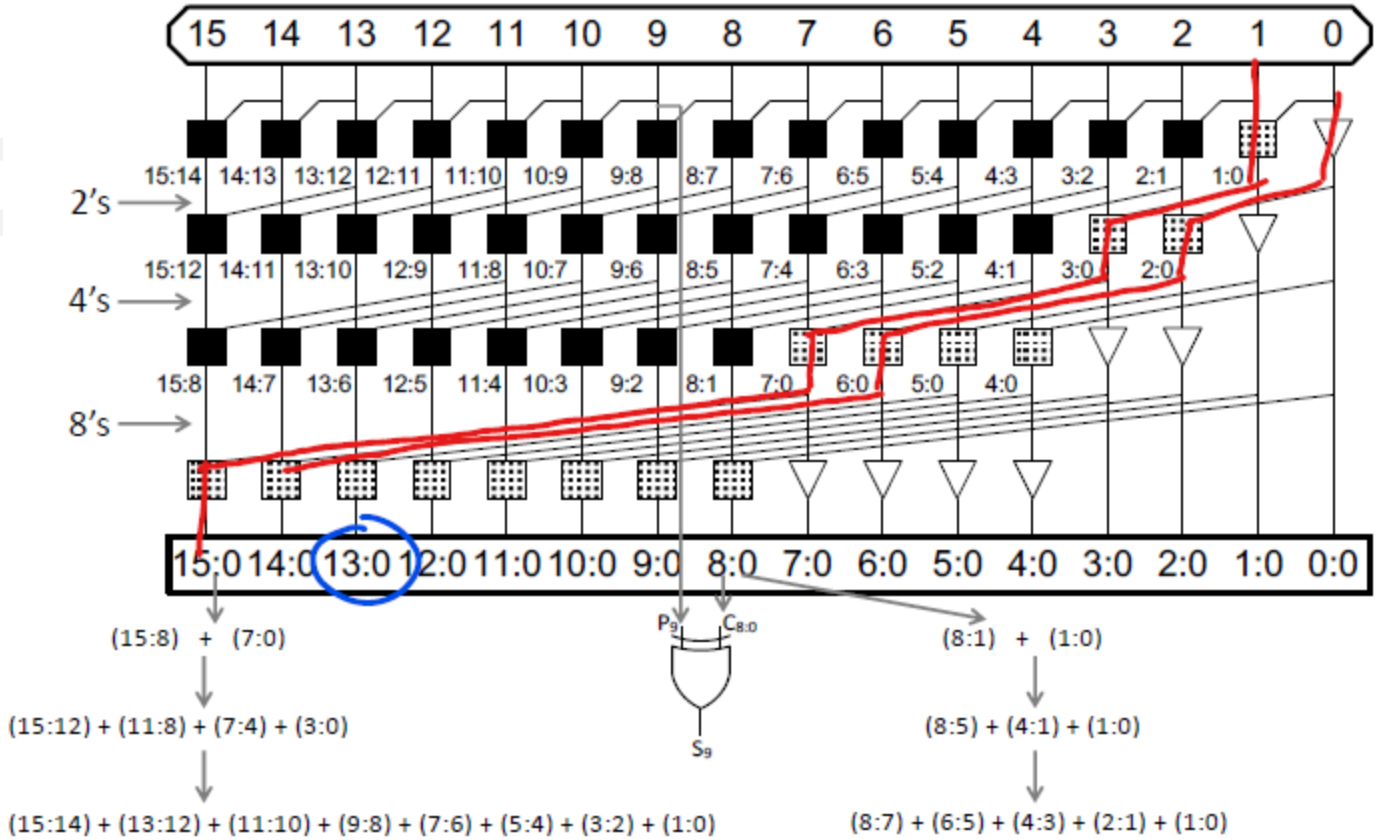
PG生成逻辑



Carry Ripple的PG图

加法器设计

- 基于PGK的加法器设计方法 – 复杂PG树加法器



$$C_o(G,P) = G + PC_i$$

$$S(G,P) = P \oplus C_i$$

$P = A \cdot B$

$P = A + B$

$\log_2(N)$

$G_{15:0} = G_{13:6} + P_{13:6}\, G_{5:0}$

$G_{13:6} = G_{13:10} + P_{13:10}\, G_{9:6}$

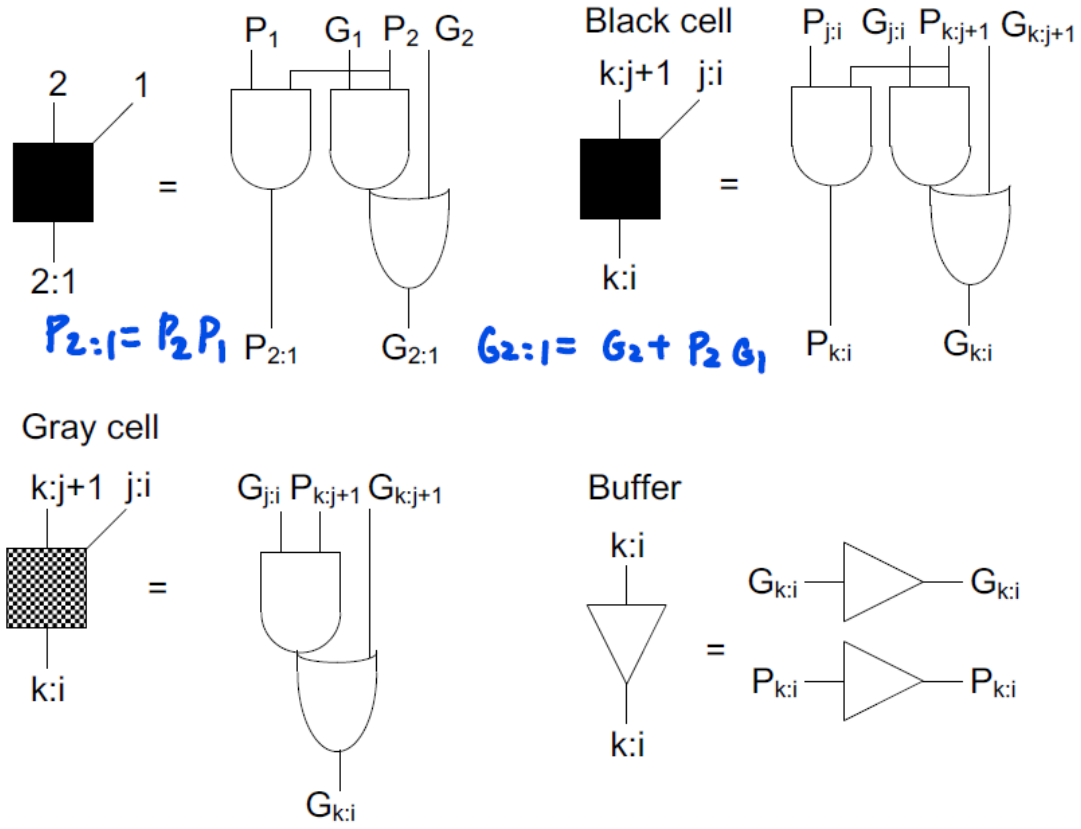
$P_{13:6} = P_{13:10}\, P_{9:6}$

$G_{5:0} = G_{5:2} + P_{5:2}\, G_{1:0}$

作业题

加法器设计

- 基于PGK的加法器设计方法



PG生成逻辑-Radix 2

$$G_{m:i} =$$

$$g(G_{m:n+1}, G_{n:k+1}, G_{k:j+1}, G_{j:i}, P_{m:n+1}, P_{n:k+1}, P_{k:j+1})$$

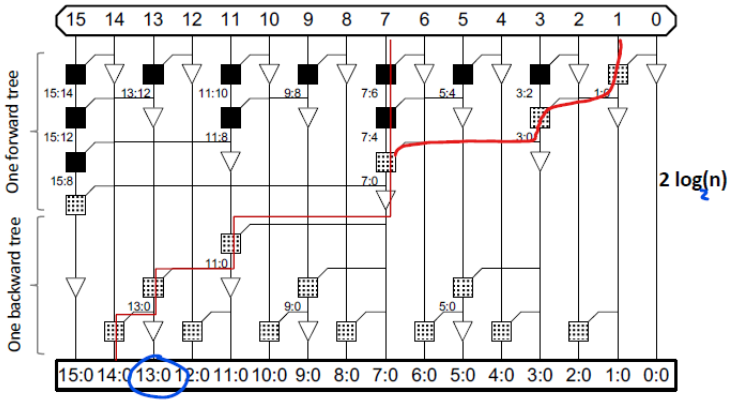
$$P_{m:i} = p(P_{m:n+1}, P_{n:k+1}, P_{k:j+1}, P_{j:i})$$

Radix 4

加法器设计

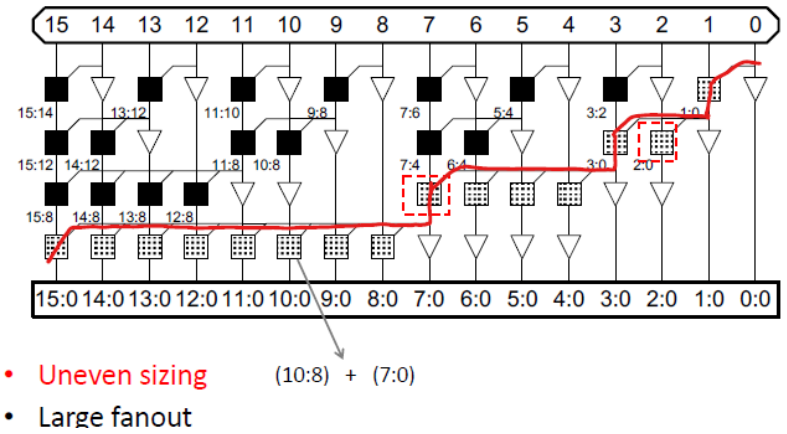
- 基于PGK的加法器设计方法 – 复杂PG树加法器

Brent-Kung



Sklansky

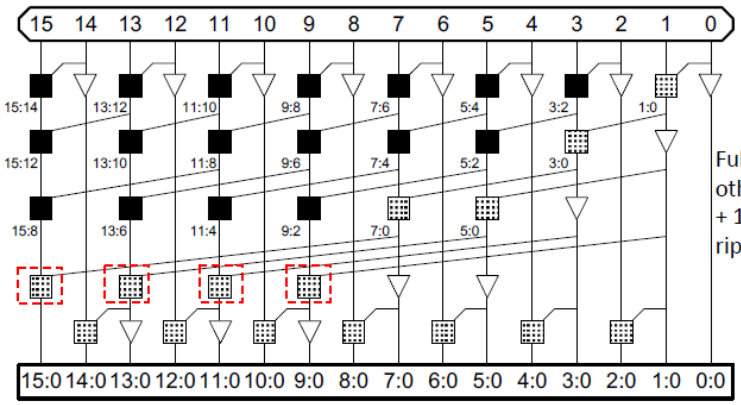
$\log_2(n)$



- Uneven sizing (10:8) + (7:0)
- Large fanout

Han-Carlson

$\log_2(n) + 1$



Low fanout, tradeoff between logic levels and wiring
Reduces wire length by half ! → half power compared to Kogge Stone

- Kogge-Stone: low logic levels, low fanout, high wiring
- Brent-Kung: low fanout, low wiring, high logic levels
- Sklansky: low logic levels, low wiring, high fanout

乘法器设计

- 乘法器设计的核心是部分和累加

Example:

```
    1100  :  12(10)        multiplicand
    0101  :  5(10)         multiplier
    1100
   0000                    partial
  1100                     products
 0000
--------
00111100  :  60(10)        product
```

M x N-bit multiplication

- Produce N M-bit partial products
- Sum these to produce M+N-bit product

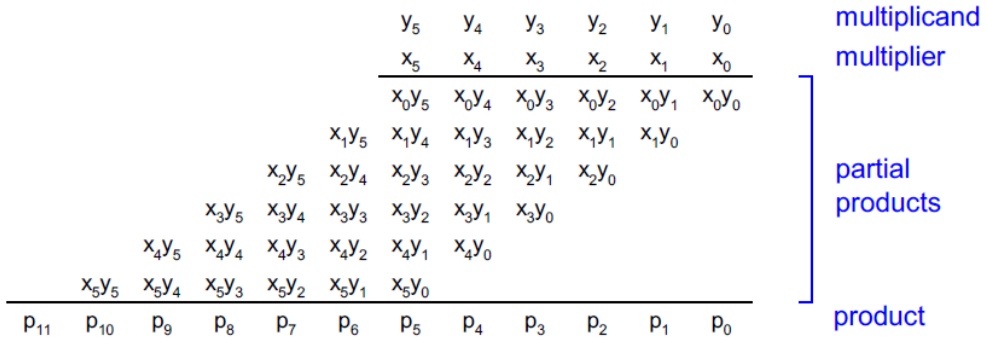
乘法器设计

- 乘法器设计的核心是部分和累加

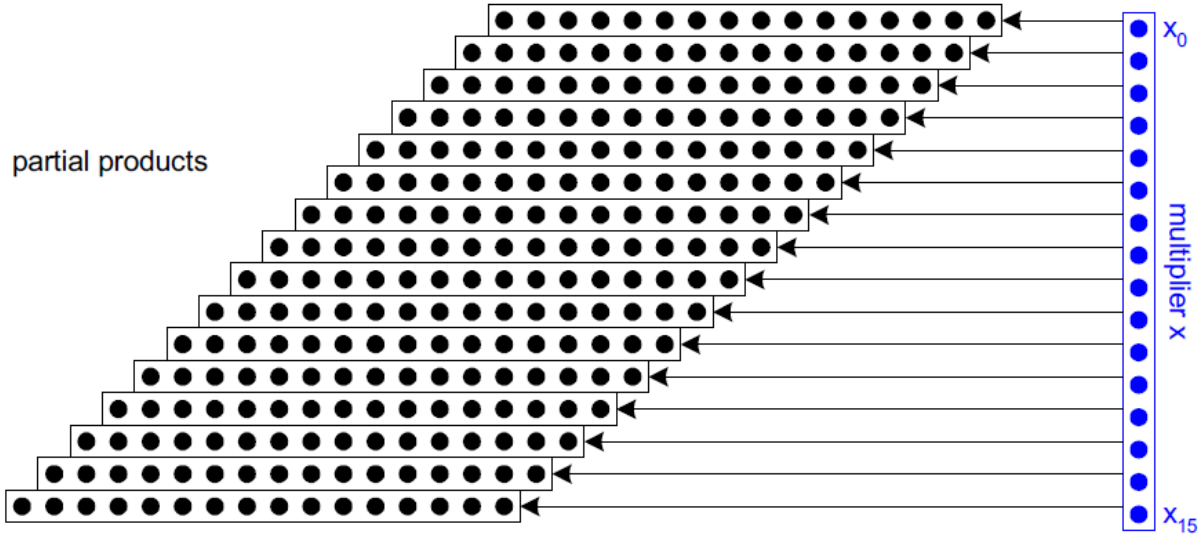
Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$

Multiplier: $X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$

Product: $$P = \left(\sum_{j=0}^{M-1} y_j 2^j\right)\left(\sum_{i=0}^{N-1} x_i 2^i\right) = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} x_i y_j 2^{i+j}$$



Each dot represents a bit



乘法器设计

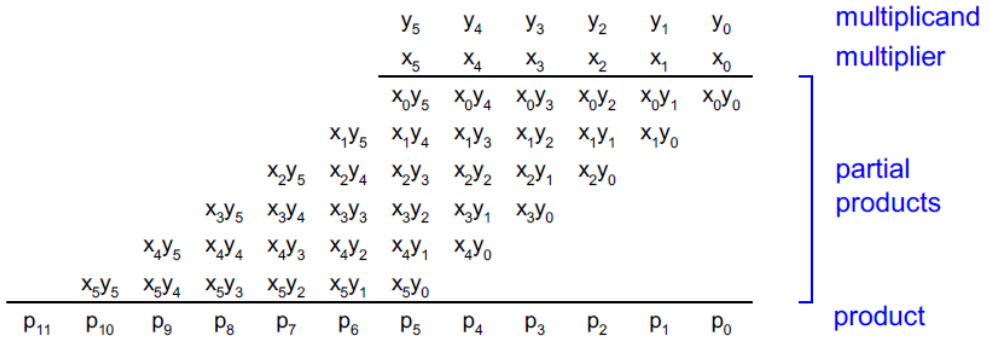
- 乘法器设计的核心是部分和累加

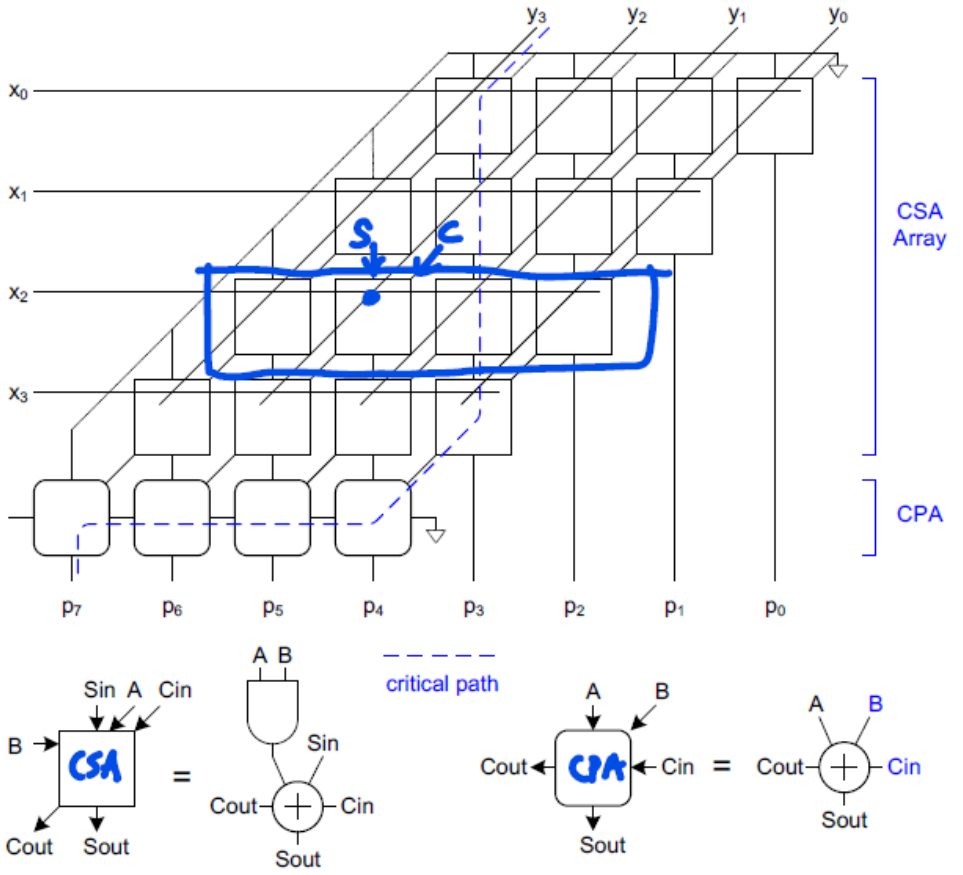
Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$

Multiplier: $X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$

Product:

$$P = \left(\sum_{j=0}^{M-1} y_j 2^j\right)\left(\sum_{i=0}^{N-1} x_i 2^i\right) = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} x_i y_j 2^{i+j}$$

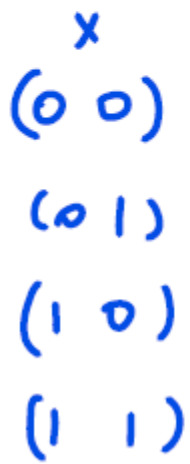


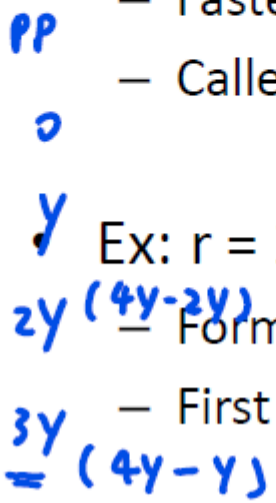


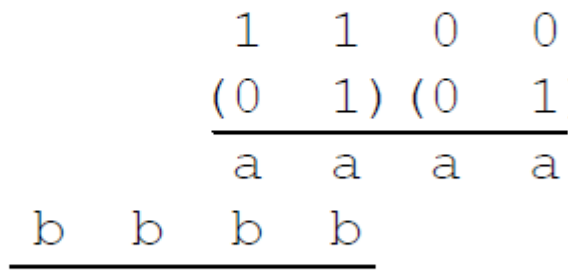
乘法器设计

- 如何减少部分和累加的次数?

- Array multiplier requires N partial products
- If we looked at groups of r bits, we could form N/r partial products.
  - Faster and smaller?
  - Called radix-2^r encoding
- Ex: r = 2: look at pairs of bits
  - Form partial products of 0, Y, 2Y, 3Y
  - First three are easy, but 3Y requires adder ☹







乘法器设计

- 如何减少部分和累加的次数 – 布斯编码（Radix-2^r）

  - Instead of 3Y, try –Y, then increment next partial product to add 4Y
  - Similarly, for 2Y, try –2Y + 4Y in next partial product

Inputs			Partial Product	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	$SINGLE_i$	$DOUBLE_i$	NEG_i
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	$2Y$	0	1	0
1	0	0	$-2Y$	0	1	1
1	0	1	$-Y$	1	0	1
1	1	0	$-Y$	1	0	1
1	1	1	$-0\ (=0)$	0	0	1

4y-2y 1 0

4y-y 1 1

Y -Y Y

作业题

乘法器设计

- 如何减少部分和累加的次数 – 布斯编码（Radix-2^r）

布斯编码的几点要求：

- 乘数、被乘数、结果均为补码
- 乘法计算前应在乘数末尾补零
- 被乘数双符号位
- 符号位参与计算

Inputs			Partial Product	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	$SINGLE_i$	$DOUBLE_i$	NEG_i
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	$2Y$	0	1	0
1	0	0	$-2Y$	0	1	1
1	0	1	$-Y$	1	0	1
1	1	0	$-Y$	1	0	1
1	1	1	$-0\ (=0)$	0	0	1

假设计算 Y x Q = -6 x -7，Q 是乘数，Y 是被乘数（4bit）

1、Y = -6 =1010　Q = -7 = 1001　-Y = 6 = 0110

2、乘数 Q 后补零，Q = 10010

3、被乘数双符号位，Y = 11010，-Y = 00110

3、乘法步骤（A为部分和、Q为乘数）

Step 1：Q = 10010

A = 11111010　Q = 1001　Q-1 = 0　补码符号扩展

Step 2：Q = 10010

A = 00110000　Q =1001　Q-1 = 0　左移补零

结果：11111010 （-6） + 00110000 （48） = 42

乘法器设计

- 如何减少部分和累加的次数 – 布斯编码（Radix-2^r）

布斯编码的几点要求：

- 乘数、被乘数、结果均为补码
- 乘法计算前应在乘数末尾补零
- 被乘数双符号位
- 符号位参与计算

Inputs			Partial Product	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	$SINGLE_i$	$DOUBLE_i$	NEG_i
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	$2Y$	0	1	0
1	0	0	$-2Y$	0	1	1
1	0	1	$-Y$	1	0	1
1	1	0	$-Y$	1	0	1
1	1	1	$-0\ (=0)$	0	0	1

假设计算 Y x Q = -6 x 7，Q 是乘数，Y 是被乘数（6bit）

1、Y = -6 =111010　Q = 7 = 000111　-Y = 6 = 000110

2、乘数 Q 后补零，Q = 0001110

3、被乘数双符号位，Y = 1111010，-Y = 0000110

3、乘法步骤（A为部分和、Q为乘数）

Step 1：Q = 0001110

A = 000000000110　Q = 000111　Q-1 = 0　补码符号扩展

Step 2：Q = 0001110

A = 111111010000　Q =000111　Q-1 = 0　左移/符号位扩展

Step3：Q = 0001110

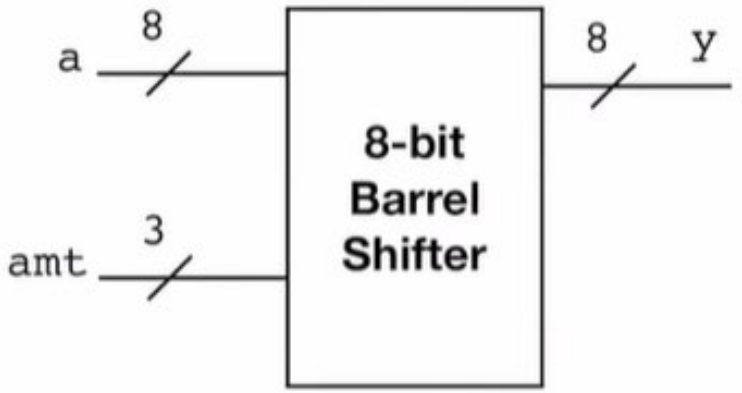
结果 = 000000000110（6）+ 111111010000（-48）= -42

位移器设计

- Shifter也是重要的数字电路模块之一

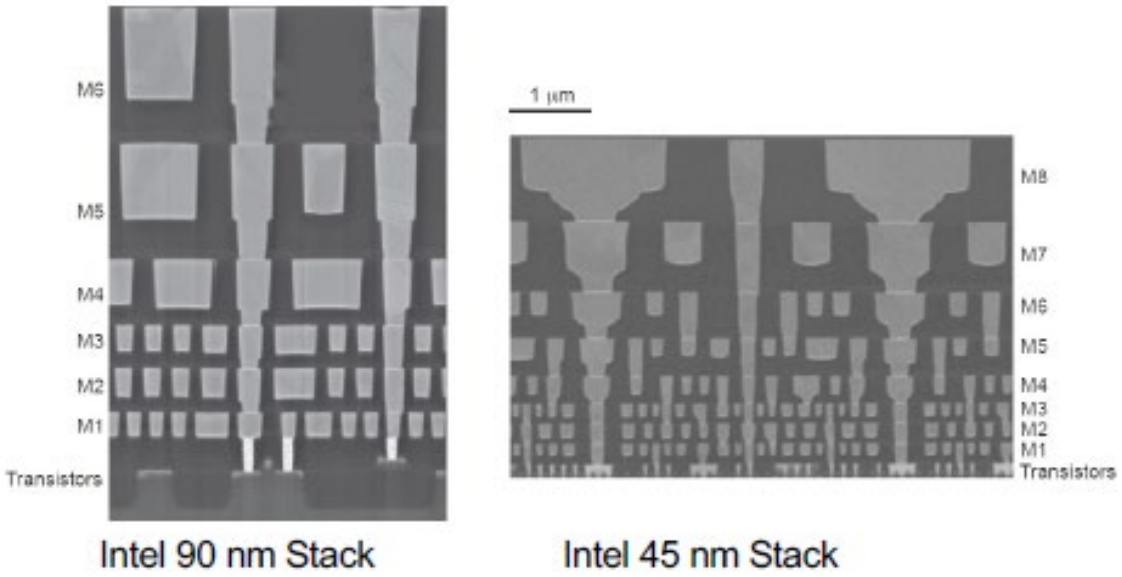
```
module barrel_shifter
    (
        input logic [7:0] a,
        input logic [2:0] amt,
        output logic [7:0] y
    );

    always_comb
        case(amt)
            3'b000: y = a;
            3'b001: y = {a[0], a[7:1]};
            3'b010: y = {a[1:0], a[7:2]};
            3'b011: y = {a[2:0], a[7:3]};
            3'b100: y = {a[3:0], a[7:4]};
            3'b101: y = {a[4:0], a[7:5]};
            3'b110: y = {a[5:0], a[7:6]};
            3'b111: y = {a[6:0], a[7]};
            default: y = a;
        endcase
endmodule
```



线路分析

- **Wire Geometry**



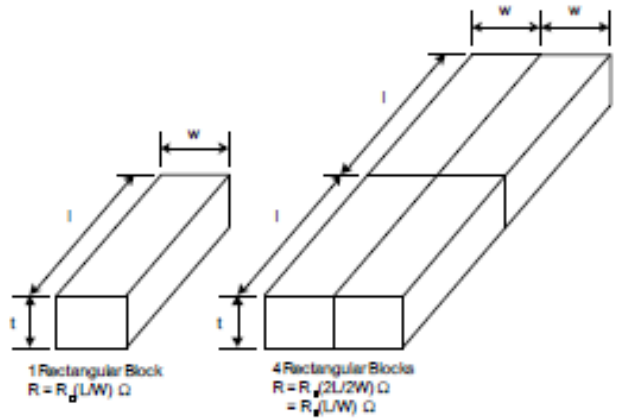
Intel 90 nm Stack　　Intel 45 nm Stack

线电阻的计算方式

- ρ = *resistivity* (Ω*m)

$$R = \frac{\rho}{t}\frac{l}{w} = R_{\square}\frac{l}{w}$$

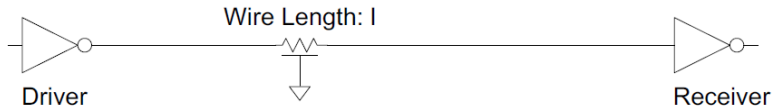
- $R_{\square}$ = *sheet resistance* (Ω/□)
  - □ is a dimensionless unit(!)
- Count number of squares
  - $R = R_{\square} * (\#\ of\ squares)$

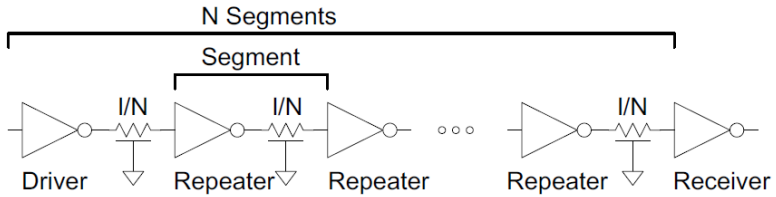


北京大学 PEKING UNIVERSITY

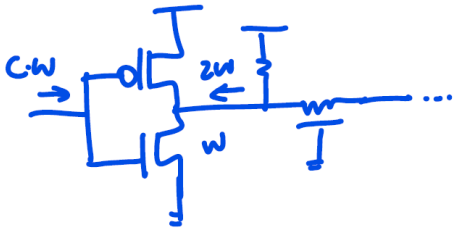
• Wire Repeaters

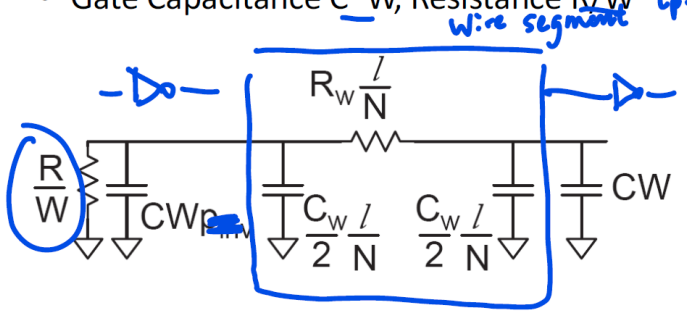
- R and C are proportional to l
- RC delay is proportional to l^2
  - Unacceptably great for long wires
- Break long wires into N shorter segments
  - Drive each one with an inverter or buffer





- How many repeaters should we use?
- How large should each one be?
- Equivalent Circuit
  - Wire length l/N
    - Wire Capacitance $C_w * l/N$, Resistance $R_w * l/N$
  - Inverter width W (nMOS = W, pMOS = 2W)
    - Gate Capacitance C*W, Resistance R/W





$$t_p = N\left(0.69\left(\frac{R}{W}CW + \frac{R}{W}\frac{CW}{2}\frac{l}{N} + \left(\frac{R}{W} + R_w\frac{l}{N}\right)\left(\frac{CW}{2}\frac{l}{N} + CW\right)\right)\right)$$

北京大学 PEKING UNIVERSITY

- Wire Repeaters

- Write equation for Elmore Delay
  - Differentiate with respect to W and N
  - Set equal to 0, solve

unit wire segment

unit inv resistance

unit inv cap

unit wire cap

unit wire res

$$\frac{l}{N} = \sqrt{\frac{2RC'}{R_w C_w}}$$

$$C' = C(1 + p_{\text{inv}})$$

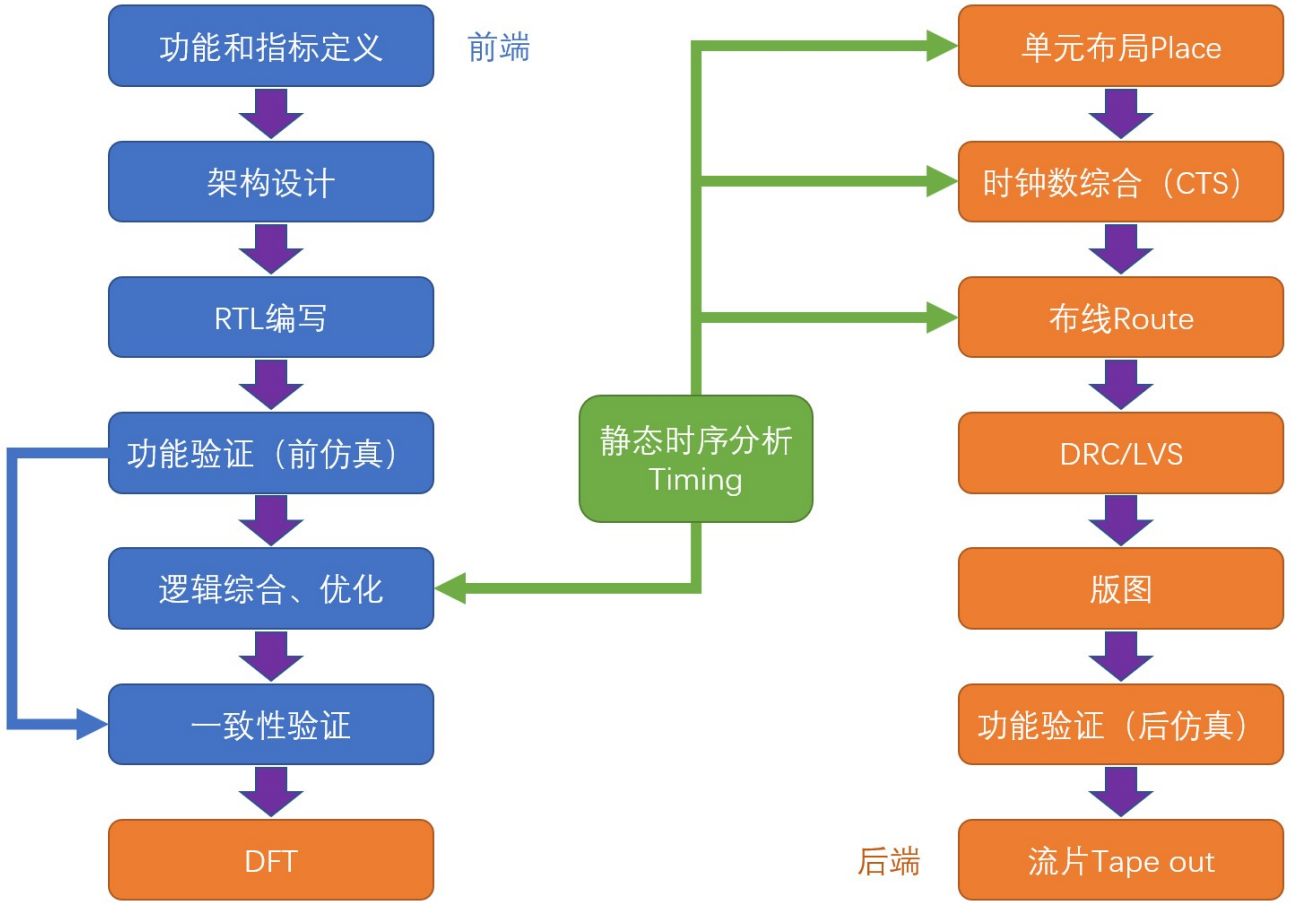
$$W = \sqrt{\frac{RC_w}{R_w C'}}$$

目 录

CONTENTS

芯片设计流程

- 美国5大芯片公司 – 英特尔、英伟达、AMD、高通、博通



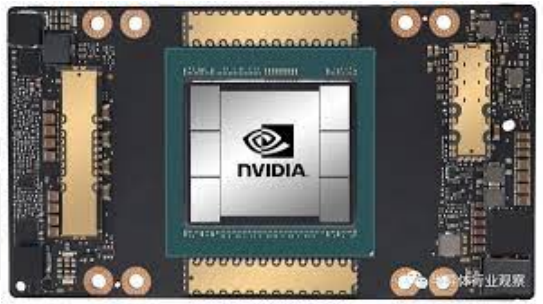
大型数字芯片设计基本流程

PC处理器CPU芯片

Qualcomm

手机SOC芯片

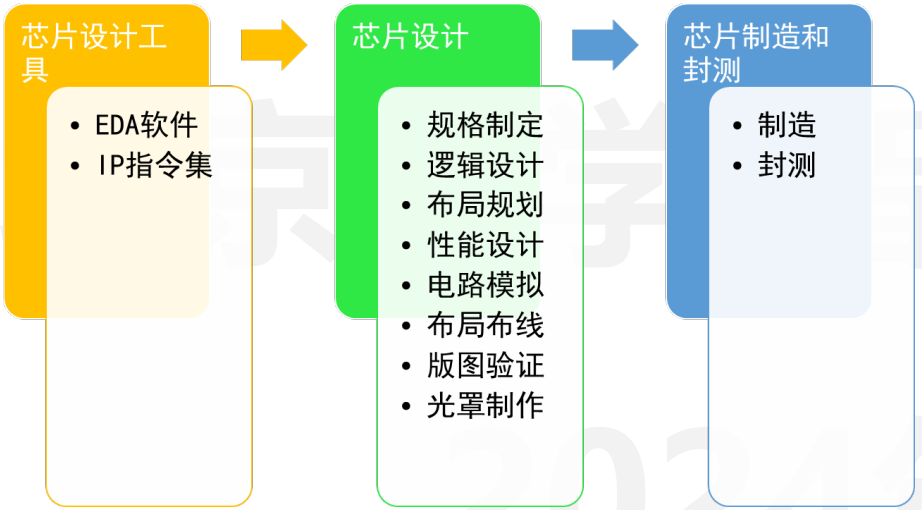
NVIDIA



图形处理芯片GPU

芯片设计流程

- 国际分工合作的庞大产业链生态



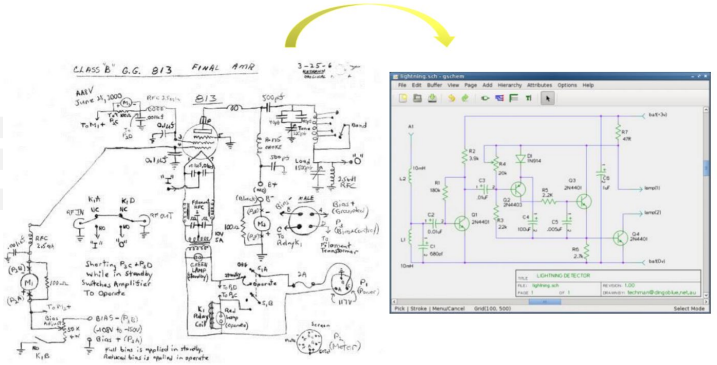
芯片设计的EDA软件工具目前由美国公司所垄断

芯片设计工具	芯片设计	芯片制造和封测
• 美国英特尔	• 美国苹果	• 美国英特尔
• 英国ARM	• 韩国三星	• 韩国三星
• 英国Imagination	• 英国Imagination	• 美国高通
• 美国IBM	• 海思	• 美国英伟达
• 美国Mips	• 紫光展锐（展讯+锐迪科）	• 台积电
• Synopsys	• 中兴微电子	• 中芯国际
• 美国Cadence	• 华大半导体	• 华虹半导体
• 美国Mentor Graphics	• 汇顶科技	• 长江存储
• 日本图研	• 兆易创新	• 合肥长鑫
• 华大九天	• 寒武纪	• 福建晋华
• 杭州中天微		• 紫光国微
• 苏州国芯		• 长电科技
• 寒武纪		• 华天科技
		• 通富微电

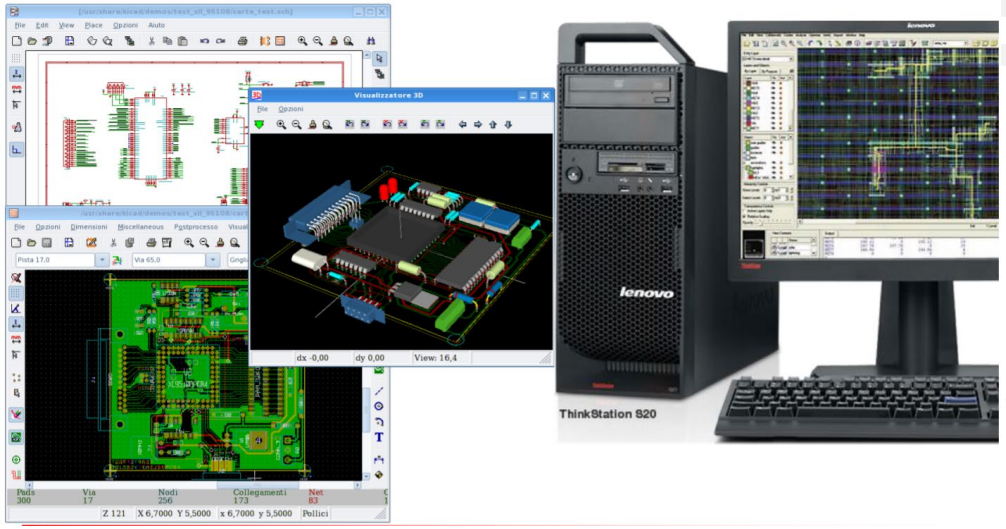
芯片设计流程

- 设计自动化软件 (EDA) 是提升芯片设计效率的关键因素，目前也由**美国公司占主导地位**

电路设计软件与仿真工具



• Electronic Design Automation



中国芯片设计产业现状 – 华为海思、紫光集团等

- 华为海思半导体、紫光集团是中国大陆最大的芯片设计公司

HUAWEI

HISILICON

SGIT 国网信通产业集团
STATE GRID INFO & TELECOM GROUP
北京智芯微电子科技有限公司
BEIJING SMARTCHIP MICROELECTRONICS TECHNOLOGY CO., LTD.

电网相关芯片

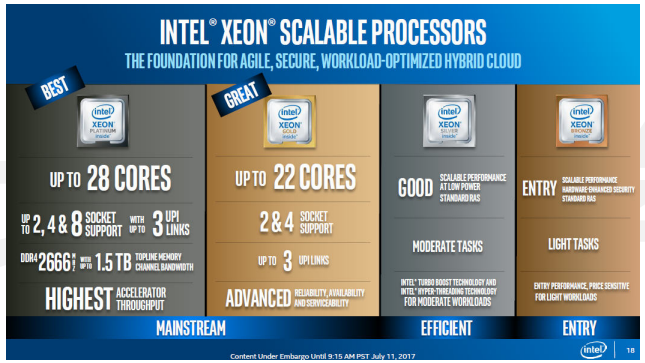
紫光集团
TSINGHUA UNIGROUP

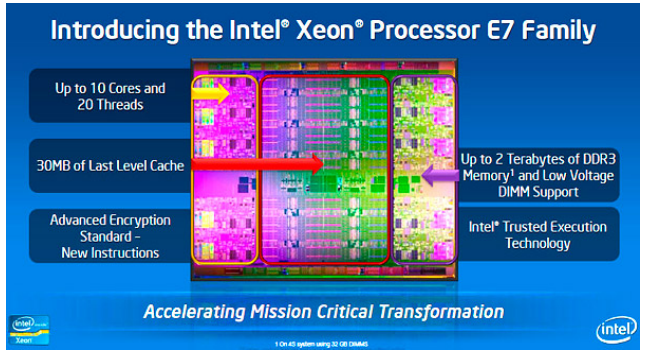
SANECHIPS 中兴微电子

通信相关芯片

中国的“卡脖子”领域：高性能处理器芯片

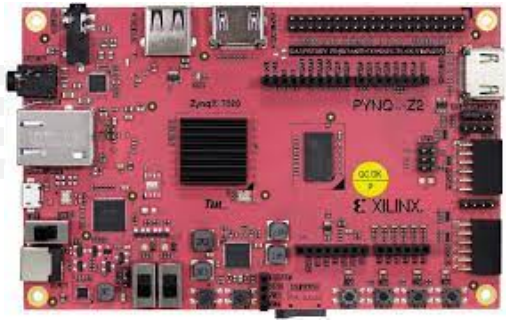
- 我国在高性能计算芯片CPU、GPU、FPGA的指令集与架构设计领域目前落后较多





高性能CPU遭美国出口管制禁运

高性能GPU遭美国出口管制禁运



高性能可编程逻辑FPGA与美国主流厂商Altera、Xilinx差距明显

国产龙芯3C5000目前已可商用，但性能与至强系列仍有显著差异

国产GPU尚处于初级阶段

国产GPU包括摩尔线程、壁仞科技、燧原科技、天数智芯、景嘉微等，与英伟达差距很大

国产FPGA包括紫光同创、安路科技、复旦微等，在并行规模、功能灵活性上急需进步

中国的“卡脖子”领域：EDA软件产业

- 我国在高性能的电路辅助设计与仿真工业软件方面目前与发达国家差距明显

现状

2019年

EDA三巨头Cadence、Synopsys、Mentor等已经对华为断供EDA工具，不再出售新license

2022年

2022年8月美国将停止对中国出口GAA相关EDA软件，GAA主要用于3nm及以下晶体管

国产EDA软件目前门类已补齐，但制程支持、设计仿真性能、与晶圆厂对接等多方面仍处于落后状态

什么是Verilog语言

北京大学 PEKING UNIVERSITY

- 为什么需要一种硬件描述语言

What is HDL, Verilog?

What is Verilog?

- Hardware Description Language - IEEE 1364-2005
  - Superseded by SystemVerilog - IEEE 1800-2009
- Two Forms
  1. Behavioral
  2. Structural
- It can be built into hardware. If you can't think of at least one (inefficient) way to build it, it might not be good.

Why do I care?

- We use Behavioral Verilog to do computer architecture here.
- Semiconductor Industry Standard (VHDL is also common, more so in Europe)

2

北京大学 PEKING UNIVERSITY

- 行为级Verilog与模块级Verilog

Behavioral vs. Sturctural

Behavioral Verilog

- Describes function of design
- Abstractions
  - Arithmetic operations (+, -, *, /)
  - Logical operations (&, |, ^, ~)

Structural Verilog

- Describes construction of design
- No abstraction
- Uses modules, corresponding to physical devices, for everything

Suppose we want to build an adder?

Verilog语言简介

- 用Verilog语言构建一个1bit的加法器

Structural Verilog

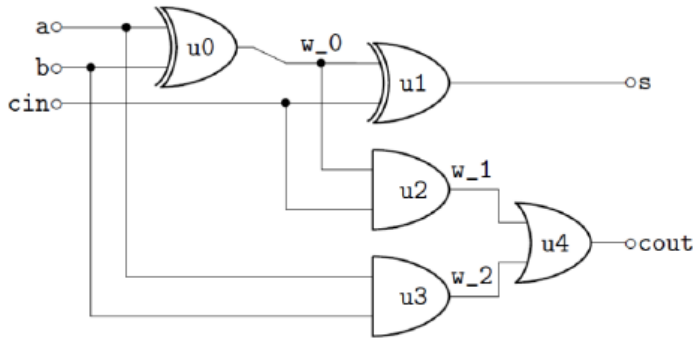


Figure: 1-bit Full Adder

```
module one_bit_adder(
    input wire a,b,cin,
    output wire sum,cout);
    wire w_0,w_1,w_2;
    xor u0(w_0,a,b);
    xor u1(sum,w_0,cin);
    and u2(w_1,w_0,cin);
    and u3(w_2,a,b);
    or u4(cout,w_1,w_2);
endmodule
```

Behavioral Verilog

```
module one_bit_adder(
    input wire a,b,cin,
    output wire sum,cout);
    assign sum = a ^ b ^ cin;
    assign cout = ((a ^ b) & cin) | a & b;
endmodule
```

OR ...

```
module one_bit_adder(
    input logic a,b,cin,
    output logic sum,cout);

    always_comb
    begin
        sum = a ^ b ^ cin;
        cout = 1'b0;
        if ((a ^ b) & cin) | (a & b))
            cout = 1'b1;
    end
endmodule
```

北京大学 PEKING UNIVERSITY

- 数据结构与可综合性

Data Types, Values

Synthesizable Data Types

`wires` Also called nets

```
wire a_wire;
wire [3:0] another_4bit_wire;
```

- Cannot hold state

`logic` Replaced `reg` in SystemVerilog

```
logic [7:0] an_8bit_register;
reg a_register;
```

- Holds state, might turn into flip-flops
- Less confusing than using `reg` with combinational logic (coming up...)

Unsynthesizable Data Types

`integer` Signed 32-bit variable

`time` Unsigned 64-bit variable

`real` Double-precision floating point variable

Four State Logic

`0` False, low

`1` True, high

`Z` High-impedance, unconnected net

`X` Unknown, invalid, don't care

7

北京大学 PEKING UNIVERSITY

- 运算符与赋值规则

Operators

Arithmetic	
*	Multiplication
/	Division
+	Addition
-	Subtraction
%	Modulus
**	Exponentiation
Bitwise	
~	Complement
&	And
|	Or
~|	Nor
^	Xor
~^	Xnor
Logical	
!	Complement
&&	And
||	Or

Shift	
>>	Logical right shift
<<	Logical left shift
>>>	Arithmetic right shift
<<<	Arithmetic left shift
Relational	
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
!=	Inequality
!==	4-state inequality
==	Equality
===	4-state equality
Special	
{,}	Concatenation
{n{m}}	Replication
?:	Ternary

Setting Values

assign Statements

- One line descriptions of combinational logic
- Left hand side must be a wire (SystemVerilog allows assign statements on logic type)
- Right hand side can be any one line verilog expression
- Including (possibly nested) ternary (?:)

Example

```
module one_bit_adder(
    input wire a,b,cin,
    output wire sum,cout);
    assign sum = a ^ b ^ cin;
    assign cout = ((a ^ b) & cin) | a & b;
endmodule
```

- 运算符与赋值规则

Setting Values

always Blocks

- Contents of `always` blocks are executed whenever anything in the sensitivity list happens
- Two main types in this class...
  - `always_comb`
    - implied sensitivity lists of every signal inside the block
    - Used for combinational logic. Replaced `always @*`
  - `always_ff @(posedge clk)`
    - sensitivity list containing only the positive transition of the `clk` signal
    - Used for sequential logic
- All left hand side signals need to be `logic` type.

Examples

Combinational Block

```
always_comb
begin
    x = a + b;
    y = x + 8'h5;
end
```

Sequential Block

```
always_ff @(posedge clk)
begin
    x <= #1 next_x;
    y <= #1 next_y;
end
```

北京大学 PEKING UNIVERSITY

- 运算符与赋值规则

Blocking vs. Non-blocking statement

Blocking Assignment

- Combinational Blocks
- Each assignment is processed in order, earlier assignments block later ones
- Uses the = operator

vs.

Nonblocking Assignment

- Sequential Blocks
- All assignments occur "simultaneously," delays are necessary for accurate simulation
- Uses the <= operator

Blocking Example

```
always_comb
begin
    x = new_val1;
    y = new_val2;
    sum = x + y;
end
```

- Behave exactly as expected
- Standard combinational logic

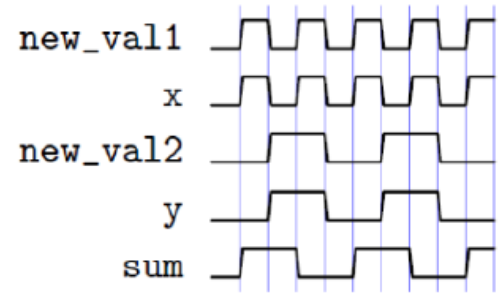


Figure: Timing diagram for the above example.

北京大学 PEKING UNIVERSITY

- 运算符与赋值规则

Nonblocking Example

```
always_ff @(posedge clock)
begin
    x <= #1 new_val1;
    y <= #1 new_val2;
    sum <= #1 x + y;
end
```

- What changes between these two examples?
- Nonblocking means that sum lags a cycle behind the other two signals

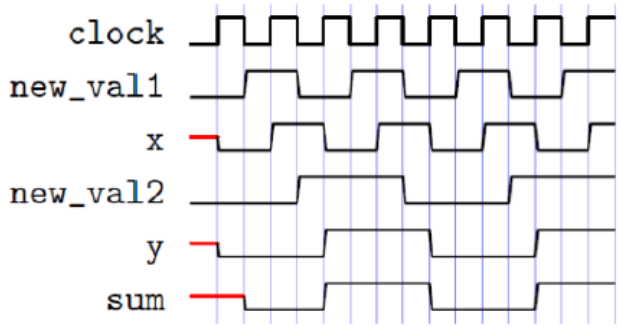


Figure: Timing diagram for the above example.

Bad Example

```
always_ff @(posedge clock)
begin
    x <= #1 y;
    z = x;
end
```

- z is updated after x
- z updates on negedge clock

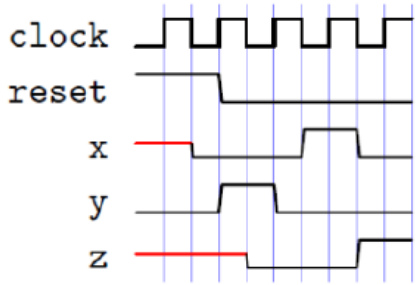


Figure: Timing diagram for the above example.

北京大学 PEKING UNIVERSITY

- 避免Latch的出现

Latches

- What is a latch?
  - Memory device without a clock
- Generated by a synthesis tool when a net needs to hold state without being clocked (combinational logic)
- Generally bad, unless designed in intentionally
- Unnecessary in this class

Latches

- Always assign every variable on every path
- This code generates a latch
- Why does this happen?

```
always_comb
begin
    if (cond)
        next_x = y;
end
```

To avoid unintentional latches

Possible Solutions to Latches

```
always_comb
begin
    next_x = x;
    if (cond)
        next_x = y;
end
```

```
always_comb
begin
    if (cond)
        next_x = y;
    else
        next_x = x;
end
```

北京大学 PEKING UNIVERSITY

- **Module的概念**

Modules

Intro to Modules

- Basic organizational unit in Verilog
- Can be reused

Module Example

```
module my_simple_mux(
    input wire select_in, a_in, b_in; //inputs listed
    output wire muxed_out); //outputs listed
    assign muxed_out = select_in ? b_in : a_in;
endmodule
```

Writing Modules

- Inputs and outputs must be listed, including size and type
  format: `<dir> <type> <[WIDTH-1:0]> <name>;`
  e.g. `output logic [31:0] addr;`
- In module declaration line or after it, inside the module

Instantiating Modules

- Two methods of instantiation
  1. e.g. `my_simple_mux m1(.a_in(a),.b_in(b),.select_in(s),.muxed_out(m));`
  2. e.g. `my_simple_mux m1(a,b,s,m);`
- The former is much safer...
- Introspection (in testbenches): `module.submodule.signal`

北京大学 PEKING UNIVERSITY

- 时序逻辑与组合逻辑的可综合性

Keys to Synthesizability

- Remember – Behavioral Verilog implies no specific hardware design
- But, it has to be synthesizable
- Better be able to build it somehow

Combinational Logic

- Avoid feedback (combinatorial loops)
- Always blocks should
  - Be `always_comb` blocks
  - Use the blocking assignment operator =
- All variables assigned on all paths
  - Default values
  - `if(...)` paired with an `else`

Sequential Logic

Sequential Logic

- Avoid clock- and reset-gating
- Always blocks should
  - Be `always_ff @(posedge clock)` blocks
  - Use the nonblocking assignment operator, with a delay `<= #1`
- No path should set a variable more than once
- Reset all variables used in the block
- `//synopsys sync_set_reset "reset"`

- 控制与流程

Flow Control

All Flow Control

- Can only be used inside procedural blocks (`always`, `initial`, `task`, `function`)
- Encapsulate multiline assignments with `begin...end`
- Remember to assign on all paths

Synthesizable Flow Control

- `if/else`
- `case`

Unsynthesizable Flow Control

- Useful in testbenches
- For example...
  - `for`
  - `while`
  - `repeat`
  - `forever`

Synthesizable Flow Control Example

```
always_comb
begin
    if (muxy == 1'b0)
        y = a;
    else
        y = b;
end
```

The Ternary Alternative

```
wire y;
assign y = muxy ? b : a;
```

Casez Example

```
always_comb
begin
    casez(alu_op)
        3'b000: r = a + b;
        3'b001: r = a - b;
        3'b010: r = a * b;
        ...
        3'b1??: r = a ^ b;
    endcase
end
```

- 测试与验证

Testing

What is a test bench?

- Provides inputs to one or more modules
- Checks that corresponding output makes sense
- Basic building block of Verilog testing

Why do I care?

- Finding bugs in a single module is hard...
- But not as hard as finding bugs after combining many modules
- Better test benches tend to result in higher project scores

Intro to Testbench

Features of the Test Bench

- Unsynthesized
  - Remember unsynthesizable constructs? This is where they're used.
  - In particular, unsynthesizable flow control is useful in testbenches (e.g. `for`, `while`)
- Programmatic
  - Many programmatic, rather than hardware design, features are available e.g. functions, tasks, classes (in SystemVerilog)

A good test bench should, in order...

1. Declare inputs and outputs for the module(s) being tested
2. Instantiate the module (possibly under the name `DUT` for Device Under Test)
3. Setup a clock driver (if necessary)
4. Setup a correctness checking function (if necessary/possible)
5. Inside an `initial` block...
   - 5.1 Assign default values to all inputs, including asserting any available `reset` signal
   - 5.2 `$monitor` or `$display` important signals
   - 5.3 Describe changes in input, using good testing practice

北京大学 PEKING UNIVERSITY

- Testbench的initial block

initial Block Example

```
initial
begin
    @(negedge clock);
    reset = 1'b1;
    in0 = 1'b0;
    in1 = 1'b1;
    @(negedge clock);
    reset = 1'b0;
    @(negedge clock);
    in0 = 1'b1;
    ...
end
```

task Example

```
task exit_on_error;
    input [63:0] A, B, SUM;
    input C_IN, C_OUT;
    begin
$display("@@@ Incorrect at time %4.0f", $time);
$display("@@@ Time:%4.0f clock:%b A:%h B:%h CIN:%b SUM:%h"
    "COUT:%b", $time, clock, A, B, C_IN, SUM, C_OUT);
$display("@@@ expected sum=%b", (A+B+C_IN) );
$finish;
    end
endtask
```

Testbench常用组成模块

task

- Reuse commonly repeated code
- Can have delays (e.g. #5)
- Can have timing information (e.g. @(negedge clock))
- Might be synthesizable (difficult, not recommended)

function

- Reuse commonly repeated code
- No delays, no timing
- Can return values, unlike a task
- Basically combinational logic

function Example

```
function check_addition;
   input wire [31:0] a, b;
   begin
       check_addition = a + b;
   end
endfunction

assign c = check_addition(a,b);
```

- Testbench的示例

System tasks and functions

`$monitor`	Used in test benches. Prints every time an argument changes. Very bad for large projects. e.g. `$monitor("format",signal,...)`
`$display`	Can be used in either test benches or design, but not after synthesis. Prints once. Not the best debugging technique for significant projects. e.g. `$display("format",signal,...)`
`$strobe`	Like display, but prints at the end of the current simulation time unit. e.g. `$strobe("format",signal,...)`
`$time`	The current simulation time as a 64 bit integer.
`$reset`	Resets the simulation to the beginning.
`$finish`	Exit the simulator, return to terminal.

More available at ASIC World.

Test Bench Setup

```
module testbench;
    logic clock, reset, taken, transition, prediction;

    two_bit_predictor(
        .clock(clock),
        .reset(reset),
        .taken(taken),
        .transition(transition),
        .prediction(prediction));

    always
    begin
        clock = #5 ~clock;
    end
```

- Testbench内的testcase

Test Bench Test Cases

```
initial
begin
    $monitor("Time:%4.0f clock:%b reset:%b taken:%b trans:%b"
        "pred:%b", $time, clock, reset, taken,
        transition, prediction);
    clock = 1'b1;
    reset = 1'b1;
    taken = 1'b1;
    transition = 1'b1;
    @(negedge clock);
    @(negedge clock);
    reset = 1'b0;
    @(negedge clock);
    taken = 1'b1;
    @(negedge clock);
    ...
    $finish;
end
```

Remember to...

- Initialize all module inputs
- Then assert reset
- Use `@(negedge clock)` when changing inputs to avoid race conditions

Verilog语言简介

北京大学 PEKING UNIVERSITY

- Verilog+Testbench的简单实例

```
module my_fir (
        input                   clk ,
        input                   reset_n,
        input  signed [7:0]     fir_in,
        input                   fir_val,
        output reg                      fir_out_val,
        output reg signed [6:0]         fir_out    ) ;


//============================
// Internal signals (wires and FFs)
//============================
localparam              h0              = -7;
localparam              h1              = 13;
...
reg             signed [7:0]        in_d0;
reg             signed [7:0]        in_d1;
...
reg             signed [7:0]        in_d4;
Reg                                 in_val_reg;
wire            signed [11:0]       mult_out0;
wire            signed [11:0]       mult_out1;
...
wire            signed [14:0]       sum_out;
...
assign          mult_out0 = in_d4*h0;       // combinational logic
assign          mult_out1 = in_d3*h1;
...
assign          sum_out = {{3, mult_out0[11]}, mult_out0} + {{3, mult_out1[11]}, mult_out1} + {{3, mult_out2[11]}, mult_out2} + {{3,
mult_out3[11]}, mult_out3} + {{3, mult_out4[11]}, mult_out4};       // sign extension and then add.
```

```
always @(posedge clk or negedge reset_n)
        begin
        if (~reset_n)
                begin
                in_d0       <= 0;
                in_d1       <= 0;
                ...
                in_val_reg          <= 1'b0;
                fir_out_val         <= 1'b0;
                fir_out             <= 0;
                end
        else
                begin
                in_val_reg          <= #1 fir_val;
                in_d0               <= #1 fir_in;
                in_d0               <= #1 in_d0;
                ...
                fir_out_val         <= #1 in_val_reg;
                fir_out             <= #1 sum_out[9:3];
                end
        end

endmodule
```

Verilog语言简介

北京大学 PEKING UNIVERSITY

- Verilog+Testbench的简单实例

```
`timescale 1 ns / 1 ps

module my_fir_tb;
parameter     CLOCK_PERIOD        = 125;       // 8MHz
reg                   clk;
reg                   reset_n;
reg  signed [7:0]       fir_in;
reg                     fir_val;
reg                     fir_out_val;
reg signed [6:0]        fir_out;
integer               finput, in_read_cnt;

my_fir UUT(
            .clk          (clk          ),
            .reset_n                (reset_n    ),
            .fir_in                 (fir_in       ),
            .fir_val      (fir_val      ),
            .fir_out      (fir_out      ),
            .fir_out_val            (fir_out_val    )
);
initial
  begin
            finput          = $fopen("input.txt", "r");
            clk             = 1'b0;
            reset_n         = 1'b1;
            in_read_cnt     = -1;
            #(CLOCK_PERIOD)
            reset_n         = 1'b0;
            #(CLOCK_PERIOD)
            reset_n         = 1'b1;
            ...
  end
always    #(CLOCK_PERIOD/2.0)   clk    = ~clk;
```

```
always @ (negedge clk)
  begin
  #1
    begin
    if (in_read_cnt < 1)
      fir_val       = 1'b0;
    else
      begin
      in_read_cnt       = $fscanf(finput,"%d %d\n", fir_in, fir_val);

      if (in_read_cnt < 1)
        begin
        $fclose(finput);
        fir_val       = 1'b0;
        end
      else
        begin
        end
      end
  end
```