Computer Organization and Architecture

4 Integer Arithmetic

Tongwei Ren

Sep. 17, 2018

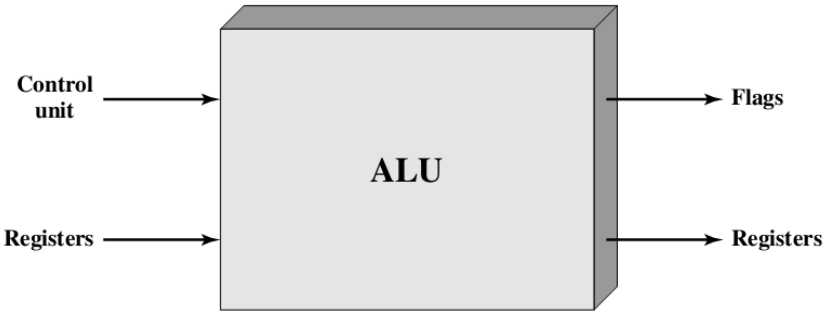
南京大學
NANJING UNIVERSITY

Review

- Integer representation
- Floating-point Representation
- Decimal representation

Arithmetic And Logic Unit (ALU)

- ALU is that part of the computer that actually performs arithmetic and logical operations on data
  - Data are presented to the ALU in registers, and the results of an operation are stored in registers
  - The flag values, also stored in registers, are set as the operation result
  - The control unit provides signals that control the operation of the ALU and the movement of the data into and out of the ALU

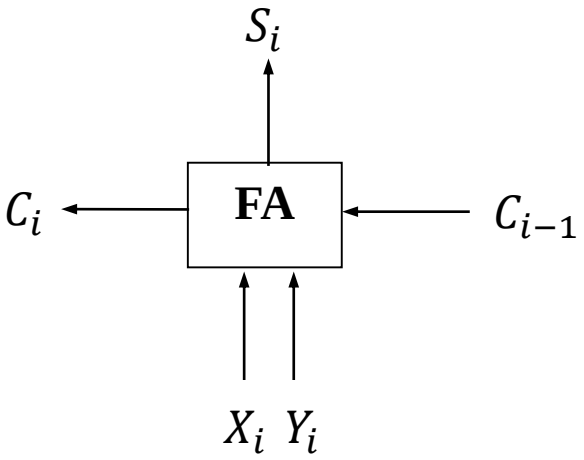


Full Adder

- Addition: $X + Y$
- One bit addition:

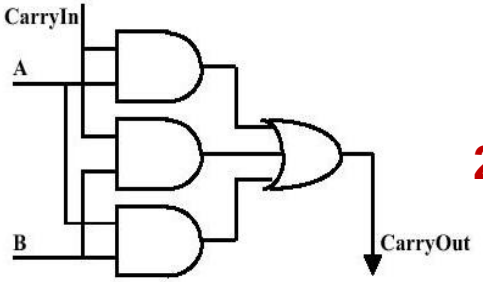
$$S_i = X_i \oplus Y_i \oplus C_{i-1}$$

$$C_i = X_i C_{i-1} + Y_i C_{i-1} + X_i Y_i$$



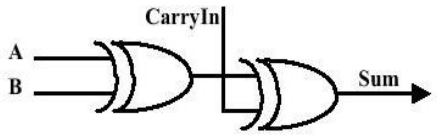
AND gate latency: 1ty
OR gate latency: 1ty
XOR gate latency: 3ty

CarryOut = B & CarryIn | A & CarryIn | A & B



2ty

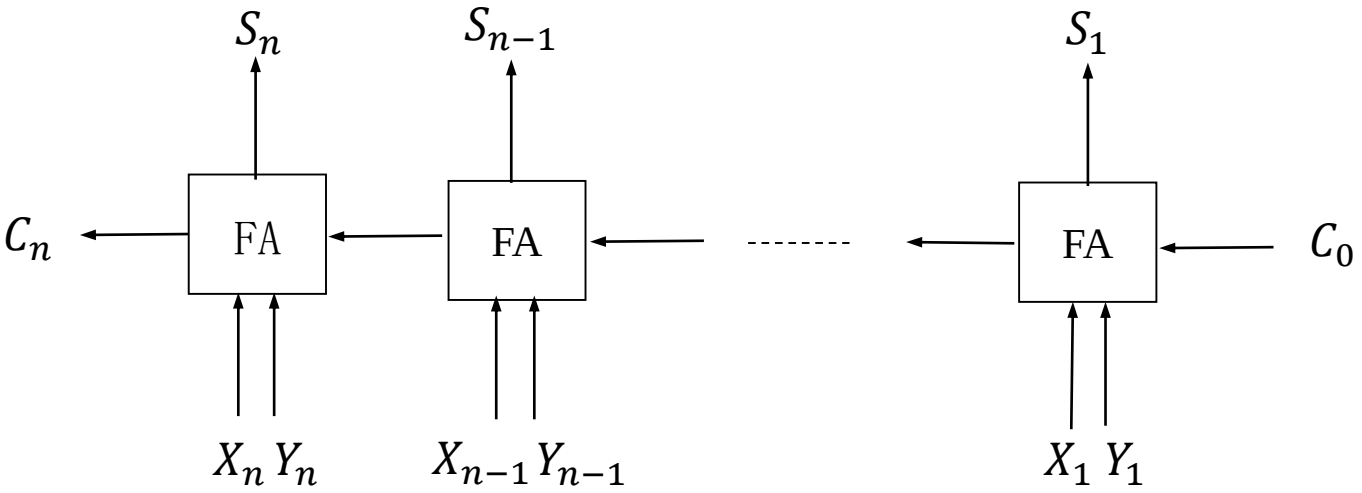
Sum = A XOR B XOR CarryIn



6ty

Serial Carry Adder

- Latency
  - *Cn*: 2n ty
  - *Sn*: (2n+1) ty
- Drawback: slow



Carry Look Ahead Adder

- Carry look ahead

$$C_i = X_i C_{i-1} + Y_i C_{i-1} + X_i Y_i$$

$$C_1 = X_1 Y_1 + (X_1 + Y_1) C_0$$

$$C_2 = X_2 Y_2 + (X_2 + Y_2) X_1 Y_1 + (X_2 + Y_2)(X_1 + Y_1) C_0$$

$$C_3 = X_3 Y_3 + (X_3 + Y_3) X_2 Y_2 + (X_3 + Y_3)(X_2 + Y_2) X_1 Y_1 + (X_3 + Y_3)(X_2 + Y_2)(X_1 + Y_1) C_0$$

$$C_4 = \ldots\ldots$$

Let

$$P_i = X_i + Y_i \qquad G_i = X_i Y_i$$

$$C_1 = G_1 + P_1 C_0$$

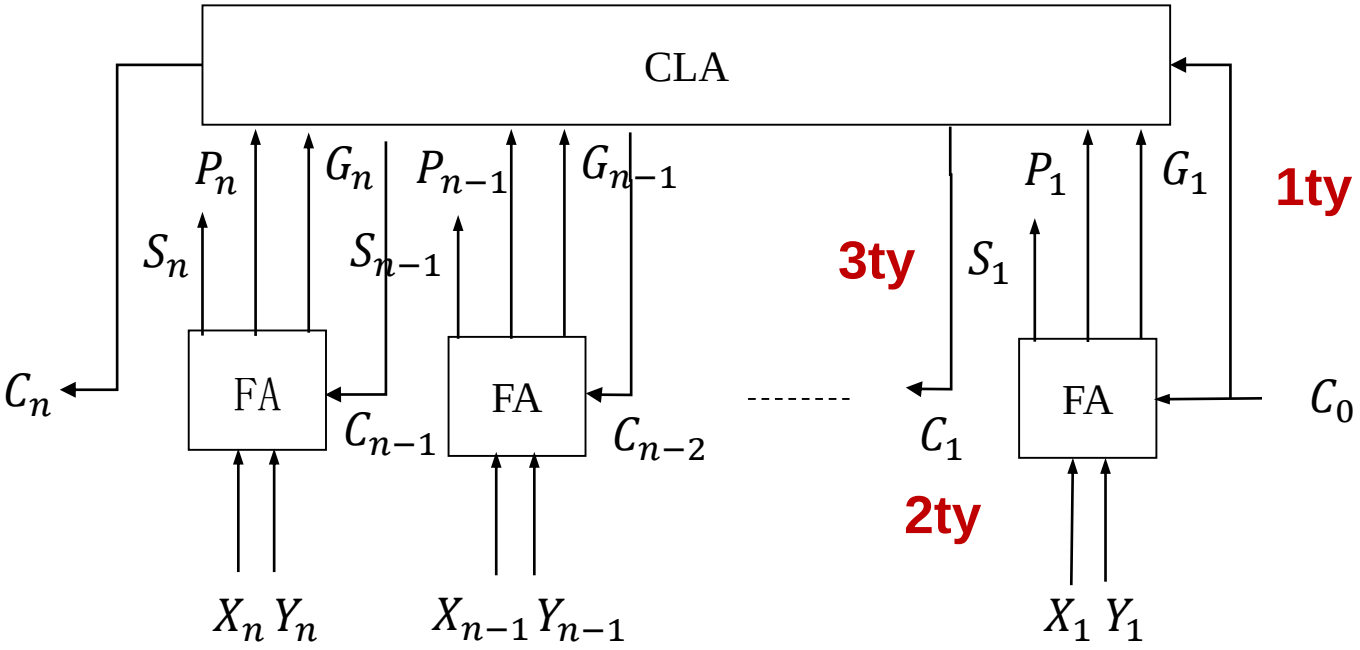
$$C_2 = G_2 + P_2 G_1 + P_2 P_1 C_0$$

$$C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_0$$

$$C_4 = \ldots\ldots$$

Carry Look Ahead Adder (cont.)

- Drawback: complex

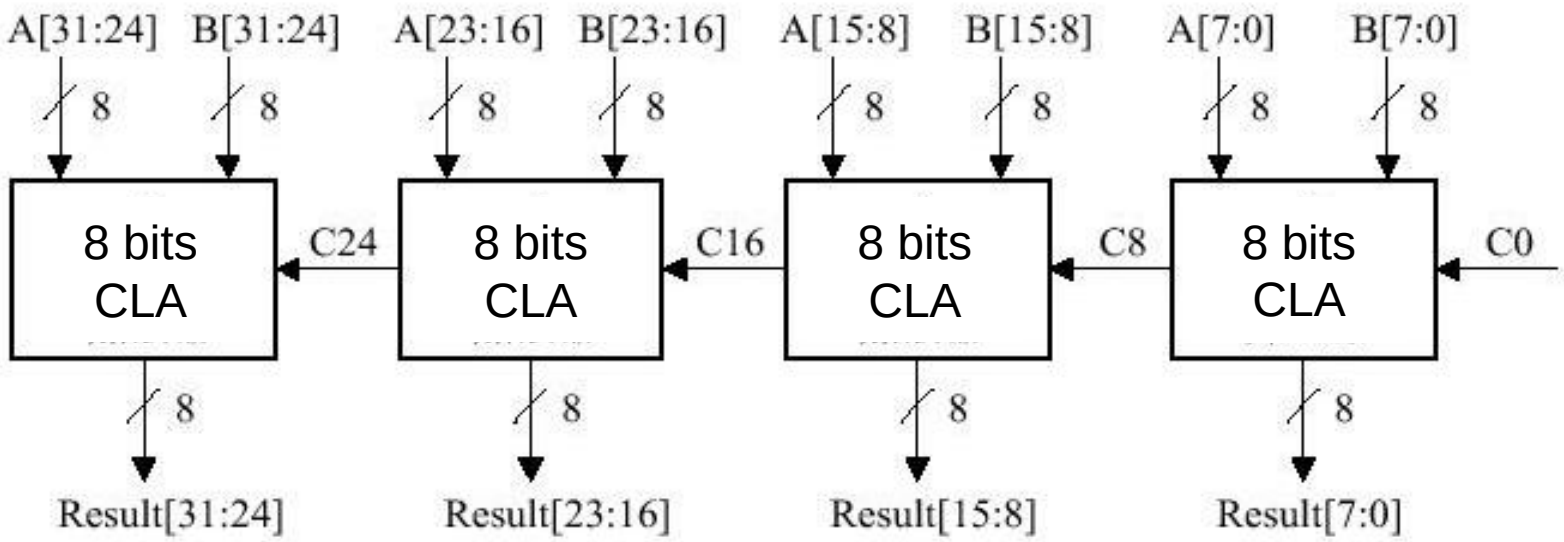


Latency = 1ty + 2ty + 3ty = 6ty

[李嘉麒, 131250089]

Partial Carry Look Ahead Adder

- Idea
  - Serially connect some CLA adders
- Example



Latency = 3ty + 2ty + 2ty + 5ty = 12ty

Addition

- $[X+Y]_C = [X]_C + [Y]_C \ (\text{MOD } 2^n)$
- Overflow

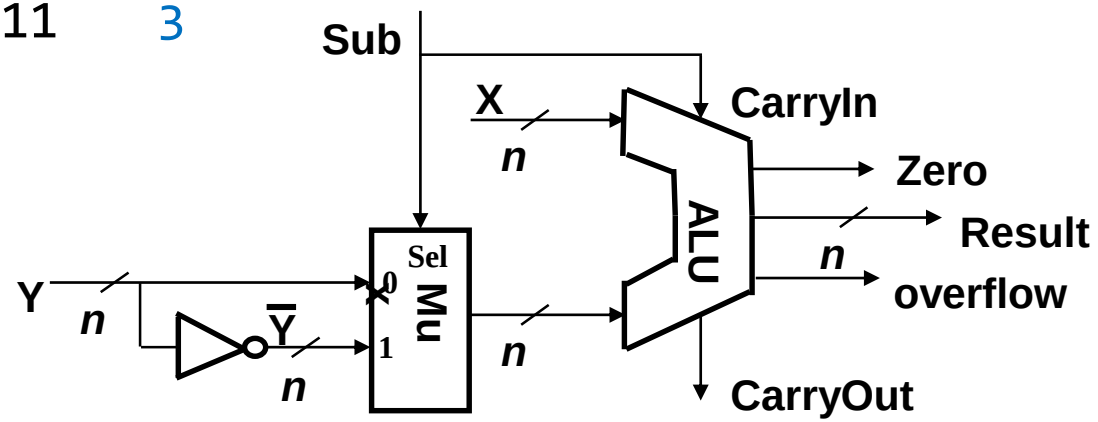
```
   -7        1001          7        0111
 + -6      + 1010        + 6      + 0110
-----     ------        ----     ------
  -13       10011   3     13       1101   -3
```

- $C_n \neq C_{n-1}$: $overflow = C_n \oplus C_{n-1}$
- $S_n \neq X_n, Y_n$,: $overflow = X_n Y_n \overline{S_n} + \overline{X_n}\ \overline{Y_n} S_n$

Subtraction

- $[X–Y]_C = [X]_C + [–Y]_C$ (MOD 2^n)
- Overflow: same to addition

```
   -7        1001
-  6      +  1010
-------    -------
  -13       10011    3
```



[袁睿, 131250088]

Multiplication

- Manual multiplication
  - If $Y_i = 0$, result is 0; otherwise, result is X
  - Left shift result in each step
  - Add all results
- Modification for computer
  - Compute **partial product** in each step
  - **Right-shift** partial product instead of left-shift results
  - **Only shift** partial product if $Y_i = 0$

```
   7          0111
 × 6        × 0110
----       ------
  42          0000
             0111
            0111
           0000
          -------
          0101010
```

Multiplication (cont.)

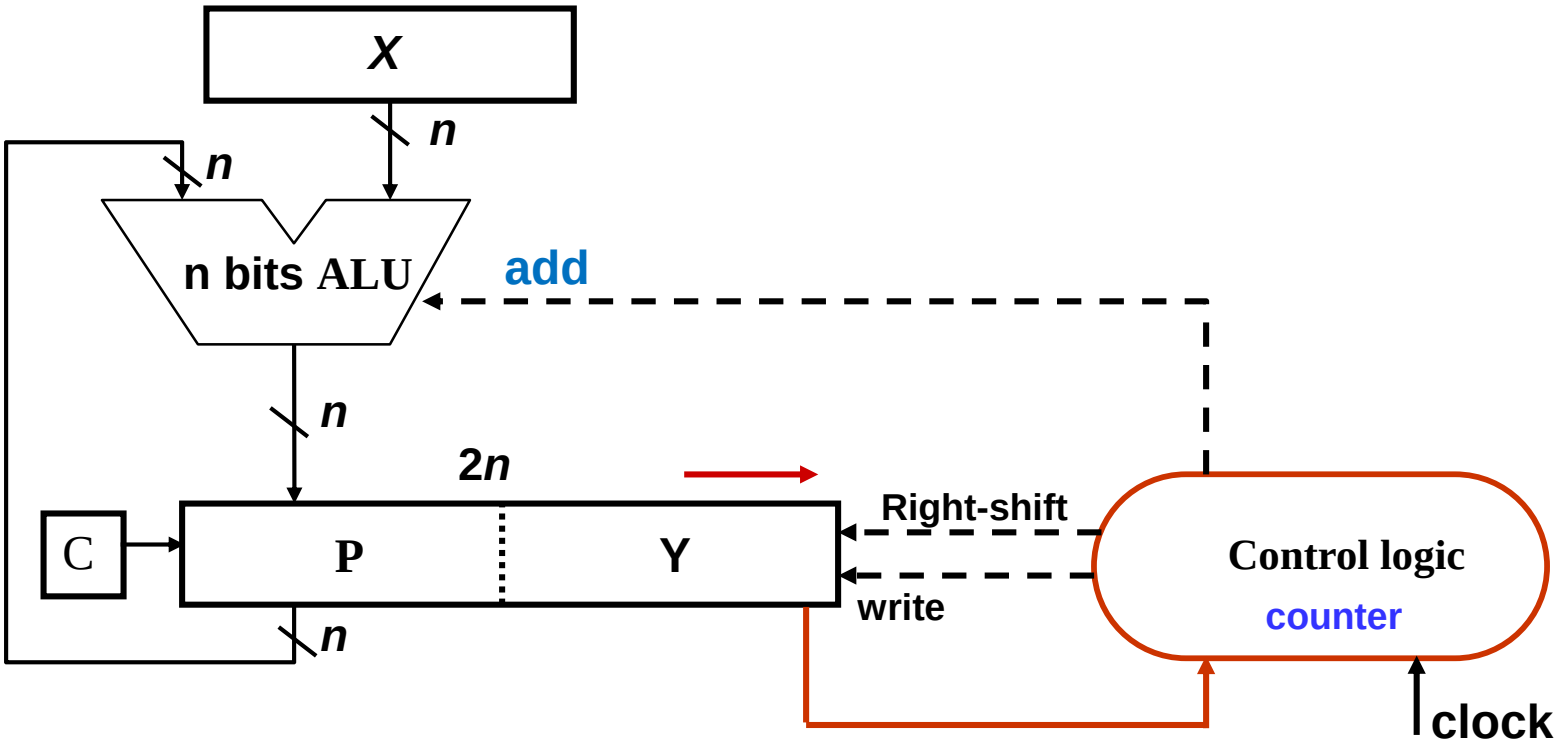
- Example

```
    7          0111
  × 6        × 0110
-----      ---------
   42          0000
              0111
             0111
            0000
          ---------
          0101010
```

		Partial product	
initial		0000	
0	->	00000	
1	+	01110	
	->	001110	
1	+	101010	
	->	0101010	
0	->	00101010	42

Multiplication (cont.)

- Implementation



Multiplication (cont.)

- Implementation (cont.)

```
    7          0111
  × 6        × 0110
-----      ---------
   42          0000
              0111
             0111
            0000
          ---------
          0101010
```

		product	Y
initial		0000	0110
0	->	0000	0011
1	+	0111	0011
	->	0011	1001
1	+	1010	1001
	->	0101	0100
0	->	0010	1010

Multiplication (cont.)

- Problem: $[X \times Y]_C \neq [X]_C \times [Y]_C$

7	0111			-7	1001		
× 6	× 0110			× -6	× 1010		
42	0101010	42		42	1011010	-38	

- Rough idea
  - Change multiplicand and multiplier from complement representation to sign magnitude representation
  - Change product from sign magnitude representation to complement representation

Multiplication (cont.)

- Booth's algorithm

$$X \times Y = X \times Y_n Y_{n-1} \dots Y_2 Y_1$$

$$= X \times (-Y_n \times 2^{n-1} + Y_{n-1} \times 2^{n-2} + \cdots + Y_2 \times 2^1 + Y_1 \times 2^0)$$

$$= X \times \begin{pmatrix} -Y_n \times 2^{n-1} + Y_{n-1} \times (2^{n-1} - 2^{n-2}) + \cdots \\ +Y_2 \times (2^2 - 2^1) + Y_1 \times (2^1 - 2^0) \end{pmatrix}$$

$$= X \times \begin{pmatrix} (Y_{n-1} - Y_n) \times 2^{n-1} + (Y_{n-2} - Y_{n-1}) \times 2^{n-2} + \cdots \\ +(Y_1 - Y_2) \times 2^1 + (Y_0 - Y_1) \times 2^0 \end{pmatrix} \quad Y_0 = 0$$

$$= 2^n \times \sum_{i=0}^{n-1} \left(X \times (Y_i - Y_{i+1}) \times 2^{-(n-i)} \right)$$

$$P_{i+1} = 2^{-1} \times \left(P_i + X \times (Y_i - Y_{i+1}) \right)$$

Multiplication (cont.)

- Booth's algorithm (cont.)
  1. Add $Y_0 = 0$
  2. According to $Y_{i+1}Y_i$, determine whether add $+X$, $-X$, $+0$
  3. Right shift partial product
  4. Repeat step 2 and step 3 n times, and get the final product

Multiplication (cont.)

- Booth's algorithm (cont.)

```
   -7
 × -6
 ----
   42

[X]c = 1001
[-X]c = 0111
[Y]c = 1010
```

		product	Y	
initial		0000	10100	
Y_0-Y_1 = 0	->	0000	01010	
Y_1-Y_2 = -1	-X	0111	01010	
	->	0011	10101	
Y_2-Y_3 = 1	+X	1100	10101	
	->	0110	01010	
Y_3-Y_4 = -1	-X	1101	01010	
	->	0110	10101	106

?

Multiplication (cont.)

- Booth's algorithm (cont.)

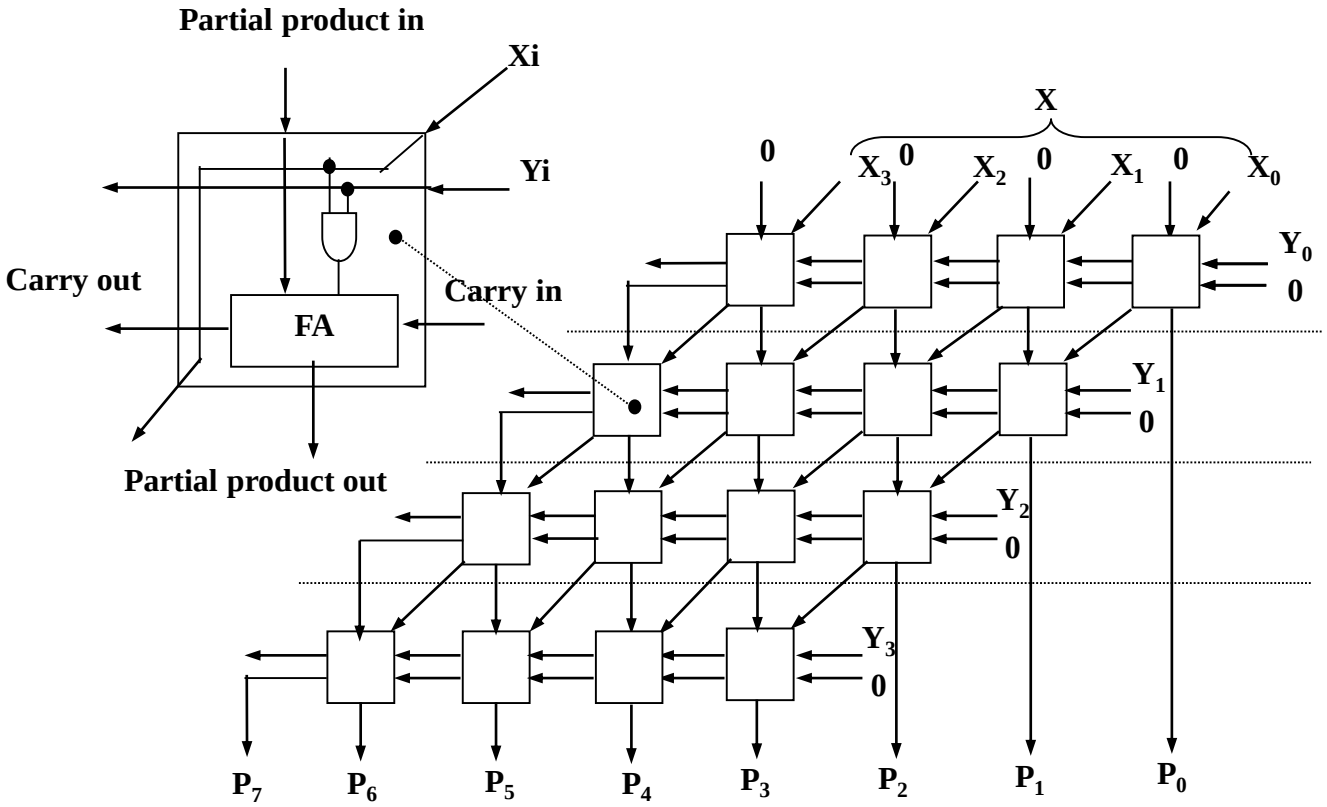
```
   -7
 × -6
 ----
   42

[X]c = 1001
[-X]c = 0111
[Y]c = 1010
```

		product	Y	
initial		0000	10100	
Y_0-Y_1 = 0	->	0000	01010	
Y_1-Y_2 = -1	-X	0111	01010	
	->	0011	10101	
Y_2-Y_3 = 1	+X	1100	10101	
	->	1110	01010	
Y_3-Y_4 = -1	-X	0101	01010	
	->	0010	10101	42

Multiplication (cont.)

- Array multiplier

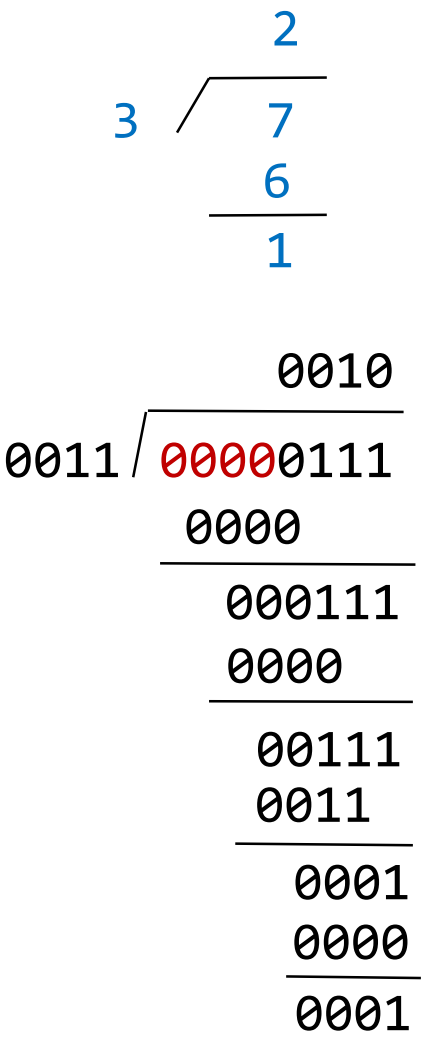


Division

- Preprocessing
  - If $X = 0$ and $Y \neq 0$: 0
  - If $X \neq 0$ and $Y = 0$: exception
  - If $X = 0$ and $Y = 0$: NaN (Not a Number)
  - If $X \neq 0$ and $Y \neq 0$: further processing

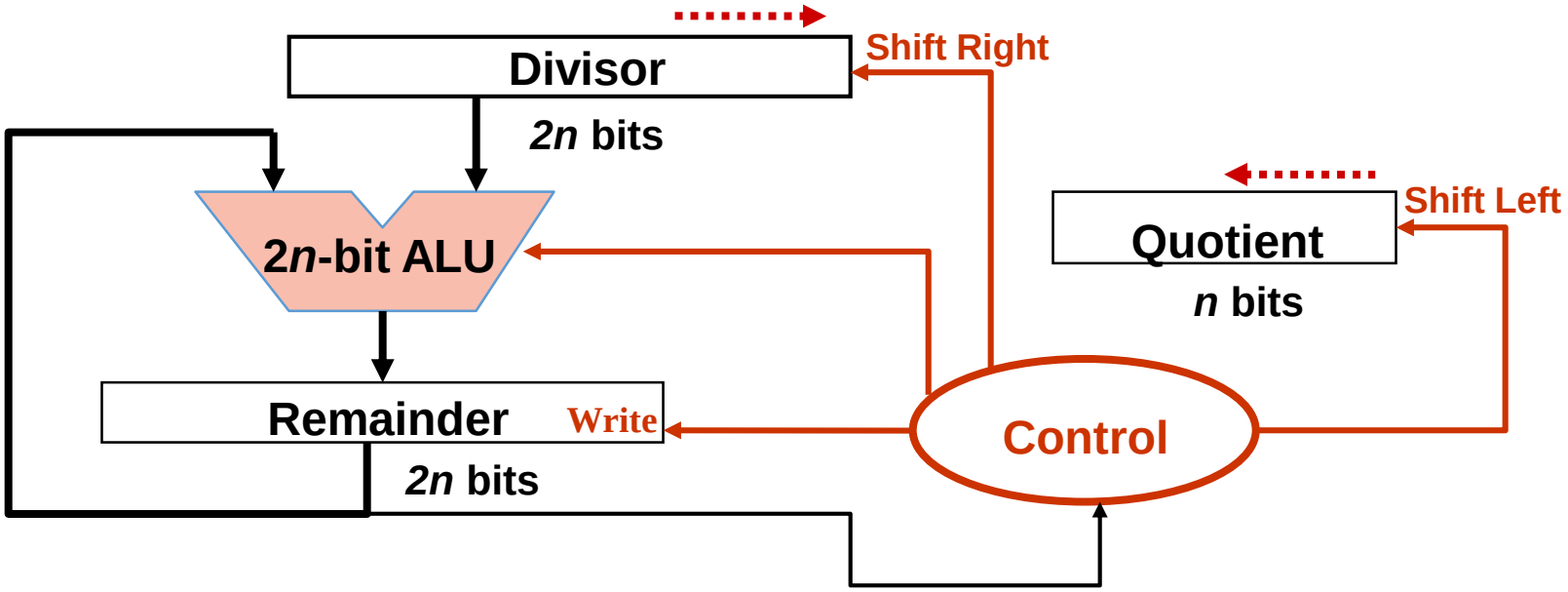
Division (cont.)

- Manual division
  - Examine dividend from left to right, until the set of bits examined represents a number greater than or equal to the divisor
  - Subtract divisor from dividend, the result is 1 if the partial remainder is larger than 0; otherwise, the results is 0
  - Right shift divisor and repeat the above step



Division (cont.)

- Implementation



Division (cont.)

- Example

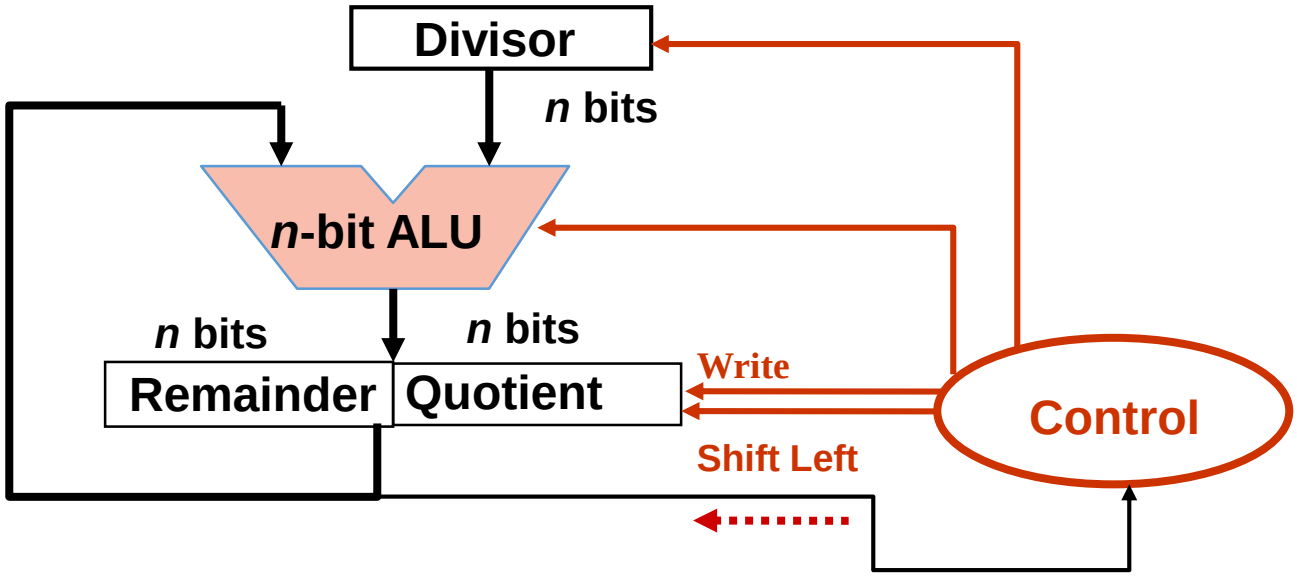
```
                0010
          ------------
0011 ) 00000111
       0000
       ------
        000111
        0000
        ------
         00111
         0011
         ----
          0001
          0000
          ----
          0001
```

```
      2
    -----
3 ) 7
    6
    -
    1
```

	remainder		divisor		quotient
initial	00000111		00110000		0000
	00000111	->	00011000		0000
not	00000111		00011000	<-	0000
	00000111	->	00001100		0000
not	00000111		00001100	<-	0000
	00000111	->	00000110		0000
enough -	00000001		00000110	<-	0001
	00000001	->	00000011		0001
not	00000001		00000011	<-	0010

Division (cont.)

- Implementation



Division (cont.)

- Example (cont.)

```
                0010
          ------------
0011 ) 00000111
       0000
       ------
         000111
         0000
         ------
           00111
           0011
           ----
             0001
             0000
             ----
             0001
```

```
    2
  -----
3 ) 7
    6
  -----
    1
```

		remainder		quotient	divisor
initial		0000		0111	0011
	<-	0000	<-	111	0011
not		0000		1110	0011
	<-	0001	<-	110	0011
not		0001		1100	0011
	<-	0011	<-	100	0011
enough	-	0000		1001	0011
	<-	0001	<-	001	0011
not		0001		0010	0011

Division (cont.)

- How to judge whether remainder is **large enough**
  - If remainder has the same sign with divisor: **subtraction**
  - If remainder has the different sign with divisor: **addition**

remainder sign	Divisor sign	Subtraction		Addition	
		0	1	0	1
0	0	Enough	Not enough	----	----
0	1	----	----	Enough	Not enough
1	0	----	----	Not enough	Enough
1	1	Not enough	Enough	----	----

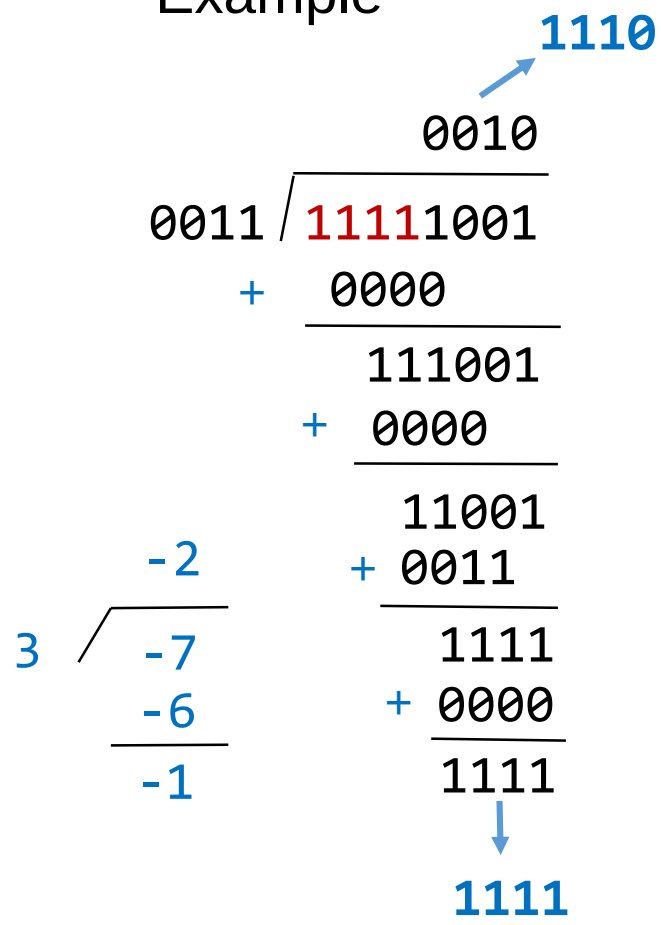
Division (cont.)

- Procedure
  - Extend the dividend by adding n bits sign in the front, and store it in the remainder and quotient registers
  - Left shift the remainder and quotient, and judge whether the remainder large enough
    - If large enough, do addition or subtraction and set quotient to 1
    - If not large enough, set quotient to 0
  - Repeat the above step
  - If the dividend has the different sign with divisor, replace quotient with its complement
  - The remainder is in the remainder register

[王子安, 141250146]

Division (cont.)

- Example



	remainder		quotient	divisor
initial	1111		1001	0011
<-	1111	<-	001	0011
+	0010		001	0011
(recover) -	1111		0010	0011
<-	1110	<-	010	0011
+	0001		010	0011
(recover) -	1110		0100	0011
<-	1100	<-	100	0011
(enough) +	1111		1001	0011
<-	1111		001	0011
+	0010		001	0011
(recover) -	1111		0010	0011
	1111		1110	

Division (cont.)

- Problem: recover remainder is high cost
- Idea: not recover remainder
  - Only consider subtraction
    - If remainder R_i is large enough

$$R_{i+1} = 2R_i - Y$$

    - If remainder is not large enough

$$R_{i+1} = 2(R_i + Y) - Y = 2R_i + Y$$

Division (cont.)

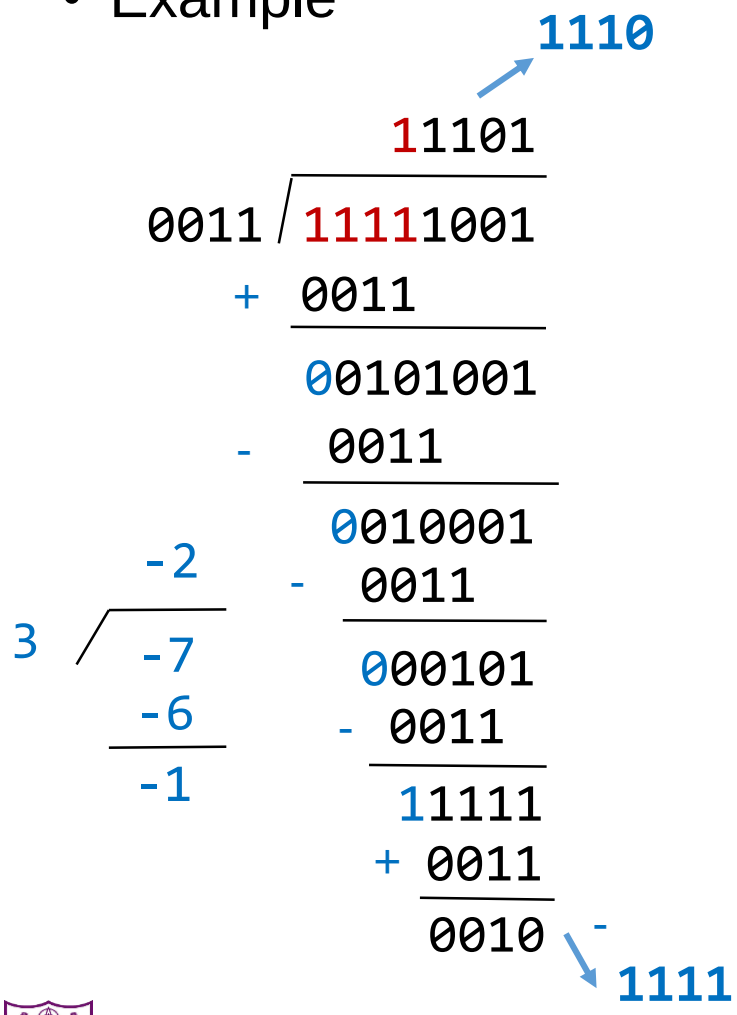
- Procedure
  - Extend the dividend by adding n bits sign in the front, and store it in the remainder and quotient registers
  - If dividend has the same sign with divisor, do subtraction; otherwise, do addition
    - If the remainder has same sign with divisor, $Q_n = 1$; otherwise, $Q_n = 0$
  - If the remainder has the same sign with divisor, $R_{i+1} = 2R_i - Y$; otherwise, $R_{i+1} = 2R_i + Y$
    - If the new remainder has the same sign to divisor, set quotient to 1; otherwise, set quotient to 0
  - Repeat the above step

Division (cont.)

- Procedure (cont.)
  - Left shift quotient, if quotient is negative (the dividend has the different sign with divisor), quotient adds 1
  - The remainder has the different sign with dividend
    - Remainder adds divisor if the dividend has the same sign with divisor, and subtracts divisor otherwise

Division (cont.)

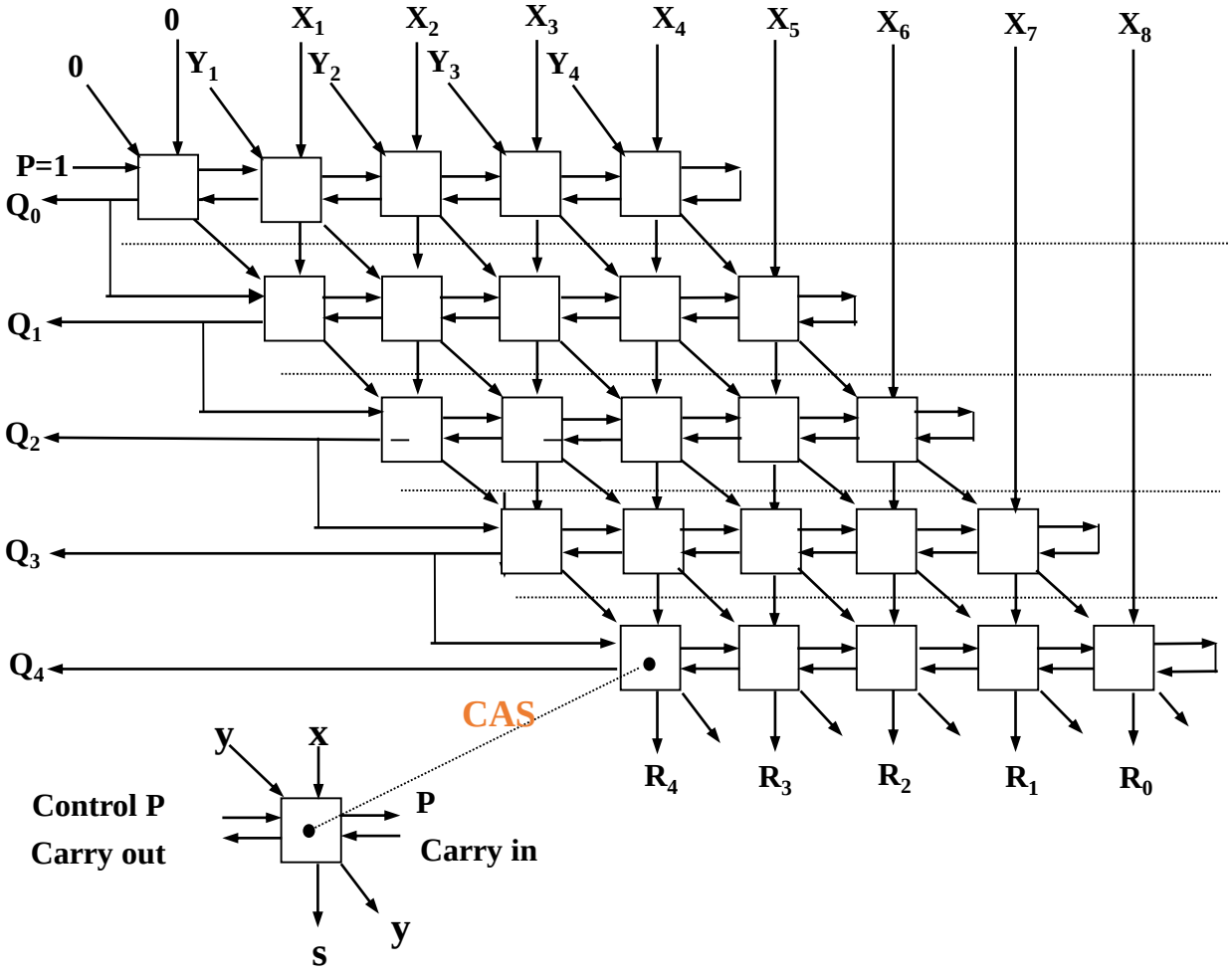
- Example



	remainder		quotient	divisor
initial	1111		1001	0011
+	0010		1001 1	0011
<-	0101	<-	0011	0011
-	0010		0011 1	0011
<-	0100	<-	0111	0011
-	0001		0111 1	0011
<-	0010	<-	1111	0011
-	1111		1111 0	0011
<-	1111	<-	1110	0011
+	0010		1110 1	0011
	↓ -		↓ <-,+1	
	1111		1110	

Division (cont.)

- Array division



Summary

- Integer representation
- ALU, full adder, serial carry adder, carry look ahead adder
- Operation
  - Addition
  - Subtraction
  - Multiplication
  - Division

Thank You

rentw@nju.edu.en

南京大學
NANJING UNIVERSITY