CS301
Embedded System and Microcomputer Principle

Lecture 14: Arithmetic

2024 Fall

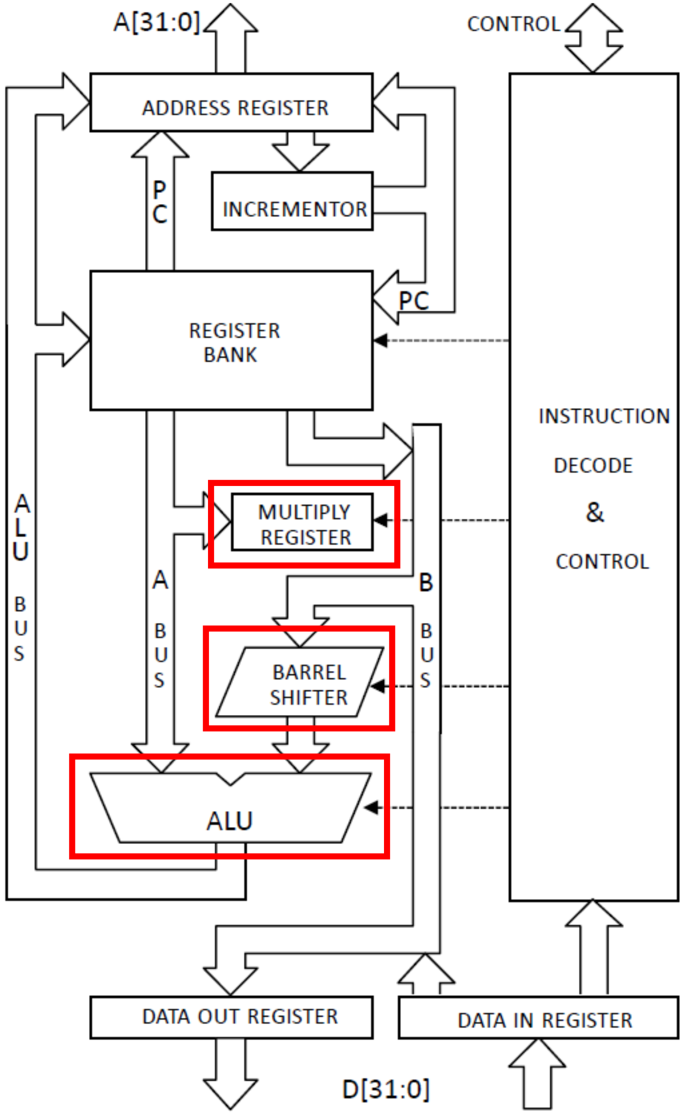
baiyh@sustech.edu.cn

Recap

- **ADC**
  - **Terms**
    - Sampling rate
    - Sampling resolution
  - **ADC output**
    - stepSize= V_REF/numOfSteps
    - Output=round(V_input/stepSize)
  - **SAR Architecture**
    - binary search
    - 12 conversion cycles for 12-bit ADC
  - **STM32 ADC**
    - 12-bit
    - Single, continues, scan conversion mode
    - continues + DMA
    - regular, injected group
    - Total conversion time = Tsampling+ Tconversion

Arithmetic Logic Unit

- The ALU is an important part of any micropro
- The ARM microprocessor divides the ALU fu ;
  - a multiplier (that uses Booth’s algorithm),
  - the ‘barrel shifter’
  - and the rest of the ALU including: the adder and lo k is generally referred to as the ALU.)



Outline

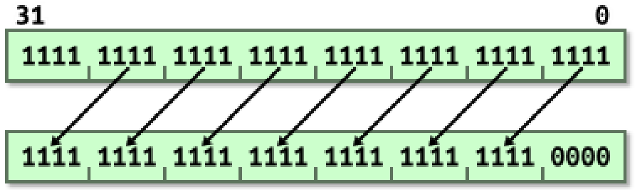
- **Barrel Shifter**
- ALU Adder
- Multiplier

Barrel Shifter

- Shift and rotate are very important operations.
- With integrated circuit techniques these are easily implemented by a barrel shifter:
- Different types of shift
  - logical shift left (LSL)
  - logical shift right (LSR)
  - arithmetic shift right (ASR)
  - Rotate right (ROR)
  - Rotate extended (RRX)

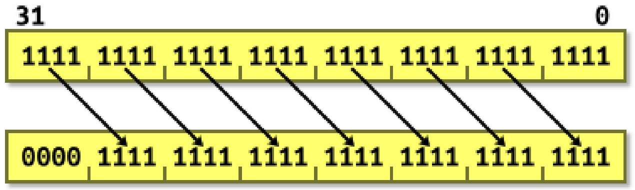
Different shifts

- Logical shift left (LSL)
  - bits are shifted to the right and the new bits added in at the left hand side are 0



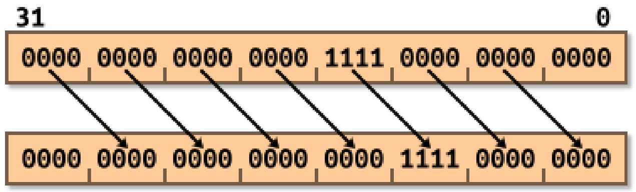
LSL#4

- Logical shift right (LSR)
  - bits are shifted to the left and the new bits added in at the right hand side are 0

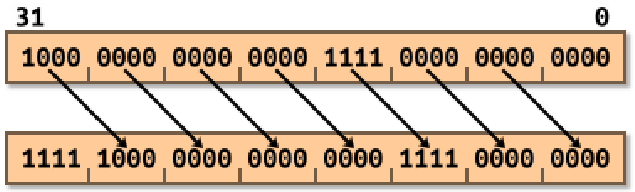


LSR#4

- Arithmetic shift right (ASR)



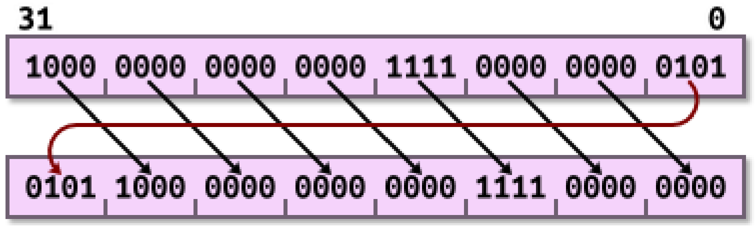
ASR#4 for positive value



ASR#4 for negative value

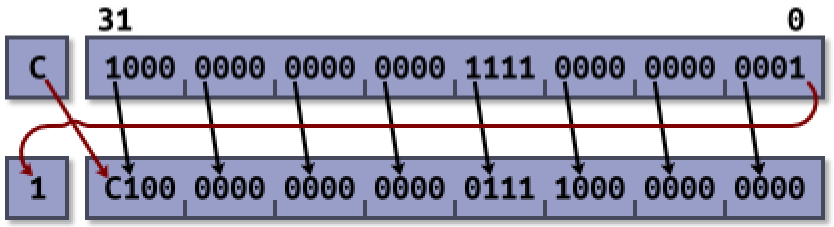
Different shifts

- Rotate right (ROR)
  - bits are rotated rightwards so that the bits shifted out at the right hand side reappear at the left hand side.



ROR#4

- Rotate extended (RRX)
  - bits are shifted right **one place only** and the carry flag is shifted into the new most significant bit. The least significant bit is shifted into the carry flag only if the mnemonic specifies an S

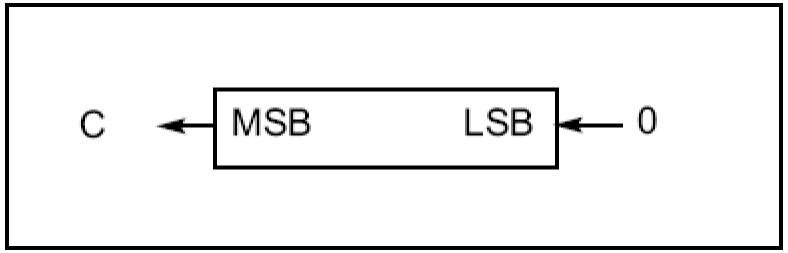


RRX#1

LSL instruction

MOVS Rd, Rn, LSL #numOfShift
LSL(S) Rd, Rn, #numOfShift ;Logical Shift Left

In LSL, as bits are shifted from right to left, 0 enters the LSB and the MSB exits to the carry flag. In other words, **in LSL 0 is moved to the LSB, and the MSB is moved to the C.**



this instruction multiplies content of the register by 2 if after LSL the carry flag is not set.

In the code you can see what happens to 00100110 after running 3 LSL instructions.

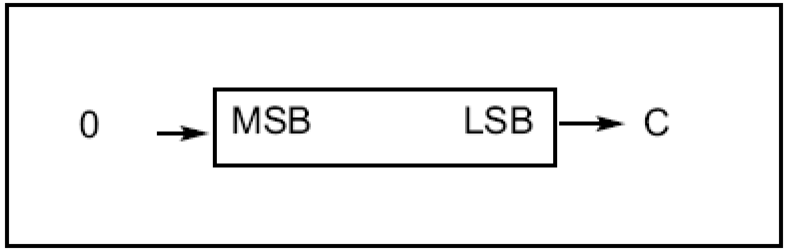
```
                ;Assume C = 0
MOV R2 , #0x26  ;R2 = 0000 0000 0000 0000 0000 0000 0010 0110 (38) C = 0
LSLS R2,R2,#1   ;R2 = 0000 0000 0000 0000 0000 0000 0100 1100 (74) C = 0
LSLS R2,R2,#1   ;R2 = 0000 0000 0000 0000 0000 0000 1001 1000 (148) C = 0
LSLS R2,R2,#1   ;R2 = 0000 0000 0000 0000 0000 0001 0011 0000 (296) C = 0
```

LSR instruction

LSR(S)Rd, Rn, #numOfShift ;Logical Shift Right

MOVS Rd, Rn, LSR #numOfShift

In LSR, as bits are shifted from left to right, 0 enters the MSB and the LSB exits to the carry flag. In other words, **in LSR 0 is moved to the MSB, and the LSB is moved to the C.**



this instruction divides content of the register by 2 and carry flag contains the remainder of division.

In the code you can see what happens to 0010 0110 after running 3 LSR instructions.

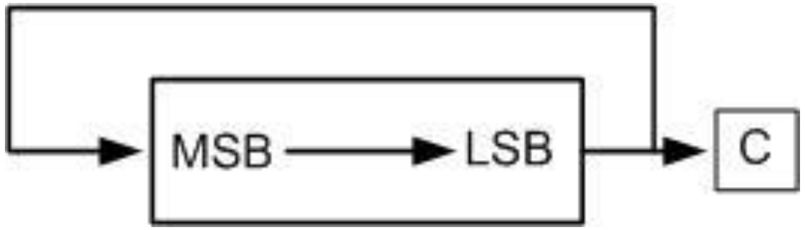
```
MOV R2, #0x26    ;R2 = 0000 0000 0000 0000 0000 0000 0010 0110 (38)
LSRS R2, R2, #1  ;R2 = 0000 0000 0000 0000 0000 0000 0001 0011 (19) C = 0
LSRS R2, R2, #1  ;R2 = 0000 0000 0000 0000 0000 0000 0000 1001 (9) C = 1
LSRS R2, R2, #1  ;R2 = 0000 0000 0000 0000 0000 0000 0000 0100 (4) C = 1
```

ROR instruction (Rotate Right)

ROR Rd, Rm, #numOfShifts **;Rotate Rm right Rn bit positions**

MOVS Rd, Rm, ROR #numOfShifts

In ROR, as bits are rotated from left to right, the LSB goes to the MSB and to the carry flag.



See what happens to 0010 0110 after running 3 ROR instructions:

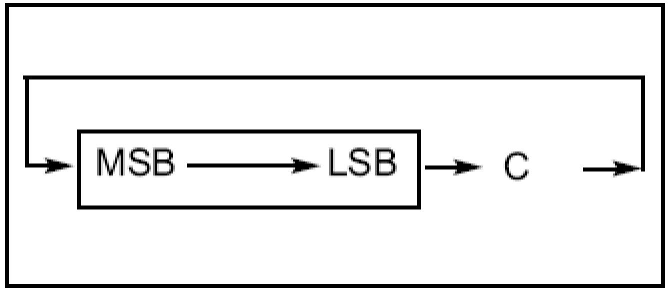
```
                          ;assume C = 0 (carry is 0 )
MOV     R2, #0x26         ;R2 = 0000 0000 0000 0000 0000 0000 0010 0110
RORS    R2, R2, #1        ;R2 = 0000 0000 0000 0000 0000 0000 0001 0011 C = 0
RORS    R2, R2, #1        ;R2 = 1000 0000 0000 0000 0000 0000 0000 1001 C = 1
RORS    R2, R2, #1        ;R2 = 1100 0000 0000 0000 0000 0000 0000 0100 C = 1
```

RRX instruction (Rotate Right with extend)

RRX(S) Rd, Rm ;Rotate Rm right 1 bit through C flag

MOVS Rd, Rm, RRX

In RRXS, as bits are rotated from left to right, the carry flag enters the MSB and the LSB exits to the carry flag. In other words, **in RRXS the C is moved to the MSB, and the LSB is moved to the C.**



这个地方写错了

0: 0000 0000 0000 0000 0000 0000 0010 0110, C=0
1: 0000 0000 0000 0000 0000 0000 0001 0011, C=0
2: 0000 0000 0000 0000 0000 0000 0000 1001, C=1
3: 1000 0000 0000 0000 0000 0000 0000 0100, C=1

See what happens to 0010 0110 after running 3 ROR instructions:

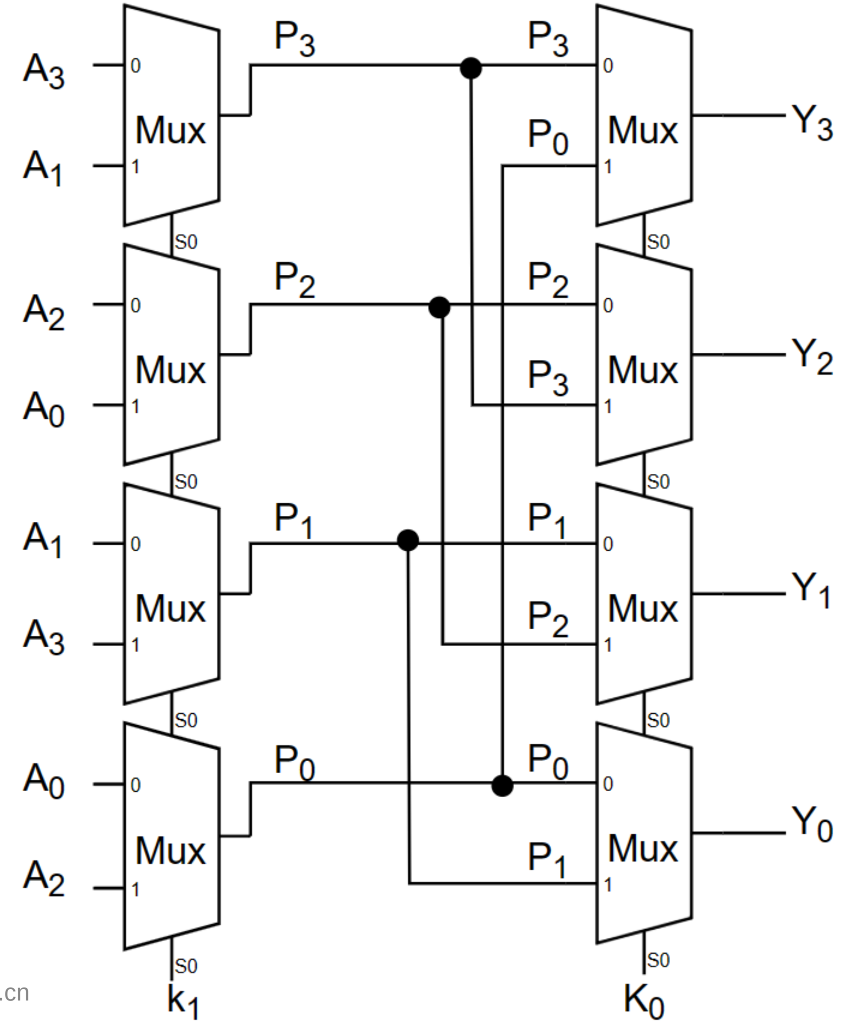
```
                      ;assume C = 0 (carry is 0 )
MOV    R2, #0x26      ;R2 = 0010 0000 0000 0000 0000 0000 0000 0110
RRXS   R2, R2         ;R2 = 0001 0000 0000 0000 0000 0000 0000 0011 C = 0
RRXS   R2, R2         ;R2 = 0000 0000 0000 0000 0000 0000 0000 1001 C = 1
RRXS   R2, R2         ;R2 = 1000 0000 0000 0000 0000 0000 0000 0100 C = 1
```

Barrel Shifter using Mux

桶移位

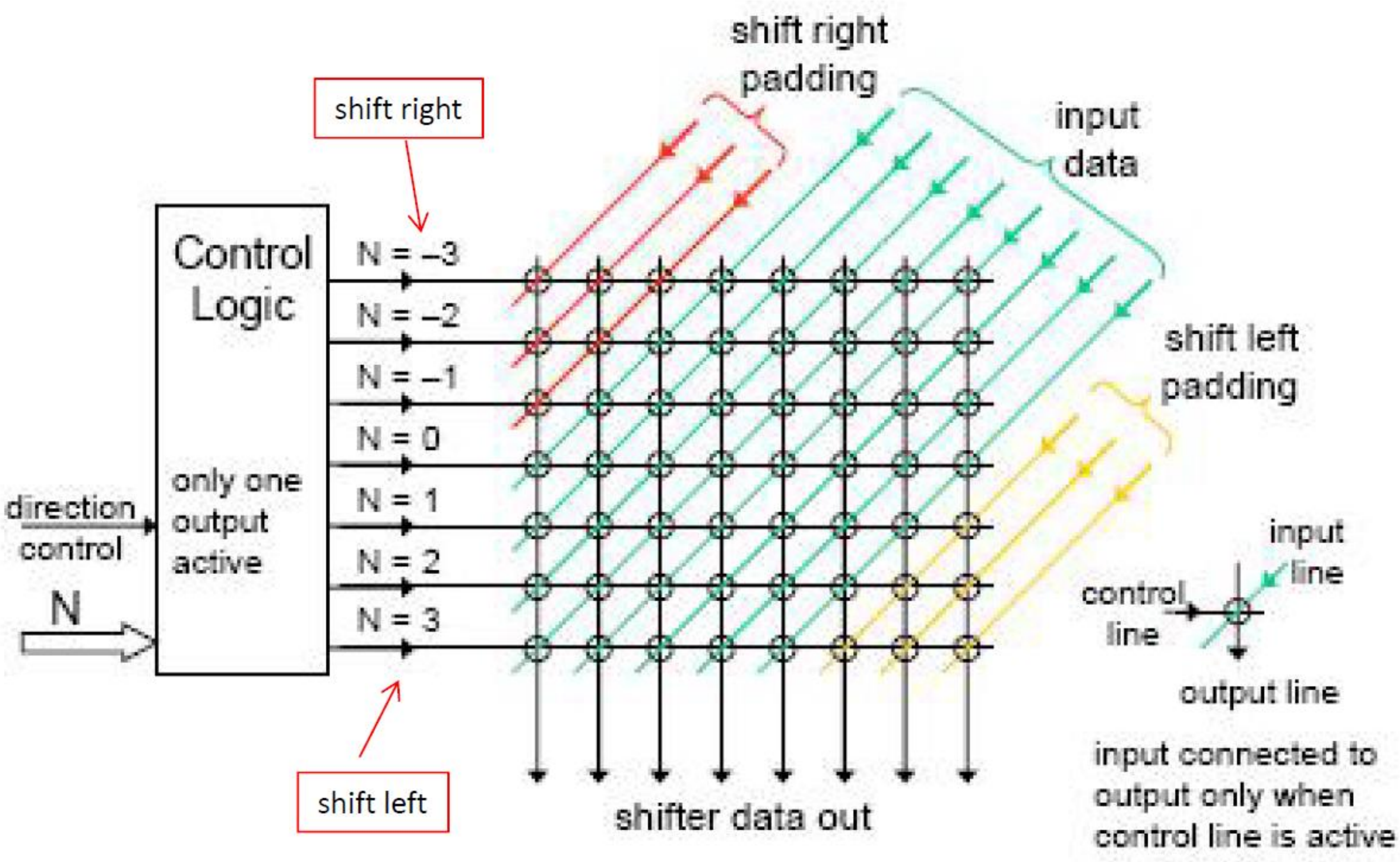
- 4 bits Barrel Shifter with 2:1 Mux example
  - K = 00, No change
  - K = 01, ROR#1
  - K = 10, ROR#2
  - K = 11, ROR#3

K_1	$P_3P_2P_1P_0$	K_0	$Y_3Y_2Y_1Y_0$
0	$A_3A_2A_1A_0$	0	$A_3A_2A_1A_0$
0	$A_3A_2A_1A_0$	1	$A_0A_3A_2A_1$
1	$A_1A_0A_3A_2$	0	$A_1A_0A_3A_2$
1	$A_1A_0A_3A_2$	1	$A_2A_1A_0A_3$

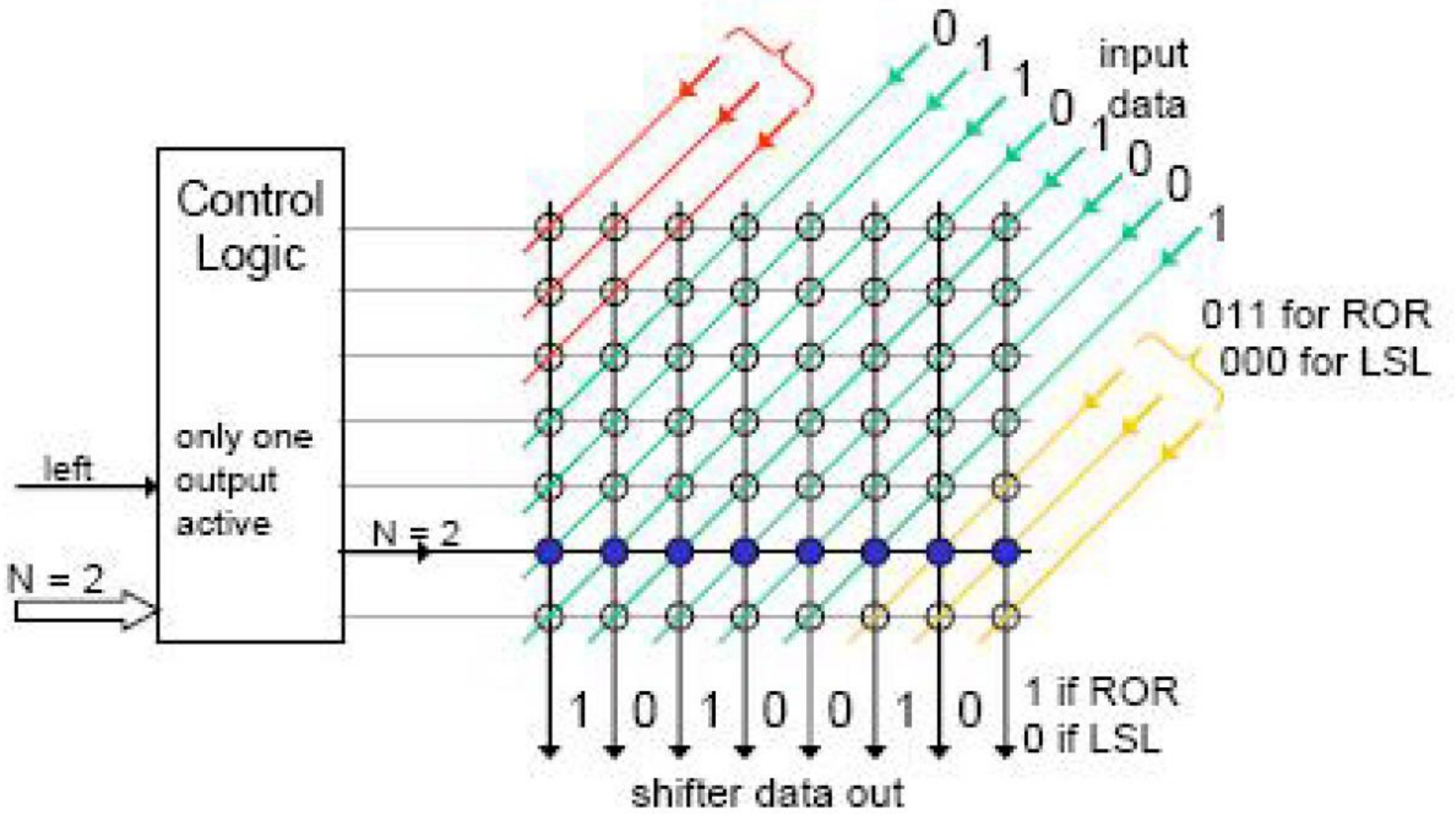


Barrel Shifter (8 bits)

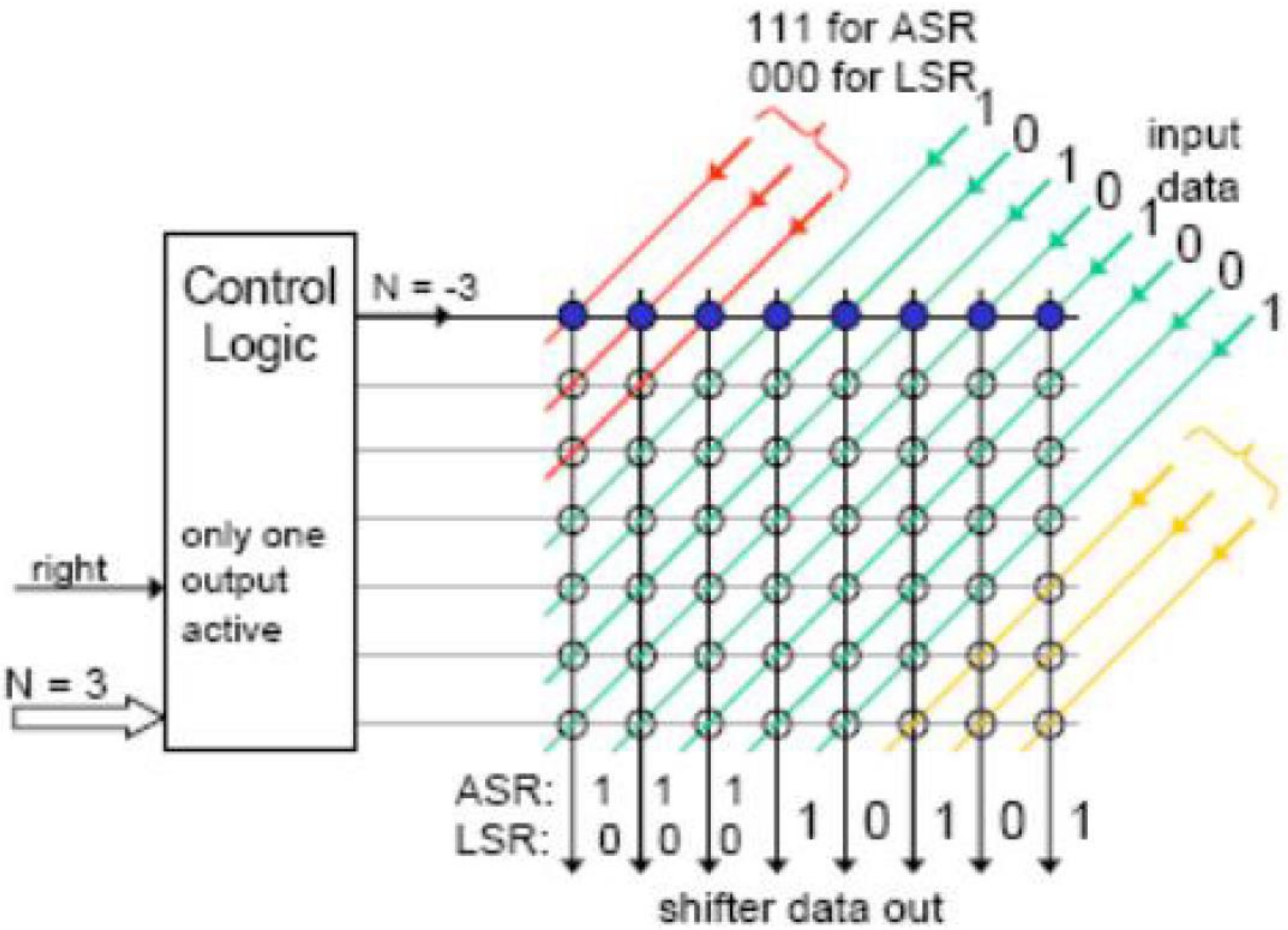
- Using Transistor



Example: Shift left by 2



Example: Shift right by 3



Shifts with other instructions

- The ARM barrel shifter is placed in the datapath so that it can be used with many instructions such as:

  MOV, ADD, ADC, SUB, RSB, AND, EOR, ORR, BIC

- The shift is done first before the output of the barrel shifter is passed onto the ALU so that the instruction:

  ADD r2, r3, r5, LSL #1

  performs the following operation:
  - the value in r5 is shifted left once (in effect doubling it’s value)
  - and then it is added to r3
  - and the sum placed in r2.

Example

- MOV r0, r0, LSL #1
  - Multiply R0 by two.
- MOV r1, r1, LSR #2
  - Divide R1 by four (unsigned).
- MOV r2, r2, ASR #2
  - Divide R2 by four (signed).
- MOV r3, r3, ROR #16
  - Swap the top and bottom halves of R3.
- ADD r4, r4, r4, LSL #4
  - Multiply R4 by 17. (N = N + N * 16)
- RSB r5, r5, r5, LSL #5
  - Multiply R5 by 31. (N = N * 32 - N

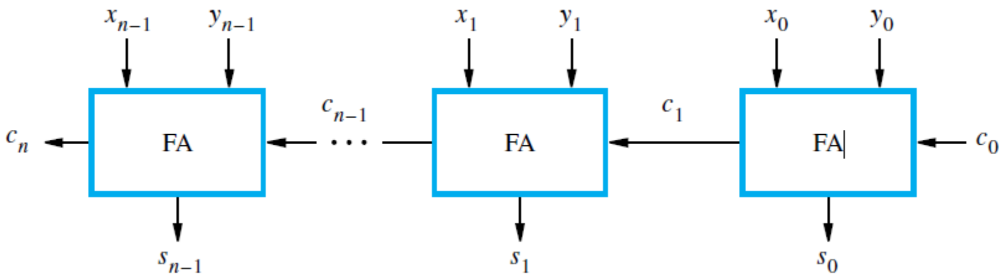
Outline

- Barrel Shifter
- **ALU Adder**
- Multiplier

Ripple carry adder

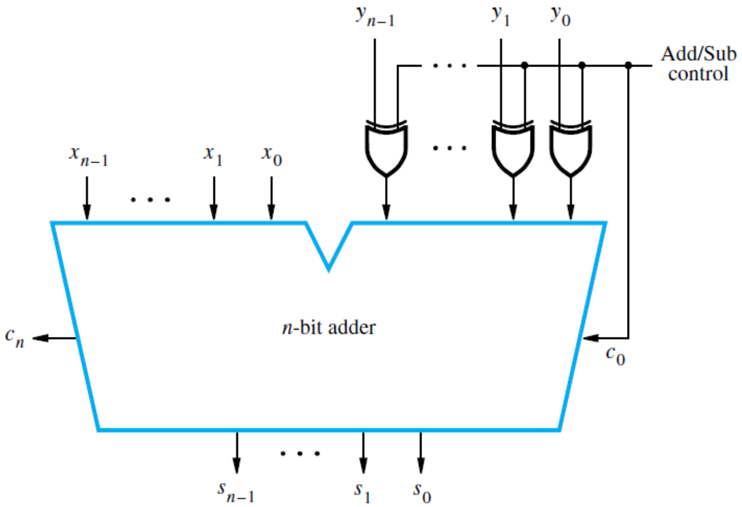
- A cascaded connection of n full-adder blocks can be used to add two n-bit numbers, carries propagate through full-adders.
  - The carry-in, c0, into the least-significant-bit (LSB) position provides a convenient means of adding 1 to a number. For instance, forming the 2’s-complement of a number involves adding 1 to the 1’s-complement of the number.

n-bit ripple-carry adder



Most significant bit (MSB) position

Least significant bit (LSB) position



Binary addition/
subtraction logic circuit

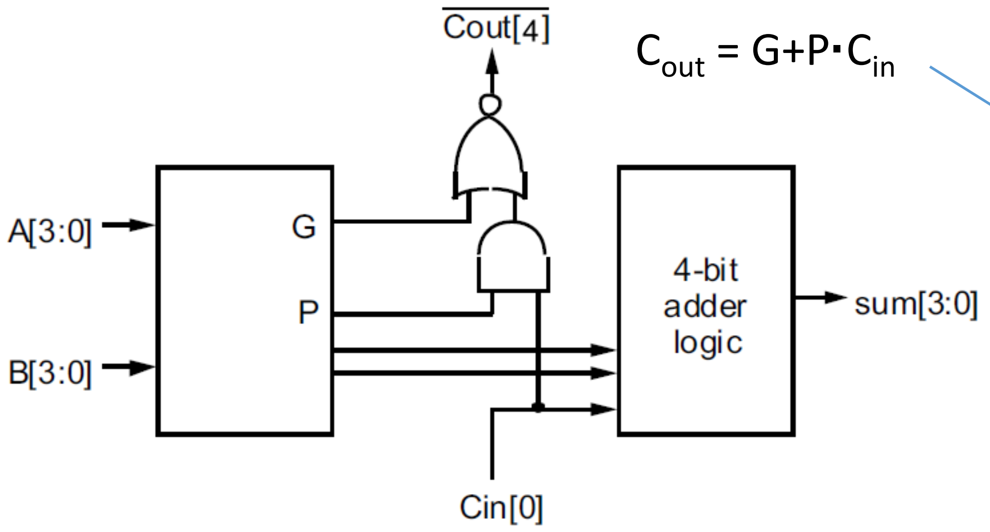
Ripple carry adder

- Problem of Ripple carry adder:
- Each cell causes a propagation delay
  - Cell for bit 1 cannot give a correct result until the cell for bit 0 has produced the carry output.
  - Cell for bit 2 has to wait for the carry from the bit 1 cell
- For an adder with many bits, the delays become very long.
  - 32 bits adder would only produce a valid result after 32 propagation delays.
  - E.g. 0xFFFFFFFF + 0x00000001

Carry Lookahead Adder

- For a full adder, define what happens to carry
  - Carry-generate: C_{out}=1 independent of C_{in}
    - **G = A·B**
  - Carry-propagate: C_{out}=C_{in}
    - **P = A $\oplus$ B** or **P = A + B**
- Fanout of G & P affect the overall delay → usually limited to 4 bits

A	B	G	P
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	x



$C_{out} = G+P·C_{in}$

4-bit carry look-ahead Adder

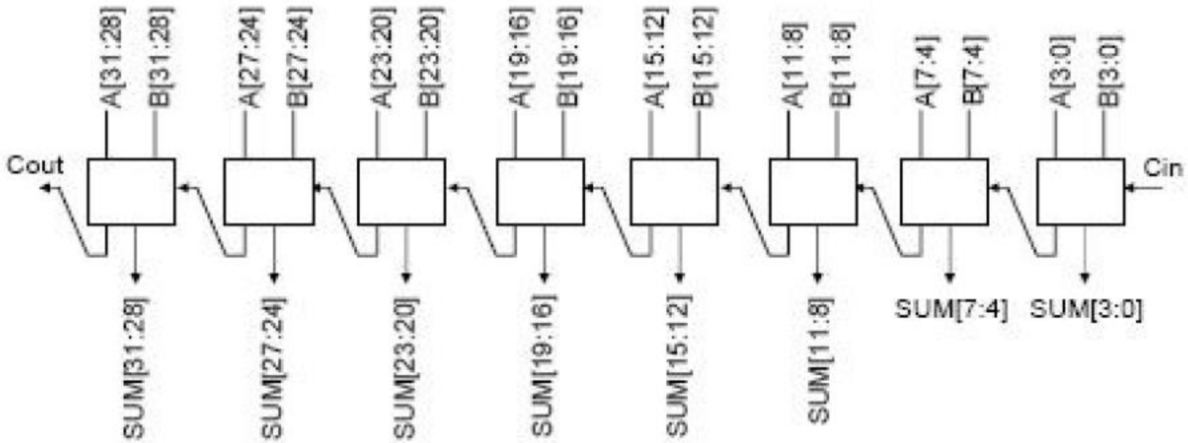
$P = P_3P_2P_1P_0$
$G = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$

C_0 =input carry,
$C_1 = G_0 + P_0C_0$,
$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0$,
$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$
$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3G_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 = G + PC_0$

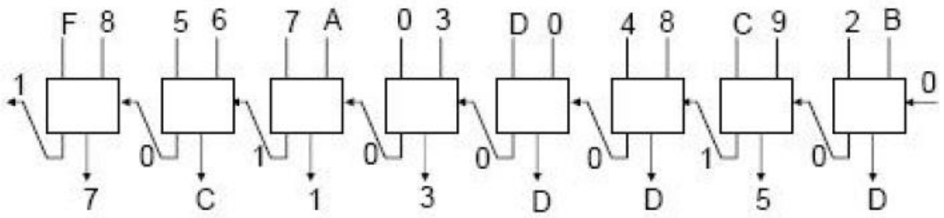
details was introduced in digital design course

Carry Lookahead Adder

- The critical carry path now has 8 propagation delays for a 32 bit adder.



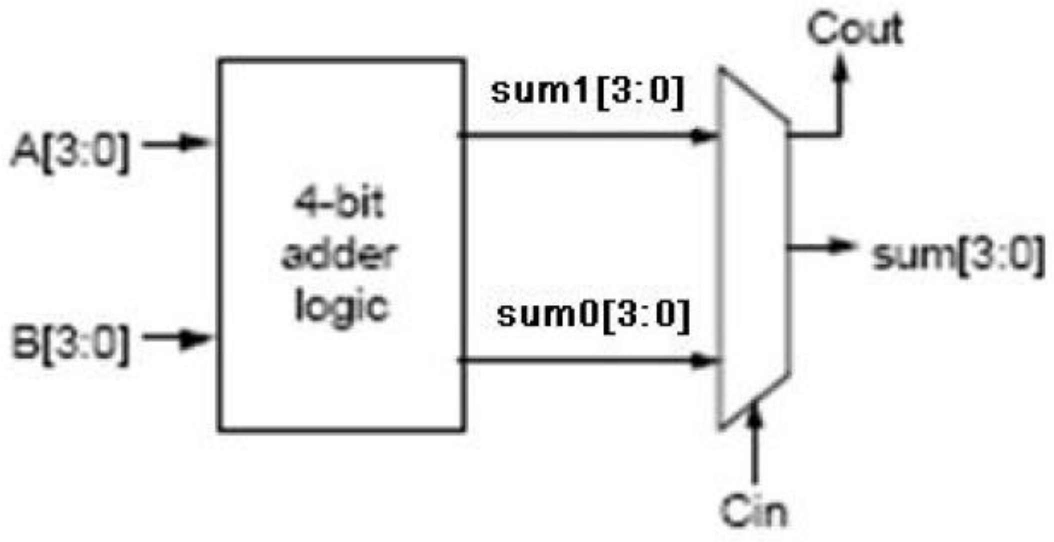
- Example: adding 0xF570D4C2 to 0x86A3089B (with C_{in} = 0):



  - The sum is 0x7C13DD5D with C_{out} = 1.

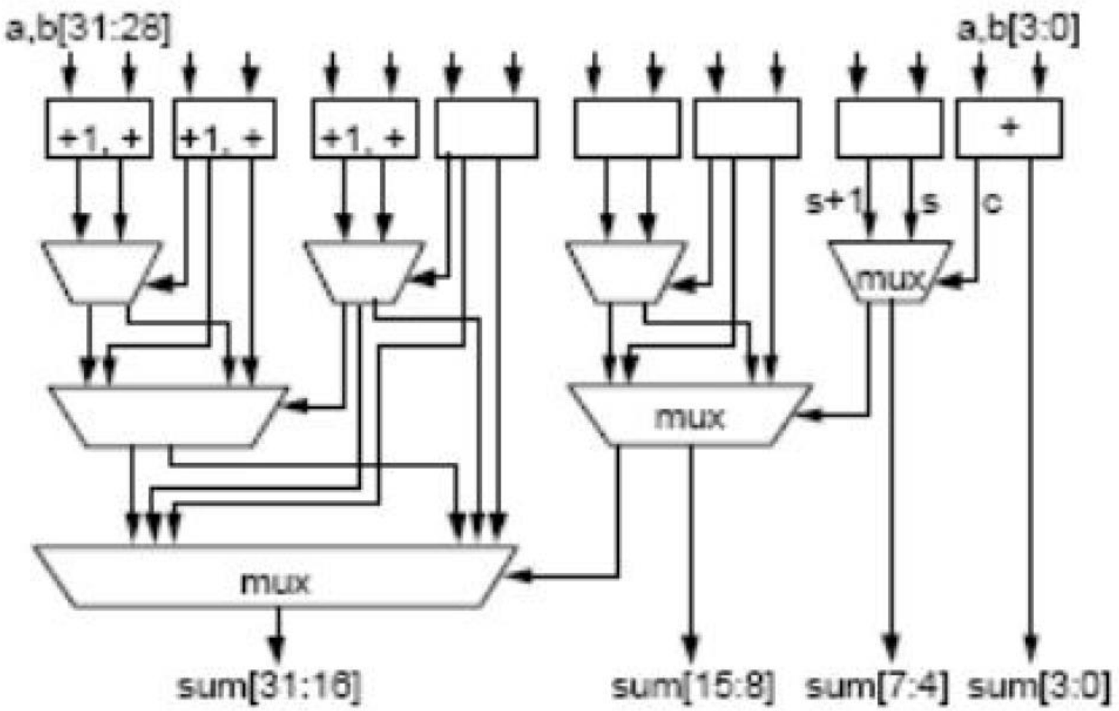
Carry select adder

- The propagation delay can be further reduced by using a ‘carry select’ scheme.
- The 4-bit adder logic produces two results; sum0 is simply (A+B) whereas sum1 is the sum; (A+B+1).
  - The carry-in is used to select one of these two results in a multiplexer
  - The output of the multiplexer is the sum and carry-out chosen by the value of the carry-in.



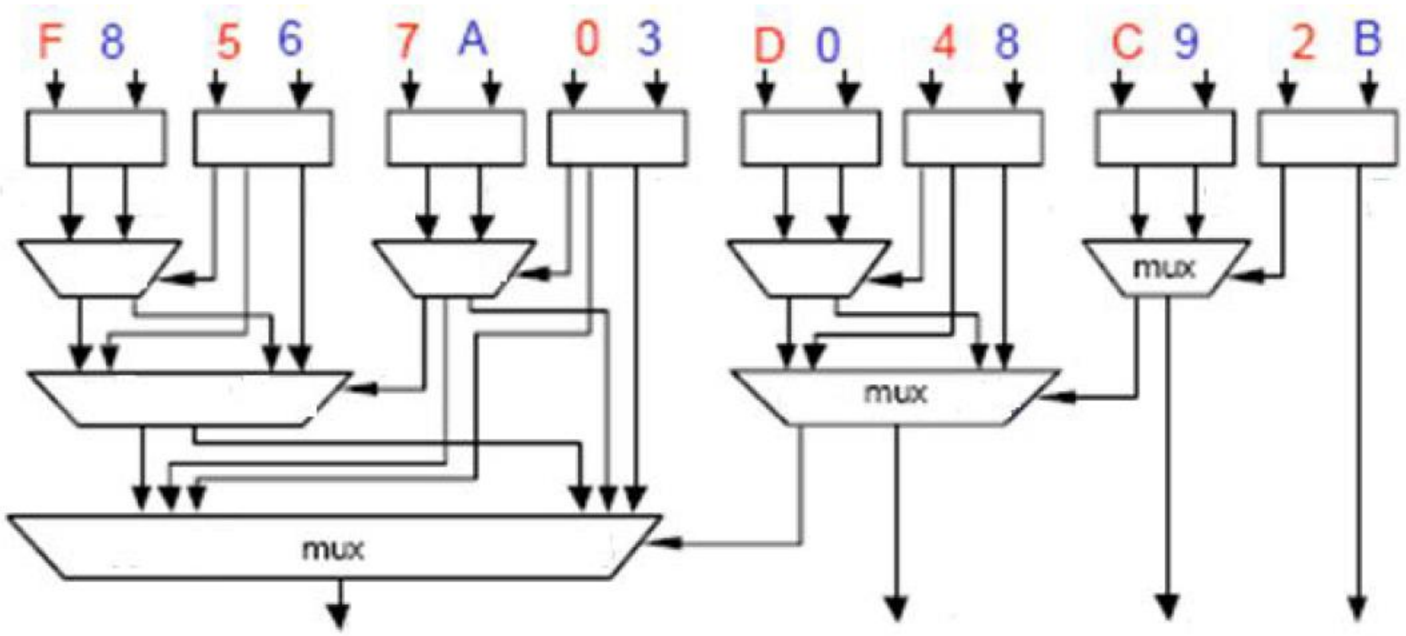
Carry select adder

- Compute sums of various fields of the word for carry-in of zero and carry-in of one
- Final result is selected by using the correct carry-in
- value to control a multiplexer
- For a 32 bit adder there are a maximum of 3 propagation delays in the carry path.



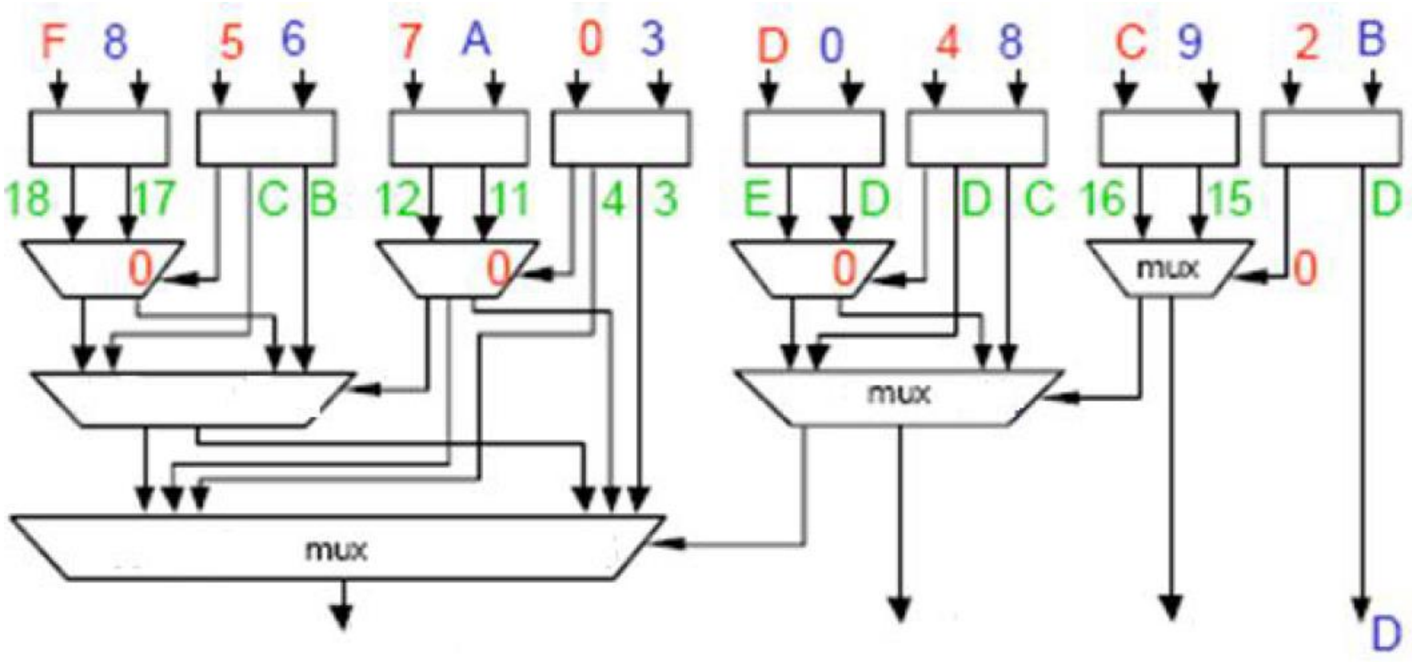
Carry select adder Example

- For example adding 0xF570D4C2 to 0x86A3089B:
  - The sum is 0x7C13DD5D with Cout = 1



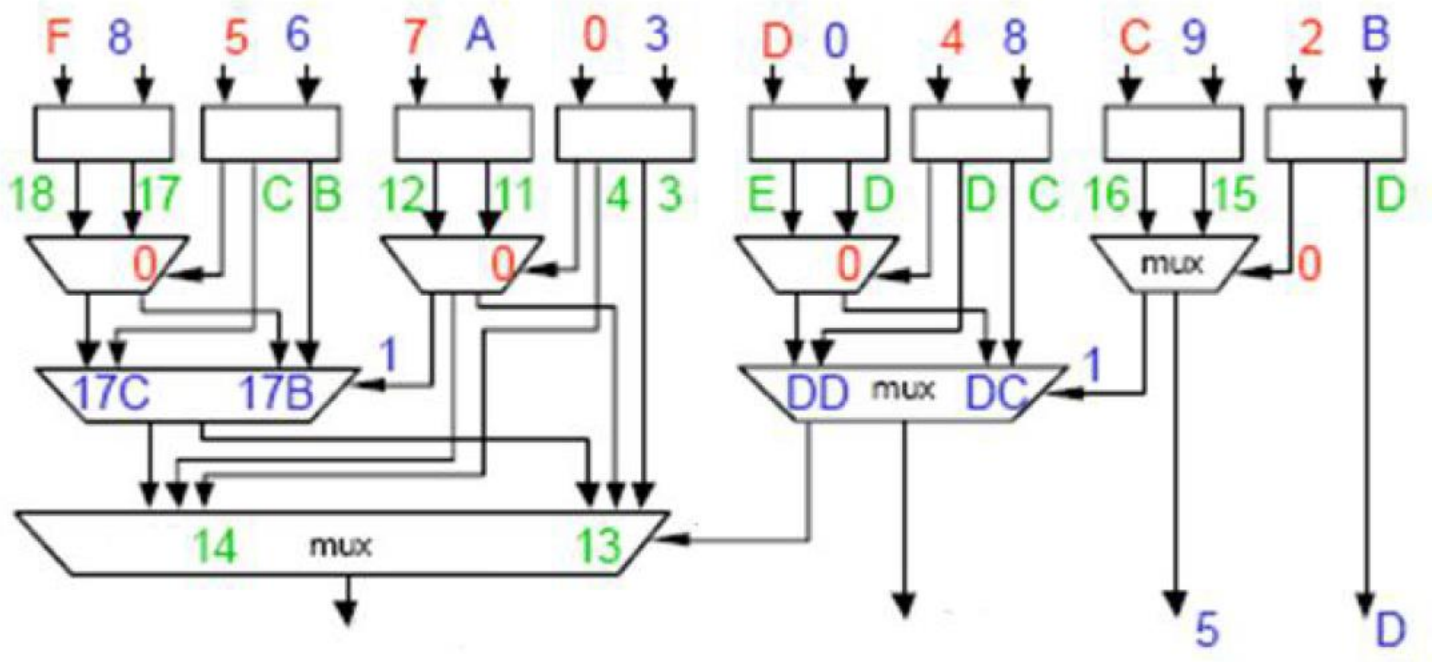
Carry select adder Example

- For example adding 0xF570D4C2 to 0x86A3089B:
  - The sum is 0x7C13DD5D with Cout = 1



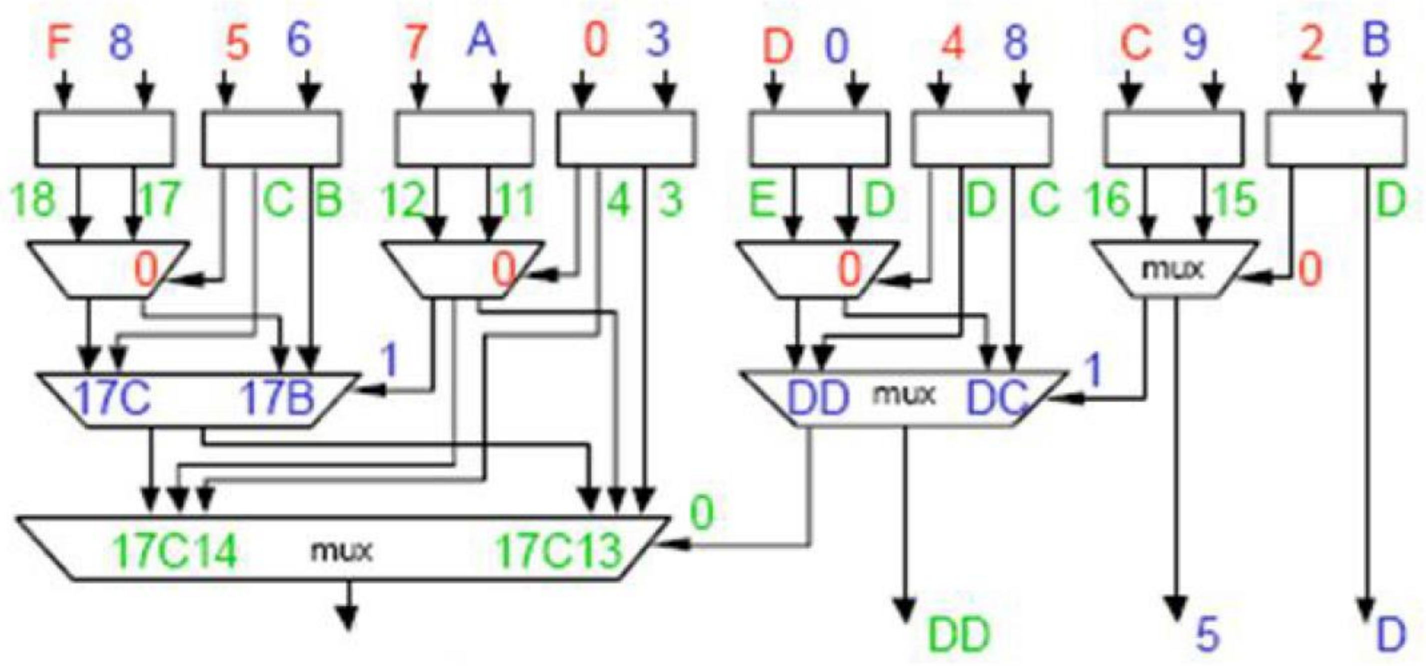
Carry select adder Example

- For example adding 0xF570D4C2 to 0x86A3089B:
  - The sum is 0x7C13DD5D with Cout = 1



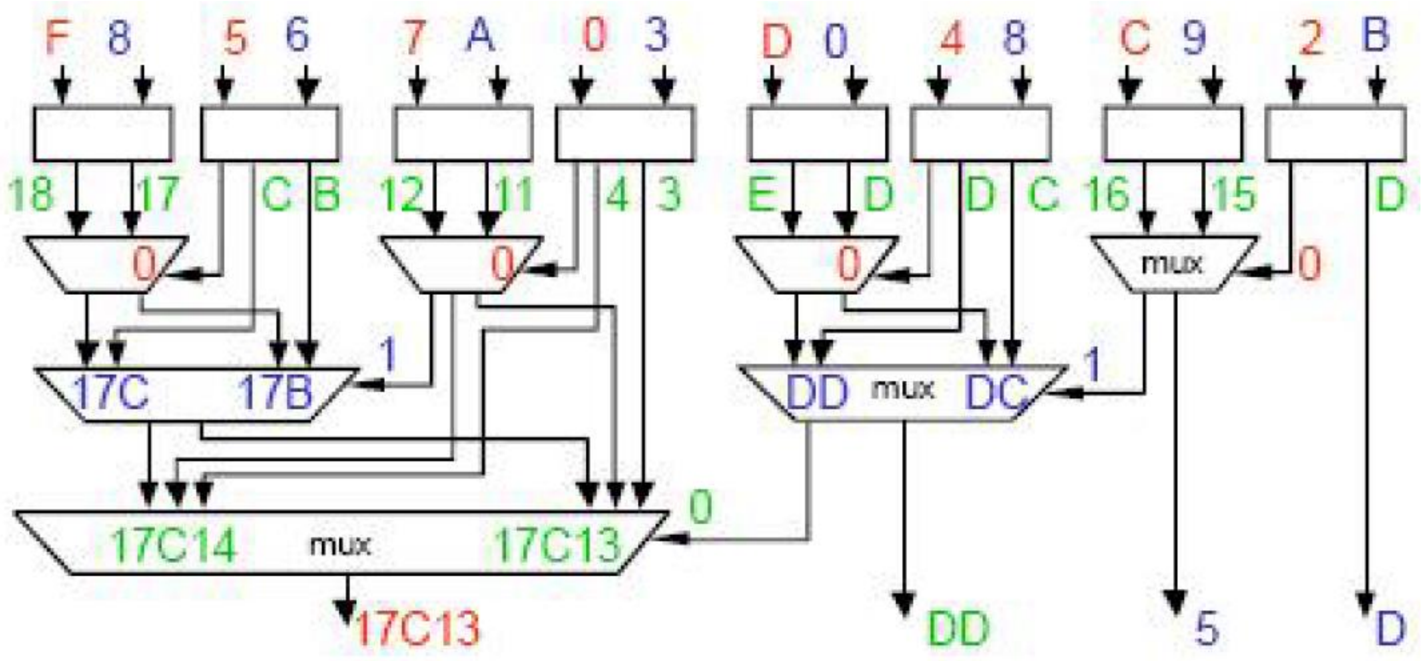
Carry select adder Example

- For example adding 0xF570D4C2 to 0x86A3089B:
  - The sum is 0x7C13DD5D with Cout = 1



Carry select adder Example

- For example adding 0xF570D4C2 to 0x86A3089B:
  - The sum is 0x7C13DD5D with Cout = 1



Performance comparison

- propagation delays on the critical carry path for the three different types of adder (assuming the carry look ahead and carry select adders use 4 bit adder blocks).

Size of adder	Ripple carry	Look ahead	Carry select
4 bits	4	1	1
8 bits	8	2	1
16 bits	16	4	2
32 bits	32	8	3
64 bits	64	16	4

Outline

- Barrel Shifter
- ALU Adder
- **Multiplier**

Multiplication circuit

- Add each partial product into a total as it is formed
  - PS: Partial sum
  - P: Product

				1	1	0	1		(13) Multiplicand M
			×	1	0	1	1		(11) Multiplier Q
				1	1	0	1		
			1	1	0	1			
		0	0	0	0				
	1	1	0	1					
1	0	0	0	1	1	1	1		(143) Product P

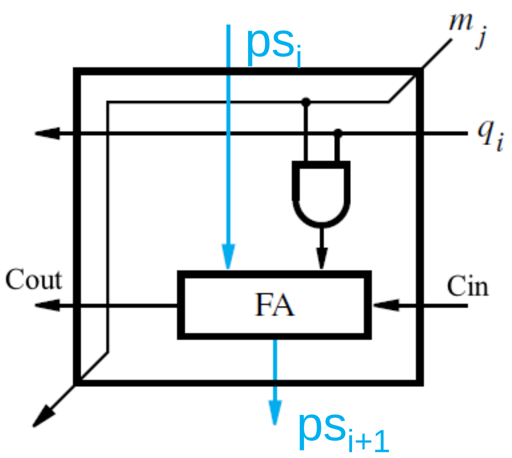
				m_3	m_2	m_1	m_0
				q_3	q_2	q_1	q_0
				m_3q_0	m_2q_0	m_1q_0	m_0q_0
			m_3q_1	m_2q_1	m_1q_1	m_0q_1	
			PS_1	PS_1	PS_1	P_1	P_0
		m_3q_2	m_2q_2	m_1q_2	m_0q_2		
		PS_2	PS_2	PS_2	P_2		
	m_3q_3	m_2q_3	m_1q_3	m_0q_3			
	PS_3	PS_3	PS_3	P_3			
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

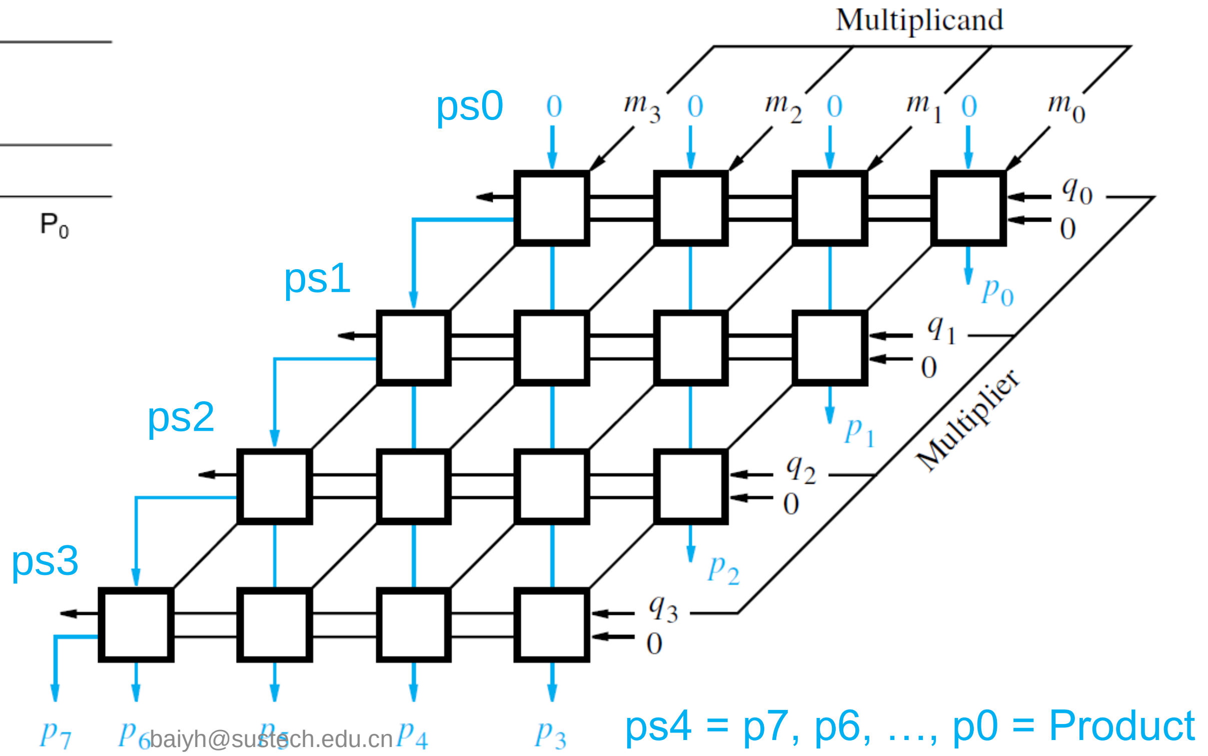
Matrix Multiplier

- Add each partial product into a total as it is formed

				m_3	m_2	m_1	m_0
				q_3	q_2	q_1	q_0
				m_3q_0	m_2q_0	m_1q_0	m_0q_0
			m_3q_1	m_2q_1	m_1q_1	m_0q_1	
			PS_1	PS_1	PS_1	P_1	P_0
		m_3q_2	m_2q_2	m_1q_2	m_0q_2		
		PS_2	PS_2	PS_2	P_2		
	m_3q_3	m_2q_3	m_1q_3	m_0q_3			
	PS_3	PS_3	PS_3	P_3			
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

Multiplicand: m
Multiplier: q
Partial sum: ps
Product: P





Carry Save Adder

- Speeding up multiplication is a matter of speeding up the summing of the partial products.
- Carry-save addition passes (saves) the carries to the output, rather than propagating them.
- With this technique, we can avoid carry propagation until final addition

Example: sum three numbers,

$3_{10} = 0011$, $2_{10} = 0010$, $3_{10} = 0011$

$$
\begin{array}{rrcl}
 & 3_{10}\ \ 0011 & & \\
+ & 2_{10}\ \ \underline{0010} & & \\
 & c\ \ 0100 & = & 4_{10} \\
 & s\ \ 0001 & = & 1_{10} \\
+ & 3_{10}\ \ \underline{0011} & & \\
 & c\ \ 0010 & = & 2_{10} \\
 & s\ \ \underline{0110} & = & 6_{10} \\
 & 1000 & = & 8_{10}
\end{array}
$$

carry-save add

carry-save add

carry-propagate add

Carry Save Adder

- Speeding up multiplication is a matter of speeding up the summing of the partial products.
- Carry-save addition passes (saves) the carries to the output, rather than propagating them.
- With this technique, we can avoid carry propagation until final addition

$$\begin{array}{ccccc} & 0 & 1 & 0 & 1 \\ & 0 & 1 & 1 & 0 \\ + & 1 & 0 & 0 & 1 \\ \hline \end{array}$$

```
e: 0101
f: 0110
g: 1001
calculate e+f+g
```

$$\begin{array}{ccccc} & 0 & 1 & 0 & 1 \\ + & 0 & {}_{1}1 & {}_{0}1 & {}_{0}0 \\ \hline & 1 & 0 & 1 & 1 \\ + & 1 & {}_{0}0 & {}_{1}0 & {}_{1}1 \\ \hline 1 & 0 & 1 & 0 & 0 \end{array}$$

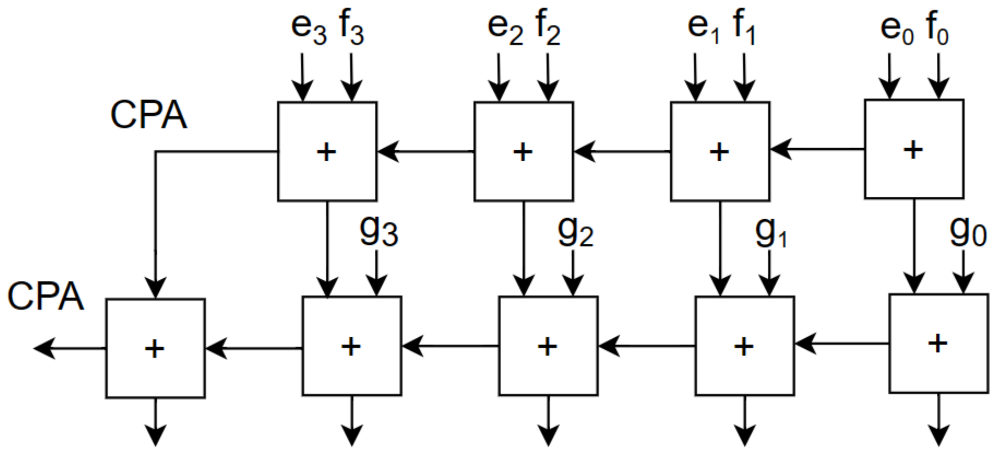
```
carry propagate
e+f, then f+g
```

$$\begin{array}{cccccc l} & & 0 & 1 & 0 & 1 & \\ & & 0 & 1 & 1 & 0 & \\ & + & 1 & 0 & 0 & 1 & \\ \hline & 0 & 1 & 0 & 1 & 0 & \text{carry} \\ + & 0 & 1 & 0 & 1 & 0 & \text{sum} \\ \hline & 1 & 0 & 1 & 0 & 0 & \end{array}$$

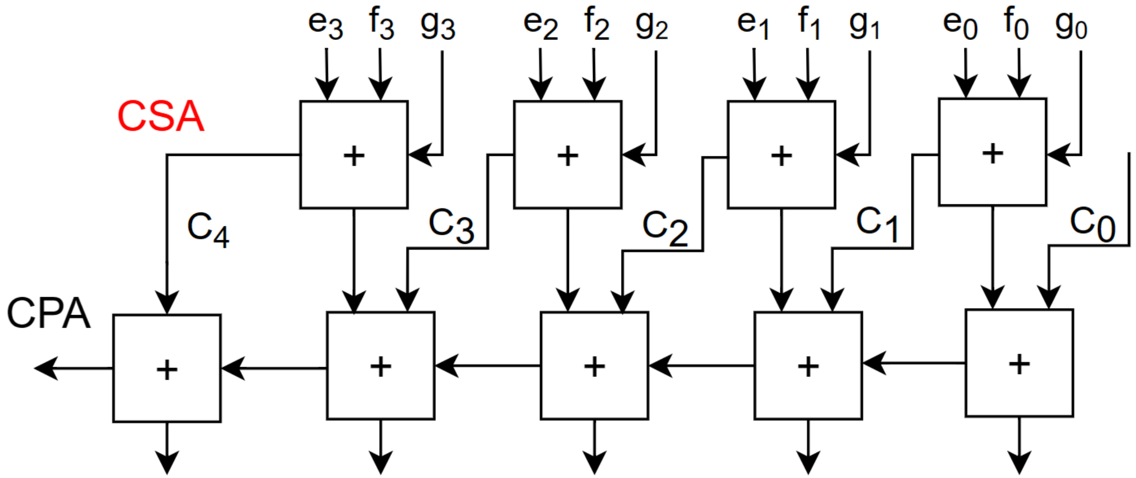
```
carry save
e+f+g = c+s
```

Carry Save Adder

```
      0   1   0   1
+     0 1 1 0 1 0 0
  ---------------------
      1   0   1   1
+     1 0 0 1 0 1 1
  ---------------------
  1   0   1   0   0
```




```
          0   1   0   1
          0   1   1   0
+         1   0   0   1
  -------------------------
      0   1   0   1   0   carry
+     0   1   0   1   0   sum
  -------------------------
      1   0   1   0   0
```

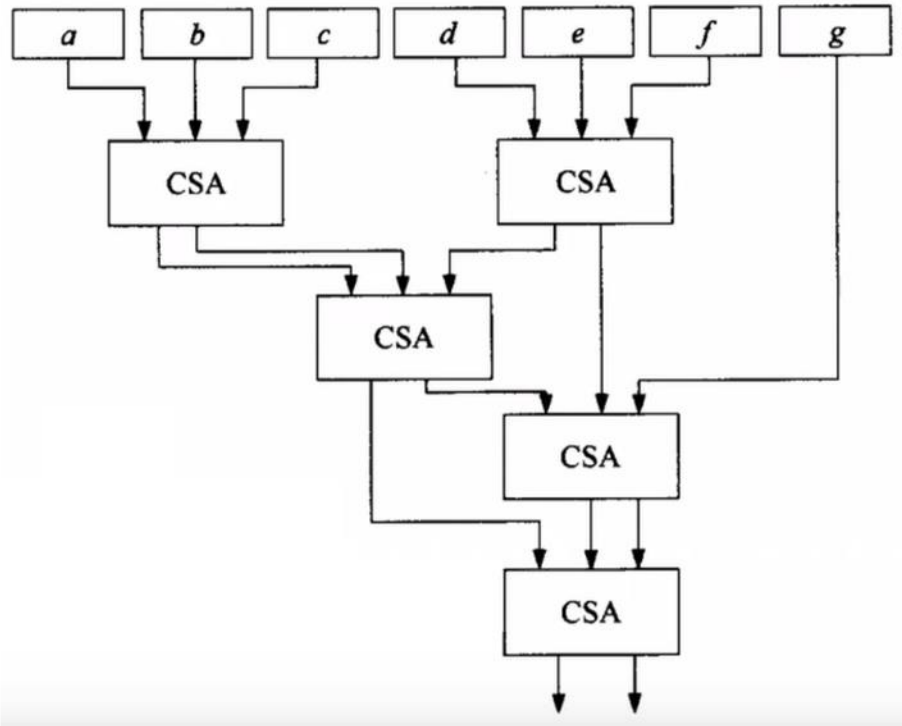


baiyh@sustech.edu.cn

Multiply with Carry Save

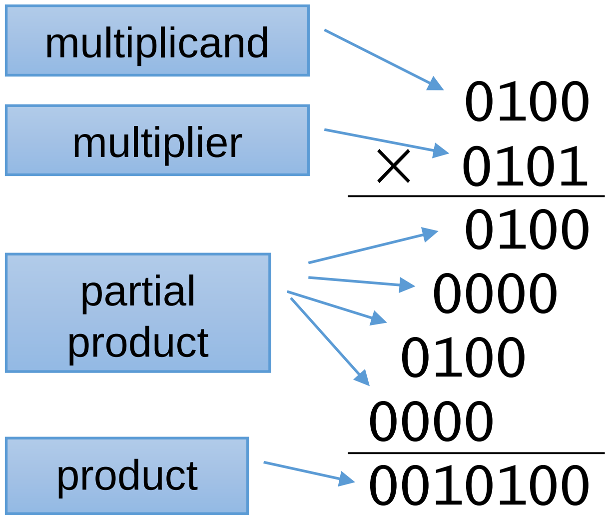
- When adding sets of numbers, carry-save can be used on all but the final sum.
- Standard adder (carry propagate) is used for final sum.
- Carry-save is fast (no carry propagation) and inexpensive (full adders)

						1	0	1	0	1	0	1	M
					x	1	1	1	1	1	1	1	Q
						1	0	1	0	1	0	1	a
					1	0	1	0	1	0	1		b
				1	0	1	0	1	0	1			c
			1	0	1	0	1	0	1				d
		1	0	1	0	1	0	1					e
	1	0	1	0	1	0	1						f
1	0	1	0	1	0	1							g



Sequential Multiplier

- This has used **sequential** actions to perform an operation that is essentially **combinational.**
- It uses mostly existing circuits, a shifter and adder, so does not add much to the gate count of the ALU.
  - In each step, one bit of the multiplier is selected
  - If the bit is logic 1, the multiplicand is shifted left to form a partial product, and it's added to the partial sum
- For unsigned multiplication. Sign bits are evaluated separately

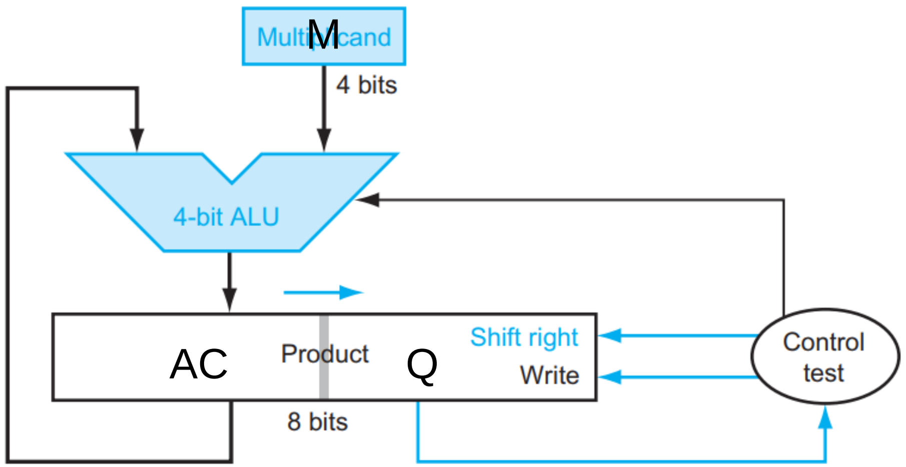


Length of product is the sum of operand lengths

Multiplier Example

- Multiply 2_{ten} (0010_{two}) by 7_{ten} (0111_{two}) :
  - M: multiplicand
  - AC: Accumulator
  - Q: multiplier
  - Final {AC, Q} will be the product

iter	M	AC	Q	Operation
ini	0010	0000	0111	
1	0010	0010	0111	**1**: AC = AC + M
	0010	0001	0011	Shift right {AC, Q}
2	0010	0011	0011	**1**: AC = AC + M
	0010	0001	1001	Shift right {AC, Q}
3	0010	0011	1001	**1**: AC = AC + M
	0010	0001	1100	Shift right {AC, Q}
4	0010	0000	1110	**0**: Shift right {AC, Q}
		res=00001110		done



Booth's multiplication algorithm

- calculate 0101101 × 0011110 (45 × 30) using normal and booth multiplication
  - multiplier: 30 = 32 – 2 → 0011110 = 0100000 - 0000010
  - **equivalent multiplier** : 0 +1 0 0 0 -1 0
- Need sign extension for complement

							0	1	0	1	1	0	1
							0	0	+1	+1	+1	+1	0
							0	0	0	0	0	0	0
						0	1	0	1	1	0	1	
					0	1	0	1	1	0	1		
				0	1	0	1	1	0	1			
			0	1	0	1	1	0	1				
		0	0	0	0	0	0	0					
	0	0	0	0	0	0	0						
0	0	0	1	0	1	0	1	0	0	0	1	1	0

normal multiplication

							0	1	0	1	1	0	1
							0	+1	0	0	0	−1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	0	1	0	0	1	1	
0	0	0	0	0	0	0	0	0	0	0	0		
0	0	0	0	0	0	0	0	0	0	0			
0	0	0	0	0	0	0	0	0	0				
0	0	0	1	0	1	1	0	1					
0	0	0	0	0	0	0	0						
0	0	0	1	0	1	0	1	0	0	0	1	1	0

booth multiplication

Booth's algorithm

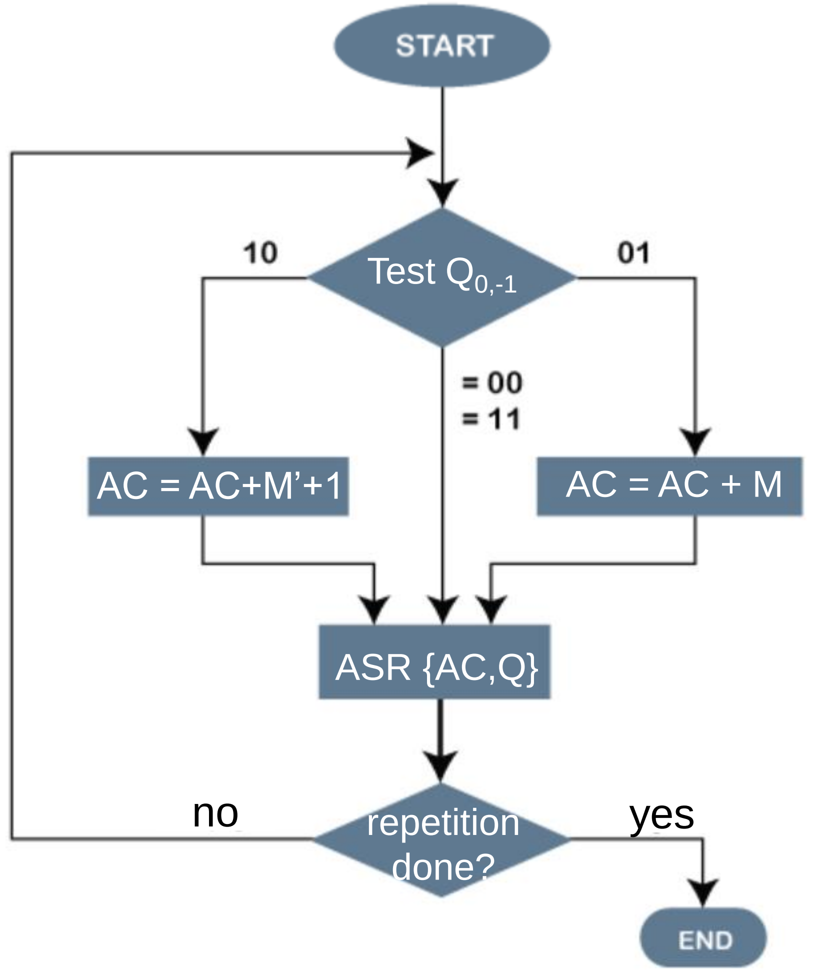
Current bit	Bit to the right	Equivalent bit (at current position)	Sequence example	Operation
0	0	0	0000111100**0**0	+0
0	1	+1	0000**1**111000	+M
1	0	-1	00001111**0**00	-M
1	1	0	000011**1**1000	+0

- Based on the current and previous bits, do one of the following
  - 00: no arithmetic operation.
  - 01: add the multiplicand to the left half of the product
  - 10: subtract the multiplicand from the left half of the product.
  - 11: no arithmetic operation.
- As in the previous algorithm, shift the product register right 1 bit

Booth's algorithm

- M: multiplicand
- AC: Accumulator
- Q: multiplier
- ASR: arithmetic shift right (sign extension)
- Final {AC, Q} will be the product

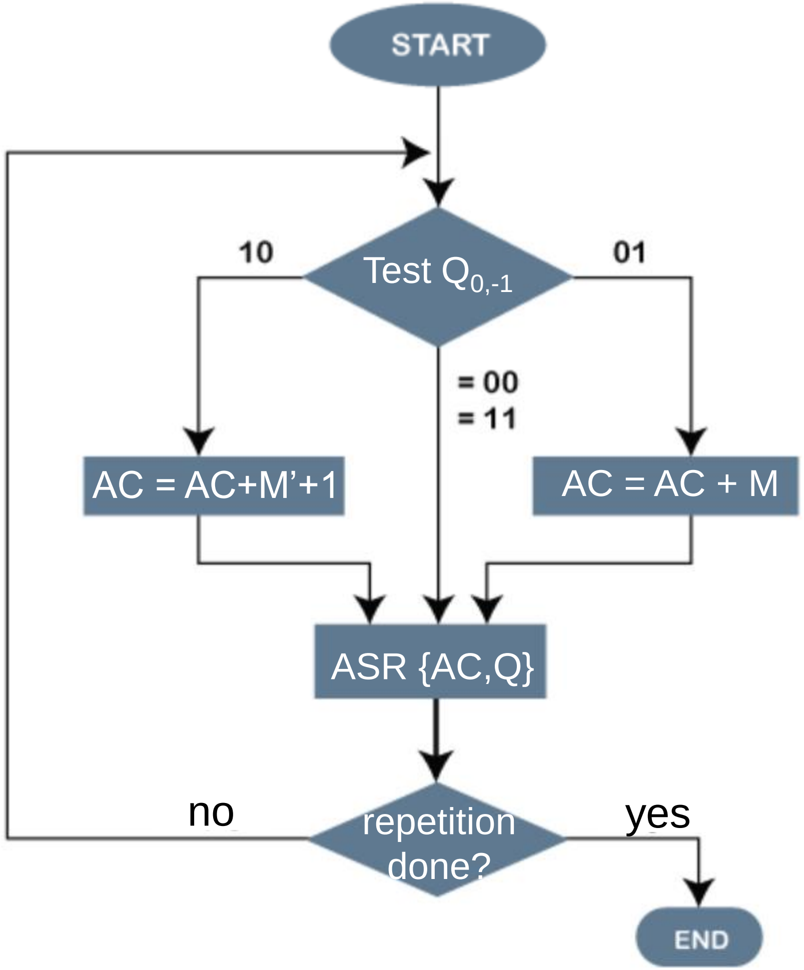
Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$



Booth's algorithm Example

- Booth's multiplier Example for signed value:
  - 0010×0111 = 00001110 (2×7 = 14)

iter	M	AC	Q	Q_{-1}	Operation
ini	0010	0000	0111	0	
1	0010	1110	0111	0	**10**: AC = AC + M' + 1
	0010	1111	0011	1	ASR {AC, Q}
2	0010	1111	1001	1	**11**: ASR {AC, Q}
3	0010	1111	1100	1	**11**: ASR {AC, Q}
4	0010	0001	1100	1	**01**: AC = AC + M
	0010	0000	1110	0	ASR {AC, Q}
		res=00001110			done



Booth’s algorithm for negative value

- Calculate 01101×11010 using Booth algorithm
  - the multiplier is equivalent to

	0	1	1	0	1		(+13)
×	1	1	0	1	0	0	(−6)

⟹

					0	1	1	0	1	
					0	−1	+1	−1	0	equivalent multiplier
0	0	0	0	0	0	0	0	0	0	
1	1	1	1	1	0	0	1	1		
0	0	0	0	1	1	0	1			
1	1	1	0	0	1	1				
0	0	0	0	0	0					
1	1	1	0	1	1	0	0	1	0	(−78)

equivalent multiplier: 0 -1 +1 -1 0

11010 = $-2^4 + 2^3 + 2^1 = -6$

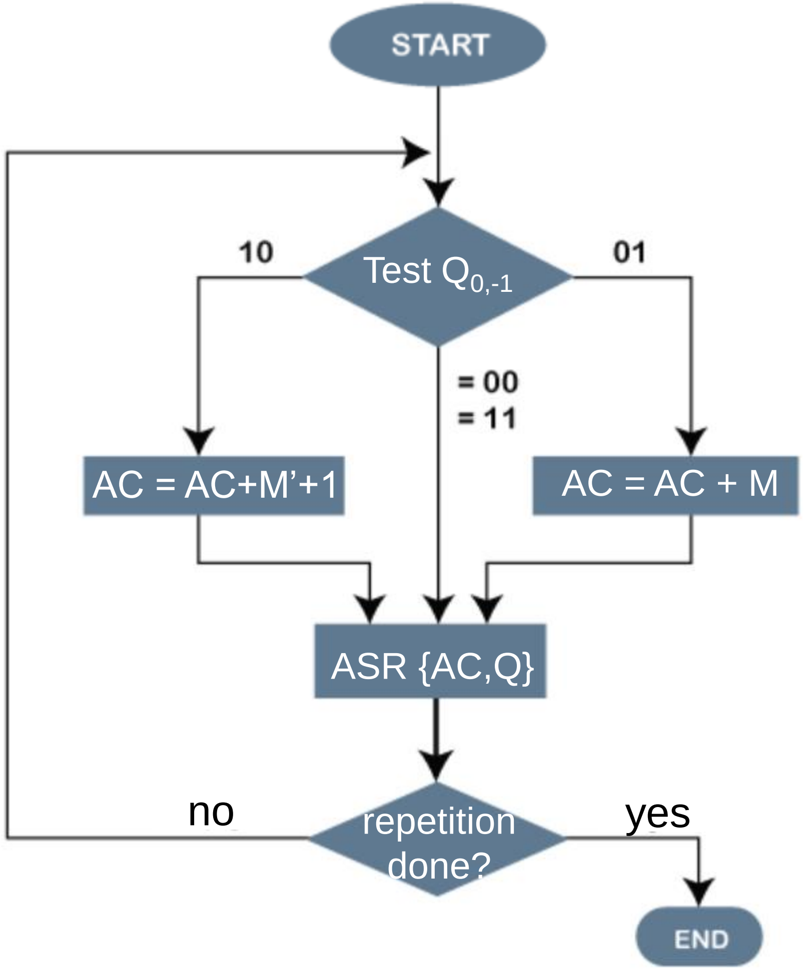
0 -1 +1 -1 0 = $-2^3 + 2^2 - 2^1 = -6$

Multiplier Bit i	Multiplier Bit $i-1$	Version of multiplicand selected by bit i
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Booth’s algorithm Example

- Booth’s multiplier Example for signed value:
  - 1011 × 1001 = 00100011 (-5 × -7 = 35)

iter	M	AC	Q	Q_{-1}	Operation
ini	1011	0000	1001	0	
1	1011	0101	1001	0	**10**: AC = AC + M’ + 1
	1011	0010	1100	1	ASR AC and Q
2	1011	1101	1100	1	**01**: AC = AC + M
	1011	1110	1110	0	ASR AC and Q
3	1011	1111	0111	0	**00**: ASR AC and Q
4	1011	0100	0111	0	**10**: AC = AC + M’ + 1
	1011	0010	0011	1	ASR AC and Q
		res=00100011			done



Booth’s algorithm performance

- Can perform negative number multiplication
- Sometimes worse than normal algorithm
- Thus, we use Booth2 algorithm

Worst-case multiplier	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
	⇓
	+1 −1 +1 −1 +1 −1 +1 −1 +1 −1 +1 −1 +1 −1 +1 −1
Ordinary multiplier	1 1 0 0 0 1 0 1 1 0 1 1 1 1 0 0
	⇓
	0 −1 0 0 +1 −1 +1 0 −1 +1 0 0 0 −1 0 0
Good multiplier	0 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1
	⇓
	0 0 0 +1 0 0 0 0 −1 0 0 0 +1 0 0 −1

Bit-pair Recoding (Booth2)

- bit-pair recoding of the multiplier results in using at most one summand for each pair of bits in the multiplier
  - +2M: left shift
  - -M: complement of M
  - -2M: complement and left shift

multiplier： 0 0 1 0 0 1 0
equivalent multiplier: +1 -2 +1

$Q_{i+1}Q_iQ_{i-1}$	Equivalent value (at position i)	Operation
000	0	+0
001	+1	+M
010	+1	+M
011	+2	+2M
100	-2	-2M
101	-1	-M
110	-1	-M
111	0	0

						0	0	0	1	1	1
x						0	0	1	0	0	1
							1		-2		1
0	0	0	0	0	0	0	0	0	1	1	1
1	1	1	1	1	1	0	0	1	0		
0	0	0	0	0	1	1	1				
0	0	0	0	0	0	1	1	1	1	1	1

- Using Booth2’s algorithm, a 32 bit multiplication can be completed in 16 cycles

baiyh@sustech.edu.cn

Bit-pair Recoding (Booth2)

- If multiplier has odd number of digit, signed extend it to even number of digits

original multiplier: 0 0 1 0 0 1 0

Basic booth multiplier 0 +1 -1 0 +1 -1

Booth 2 eqv. multiplier: +1 -2 +1

$Q_{i+1}Q_i$ Q_{i-1}	Booth		Equivalent value (at position i)	Booth2 Operation
000	0	0	0	+0
001	0	+1	+1	+M
010	+1	-1	+1	+M
011	+1	0	+2	+2M
100	-1	0	-2	-2M
101	-1	+1	-1	-M
110	0	-1	-1	-M
111	0	0	0	0

Booth2 Algorithm

- Advantage of Booth2's algorithm
  - Multiplication requiring only n/2 summands

$$
\begin{array}{rccccccccccl}
 & & & & & & 0 & 1 & 1 & 0 & 1 & (+13) \\
 & & & & & \times & 1 & 1 & 0 & 1 & 0 & (-6) \\
\hline
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{cccccccccc}
 & & & & & 0 & 1 & 1 & 0 & 1 \\
 & & & & & 0 & -1 & +1 & -1 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & & \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & & & \\
0 & 0 & 0 & 0 & 0 & 0 & & & & \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \quad (-78)
\end{array}
\quad \Rightarrow \quad
\begin{array}{cccccccccc}
 & & & & & 0 & 1 & 1 & 0 & 1 \\
 & & & & & 0 & & -1 & & -2 \\
\hline
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & & \\
0 & 0 & 0 & 0 & 0 & 0 & & & & \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0
\end{array}
$$

Booth | Booth2