Week 8 - T1 2020

Arithmetic Circuits

ELEC2141: Digital Circuit Design

UNSW SYDNEY

Overview

Arithmetic circuits

- Iterative combinational circuits

Binary adders

Unsigned binary addition

Signed binary addition

Overflow

Reading: Mano - Chapter 3: 3.8-3.11

Arithmetic circuits

Circuits that perform arithmetic operations are important part of digital design

These circuits from the basis for a microprocessor's ALU and commonly used in many other applications

Arithmetic circuits often designed to operate on binary input vectors and produce binary output vectors

Iterative combinational circuits

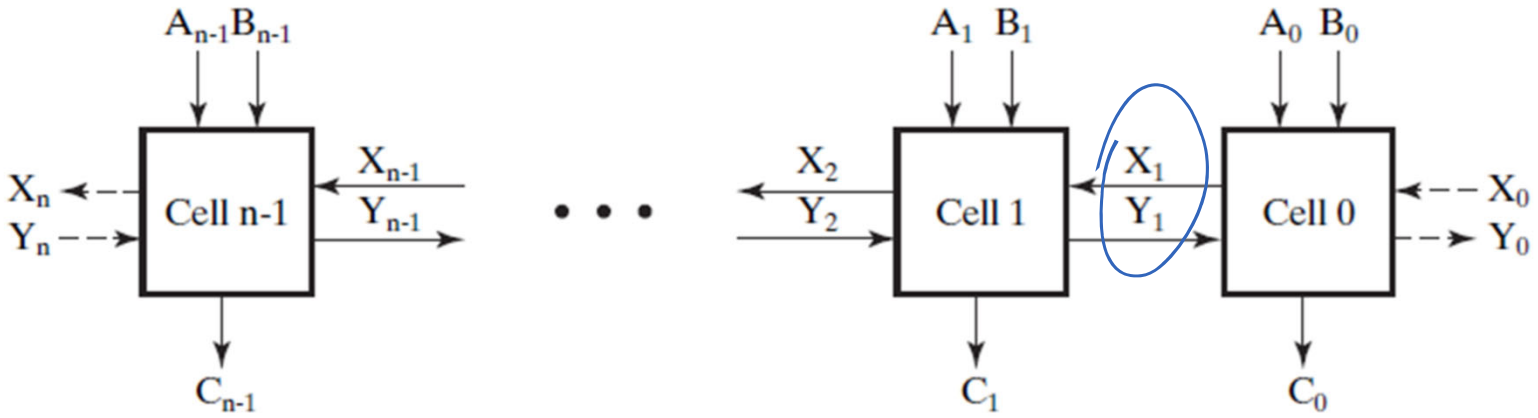
The function implemented requires that the same sub-function be applied to each bit position

Can design functional block for the sub-function and duplicate it to obtain functional block for an overall function

Each sub-function block is referred to as a *cell*

An array of interconnected cells forms an *iterative array*

Iterative combinational circuits



Example: *n* = 32 means 64 inputs and 32 outputs

2^{64} truth table rows – impractical!

Iterative array takes advantage of the regularity to make the design feasible

UNSW SYDNEY

Binary adders

Binary addition is used frequently

Similar to decimal addition but with two possible digits: 0 and 1

Example – 8-bit addition:

$$\begin{array}{r} {\scriptstyle 1} \\ 77 \\ +\ 86 \\ \hline 163 \end{array} \qquad \begin{array}{r} {\scriptstyle 1\ \ 1\ 1\ 1\ 1\ 1\ 1\ 1} \\ 01001101 \\ +\ 01010110 \\ \hline (10100011)_2 \end{array}$$

Half adder

A *half adder* generates the sum of two input bits

The two inputs are represented by X and Y

The sum is expressed as a *sum* (S) and a *carry* (C)

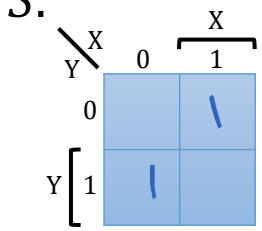
X	0	1	0	1
+ Y	+ 0	+ 0	+ 1	+ 1
C S	0 0	0 1	0 1	1 0

Half adder

The half adder can be specified as a truth table and the output equations can be derived

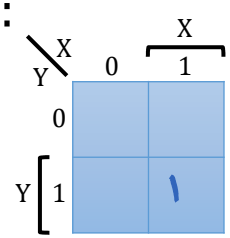
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S:



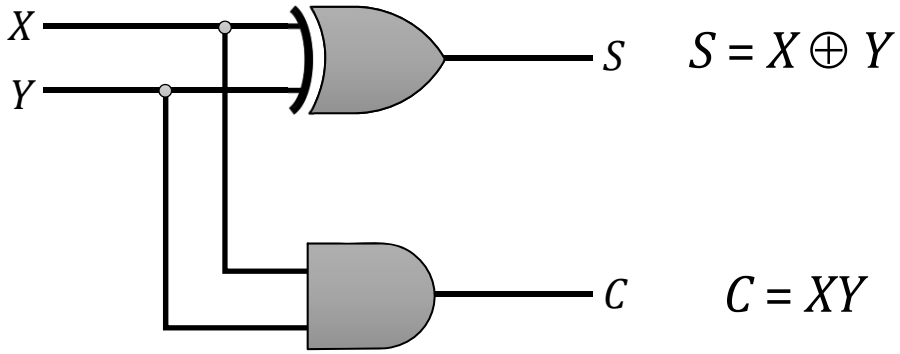
$S = \bar{X}Y + X\bar{Y}$
$= X \oplus Y$

C:



$C = XY$

The logic diagram for the half adder



$S = X \oplus Y$

$C = XY$

UNSW SYDNEY

Full adder

A full adder generates the sum of three input bits

Two inputs (X and Y) represent the two significant bits to be added

The third input (Z) represents the carry from the previous significant bit

Two outputs represent the *sum* (S) and *carry* (C)

Z	0	0	0	0	1	1	1	1
X	0	0	1	1	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0	0 1	1 0	1 0	1 1

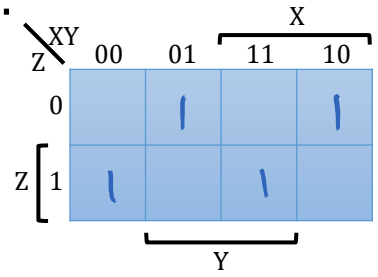
$X + Y = XY + X \oplus Y$

Full adder

A Full Adder can be specified as a truth table and the output equations can be derived

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S:

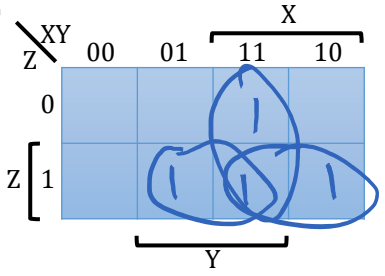


$S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + XYZ + X\bar{Y}\bar{Z}$

$= \bar{X}(\bar{Y}Z + Y\bar{Z}) + X(YZ + \bar{Y}\bar{Z})$

$= X \oplus Y \oplus Z$

C:



$C = XY + YZ + XZ$

$= XY + Z(X + Y)$

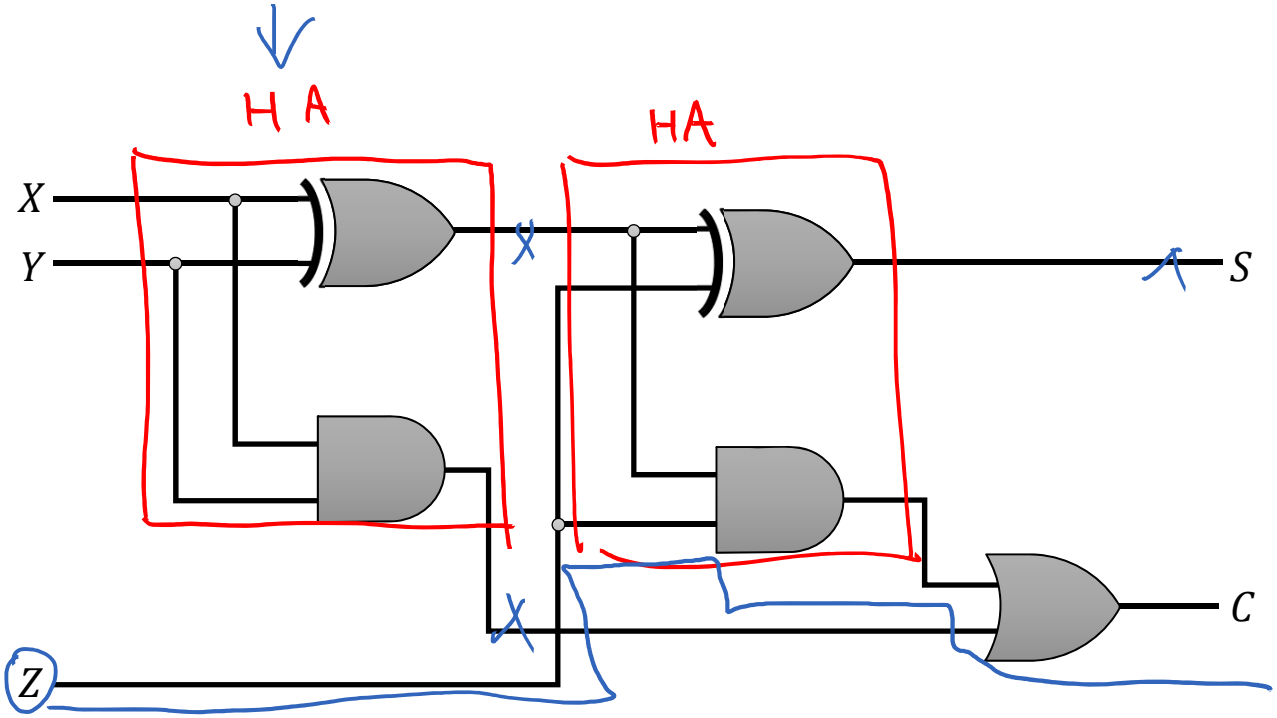
$= XY + Z(XY + X \oplus Y)$

$= XY(Z + \bar{Z}) + Z(XY) + Z(X \oplus Y)$

$= XY + Z(X \oplus Y)$

UNSW SYDNEY

Full adder logic diagram



$$S = X \oplus Y \oplus Z$$

$$C = XY + Z(X \oplus Y)$$

UNSW SYDNEY

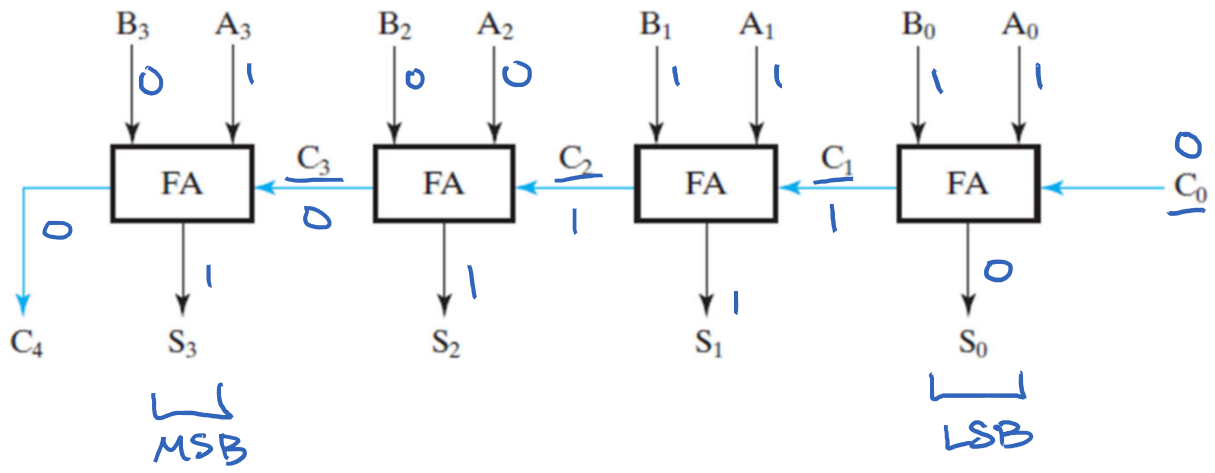
Binary ripple-carry adder

An *n*-bit parallel adder formed by cascading *n* full adders

Apply all *n*-bit inputs simultaneously

Connect the carry output from one full adder to the carry input of the next

The carry propagate through, from the LSB to the MSB



C_i	0	0	1	1	0
A_i		1	0	1	1
B_i	+	0	0	1	1
S_i		1	1	1	0
C_{i+1}		0	0	1	1

Binary ripple-carry adder

Ripple-carry adders have long circuit delays

The carry has to propagate through many gates until the final result is obtained

Each of the *n* full-adders introduce two gate-delays in the carry path

Example: for a 16-bit adder, the delay is 32 gate delays (+ overhead)

Binary carry lookahead adder

Define: Carry *Generate*

$$G_i = A_i B_i$$

Must generate carry when $A = B = 1$

Define: Carry *Propagate*

$$P_i = A_i \oplus B_i$$

The carry-out will equal carry-in

C_i	0	1	0	0	0
A_i		0	1	0	0
B_i	+	0	1	0	0
S_i		1	0	0	0
C_{i+1}		0	1	0	0

Express the sum (S) and carry (C) in terms of generate/propagate:

$$S_i = (A_i \oplus B_i) \oplus C_i$$
$$= P_i \oplus C_i$$
$$C_{i+1} = A_i B_i + C_i(A_i \oplus B_i)$$
$$= G_i + C_i P_i$$

C_i	0	1	1	0	0
A_i		0	0	1	0
B_i	+	0	1	1	0
S_i		1	0	0	0
C_{i+1}		0	1	1	0

Binary carry lookahead adder

Re-expressing the carry equations

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

All carry bits can be calculated straight from the inputs

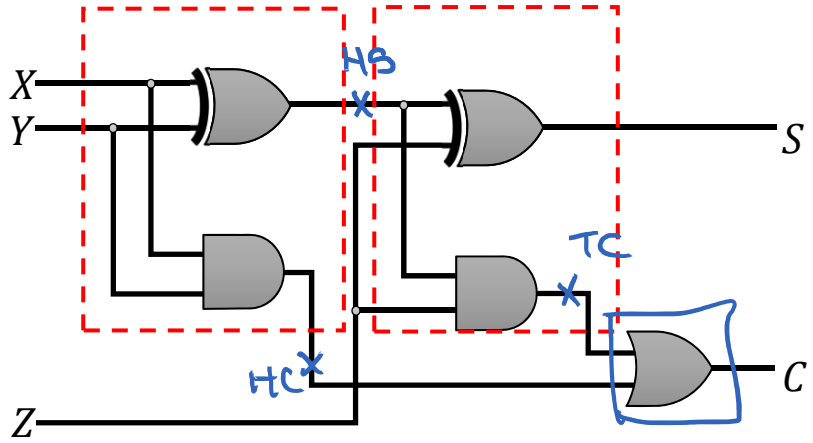
3 gate-delays regardless of number of bits, at the cost of more complex logic

In practice, gates have limited number of inputs

For larger adders, can cascade 4-bit CLAs and connect to a group lookahead carry unit

Dataflow Description of 4-Bit Adder

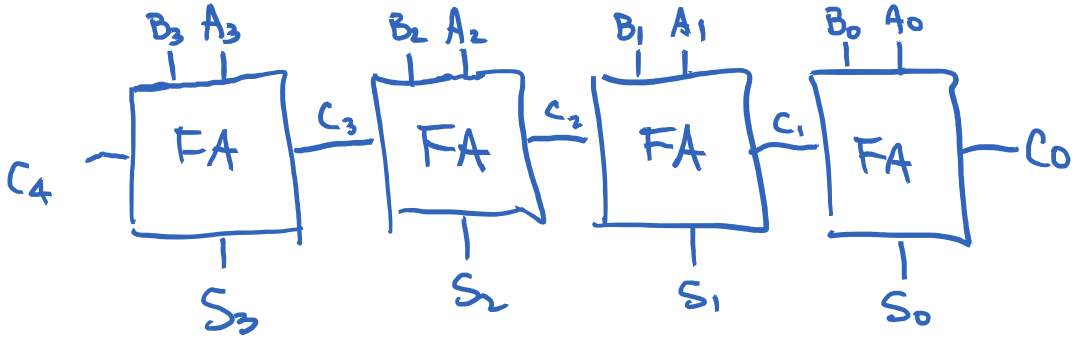
```
// 4-bit ripple carry adder: dataflow Verilog description
// based on logic diagram earlier in slides

module   half_adder(X, Y, C, S);
      input   X, Y;
      output S, C;

      assign S = X ^ Y;
      assign C = X & Y;

endmodule

module   full_adder(X, Y, Z, C, S);
      input   X, Y, Z;
      output S, C;
      wire HS, HC, TC;

   - half_adder HA1(X, Y, HC, HS);
   - half_adder HA1(HS, Z, TC, S);

   - assign C = TC|HC;

endmodule
```



UNSW SYDNEY

Dataflow Description of 4-Bit Adder

```
// continued from previous slide

module  adder_4bit(A, B, C0, S, C4);

    input   [3:0]   A, B;   // A3A2A1A0 , B3B2B1B0
    input   C0;             // 1st stage carry in
    output [3:0]  S;        // sum
    output C4;              // carry out last stage

    wire [3:1]  C;

    full_adder Bit0(A[0], B[0], C0, C[1], S[0]);
    full_adder Bit1(A[1], B[1], C[1], C[2], S[1]);
    full_adder Bit2(A[2], B[2], C[2], C[3], S[2]);
    full_adder Bit3(A[3], B[3], C[3], C4, S[3]);

endmodule
```



UNSW SYDNEY

Behavioural description of 4-bit adder

```
// 4-bit full adder: behavioural Verilog description

module   adder_4_bit(A, B, C0, S, C4);

    input   [3:0]   A, B;
    input   C0;
    output [3:0]  S;
    output C4;

    assign {C4, S} = A + B + C0;

endmodule
```

5 bit

Unsigned binary subtraction

Subtraction involves comparing the subtrahend with the minuend and subtracting the smaller from the larger

If the minuend is larger than the subtrahend, then the result is a positive number; otherwise, negative

Subtraction with comparison is inefficient as comparison operation results in costly circuitry

As an alternative, we can simply subtract the subtrahend from the minuend and correct the result if negative

$$\begin{array}{r} M \\ -\,N \\ \hline \end{array} \qquad \begin{array}{l} M > N \quad +ve \\ M < N \quad -ve \end{array}$$

Unsigned binary subtraction

Example

$$\begin{array}{r} 30 \\ -\ 19 \\ \hline 11 \end{array} \qquad \begin{array}{r} 119 \\ -\ 30 \\ \hline 89 \end{array} \qquad \begin{array}{r} 100 \\ 89 \\ \hline -11 \end{array}$$

30 → 11110
19 → −10011
11 → 01011

19 → 10011
30 → −11110
21 → 10101

If no borrow occurs into the most significant position, then we know that the subtrahend is not larger than the minuend and the result is positive and correct

If a borrow does occur into the most significant position, then we know the subtrahend is larger than the minuend. The result must then be negative, and so needs correction

Unsigned binary subtraction

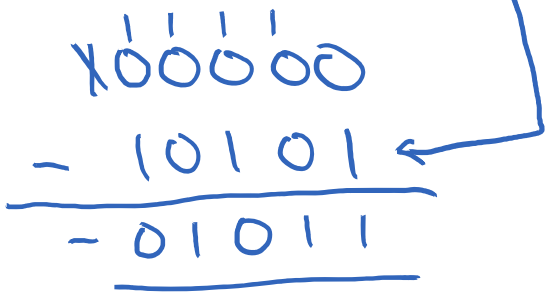
The result when a borrow occurs is

$$M - N + 2^n$$

Instead the result we desire is $\mathbf{N - M}$ in magnitude. The correct result can be obtained by

$$2^n - (M - N + 2^n) = \mathbf{N - M}$$

In the previous example, the correct magnitude is 100000 – 10101 = 01011



UNSW SYDNEY

Unsigned binary subtraction

In general, the subtraction of two n-digit numbers, $M - N$, in base 2 can be done as follows:

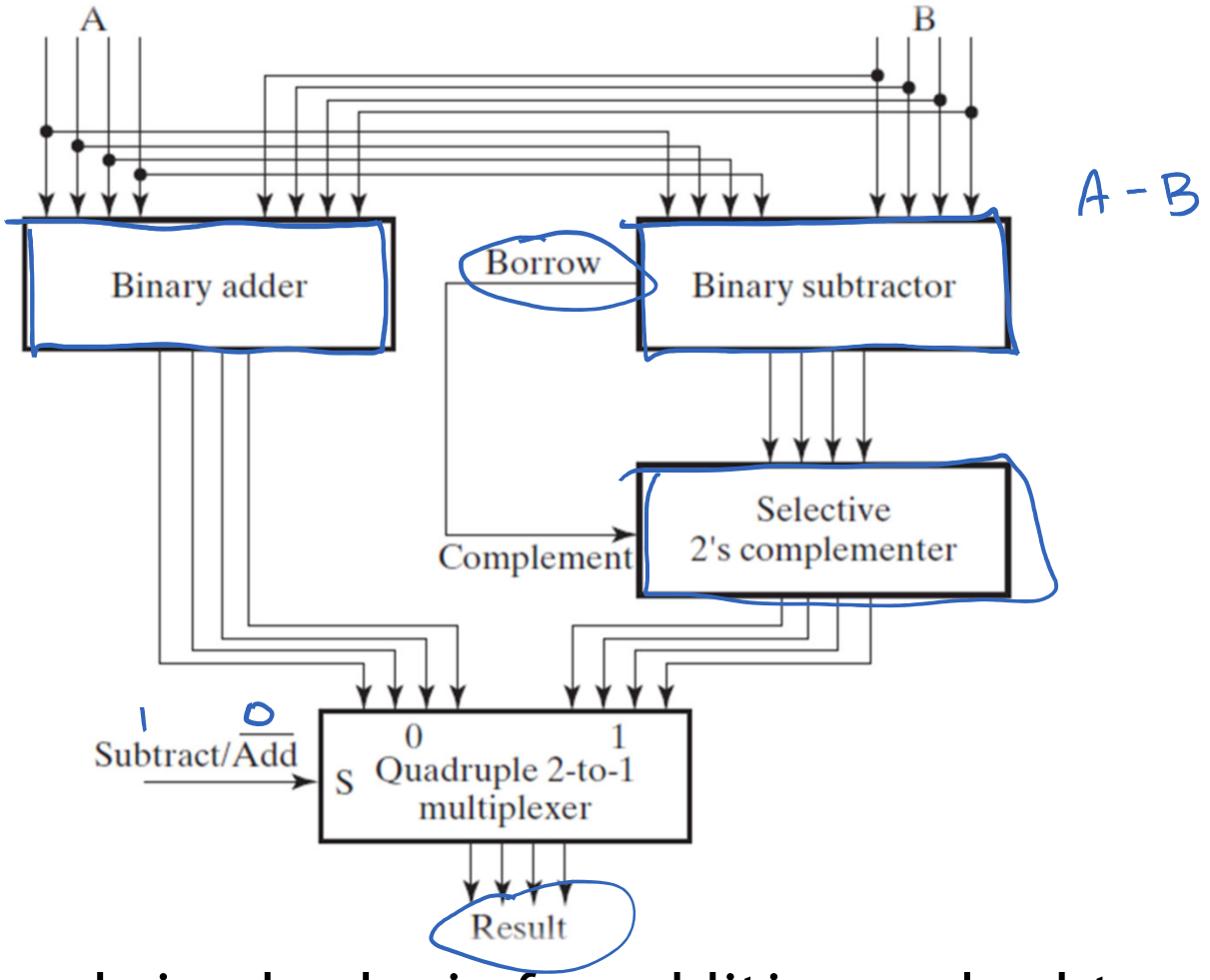
1. Subtract the subtrahend N from the minuend M
2. If no end borrow occurs, then $M \geq N$, and the result is nonnegative and correct
3. If an end borrow occurs, then $N > M$, and the difference, $M - N + 2^n$, is subtracted from 2^n, and a minus sign is appended to the result

$2^n - (M - N + 2^n) = N - M$

The subtraction $2^n - N$ is called taking the 2's complement of N

To do both unsigned addition and unsigned subtraction requires a complex circuit (next slide)

Unsigned binary subtraction



Need shared simpler logic for addition and subtraction

Complements are used to achieve that

UNSW SYDNEY

Complements

For any radix r, the *Diminished Radix Complement* – called $(r - 1)$'s complement for radix r – is defined as $(r^n - 1) - N$, for some number N

The *Radix Complement* – called r's complement for radix r is defined as $r^n - N$, for some number N

Example

$$r = 2,\ N = 0111\ 0011,\ n = 8$$

$$r^n - 1 = 100000000 - 1 = 11111111$$

The 1's complement of N can be easily obtained by inverting all the bits in N

$$\begin{array}{r} 1\,1\,1\,1\,1\,1\,1\,1 \\ -\,0\,1\,1\,1\,0\,0\,1\,1 \\ \hline 1\,0\,0\,0\,1\,1\,0\,0 \end{array}$$

Complements

Example

$$r = 2,\ N = 0111\ 0011,\ n = 8$$

$$r^n = 100000000$$

So the 2's complement of N

$$\begin{array}{r} 100000000 \\ -\ 01110011 \\ \hline 10001101 \end{array}$$

Note the result is the 1's complement plus 1, a fact that can be used in designing hardware

By using complements, it is possible to simplify hardware by sharing adder and subtractor logic

Problem

Obtain the 1's and 2's complement of the following numbers

	10011100	10101000	11001101	01101010
1's	01100011	01010111	00110010	10010101
2's	+ 1	+ 1	+ 1	+ 1
	01100100	01011000	00110011	10010110

Complements

M
− N ← 2's

The algorithm for subtracting two unsigned binary numbers using addition

1. Add the 2's complement of the subtrahend N to the minuend M. This performs $M + (2^n - N) = M - N + 2^n$ – carry out
2. If $M \geq N$, the sum produces an end carry, 2^n. Discard the end carry, leaving result $M - N$
3. If $M < N$, the sum does not produce an end carry, since it is equal to $2^n - (N - M)$, the 2's complement of $N - M$. Perform a correction, taking the 2's complement of the sum and placing a minus sign in front to obtain the result $-(N - M)$

$2^n - (2^n - N - M) = N - M$
↑
-ve

UNSW SYDNEY

Complements

Example: Given X = 1010100 and Y = 1000011, perform the subtraction $X - Y$ and $Y - X$

$(84)_{10}$ $(67)_{10}$

$$
\begin{array}{rr}
X = & 1\ 0\ 1\ 0\ 1\ 0\ 0 \\
\text{2's complement of } Y = & +0\ 1\ 1\ 1\ 1\ 0\ 1 \\
\hline
\text{Sum} = & 0\ 0\ 1\ 0\ 0\ 0\ 1
\end{array}
$$

$(17)_{10}$

$$
\begin{array}{rr}
Y = & 1\ 0\ 0\ 0\ 0\ 1\ 1 \\
\text{2's complement of } X & \\
= & +\ 0\ 1\ 0\ 1\ 1\ 0\ 0 \\
\hline
\text{Sum} = & 1\ 1\ 0\ 1\ 1\ 1\ 1
\end{array}
$$

$Y - X$ = -(2's complement of 1101111) = -0010001

Signed binary numbers

So far we dealt with unsigned binary numbers

Positive numbers and zero are represented in the “usual” way

Negative numbers are identified by adding a minus sign on paper, but no digital implementation

Need to represent negative numbers in hardware

This is done by defining the most significant bit to be the *sign* bit

Use 0 for positive numbers and 1 for negative numbers

Possible ways to represent integers:
Signed-magnitude or *Signed-complement*

Signed-magnitude

The MSB stands for the sign (0 for +, 1 for –)

The remaining bits are of positive magnitude

8-bit example

$00011001_2 = +25_{10}$

$10100101_2 = -37_{10}$

Arithmetic using signed-magnitude is complex

It requires checking of sign bits and choosing between addition or subtraction

The carry-out or borrow determines whether a correction step should be applied to the result

Signed-complement

Negative numbers are the complements of the respective positive ones

Two possibilities:

1. Signed 1's complement – uses 1s complement arithmetic
2. Signed 2's complement – uses 2s complement arithmetic

Both will make the MSB correspond to the sign

With 2's complement arithmetic we can use the same hardware as for unsigned numbers alone

Signed-complement

Example: 3-bit numbers

Number	Sign-mag.	1's Comp.	2's Comp.
+3	011	011	011
+2	010	010	010
+1	001	001	001
+0	000	000	000
-0	100	111	-
-1	101	110	111
-2	110	101	110
-3	111	100	101
-4	-	-	100

UNSW SYDNEY

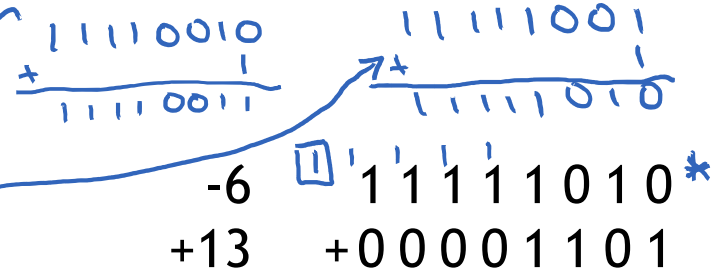
2's complement arithmetic

2's complement arithmetic does not require special hardware – can use simple adders

The addition of two signed binary numbers with negative numbers represented in signed 2s complement form is obtained from the addition of the two numbers

A carry-out of the MSB bit position is discarded

8 bit example:



+6	0 0 0 0 0 1 1 0	-6	1 1 1 1 1 0 1 0 *
+13	+0 0 0 0 1 1 0 1	+13	+0 0 0 0 1 1 0 1
+19	0 0 0 1 0 0 1 1	+7	0 0 0 0 0 1 1 1
+6	0 0 0 0 0 1 1 0	-6	1 1 1 1 1 0 1 0
-13	+1 1 1 1 0 0 1 1 *	-13	+1 1 1 1 0 0 1 1
-7	1 1 1 1 1 0 0 1	-19	1 1 0 1 1 0 1

2's complement arithmetic

To convert negative binary numbers to decimal, use one of two methods:

1. Take the 2's complement of the negative number to obtain the corresponding positive one, and add a minus sign at the front
2. Convert as usual but subtract the MSBs value instead of adding

Example: for the 8-bit number 1110 1101:

1. 2's complement: $00010010 + 1 = 00010011$ $\Leftrightarrow 19$, $= -19$, -19
2. $1110\ 1101_2 = 1 + 4 + 8 + 32 + 64 - 128 = -19_{10}$
   $2^7\ 2^6\ 2^5\ 2^3\ 2^2\ 2^0$

2's complement subtraction

Subtraction with 2's complement is simple

Take the 2s complement of the subtrahend and add it to the minuend

A carry-out of the sign bit position is discarded

Works because: $A - B = A + (-B)$

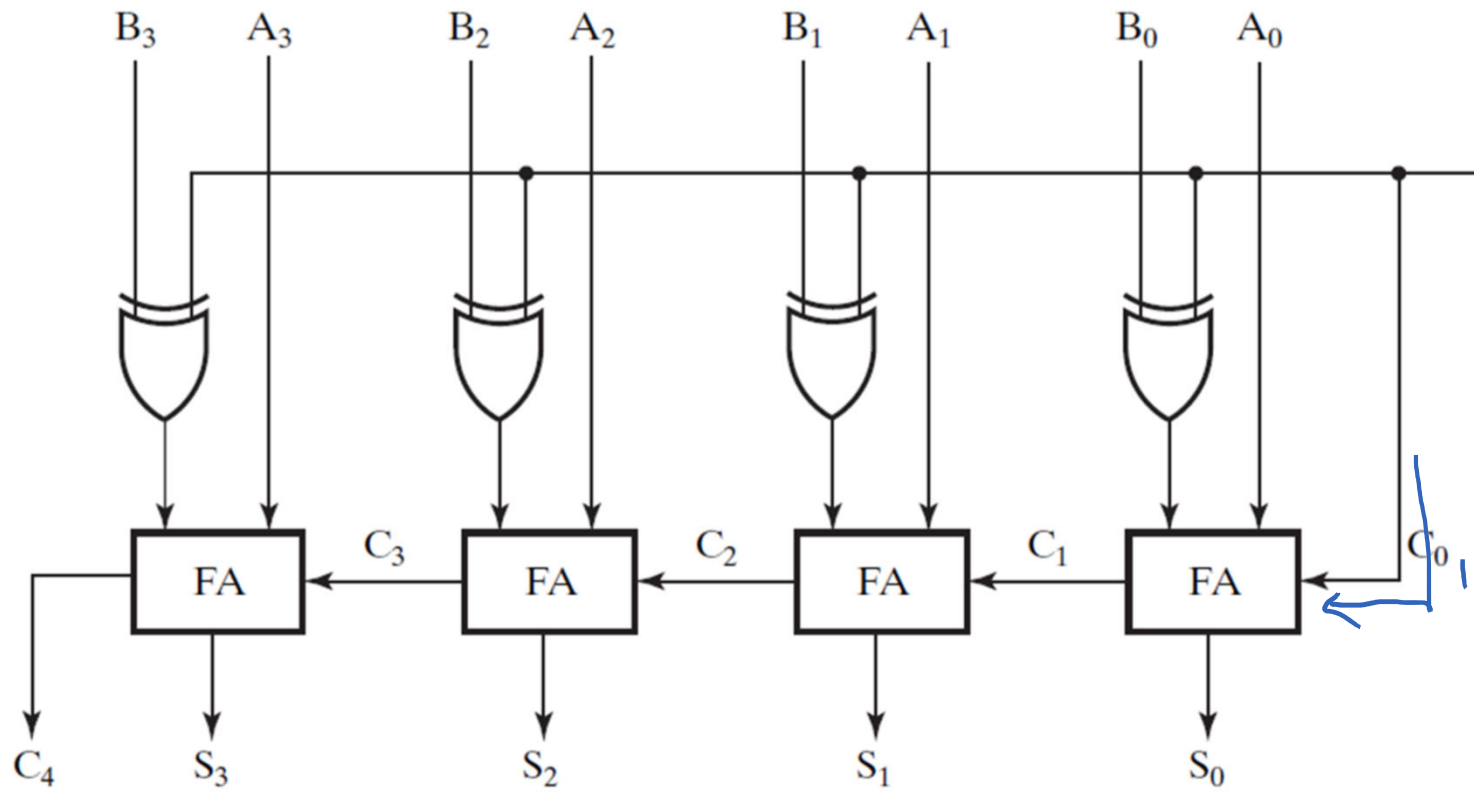
8-bit example:

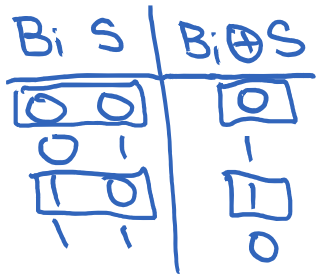
-6	11111010
- -13	-11110011
+7	

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

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

2's complement adder/subtractor





$S = A + B \quad \overline{\text{Add}}/\text{Sub} = 0$

$S = A - B \quad \overline{\text{Add}}/\text{Sub} = 1$

UNSW SYDNEY

2's complement adder/subtractor

Signal $\overline{\text{Add}}/\text{Sub}$ selects between addition or subtraction

If $\overline{\text{Add}}/\text{Sub} = 0$, vector *B* propagate through the XOR gates and the carry-in is 0 – computes *A* + *B*

If $\overline{\text{Add}}/\text{Sub} = 1$, vector *B* is complemented and the carry-in is 1 to obtain the 2s complement of *B* – computes *A* – *B*

Sign extension

0000 1011

To represent an *n*-bit number with larger number of bits (*n* + *m*):

For *unsigned* numbers add *m* 0's at the front

For *signed* numbers, extend the MSB at the front of the number

4 to 8 bits signed example:

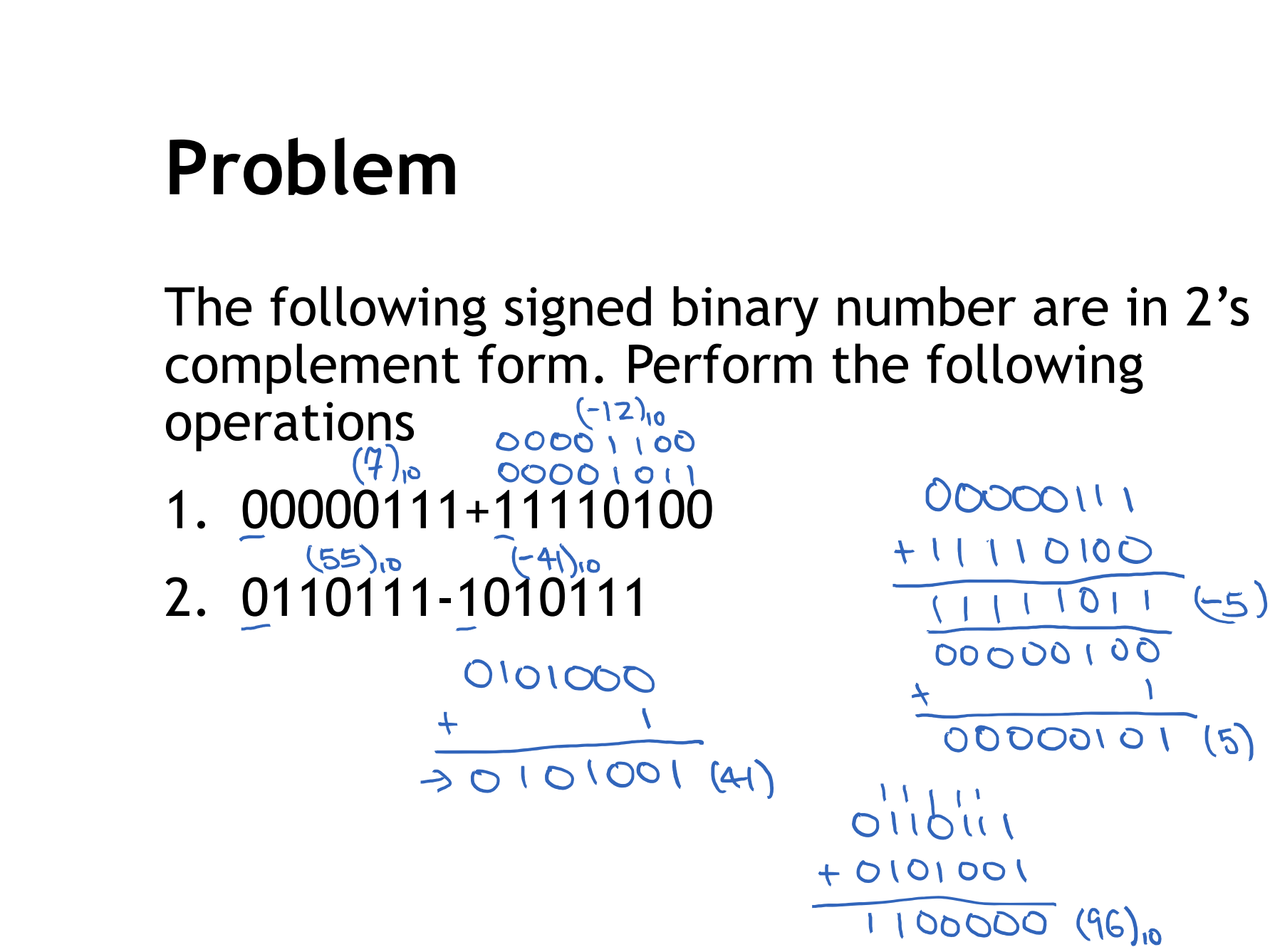
+7 = 0111 = 0000 0111

-7 = 1001 = 1111 1001

Problem

The following signed binary number are in 2's complement form. Perform the following operations

1. 00000111+11110100
2. 0110111-1010111



Overflow

Overflow occurs if (n + 1) bits are required to contain the result from an n-bit addition or subtraction

Example: 8-bits can represent values between -128 to +127. Calculate 70 + 80:

2's comp.

```
  1 0
  0 1 0 0 0 1 1 0    70₁₀
+ 0 1 0 1 0 0 0 0  + 80₁₀
-----------------
  1 0 0 1 0 1 1 0    150₁₀
  0 1 1 0 1 0 1 0    (-106)₁₀
```

Negative number! (MSB = 1)

For *unsigned* addition:

$C = 0 \rightarrow$ No Overflow

$C = 1 \rightarrow$ Overflow

For *signed* addition (and subtraction):

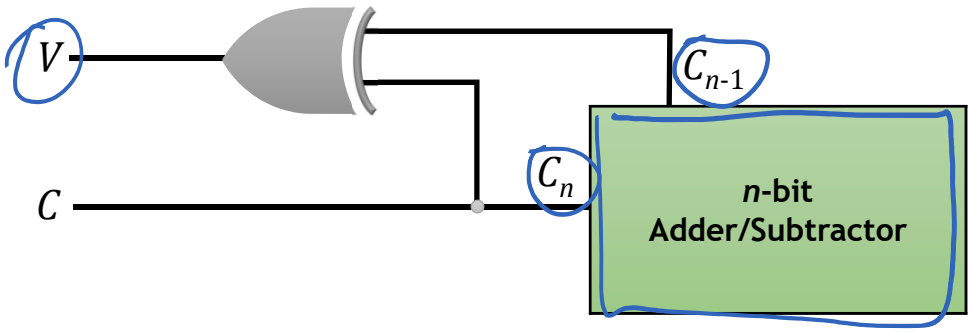
$V = 0 \rightarrow$ No Overflow

$V = 1 \rightarrow$ Overflow

Overflow

Overflow occurs when the carry-in to the MSB is not equal to the carry-out from the MSB

A simple circuit to detect overflow



C_n	C_{n+1}	V
0	0	0
0	1	1
1	0	1
1	1	0

N, Z, C, V – Status flags

Status signals that provide information from the arithmetic unit in microprocessors

N (*Negative*): 1 if the sum is negative; 0 if positive

Z (*Zero*): 1 if the sum is zero; 0 if non-zero

C (*Carry*): 1 if there is a carry-out; 0 if no carry

V (*oVerflow*): 1 if signed overflow detected; 0 for no-overflow

Can be used to compare numbers

N, Z, C, V - Status flags

These can be implemented by:

$N = S_{n-1}$ (value at MSB of sum)

$Z = \overline{S_{n-1} + S_{n-2} + \ldots + S_0}$ (detect if all sum bits are 0)

$C = C_n$ (carry out from MSB)

$V = C_n \oplus C_{n-1}$ (MSB carry-in ≠ carry-out)

C_n C_{n-1}

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

$N = 1$; $Z = 0$; $C = 0$; $V = 1$

N, Z, C, V - Status flag

```
// status flags

module   add32_status_flags(a, b, s, c, z, n, v);

      input reg signed [31:0] a,b;   //ab - binary values
      output reg signed [31:0] s;    // sum
      output z,n,v,c;                // status flags
      assign {c,s} = a + b;          // carry + sum
      assign z = ~|s;                // NOR S
      assign n = s[31];              // MSB
      assign v = a[31]^b[31]^s[31]^c;  // MSB a, b, s, c  a⊕b⊕s⊕c

endmodule
```

BCD addition

0~9 10~15 4-bit

BCD addition is complicated by the fact that the sum can exceed 9

X+*Y* >9

X	0 1 1 1	7
+ Y	+ 0 1 0 1	+ 5
Z	1 1 0 0	12
	+ 0 1 1 0	+ 6
carry	0 0 1 0 0 1 0	
	S = 2	

Z=*X*+*Y*

If *Z* ≤ 9, then *S*=*Z* & *C*=0

If Z ≥ 9, then *S*=Z+6 & *C*=1

X+*Y* >15

X	1 0 0 0	8
+ Y	+ 1 0 0 1	+ 9
Z	1 0 0 0 1	17
	+ 0 1 1 0	+ 6
carry	0 0 1 0 1 1 1	
	S = 7	

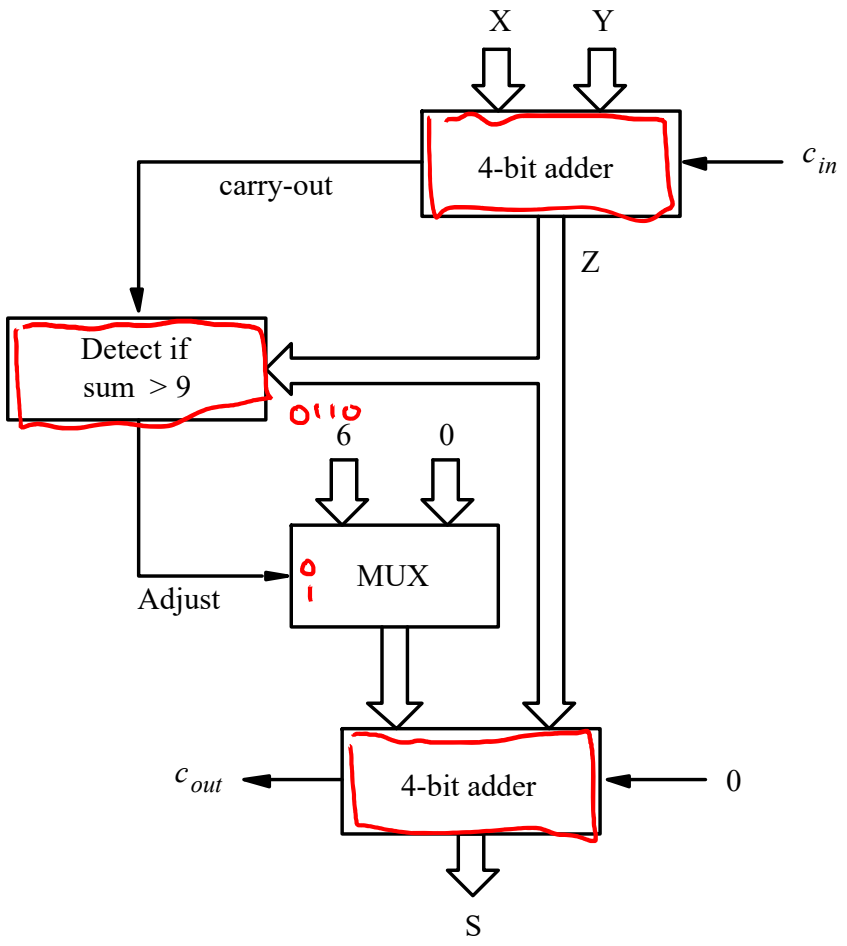
Z=*X*+*Y*

If *Z* ≥ 16, then *S*=Z+6 & *C*=1

BCD addition

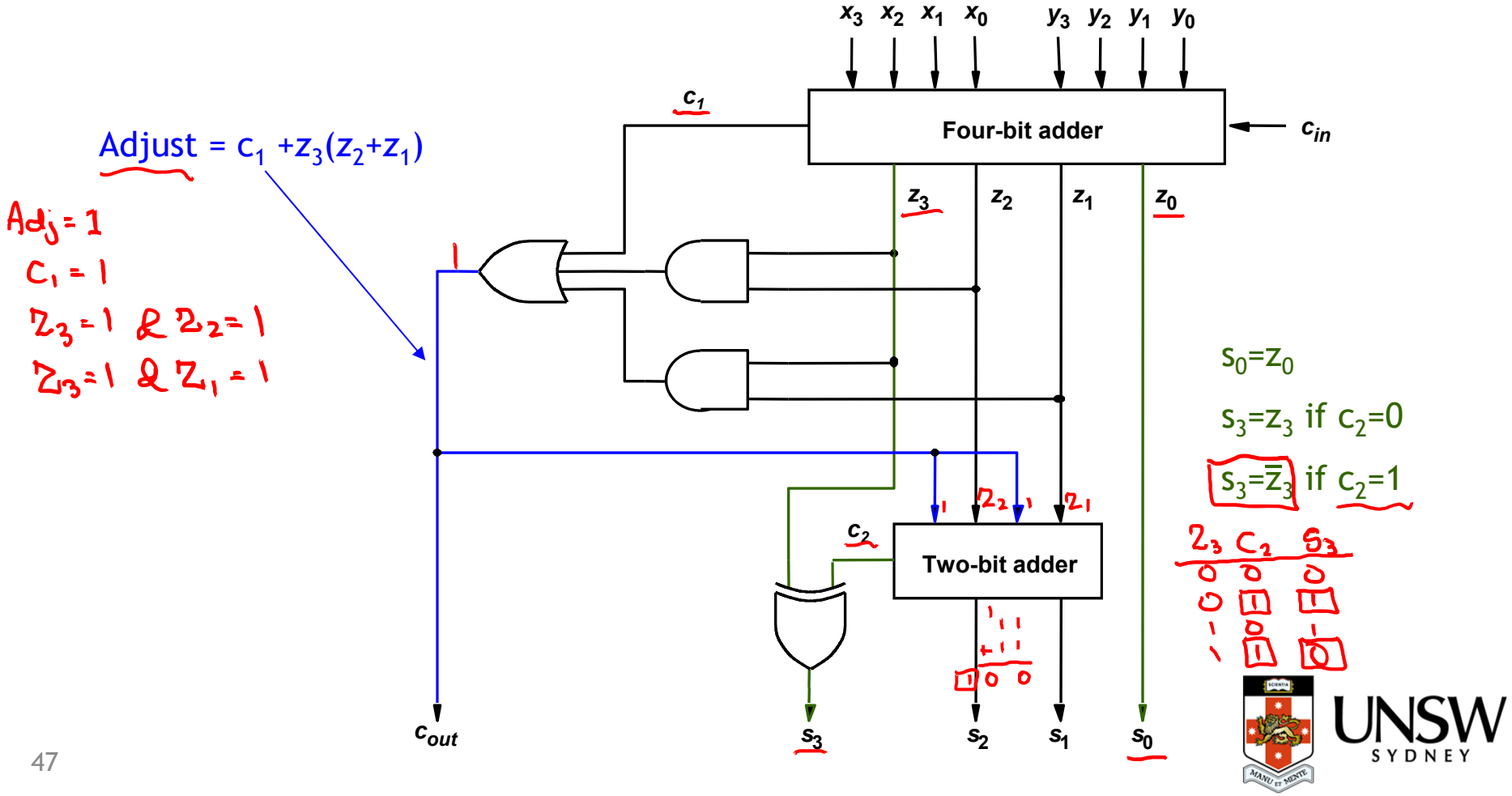
As the correction for when the sum is greater than 9 or 16 is the same, only need to check whether sum > 9

The signal Adjust controls whether the MUX provides correction or not



BCD addition

As 6 is 0110, the correction can be done using a 2-bit adder instead



Binary multiplication

The binary digit multiplication table is trivial

a x b	b = 0	b = 1
a = 0	0	0
a = 1	0	1

This is simply the Boolean AND function

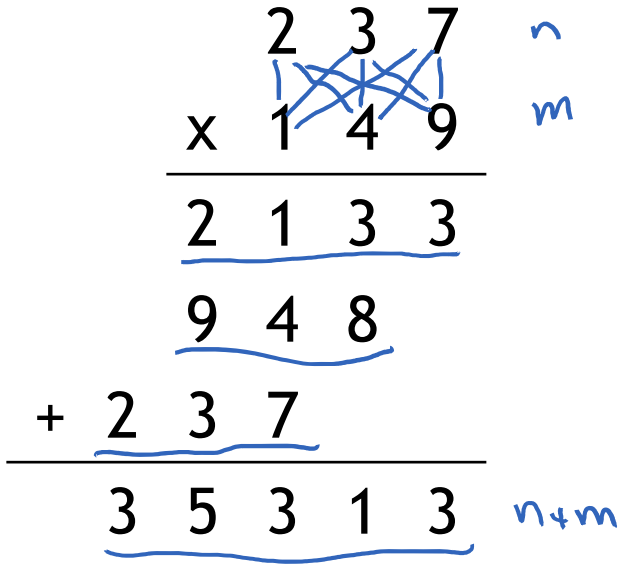
Larger binary products are formed just like larger products are formed in base 10

Binary multiplication

In decimal multiplication, partial products are computed and then summed up

The partial product summation for n digits, requires adding up to n digits (with carries)

Also an n × m digits multiply generates up to an m + n digit result



UNSW SYDNEY

Binary multiplication

$10^2 \rightarrow 100$

$10^{-2} \rightarrow 0.01$

Multiplying a number by 2^k, shifts it to the left by k bit positions

Dividing a number by 2^k, shifts it to the right by k bit positions

For unsigned numbers 0s are added to the left of the MSB

00110

In signed numbers, as it is necessary to preserve the sign, bits are shifted to the right and the value of the sign bit filled from the left

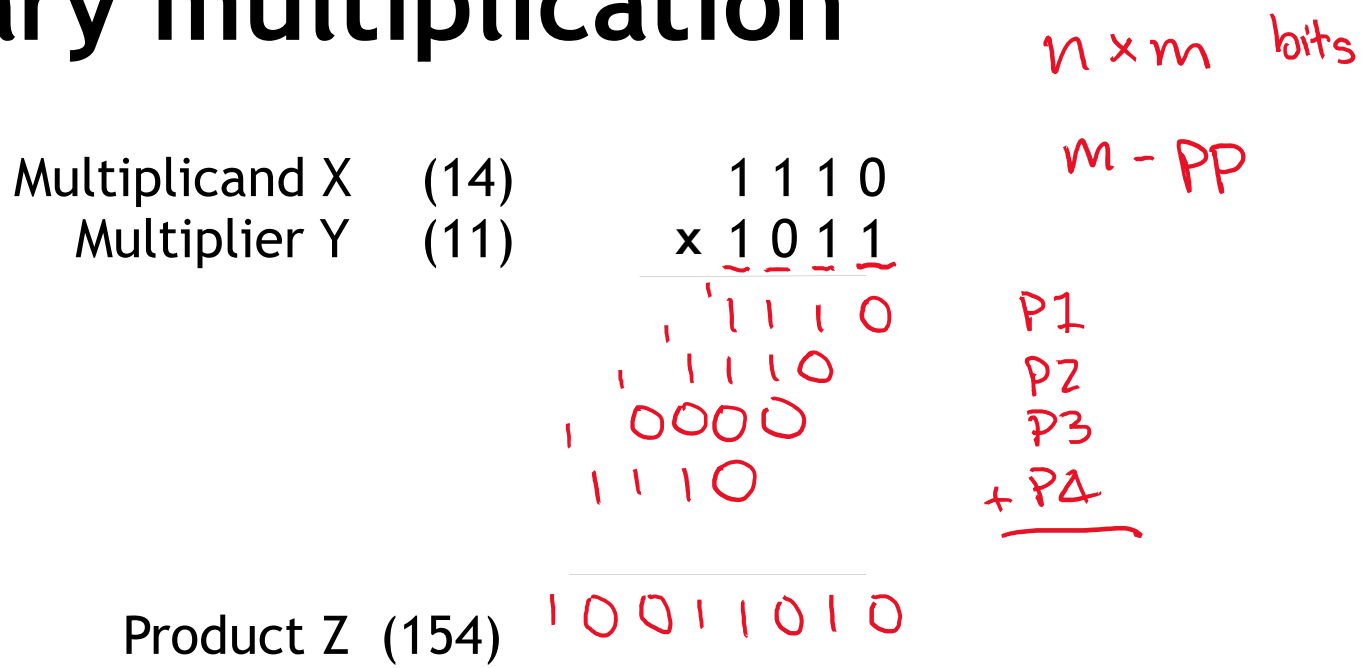
A = 011000

00011000

B = 101000

11101000

Binary multiplication



Partial products are

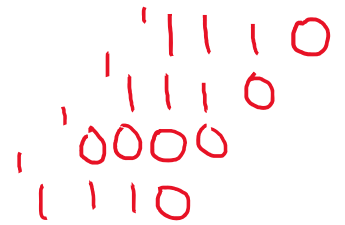
either all zero (if the multiplier digit is zero) or
the same as the multiplicand (if the multiplier digit is one)

Note: No carries are added in partial product formation

UNSW SYDNEY

Binary multiplication

Multiplicand X	(14)	1 1 1 0
Multiplier Y	(11)	x 1 0 1 1
Product Z	(154)	1 0 0 1 1 0 1 0

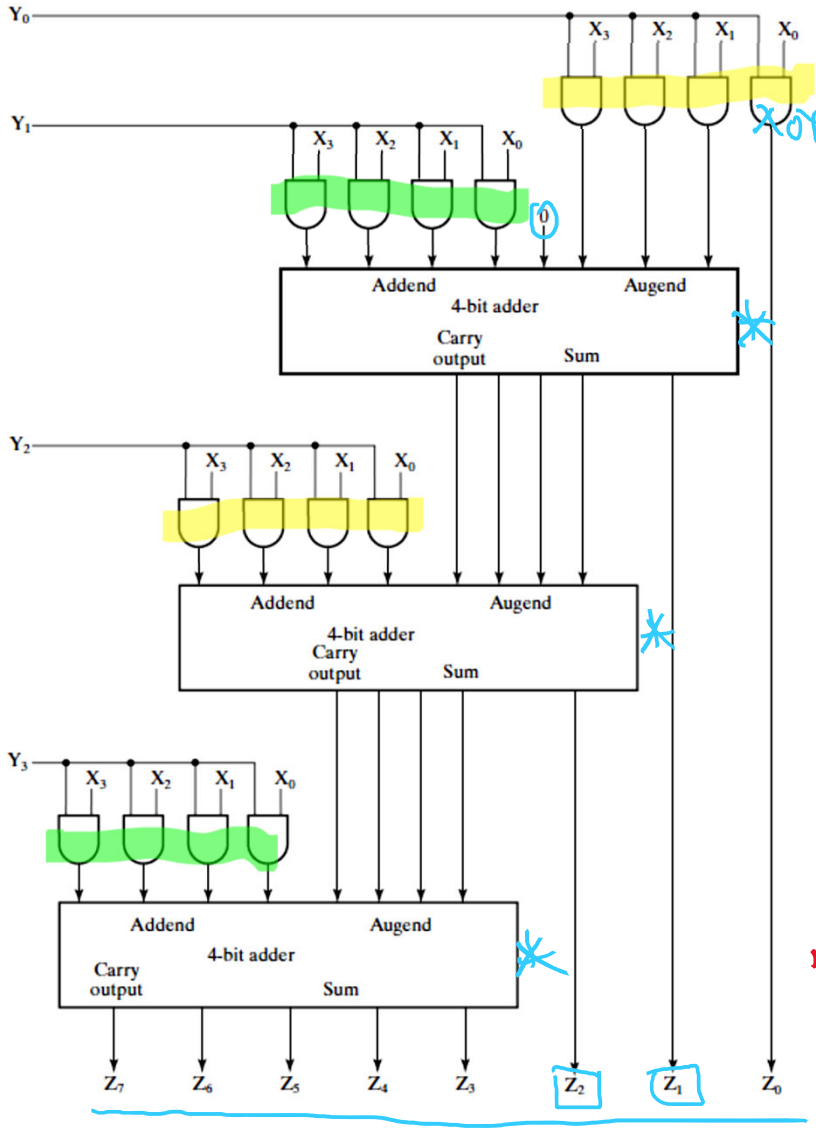


Partial products are

- either all zero (if the multiplier digit is zero) or
  the same as the multiplicand (if the multiplier digit is one)

Note: No carries are added in partial product formation

Binary multiplication



Partial products are formed using an n × m array of AND gates

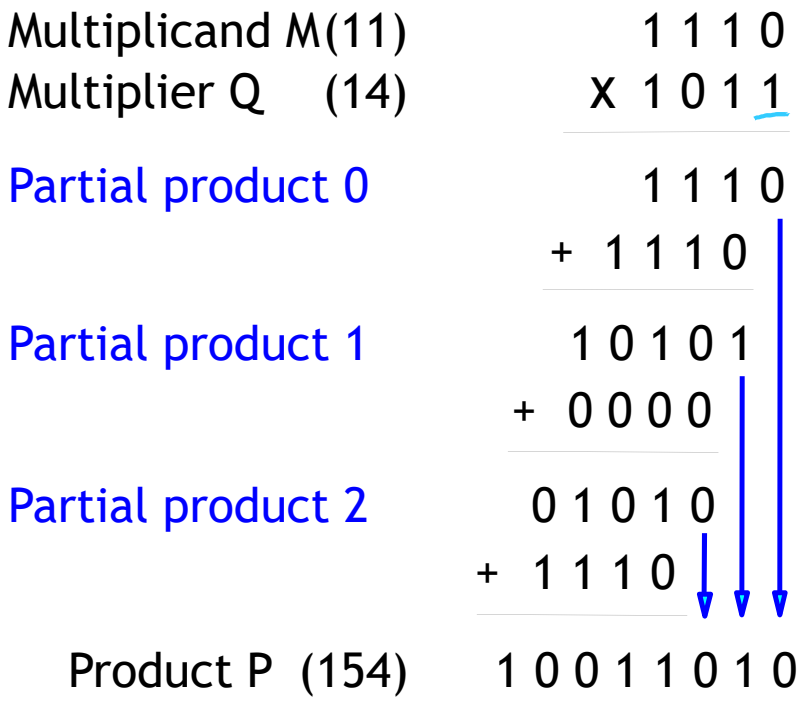
The partial products are then summed using adder trees

Will need m – 1 adders of width n bits

$$\begin{array}{cccc} X_3 & X_2 & X_1 & X_0 \\ \times\ Y_3 & Y_2 & Y_1 & Y_0 \end{array}$$

Binary multiplication

In hardware, a sequential approach is adopted with adders used to compute partial products



Faster to use multiple adders instead of a single adder

UNSW SYDNEY

Binary multiplication

Sequential multiplication circuits are slow

Can be made faster by performing addition in array form

$M = m_3 m_2 m_1 m_0$

$Q = q_3 q_2 q_1 q_0$

4x4 multiplication

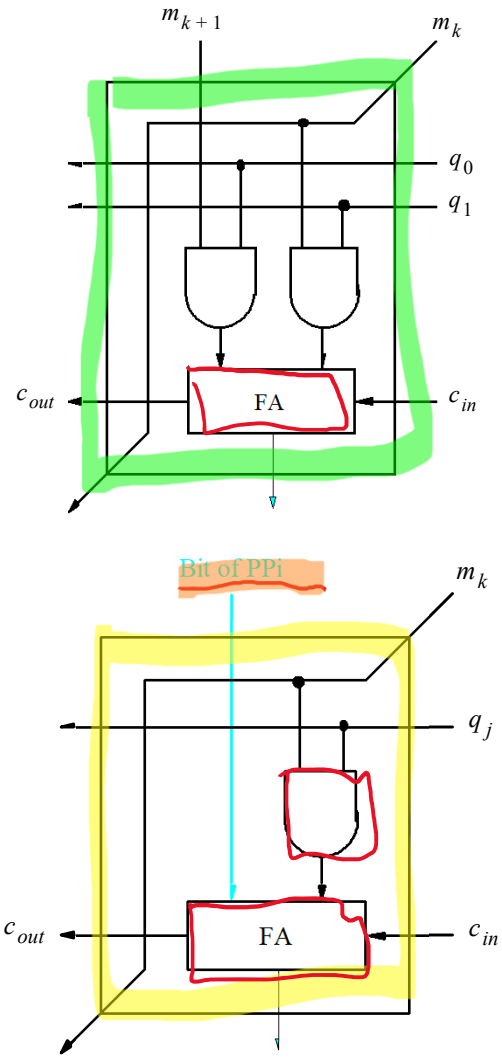
$PP0 = m_3 q_0 \;\; m_2 q_0 \;\; m_1 q_0 \;\; m_0 q_0$

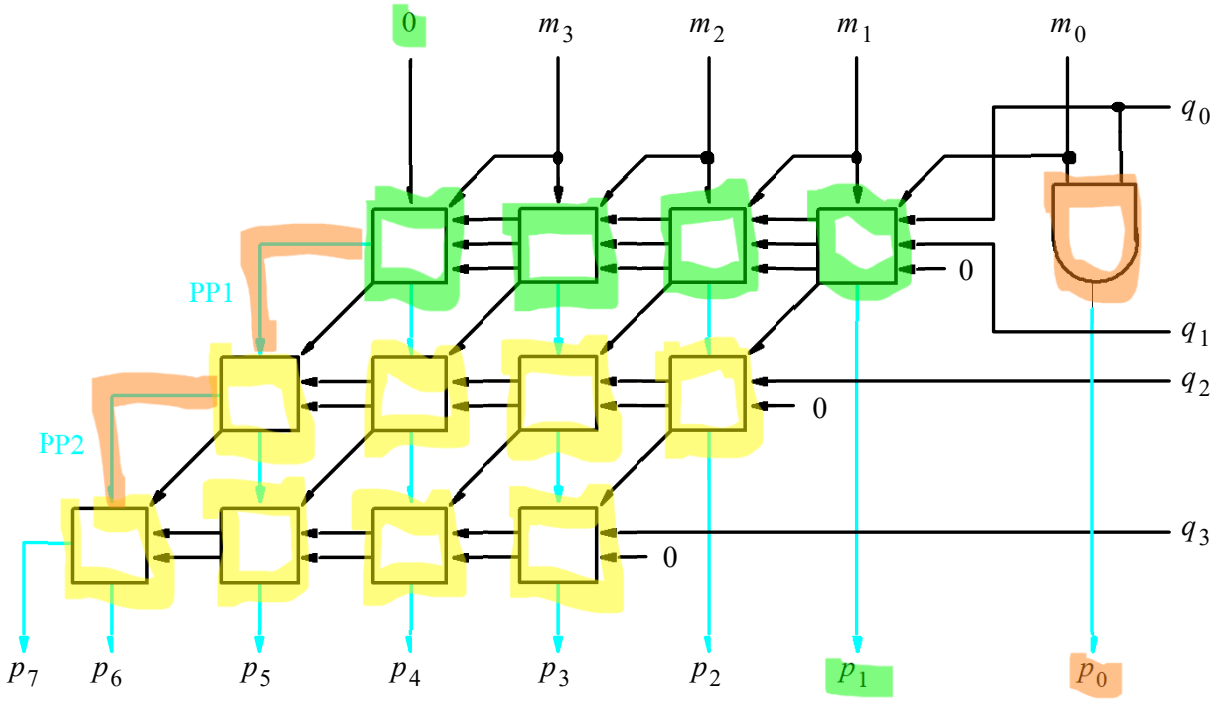
$$
\begin{array}{lccccc}
PP0: & 0 & pp0_3 & pp0_2 & pp0_1 & pp0_0 \\
+ & m_3 q_1 & m_2 q_1 & m_1 q_1 & m_0 q_1 & 0 \\
\hline
PP1: & pp1_4 & pp1_3 & pp1_2 & pp1_1 & pp1_0
\end{array}
$$

$$
\begin{array}{l}
\quad m_3\ m_2\ m_1\ m_0 \\
\times\ q_3\ q_2\ q_1\ q_0 \\
\hline
\quad m_3q_0\ m_2q_0\ m_1q_0\ m_0q_0 \\
m_3q_1\ m_2q_1\ m_1q_1\ m_0q_1 \\
\hline
pp1_4\ pp1_3\ pp1_2\ pp1_1\ pp1_0
\end{array}
$$

Binary multiplication

$$
\begin{array}{cccccccc}
 & & m_3 & m_2 & m_1 & m_0 & & \\
\times & & q_3 & q_2 & q_1 & q_0 & & \\
\hline
 & & m_3q_0 & m_2q_0 & m_1q_0 & m_0q_0 & & \\
 & m_3q_1 & m_2q_1 & m_1q_1 & m_0q_1 & & & \\
 & m_3q_2 & m_2q_2 & m_1q_2 & m_0q_2 & & & \\
m_3q_3 & m_2q_3 & m_1q_3 & m_0q_3 & & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$





UNSW SYDNEY