Digital Logic

Chapter 3:
Number Representation and Arithmetic Circuits

Ru Han

1

1

Outline

Chapter 3:

- Representation of numbers in computers
- Circuits used to perform arithmetic operations
- Performance issues in large circuits
- Use of Verilog to specify arithmetic circuits

2

2

3.1 Positional Number Representation

Unsigned Integers

Numbers that are positive only are called *unsigned*, and numbers that can also be negative are called *signed*.

An n-bit unsigned number: $B = b_{n-1}b_{n-2}\cdots b_1 b_0$

$$V(B) = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

$$= \sum_{i=0}^{n-1} b_i \times 2^i$$

3

3

3.1 Positional Number Representation

Octal and Hexadecimal Representations

$K = k_{n-1}k_{n-2}\cdots k_1 k_0$

$$V(k) = \sum_{i=0}^{n-1} k_i \times r^i$$

Decimal	Binary	Octa	Hexadecimal
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10
17	10001	21	11
18	10010	22	12

4

4

3.2 Addition of Unsigned Numbers

1-Bit Addition

- Two 1-bit addends: x, y ∈ {0, 1}
  ⇒ sum ∈ {0, 1, 2}

x	0	0	1	1
$+y$	$+0$	$+1$	$+0$	$+1$
$c\ s$	0 0	0 1	0 1	1 0

Carry — Sum

(a) The four possible cases

5

5

1-Bit Addition

- Truth Table

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- SOP Realization

s

x \ y	0	1
0	0	1
1	1	0

c

x \ y	0	1
0	0	0
1	0	1

$s = \bar{x}\cdot y + x\cdot\bar{y}$ $\quad c = x\cdot y$

6

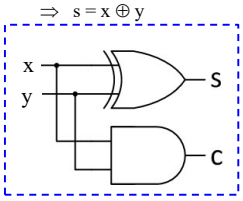
6

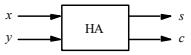
1-Bit Addition

$s = \bar{x}\cdot y + x\cdot\bar{y}$ $\qquad$ $c = x\cdot y$

Note that this is just an exclusive-or operation

$\Rightarrow\ s = x \oplus y$





(d) Graphical symbol

HALF-ADDER:
- Uses only 2 gates
- Does not require inverted inputs

7

7

Multi-bit Addition

- To extend our results to larger inputs & outputs, consider using a half-adder to add each pair of bits.
- Ex:

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

- **Problem:**
  If a carry is generated at one stage, it must be added to the next stage. *No input available for this on half adder.*

8

8

Full Adder

- 2 1-bit addends: a, b ∈ {0, 1}
  1 1-bit carry-in: $c_{in} \in \{0, 1\}$

still only need 2-bit output

low bit: **s** (sum)
high bit: $\mathbf{c_{out}}$ (carry-out)

$\Rightarrow$ sum ∈ {0, 1, 2, 3}

9

9

Full Adder

- Truth Table

a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

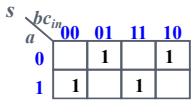
- s = 1 if 1 or 3 inputs = 1
- c_{out} = 1 if 2 or 3 inputs = 1

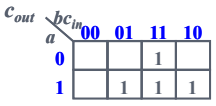
10

10

Full Adder

- SOP Realization





$s = \bar{a}\bar{b}c_{in} + \bar{a}b\bar{c}_{in} + a\bar{b}\bar{c}_{in} + abc_{in}$ $\qquad$ $c_{out} = ab + ac_{in} + bc_{in}$

Cost = 5 + 16 = 21 $\qquad$ Cost = 4 + 9 = 13

Total cost = 34

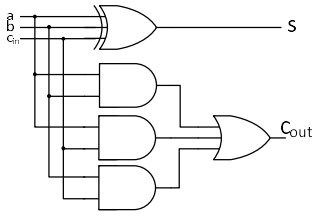
11

11

Full Adder

- Use of XOR

$s = a \oplus b \oplus c_{in}$ $\qquad$ $c_{out} = ab + ac_{in} + bc_{in}$



Total cost = 5 + 12 = 17

12

12

XOR Gates

- XOR operations are sometimes very useful in optimizing logic circuits because ***they cover non-adjacent squares*** in a K-map.
- 2-input XOR

$f = a \oplus b$
$= \bar{a}\cdot b + a\cdot\bar{b}$
$= b \oplus a$

a \ b	0	1
0	0	1
1	1	0

13

13

XOR Gates

- XOR operations are sometimes very useful in optimizing logic circuits because ***they cover non-adjacent squares*** in a K-map.
- 2-input XOR

$f = a \oplus b$
$= \bar{a}\cdot b + a\cdot\bar{b}$
$= b \oplus a$

a \ b	0	1
0	0	1
1	1	0

For the Two-input XOR , one input can be thought as the control signal to determine whether the true or complemented of the other input.

14

14

XOR Gates – 3 input XOR

$f = a \oplus b \oplus c = (a \oplus b) \oplus c = a \oplus (b \oplus c)$

- $f = 1$ if an odd number of inputs = 1
- $f = 0$ if an even number of inputs = 1

f \ bc / a	00	01	11	10
0	0	1	0	1
1	1	0	1	0

SOP: 5 gates
XOR: 1 gate

$b \oplus c$ \ bc / a	00	01	11	10
0	0	1	0	1
1	0	1	0	1

$a \oplus b$ \ bc / a	00	01	11	10
0	0	0	1	1
1	1	1	0	0

$a \oplus c$ \ bc / a	00	01	11	10
0	0	1	1	0
1	1	0	0	1

SOP: 3 gates; XOR: 1 gate

15

15

XOR Gates – 4 input XOR

$f = a \oplus b \oplus c \oplus d$

- $f = 1$ if + only if an odd number of inputs = 1

ab \ cd	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

SOP: 9 gates
XOR: 1 gate

XOR is also referred to as the odd function。

$a \oplus b \oplus d$ \ cd / ab	00	01	11	10
00	0	1	1	0
01	1	0	0	1
11	0	1	1	0
10	1	0	0	1

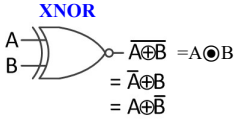
$b \oplus c$ \ cd / ab	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	1	1	0	0
10	0	0	1	1

16

16

XOR Gates – Features of XOR

- XOR is also referred to as the odd function。
- For the Two-input XOR , one input can be thought as the control signal to determine whether the true or complemented of the other input.
- XNOR, which is the complement of XOR, is also referred to as the coincidence operation.
- The symbol of the XNOR.



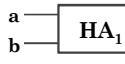
17

17

3.2.1 Decomposed full adder

c_{out} \ bc_{in} / a	00	01	11	10
0			1	
1		1	1	1

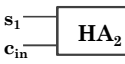
- Can we make a full adder using two half adders?
- Use HA_1 to add a, b



$s_1 = a \oplus b$
$c_1 = ab \quad = ab\bar{c}_{in} + abc_{in}$

- Use HA_2 to add s_1, c_{in}

Let $s = s_2$ $c_2 = c_{out}$?



$s_2 = s_1 \oplus c_{in} = (a \oplus b) \oplus c_{in}$
$c_2 = s_1 \cdot c_{in}$
$= (a \oplus b) \cdot c_{in} = \bar{a}bc_{in} + a\bar{b}c_{in}$

Let $c_{out} = c_1 + c_2 = \bar{a}bc_{in} + a\bar{b}c_{in} + ab\bar{c}_{in} + abc_{in}$

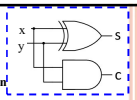
18

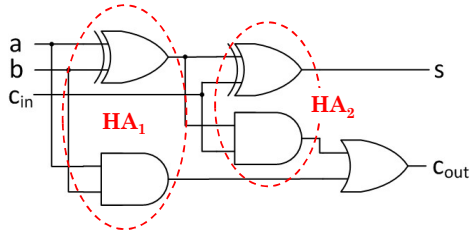
18

Multi-bit Addition

$s_1 = a \oplus b$
$c_1 = ab$
$s_2 = s_1 \oplus c_{in} = (a \oplus b) \oplus c_{in}$
$c_2 = s_1 \cdot c_{in}$
$s = s_2$
$c_{out} = c_1 + c_2$

- Circuit





Total cost = 5 + 10 = 15

19

19

3.3.2 Ripple-Carry Adder

- We can make an n-bit adder by using n full adders

$$\begin{array}{r} 1\ 0\ 1\ 1 \\ +\ 0\ 1\ 1\ 0 \\ \hline \end{array}$$

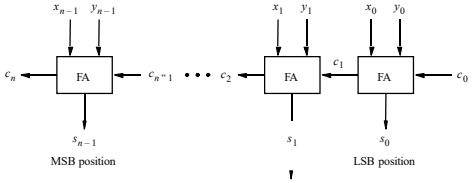


Figure 3.5. An n-bit ripple-carry adder.

20

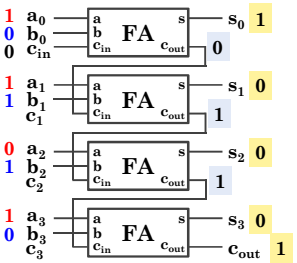
20

Ripple-Carry Adder

- We can make an n-bit adder by using n full adders
- Ex: $n = 4$

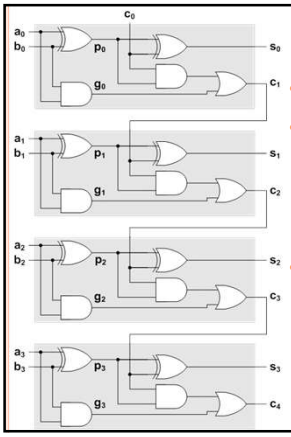
a+b

$$\begin{array}{r} 1\ 0\ 1\ 1 \\ +\ 0\ 1\ 1\ 0 \\ \hline \end{array}$$



21

21



4-Bit Ripple-Carry Adder

- The delay increases as more bits are added.
- Ex: 4-bit ripple-carry adder:
  Max delay from inputs to c_1 = 3 gates
  Max delay from inputs to c_2 = 5 gates
  Max delay from inputs to c_3 = 7 gates
  Max delay from inputs to c_{out} = 9 gates

Critical path delay

- Note that delays to s_1, s_2, s_3 are each one gate more than delays to c_1, c_2, c_3.
  ⇒ If we can reduce delays to carry bits, delay to sum bits will also be reduced.

22

22

Analysis of the Ripple-Carry Adder

- For ripple-carry adder, the total delay depends on the size of the number.
- When 32 or 64 bit number, the delay is unacceptable.
- As a result, we need to design a new architecture in the following chapter.
- Until now, we talk about the unsigned adder.

23

23

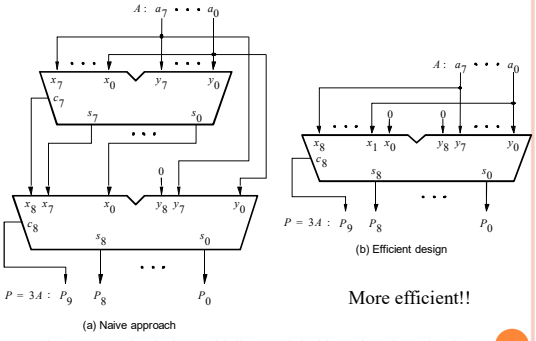
3.2.3 Design Example

- Suppose that we need a circuit that multiplies an eight-bit unsigned number by 3. Let $A = a_7a_6 \cdots a_1a_0$ denote the number and $P = p_9p_8 \cdots p_1p_0$ denote the product $P = 3A$.

24

24

Design Example



More efficient!!

Figure 3.6. Circuit that multiplies an eight-bit unsigned number by 3.

25

3.3 Signed Numbers

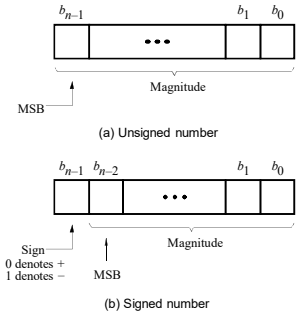


Figure 3.7. Formats for representation of integers.

26

3.3.1 Negative Numbers

- Negative numbers can be represented in three different ways:

  sign-and-magnitude,

  1's complement,

  2's complement.

27

Negative Numbers

- Negative numbers can be represented in three different ways:

（1）sign-and-magnitude

The sign symbol distinguishes a number as being positive or negative.

（2）1's complement

In the *1's complement* scheme, an n-bit negative number, K, is obtained by subtracting its equivalent positive number, P, from $2^n - 1$; that is, $K = (2^n - 1) - P$.

（3）2's complement

In the 2's complement scheme, a negative number, K, is obtained by subtracting its equivalent positive number, P, from 2^n; namely, $K = 2^n - P$.

28

Negative Numbers

- **Rule for Finding 1's and 2's Complements**

Given a number $B = b_{n-1}b_{n-2} \cdots b_1b_0$,

- **1's Complements**

1's complement can be obtained simply by complementing each bit of the number, including the sign bit.

- **2's Complements**

(1) a simpler way of finding a 2's complement of a number is to add 1 to its 1's complement

(2) Its 2's complement, $K = k_{n-1}k_{n-2} \cdots k_1k_0$, can be found by examining the bits of B from right to left and taking the following action:

（a）copy all bits of B that are 0 and the first bit that is 1;

（b）then simply complement the rest of the bits.

29

Negative Numbers

$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	–0	–7	–8
1001	–1	–6	–7
1010	–2	–5	–6
1011	–3	–4	–5
1100	–4	–3	–4
1101	–5	–2	–3
1110	–6	–1	–2
1111	–7	–0	–1

Table 3.2. Interpretation of four-bit signed integers.

30

- **Assignment: Page 184: 3.1, 3.2, 3.4**

31

31

3.3.2 Addition and Subtraction

- **1's Complement Addition**

(+5)	0 1 0 1	(–5)	1 0 1 0
+(+2)	+0 0 1 0	+(+2)	+0 0 1 0
(+7)	0 1 1 1	(-3)	1 1 0 0
(+5)	0 1 0 1	(–5)	1 0 1 0
+(–2)	+1 1 0 1	+(–2)	+1 1 0 1
(+3)	1 0 0 1 0	(–7)	1 0 1 1 1
	↳ 1		↳ 1
	0 0 1 1		1 0 0 0

Figure 3.8. Examples of 1's complement addition.

32

32

Addition and Subtraction

- **2's Complement Addition**

(+5)	0 1 0 1	(–5)	1 0 1 1
+ (+2)	+ 0 0 1 0	+ (+2)	+ 0 0 1 0
(+7)	0 1 1 1	(–3)	1 1 0 1
(+5)	0 1 0 1	(–5)	1 0 1 1
+ (–2)	+ 1 1 1 0	+ (–2)	+ 1 1 1 0
(+3)	1 0 0 1 1	(–7)	1 1 0 0 1
	↑		↑
	ignore		ignore

Figure 3.9. Examples of 2's complement addition.

33

33

Addition and Subtraction

- **2's Complement Subtraction**

The easiest way of performing subtraction is to negate the subtrahend and add it to the minuend.

(+5)	0 1 0 1		0 1 0 1		
– (+2)	– 0 0 1 0	⇒	+ 1 1 1 0		
(+3)			1 0 0 1 1		
			↑		
			ignore		
(–5)	1 0 1 1		1 0 1 1		
– (+2)	– 0 0 1 0	⇒	+ 1 1 1 0		
(–7)			1 1 0 0 1		
			↑		
			ignore		
(+5)	0 1 0 1		0 1 0 1		
– (–2)	– 1 1 1 0	⇒	+ 0 0 1 0		
(+7)			0 1 1 1		
(–5)	1 0 1 1		1 0 1 1		
– (–2)	– 1 1 1 0	⇒	+ 0 0 1 0		
(–3)			1 1 0 1		

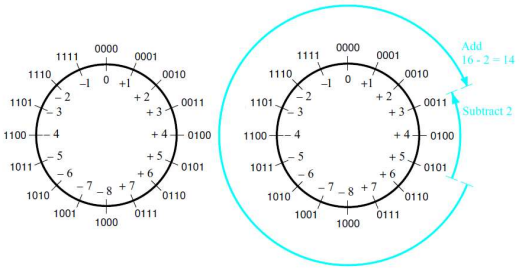
Figure 3.10. Examples of 2's complement subtraction.

34

34

Addition and Subtraction

- **2's Complement Subtraction**



(a) The number circle (b) Subtracting 2 by adding its 2's complement

Graphical interpretation of four-bit 2's complement numbers

35

35

3.3.3 Adder and Subtraction Unit

- For X-Y = X+(-Y), we need the 2's complement of Y, which can be obtained by adding 1 to the 1's complement of Y. In other words, **Subtraction** can be completed by the addition.
- So :We need to perform X+Y and X+$\overline{Y}$+1 in just one circuit.
- How can we build the unify circuit to perform addition and subtraction.
- Think about the feature of XOR.

36

36

3.3.3 Adder and Subtractor Unit

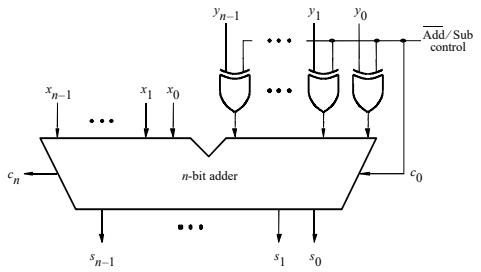


Figure 3.12. Adder/subtractor unit.

37

37

Hints:

- When design the digital circuit:
  - As flexible as possible
  - As many tasks as possible
- To minimize the area and reduce the wiring complexity

38

38

3.3.5 Arithmetic Overflow

- What is overflow?
  - The result beyonds the range of the 2's complement of n-bit singed number.
- So: The occurrence of overflow should be detected and reported the system.
- Investigate some example in the textbook.

39

39

Arithmetic Overflow

- The four cases where 2's-complement numbers with magnitudes of 7 and 2 are added.

(+7)	0 1 1 1	(–7)	1 0 0 1
+ (+2)	+ 0 0 1 0	+ (+2)	+ 0 0 1 0
(+9)	1 0 0 1	(–5)	1 0 1 1
	$c_4 = 0$		$c_4 = 0$
	$c_3 = 1$		$c_3 = 0$
(+7)	0 1 1 1	(–7)	1 0 0 1
+ (–2)	+ 1 1 1 0	+ (–2)	+ 1 1 1 0
(+5)	1 0 1 0 1	(–9)	1 0 1 1 1
	$c_4 = 1$		$c_4 = 1$
	$c_3 = 1$		$c_3 = 0$

Figure 3.13. Examples of determination of overflow.

For n-bit numbers we have

Overflow = $c_{n-1} \oplus c_n$
$= x_3 y_3 s_3 + \bar{x}_3 \bar{y}_3 s_3$

40

40

How to detect the overflow?

- Analysis Figure 3.9 and Figure 3.13
- We can draw the conclusion:
- for the n-bit numbers ,we have

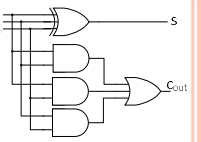
Overflow = $c_{n-1} \oplus c_n$
Note: c_{n-1} is the carry out from the MSB position
C_n is the carry out form the Sign position

- Another way to detect the overflow is to compare the sign bit of sum with the sign of summands.
- So: ??
- Understand the difference of carry out and the overflow

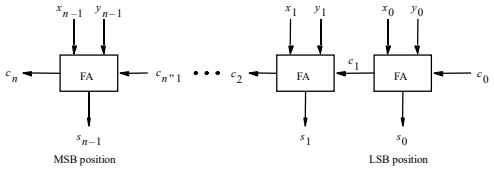
41

41

3.3.6 Performance Issues



- A commonly used indicator of the value of a system is its *price/performance ratio.*

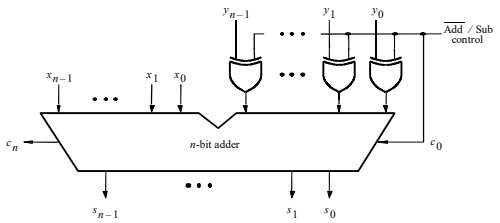


- Circuit delay: (2n)△t

42

42

Performance Issues



- Circuit delay: (2n+1)△t
- The longest delay is often referred to as the *critical-path delay*。
- The path that causes this delay is called the *critical path.*

43

43

3.4 Fast Adder

- We can reduce the delays to the carry bits by calculating them directly from the inputs, not from the outputs from the previous stage.
- The trade-off is an increase in complexity for a decrease in delay.
- There are two ways to produce a carry output from each stage.
  - Carry Generation
  - Carry Propagate

44

44

Fast Adder

- Carry Generation: in a given stage, a carry is generated if both *a* and *b* are 1.

  ⇒ $c_{n+1} = 1$ if $a_n = b_n = 1$

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

No carry at stage 0

Carry generated at stage 1

⇒ $g_n = a_n\, b_n$

45

45

Fast Adder

- Carry Propagate: if a carry comes in to a given stage, we will propagate that carry to the next stage if either *a* or *b* is 1.

  ⇒ $c_{n+1} = 1$ if $c_n = 1$ and ($a_n = 1$ or $b_n = 1$)

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

No carry

Generate

Propagate

Note: if $a_n = b_n = c_n = 1$, it is customary to count this as a generate, but not a propagate. (Book is different.)

⇒ $p_n = a_n \oplus b_n$

46

46

Fast Adder

- Truth Table of full adder

a	b	c_{in}	s	c_{out}	
0	0	0	0	0	
0	0	1	1	0	
0	1	0	1	0	
0	1	1	0	1	$p_n = a_n \oplus b_n$
1	0	0	1	0	
1	0	1	0	1	
1	1	0	0	1	
1	1	1	1	1	$g_n = a_n\, b_n$

- Note: g_n and p_n are already produced by the full adder circuit!!!
- Adder outputs

  $s_n = a_n \oplus b_n \oplus c_n = p_n \oplus c_n$

  $c_{n+1} = a_n\, b_n + (a_n \oplus b_n)\, c_n = g_n + p_n\, c_n$

47

47

Carry-Lookahead Adder

- To reduce delays to carry outputs, rewrite c_{n+1} in terms of c_0, g, and p signals
  - Bit 0: $c_1 = g_0 + p_0 c_0$
  - Bit 1: $c_2 = g_1 + p_1 c_1$
    $= g_1 + p_1 g_0 + p_1 p_0 c_0$
  - Bit 2: $c_3 = g_2 + p_2 c_2$
    $= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$
  - Bit 3: $c_4 = g_3 + p_3 c_3$
    $= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$
- Result: Carry-lookahead adder

48

48

COMPARE!!

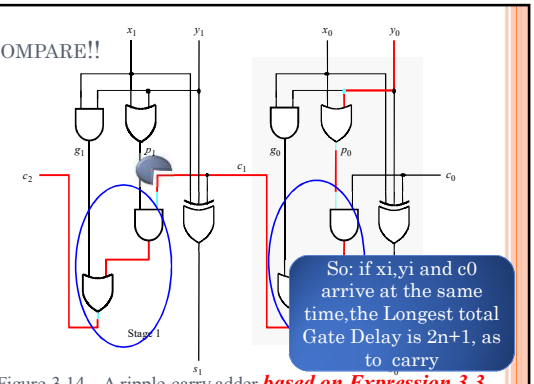


Figure 3.14. A ripple-carry adder ***based on Expression 3.3.***

49

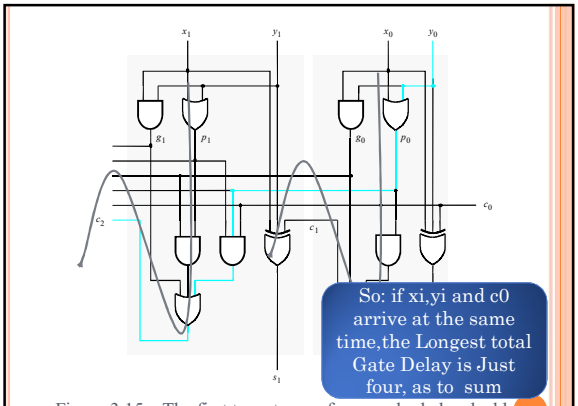


Figure 3.15. The first two stages of a carry-lookahead adder.

50

CARRY-LOOKAHEAD ADDER

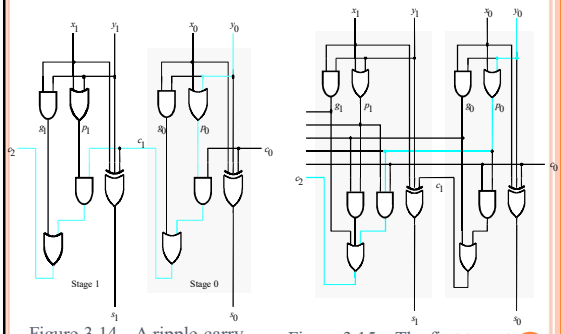
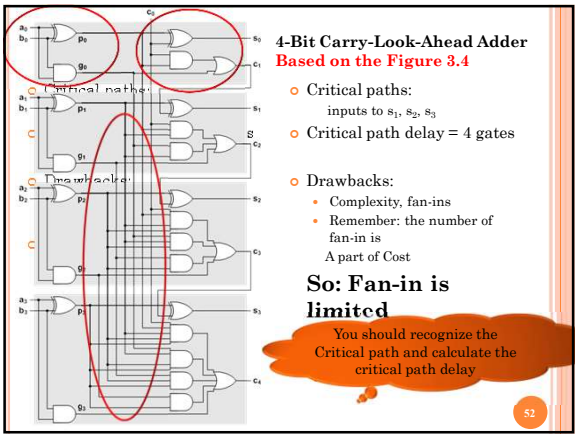


Figure 3.14. A ripple-carry adder based on Expression 3.3.

Figure 3.15. The first two stages of a carry-lookahead adder.

51



52

HOW TO BUILD LARGER ADDER: PARALLELIZATION INSIDE BLOCK SERIALIZATION BETWEEN BLOCKS

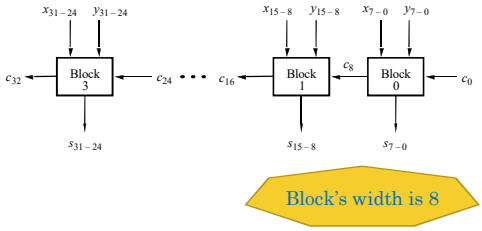
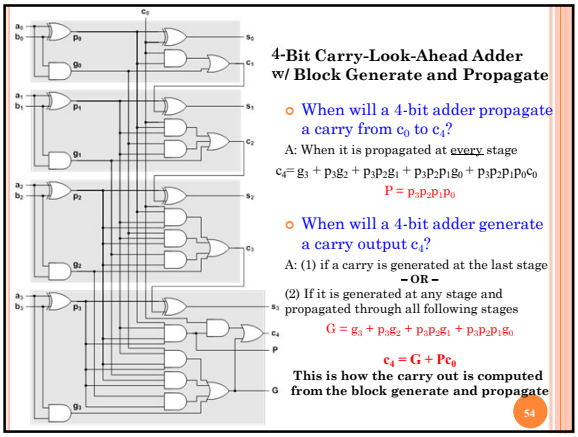


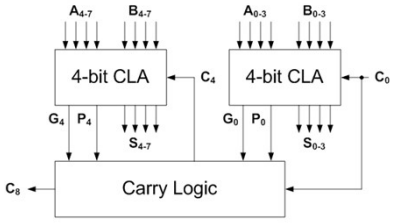
Figure 3.16. A hierarchical carry-lookahead adder with ripple-carry between blocks.

53



54

Carry-Lookahead Adder



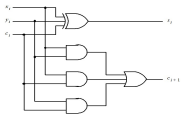
Creating Larger Adders Using
4-bit Carry-Look-Ahead Blocks

55

55

3.5 Design of Arithmetic Circuits Using CAD Tools

How to write the hierarchical code for a ripple-carry adder?



```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   xor (s, x, y, Cin);
   and (z1, x, y);
   and (z2, x, Cin);
   and (z3, y, Cin);
   or (Cout, z1, z2, z3);

endmodule
```

Figure 3.18. Verilog code for the full-adder using gate level primitives.

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   xor (s, x, y, Cin);
   and (z1, x, y) ,
       (z2, x, Cin) ,
       (z3, y, Cin);
   or (Cout, z1, z2, z3);

endmodule
```

Not Suggested!!

Figure 3.19. Another version of Verilog code from Figure 3.18.

56

56

Design of Arithmetic Circuits Using Verilog

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Figure 3.20. Verilog code for the full-adder using continuous assignment.

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin,
          Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Not Suggested!!

Figure 3.21. Another version of Verilog code from Figure 3.20.

57

57

Design of Arithmetic Circuits Using Verilog

```
module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
   input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
   output s3, s2, s1, s0, carryout;

   fulladd stage0 (carryin, x0, y0, s0, c1);
   fulladd stage1 (c1, x1, y1, s1, c2);
   fulladd stage2 (c2, x2, y2, s2, c3);
   fulladd stage3 (c3, x3, y3, s3, carryout);

endmodule

module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin,
   assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Maintain the sequence

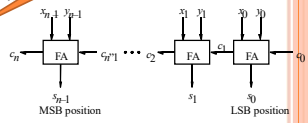


Figure 3.22. Verilog code for a four-bit adder.

58

58

Using Vectored Signals

Multibit signals can be represented in Verilog code as a multibit vector, An example of an input vector is

input [3:0] X; **wire** [3:1] C;

```
module adder4 (carryin, X, Y, S, carryout);
   input carryin;
   input [3:0] X, Y;
   output [3:0] S;
   output carryout;
   wire [3:1] C;

   fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
   fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
   fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
   fulladd stage3 (C[3], X[3], Y[3], S[3], carryout);

endmodule
```

Support partial selection

Figure 3.23. A four-bit adder using vectors.

59

59

Using a Generic Specification

How to define a module that could be used to implement an adder of any size?

Verilog allows the use of general parameters that can be given a specific value as desired.

For example: $X[n-1:0]$.
parameter n = 4;
the bit range of X is [3:0]

Default Value, may be updated When instantiation

60

60

USING A GENERIC SPECIFICATION

```
module addern (carryin, X, Y, S, carryout);
   parameter n=32;
   input carryin;
   input [n-1:0] X, Y;
   output reg [n-1:0] S;
   output reg carryout;
   reg [n:0] C;
   integer k;

   always @(X, Y, carryin)
   begin
      C[0] = carryin;
      for (k = 0; k < n; k = k+1)
      begin
         S[k] = X[k] ^ Y[k] ^ C[k];
         C[k+1] = (X[k] & Y[k]) | (X[k] & C[k]) | (Y[k] & C[k]);
      end
      carryout = C[n];
   end

endmodule
```

'For' statement Not Suggested!!

Figure 3.24. A generic specification of a ripple-carry adder.

61

61

USING THE GENERATE CAPABILITY

```
module addern (carryin, X, Y, S, carryout);
   parameter n=32;
   input carryin;
   input [n-1:0] X, Y;
   output [n-1:0] S;
   output carryout;
   wire [n:0] C;

   genvar i;
   assign C[0] = carryin;
   assign carryout = C[n];

   generate
      for (i = 0; i <= n – 1; i = i+1)
      begin:addbit
         fulladd stage (C[i], X[i], Y[i], S[i],
C[i+1]);
      end
   endgenerate

endmodule
```

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin,
   assign Cout = (x & y) | (x & Cin) | (y
& Cin);

endmodule
```

Figure 3.25. A ripple-carry adder specified by using the **generate** statement.

62

62

NETS AND VARIABLES IN VERILOG

- ***Nets***: Connections between logic elements are defined using *nets*.

1). A net represents a node in a circuit.

2). It can be a *scalar* that represents a single connection or a *vector* that represents multiple connections.

- *Variables:* Signals produced by procedural statements are referred to as *variables*.

1). A variable can be assigned a value in one Verilog statement, and it retains this value until it is overwritten by a subsequent assignment statement.

2). There are two types of variables: **reg** and **integer**.

63

63

ARITHMETIC ASSIGNMENT STATEMENTS

- Verilog implements such operations using arithmetic assignment statements and vectors.

For example: **input** [n−1:0] X, Y;
output [n−1:0] S;
S = X + Y;

```
module addern (carryin, X, Y, S);
   parameter n = 32;
   input carryin;
   input [n-1:0] X, Y;
   output reg [n-1:0] S;

   always @(X, Y, carryin)
      S = X + Y + carryin;

endmodule
```

EDA tool will synthesize the "+" into the connection of basic logic gates

Figure 3.26. Specification of an *n*-bit adder using arithmetic assignment.

64

64

ARITHMETIC ASSIGNMENT STATEMENTS

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n-1:0] X, Y;
   output reg [n-1:0] S;
   output reg carryout, overflow;

   always @(X, Y, carryin)
   begin
      S = X + Y + carryin;
      carryout = (X[n-1] & Y[n-1]) | (X[n-1] & ~S[n-1]) | (Y[n-1] & ~S[n-1]);
      overflow = (X[n-1] & Y[n-1] & ~S[n-1]) | (~X[n-1] & ~Y[n-1] & S[n-1]);
   end

endmodule
```

Figure 3.27. An *n*-bit adder with carry-out and overflow signals.

65

65

ARITHMETIC ASSIGNMENT STATEMENTS

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n-1:0] X, Y;
   output reg [n-1:0] S;
   output reg carryout, overflow;
   reg [n:0] Sum;

   always @(X, Y, carryin)
   begin
      Sum = {1'b0,X} + {1'b0,Y} + carryin;
      S = Sum[n-1:0];
      carryout = Sum[n];
      overflow = (X[n-1] & Y[n-1] & ~S[n-1]) | (~X[n-1] & ~Y[n-1] & S[n-1]);
   end

endmodule
```

Figure 3.28. An alternative specification of an *n*-bit adder with carry-out and overflow signals.

66

66

Arithmetic Assignment Statements

```
module addern (carryin, X, Y, S, carryout, overflow);
    parameter  n = 32;
    input carryin;
    input [n-1:0] X, Y;
    output reg [n-1:0] S;
    output reg carryout, overflow;

    always @(X, Y, carryin)
    begin
        {carryout, S} = X + Y + carryin;
        overflow = (X[n-1] & Y[n-1] & ~S[n-1]) | (~X[n-1] & ~Y[n-1] & S[n-1]);
    end

endmodule
```

Figure 3.29. Simplified complete specification of an *n*-bit adder.

67

67

Arithmetic Assignment Statements

```
module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output reg s, Cout;

    always @(x, y, Cin)
      {Cout, s} = x + y + Cin;

endmodule
```

Figure 3.30. Behavioral specification of a full-adder.

68

68

Module Hierarchy in Verilog Code

```
module adder_hier (A, B, C, D, S, T, overflow);
    input [15:0] A, B;
    input [7:0] C, D;
    output [16:0] S;
    output [8:0] T;
    output overflow;

    wire o1, o2;  // used for the overflow signals

    addern U1 (1'b0, A, B, S[15:0], S[16], o1);
    defparam U1.n = 16;
    addern U2 (1'b0, C, D, T[7:0], T[8], o2);
    defparam U2.n = 8;

    assign overflow = o1 | o2;

endmodule
```

Figure 3.31. An example of setting parameter values in Verilog code.

69

69

Module Hierarchy in Verilog Code

```
module adder_hier (A, B, C, D, S, T, overflow);
    input [15:0] A, B;
    input [7:0] C, D;
    output [16:0] S;
    output [8:0] T;
    output overflow;

    wire o1, o2;  // used for the overflow signals

    addern #(16) U1 (1'b0, A, B, S[15:0], S[16], o1);
    addern #(8) U2 (1'b0, C, D, T[7:0], T[8], o2);

    assign overflow = o1 | o2;

endmodule
```

Figure 3.32. Using the Verilog # operator to set the values of parameters.

70

70

Representation of Numbers in Verilog Code

- Numbers can be given as constants in Verilog code.
- They can be given as **binary (b), octal (o), hexadecimal (h), or decimal (d)** numbers.

12'b100010101001	'b100010110
12'o4251	'o426
12'h8A9	'h116
12'd2217	278

71

71

Representation of Numbers in Verilog Code

- The value of a positive number does not change if 0s are appended as the most-significant bits;
- The value of a negative number does not change if 1s are appended as the most-significant bits.
- Such replication of the sign bit is called *sign extension*.

Suppose that *A* is an eight-bit vector and *B* is a four-bit vector.

```
S = A + B;

S = A + {4{B[3]}, B};
```

72

72

3.6 Multiplication of Unsigned Numbers

Multiplication by hand

Multiplicand M	(14)	1 1 1 0
Multiplier Q	(11)	1 0 1 1
		1 1 1 0
		1 1 1 0
		0 0 0 0
		1 1 1 0
Product P	(154)	1 0 0 1 1 0 1 0

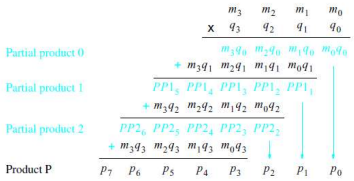
Implementation in hardware

Multiplicand M	(14)	1 1 1 0
Multiplier Q	(11)	1 0 1 1
Partial product 0		1 1 1 0
		+ 1 1 1 0
Partial product 1		1 0 1 0 1
		+ 0 0 0 0
Partial product 2		0 1 0 1 0
		+ 1 1 1 0
Product P	(154)	1 0 0 1 1 0 1 0

73

73

Array Multiplier for Unsigned Numbers



c) Hardware implementation

74

74

Array Multiplier for Unsigned Numbers

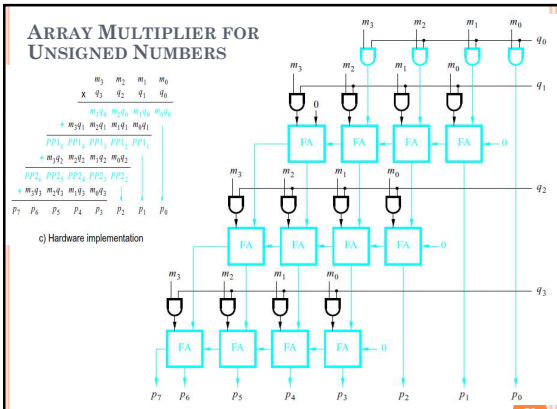


Figure 3.35. A 4x4 multiplier circuit.

75

75

Multiplication of Signed Numbers

Multiplicand M	(+14)	0 1 1 1 0	Multiplicand M	(–14)	1 0 0 1 0
Multiplier Q	(+11)	X 0 1 0 1 1	Multiplier Q	(+11)	X 0 1 0 1 1
Partial product 0		0 0 0 1 1 1 0	Partial product 0		1 1 1 0 0 1 0
		+ 0 0 1 1 1 0			+ 1 1 0 0 1 0
Partial product 1		0 0 1 0 1 0 1	Partial product 1		1 1 0 1 0 1 1
		+ 0 0 0 0 0 0			+ 0 0 0 0 0 0
Partial product 2		0 0 0 1 0 1 0	Partial product 2		1 1 1 0 1 0 1
		+ 0 0 1 1 1 0			+ 1 1 0 0 1 0
Partial product 3		0 0 1 0 0 1 1	Partial product 3		1 1 0 1 1 0 0
		+ 0 0 0 0 0 0			+ 0 0 0 0 0 0
Product P	(+154)	0 0 1 0 0 1 1 0 1 0	Product P	(–154)	1 1 0 1 1 0 0 1 1 0

(b) Negative multiplicand

Figure 3.36. Multiplication of signed numbers.

76

76

3.7 Other Number Representations

- **Fixed-Point Numbers** **skip**

77

77

Binary-Coded-Decimal Representation

- **BCD:** In digital systems it is possible to represent decimal numbers simply by encoding each digit in binary form.

Decimal digit	BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Table 3.3. Binary-coded decimal digits.

78

78

BCD Addition

- If $X + Y \leq 9$, then the addition is the same as the addition of 2 four-bit unsigned binary numbers.
- if $X + Y > 9$, then the result requires two BCD digits.
- Two cases where some correction has to be made:

  Z=X+Y

  If Z ≤9, then S =Z and carry-out=0

  If Z > 9, then S = Z + 6 and carry-out = 1

X	0111	7
+ Y	+ 0101	+ 5
Z	1100	12
	+ 0110	
carry →	10010	
	S = 2	

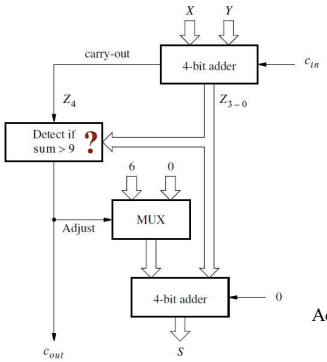
X	1000	8
+ Y	+ 1001	+ 9
Z	10001	17
	+ 0110	
carry →	10111	
	S = 7	

Figure 3.38. Addition of BCD digits.

79

79

BCD Addition--- one-digit BCD adder



Decimal digit	BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Adjust = $C_{out} + z_3 (z_2 + z_2)$

Figure 3.39. Block diagram for a one-digit BCD adder.

80

80

BCD Addition---Circuit

Adjust = $C_{out} + z_3 (z_2 + z_2)$

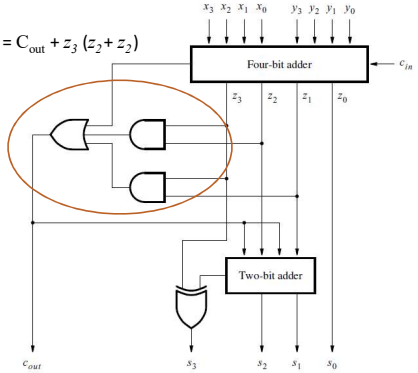


Figure 3.41. Circuit for a one-digit BCD adder.

81

81

BCD Addition--- Verilog code

```
module bcdadd(Cin, X, Y, S, Cout);
   input Cin;
   input [3:0] X,Y;
   output reg [3:0] S;
   output reg Cout;
   reg [4:0] Z;

   always@ (X, Y, Cin)
   begin
      Z = X + Y + Cin;
      if (Z < 10)
         {Cout, S} = Z;
      else
         {Cout, S} = Z + 6;
   end

endmodule
```

Figure 3.40. Verilog code for a one-digit BCD adder.

82

82

3.8 Examples of Solved Problems

83

83

Conclusion

- Understanding the ADDER and fast ADDER
- XOR Gate
- Difference of the ripple-carry and look-ahead carry adder
- Some concept like delay, critical path, fan-in,fan-out ..
- Advanced Verilog

- Assignment: **3.5, 3.7(show->prove),3.14**
- **3.21, 3.22**

84

84