DIGITAL SYSTEM DESIGN USING VERILOG

COURSE CODE: 19EC5DCDSV (3 CREDITS) MODULE-4A

Faculty In-Charge: Dr. Dinesha P

drdinesh-ece@dayanandasagar.edu

Numeric Basics:

Unsigned and Signed Integers, Fixed and Floating-point Numbers. Design examples: BCD to 7 segment decoder, 32 bit adders, Shift and add Multiplier, Array Multiplier, signed integer / fraction multiplier.

[Text Book 1 and 2]

Numeric Basics

- One of the most common kinds of information processed by digital systems is numeric information.
  - unsigned integers
  - signed integers
  - fixed-point real numbers
  - floating-point real numbers
  - complex numbers

Unsigned Integers

- Non-negative numbers (including 0)
  - Represent real-world data
    - e.g., temperature, position, time, …
  - Also used in controlling operation of a digital system
    - e.g., counting iterations, table indices

CODING UN SIGNED INTEGERS

- Coded using unsigned binary (base 2) representation
  - analogous to decimal representation

Binary Representation

- Decimal: base 10
  - $124_{10} = 1\times10^2 + 2\times10^1 + 4\times10^0$
- Binary: base 2
  - 124_{10}
    $= 1\times2^6+1\times2^5+1\times2^4+1\times2^3+1\times2^2+0\times2^1+0\times2^0$
    $= 1111100_2$
- In general, a number x is represented using n bits as x_{n-1}, x_{n-2}, …, x_0, where

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_0 2^0$$

EXAMPLE 3.1 What number is represented by the unsigned binary number 101101_2?

SOLUTION Express the number as a sum of powers of two and calculate the result:

$$\begin{aligned}101101_2 &= 1\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0\\ &= 1\times32 + 0\times16 + 1\times8 + 1\times4 + 0\times2 + 1\times1\\ &= 45_{10}\end{aligned}$$

Binary Representation

- Unsigned binary is a code for numbers
  - n bits: represent numbers from 0 to $2^n - 1$
    - 0: 0000…00; $2^n - 1$: 1111…11
  - To represent x: $0 \leq x \leq N - 1$, need $\lceil \log_2 N \rceil$ bits
- Computers use
  - 8-bit bytes: 0, …, 255
  - 32-bit words: 0, …, ~4 billion
- Digital circuits can use what ever size is appropriate

EXAMPLE 3.2 Suppose we are designing a scientific instrument to measure the time interval between two random events very precisely, with a resolution of nanoseconds (1ns = 10^{-9} seconds). Events may occur as much as a day apart. How many bits are needed to represent the interval as a number of nanoseconds?

SOLUTION There are 10^9 nanoseconds per second, and $60 \times 60 \times 24 = 86{,}400$ seconds per day, so the largest number we need to allow for is 8.64×10^{13}. The number of bits needed is

$$\lceil \log_2(8.64 \times 10^{13}) \rceil = \left\lceil \frac{\log(8.64 \times 10^{13})}{\log 2} \right\rceil = \lceil 46.296\ldots \rceil = 47$$

So at least 47 bits are needed.

- Develop a Verilog model of a 4-to-1 multiplexer that selects among four unsigned 6-bit integers.
- Use vectors as the representation
  - Can apply arithmetic operations

```
module multiplexer_6bit_4_to_1
  ( output reg [5:0] z,
    input      [5:0] a0, a1, a2, a3,
    input      [1:0] sel );
  always @*
    case (sel)
      2'b00: z = a0;
      2'b01: z = a1;
      2'b10: z = a2;
      2'b11: z = a3;
    endcase
endmodule
```

Octal and Hexadecimal

- Short-hand notations for vectors of bits
- Octal (base 8)
  - Each group of 3 bits represented by a digit
  - 0: 000, 1:001, 2: 010, …, 7: 111
  - $253_8 = 010\ 101\ 011_2$
  - $11001011_2 \Rightarrow 11\ 001\ 011_2 = 313_8$
- Hex (base 16)
  - Each group of 4 bits represented by a digit
  - 0: 0000, …, 9: 1001, A: 1010, …, F: 1111
  - $3CE_{16} = 0011\ 1100\ 1110_2$
  - $11001011_2 \Rightarrow 1100\ 1011_2 = CB_{16}$

Examples: Octal to decimal conversion:

$$253_8 = 2\times 8^2 + 5\times 8^1 + 3\times 8^0$$
$$= 2\times 64 + 5\times 8 + 3\times 1$$
$$= 128 + 40 + 3 = 171_{10}$$

Octal to Binary conversion:

$$253_8 = 2\times 8^2 + 5\times 8^1 + 3\times 8^0$$
$$= (0\times 2^2 + 1\times 2^1 + 0\times 2^0)\times 8^2 + (1\times 2^2 + 0\times 2^1 + 1\times 2^0)\times 8^1 + (0\times 2^2 + 1\times 2^1 + 1\times 2^0)\times 8^0$$
$$= (0\times 2^2 + 1\times 2^1 + 0\times 2^0)\times 2^6 + (1\times 2^2 + 0\times 2^1 + 1\times 2^0)\times 2^3 + (0\times 2^2 + 1\times 2^1 + 1\times 2^0)\times 2^0$$
$$= (0\times 2^8 + 1\times 2^7 + 0\times 2^6) + (1\times 2^5 + 0\times 2^4 + 1\times 2^3) + (0\times 2^2 + 1\times 2^1 + 1\times 2^0)$$
$$= 010101011_2$$

Binary to octal: $11001011_2 \Rightarrow 11\ 001\ 011 \Rightarrow 313_8$

- Hexadecimal is another form of positional number system, like octal, but based on powers of 16.
- i.e 0 to 9 and $A_{16} = 10_{10}$ $B_{16} = 11_{10}$ $C_{16} = 12_{10}$ $D_{16} = 13_{10}$ $E_{16} = 14_{10}$ $F_{16} = 15_{10}$
- Hexa decimal to Decimal conversion:

Thus, for example,

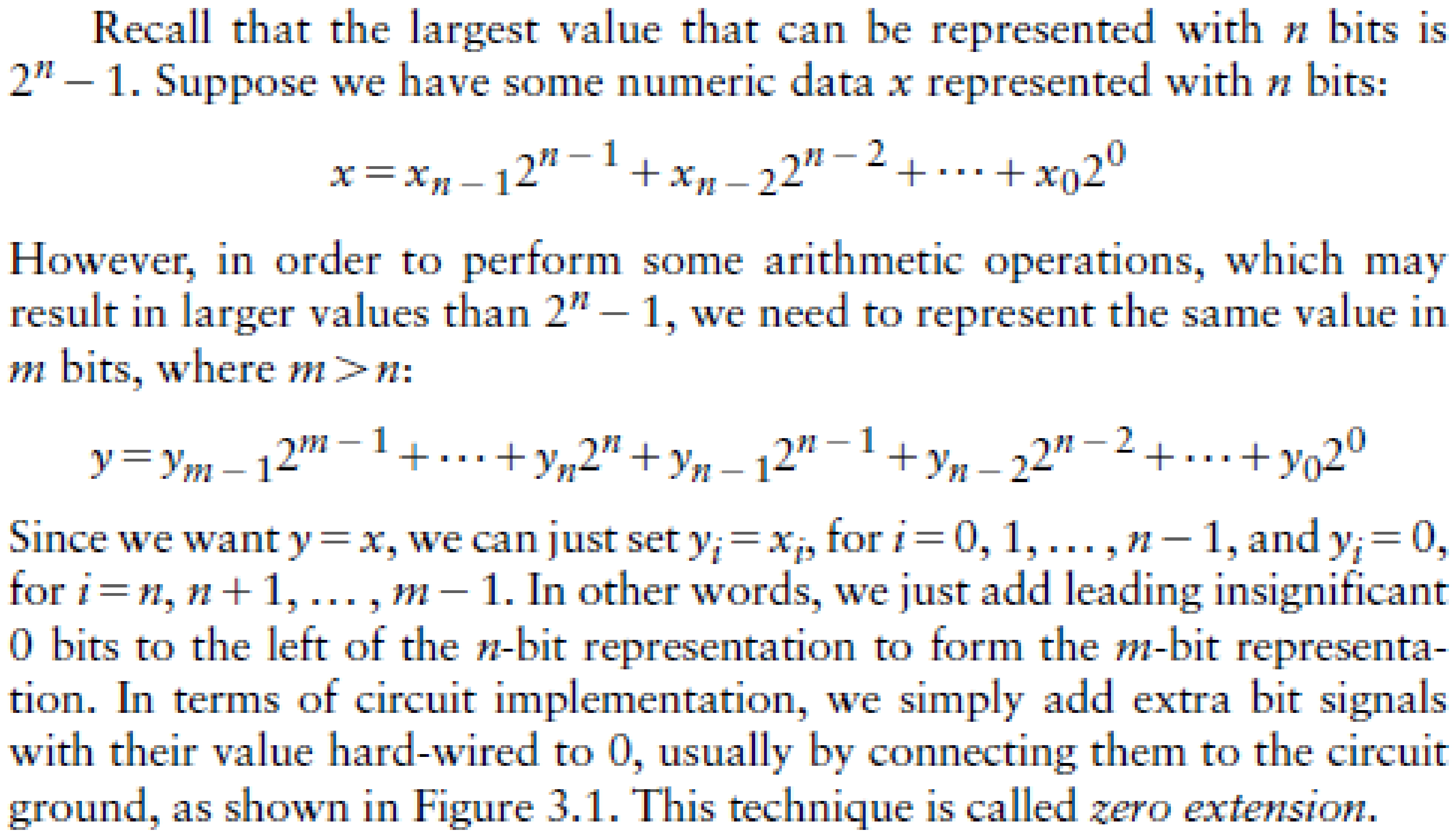
$$3CE_{16} = 3 \times 16^2 + 12 \times 16^1 + 14 \times 16^0$$
$$= 3 \times 256 + 12 \times 16 + 14 \times 1$$
$$= 768 + 192 + 14 = 974_{10}$$

- Binary to Hexa decimal conversion:

$$11001011_2 \Rightarrow 1100\ 1011 \Rightarrow CB_{16}$$

OPERATIONS ON UNSIGNED INTEGERS
EXTENDING UNSIGNED NUMBERS

- To extend an *n*-bit number to *m* bits
  - Add leading 0 bits
  - e.g., $72_{10} = 1001000 = 000001001000$



Recall that the largest value that can be represented with n bits is $2^n - 1$. Suppose we have some numeric data x represented with n bits:

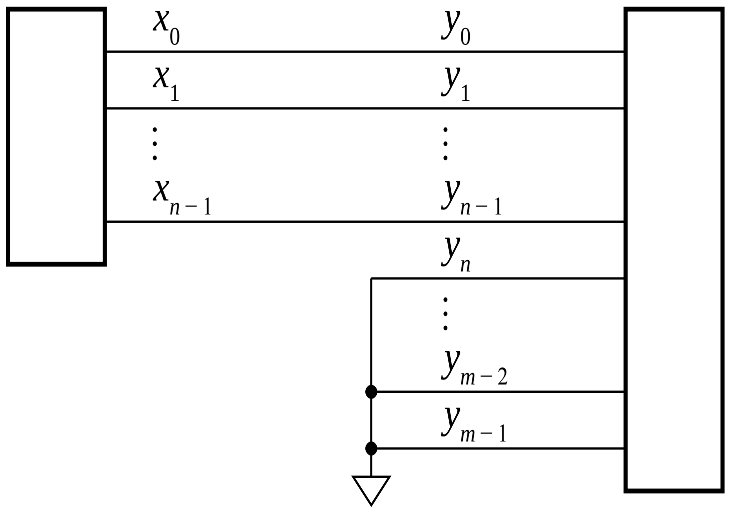
$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_0 2^0$$

However, in order to perform some arithmetic operations, which may result in larger values than $2^n - 1$, we need to represent the same value in m bits, where $m > n$:

$$y = y_{m-1}2^{m-1} + \cdots + y_n 2^n + y_{n-1}2^{n-1} + y_{n-2}2^{n-2} + \cdots + y_0 2^0$$

Since we want $y = x$, we can just set $y_i = x_i$, for $i = 0, 1, \ldots, n-1$, and $y_i = 0$, for $i = n, n+1, \ldots, m-1$. In other words, we just add leading insignificant 0 bits to the left of the n-bit representation to form the m-bit representation. In terms of circuit implementation, we simply add extra bit signals with their value hard-wired to 0, usually by connecting them to the circuit ground, as shown in Figure 3.1. This technique is called *zero extension*.

Implementation of zero extension in a circuit.



We can express zero extension in a Verilog model by concatenating a string of 0 bits to the left of a vector representing an unsigned integer. For example, given nets declared as

```
wire [3:0] x;
wire [7:0] y;


assign y = {4'b0000, x};
```

```
assign y = {4'b0, x};
```

Truncating Unsigned Numbers

- To truncate from m bits to n bits
  - Discard leftmost bits
  - Value is preserved if discarded bits are 0
  - Result is x mod 2^n

An alternative view of truncation of y from m bits to n bits is that it implements the operation $y \bmod 2^n$. We can demonstrate this as follows:

$$y \bmod 2^n$$

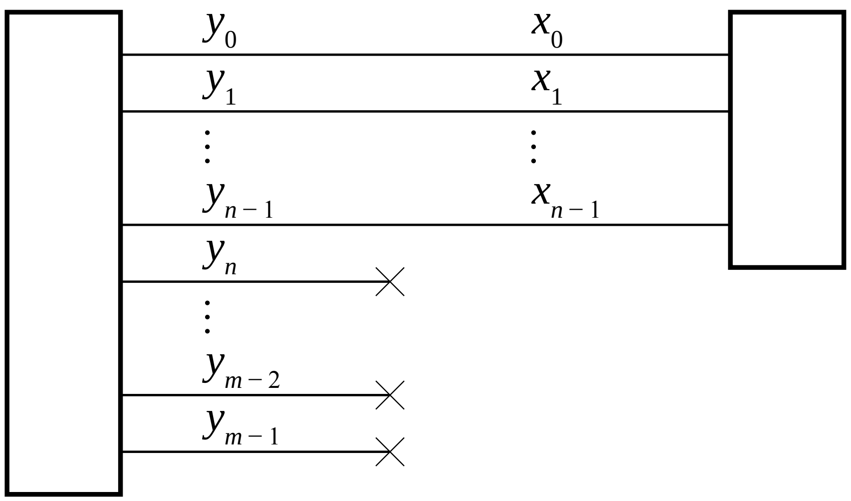
$$= (y_{m-1}2^{m-1} + \dots + y_n 2^n + y_{n-1}2^{n-1} + \dots + y_0 2^0) \bmod 2^n$$

$$= ((y_{m-1}2^{m-n-1} + \dots + y_n 2^0)2^n + y_{n-1}2^{n-1} + \dots + y_0 2^0) \bmod 2^n$$

$$= y_{n-1}2^{n-1} + \dots + y_0 2^0$$

Thus, if we want to compute y *mod* 2^n, *we just truncate* y *to* n *bits,* regardless of the values of any of the discarded bits.

Implementation of truncation in a circuit.



```
assign x = y[3:0];
```

```
assign y = x;
```

Addition of Unsigned Integers

- Performed in the same way as decimal

$$\begin{array}{ccccccccccc}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & \\
 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
\hline
 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0
\end{array}
\qquad
\begin{array}{cccccc}
1 & 1 & 0 & 0 & 1 & \\
 & 0 & 1 & 0 & 0 & 1 \\
 & 1 & 1 & 1 & 0 & 1 \\
\hline
1 & 0 & 0 & 1 & 1 & 0
\end{array}$$

carry bits

overflow

Addition Circuits

- Half adder
  - for least-significant bits

$$s_0 = x_0 \oplus y_0$$

$$c_1 = x_0 \cdot y_0$$

- Full adder
  - for remaining bits

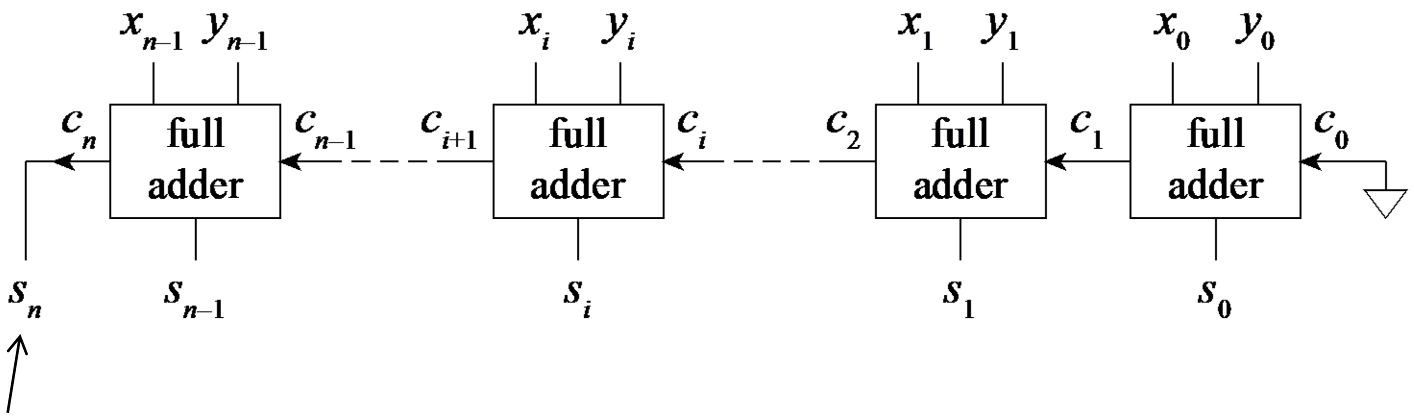
$$s_i = \left(x_i \oplus y_i\right) \oplus c_i$$

$$c_{i+1} = x_i \cdot y_i + \left(x_i \oplus y_i\right) \cdot c_i$$

x_i	y_i	c_i	s_i	c_{i+1}
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

Ripple-Carry Adder

- Full adder for each bit, $c_0 = 0$



overflow

- Worst-case delay
  - from x_0, y_0 to s_n
  - carry must ripple through intervening stages, affecting sum bits

Improving Adder Performance

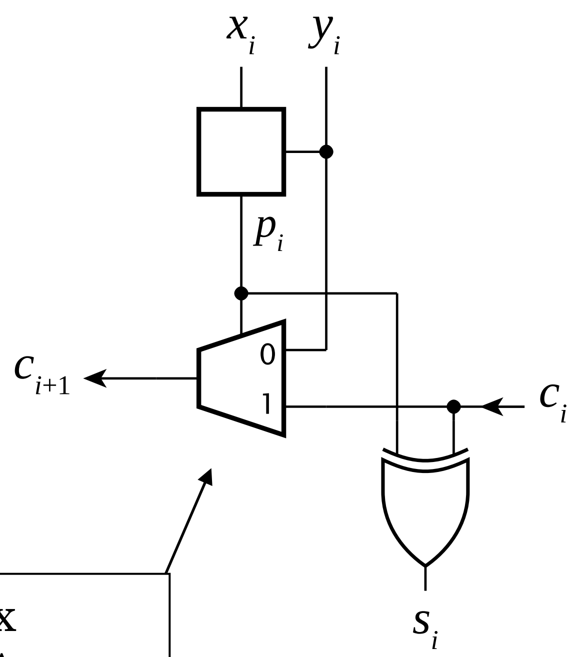
- Carry kill: $k_i = \overline{x_i} \cdot \overline{y_i}$
- Carry propagate: $p_i = x_i \oplus y_i$
- Carry generate: $g_i = x_i \cdot y_i$
- Adder equations

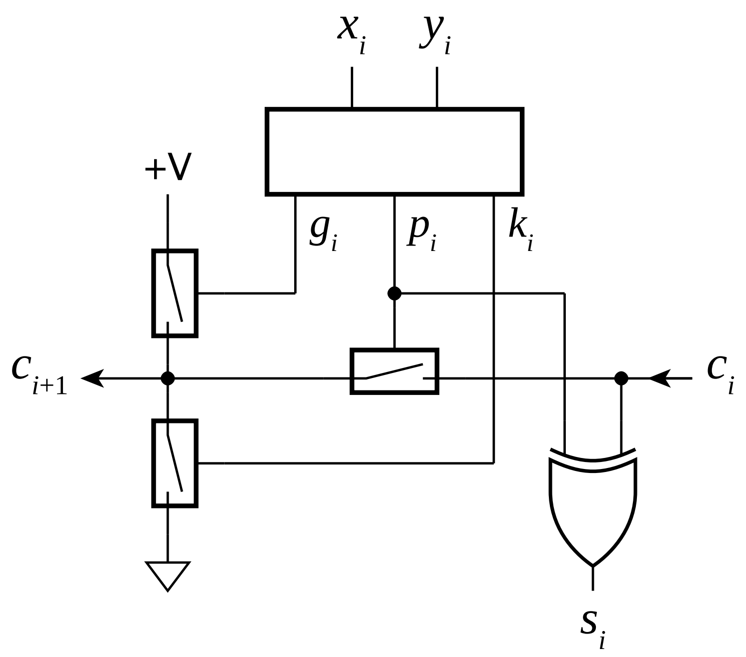
$$s_i = p_i \oplus c_i \qquad c_{i+1} = g_i + p_i \cdot c_i$$

x_i	y_i	c_i	s_i	c_{i+1}
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	1	1
1	1	1	1	1

Fast-carry-chain full-adder cells.

- Also called Manchester adder





Xilinx FPGAs include this structure

Carry Look ahead

$$\boxed{c_{i+1} = g_i + p_i \cdot c_i}$$

$$c_1 = g_0 + p_0 \cdot c_0$$

$$c_2 = g_1 + p_1 \cdot \left(g_0 + p_0 \cdot c_0\right) = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0$$

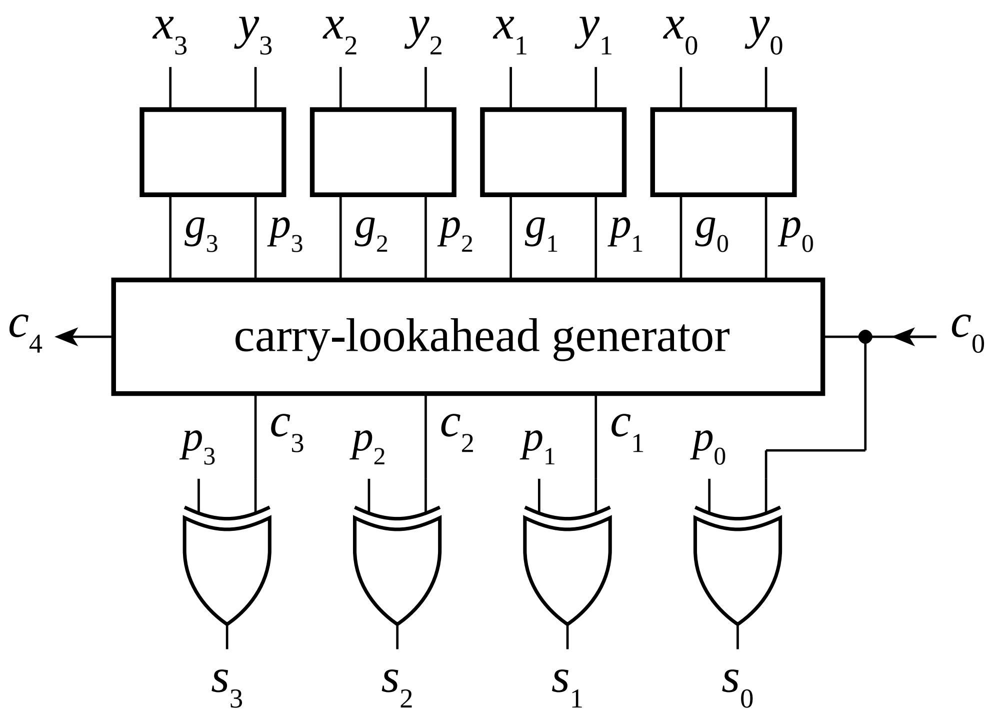
$$c_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0$$

$$c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1$$

$$+ p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$$

Carry-Lookahead Adder

- Avoids chained carry circuit



x_3 y_3 x_2 y_2 x_1 y_1 x_0 y_0
g_3 p_3 g_2 p_2 g_1 p_1 g_0 p_0
c_4
carry-lookahead generator
c_0
p_3 c_3 p_2 c_2 p_1 c_1 p_0
s_3 s_2 s_1 s_0

- Use multilevel lookahead for wider numbers

Other Optimized Adders

- Other adders are based on other reformulations of adder equations
- Choice of adder depends on constraints
  - e.g., ripple-carry has low area, so is ok for low performance circuits
  - e.g., Manchester adder ok in FPGAs that include carry-chain circuits

Adders in Verilog

- Use arithmetic "+" operator

Given the Verilog declaration of three nets:

```
wire [7:0] a, b, s;
```

…

Verilog statement to assign the sum of a and b to s.

```
assign s = a + b;
```

Revise the statements to produce a carry-out bit, c.

```
wire [8:0] tmp_result;
wire       c;
```

…

The required statements are

```
assign tmp_result = {1'b0, a} + {1'b0, b};
assign c          = tmp_result[8];
assign s          = tmp_result[7:0];
```

An alternative way of writing these assignments is

```
assign {c, s} = {1'b0, a} + {1'b0, b};
```

```
assign {c, s} = a + b;
```

EXAMPLE 3.8 Revise the declaration and statement in Example 3.6 to use `integer` variables instead of vector nets.

SOLUTION The revised declaration is

```
integer a, b, s;
```

Since we are using variables instead of nets, the assignment must be in a procedural block. We replace the assignment statement with the always block:

```
always @*
  s = a + b;
```

Unsigned Subtraction

- As in decimal

$$
\begin{array}{lccccccccc}
b: & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & \\
x: & & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
y: & - & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
\hline
d: & & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0
\end{array}
$$

borro
w bits

Subtraction Circuits

- For least-significant bits

$$d_0 = x_0 \oplus y_0$$

$$b_1 = \overline{x_0} \cdot y_0$$

- For remaining bits

$$d_i = (x_i \oplus y_i) \oplus b_i$$

$$b_{i+1} = \overline{x_i} \cdot y_i + \overline{(x_i \oplus y_i)} \cdot b_i$$

x_i	y_i	b_i	s_i	b_{i+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	1	0
1	1	1	1	1

Adder/Subtracter Circuits

- Many systems add and subtract
  - Trick: use complemented borrows

<u>Addition</u>

$$s_i = (x_i \oplus y_i) \oplus c_i$$

$$c_{i+1} = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_i$$

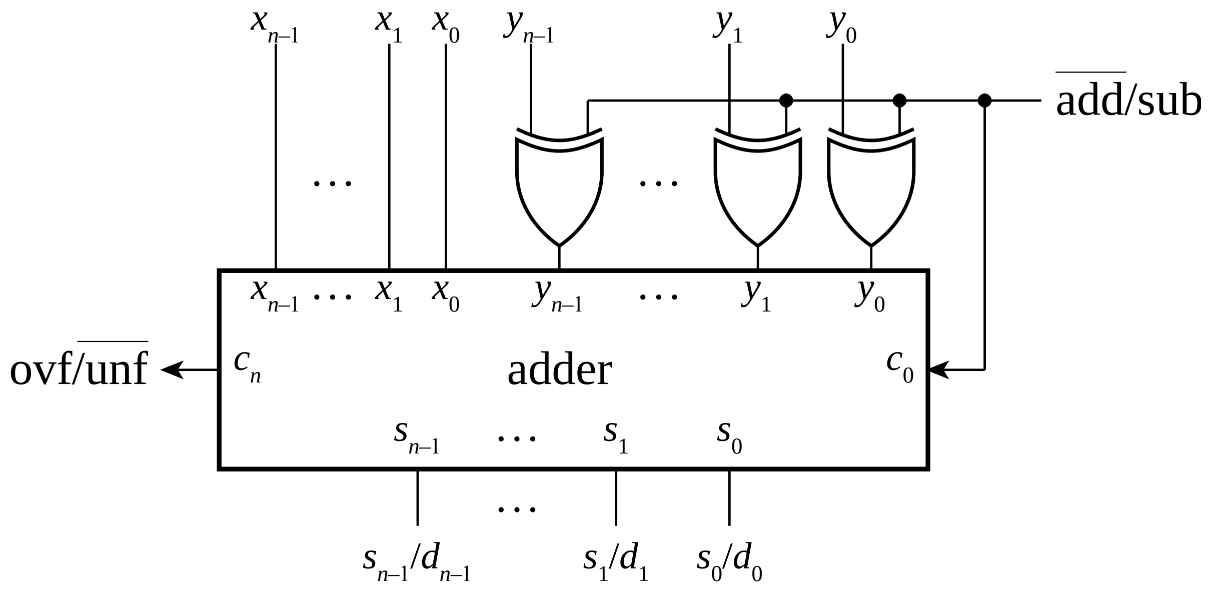
<u>Subtraction</u>

$$d_i = (x_i \oplus \overline{y_i}) \oplus \overline{b_i}$$

$$\overline{b_{i+1}} = x_i \cdot \overline{y_i} + (x_i \oplus \overline{y_i}) \cdot \overline{b_i}$$

- Same hardware can perform both
  - For subtraction: complement y, set $\overline{b_0} = 1$

Adder/Subtracter Circuits



- Adder can be any of those we have seen
  - depends on constraints

Develop a Verilog behavioral model of an adder/subtracter for 12-bit unsigned binary numbers. The circuit has data inputs x and y, a data output s, a control input mode that is 0 for addition and 1 for subtraction, and an output ovf_unf that is 1 when an addition overflow or a subtraction underflow occurs.

```
module adder_subtracter ( output [11:0] s,
                          output        ovf_unf,
                          input  [11:0] x, y,
                          input         mode );
  assign {ovf_unf, s} = !mode ? (x + y) : (x - y);
endmodule
```

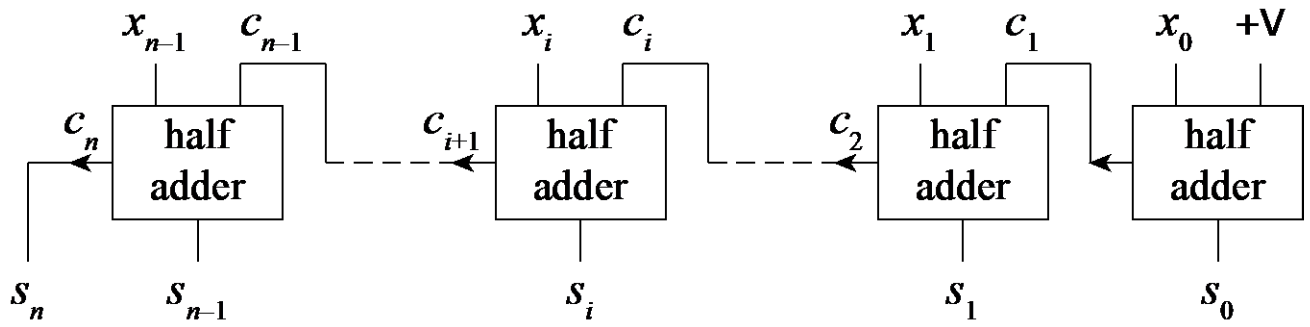
Increment and Decrement

The increment operation involves adding the constant value 1, and the decrement operation involves subtracting the constant value 1.

- A straightforward way to design an increment circuit would be to use an adder with one operand input hard wired to the unsigned binary representation of 1, namely, 0 . . . 001. Alternatively, we could hard wire one input to the representation of 0 and the carry in to 1

$$s_i = x_i \oplus c_i \qquad c_{i+1} = x_i \cdot c_i$$

  - These are equations for a half adder



  - Similarly for decrementing: subtracting 1

Increment/Decrement in Verilog

- In Verilog models, we can express the increment or decrement operation by adding or subtracting the literal value 1 to an operand.

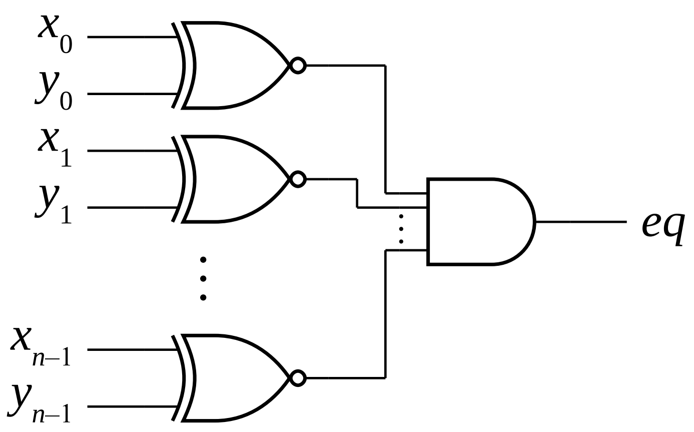
```
wire [15:0] x, s;
...

assign s = x + 1;  // increment x

assign s = x - 1;  // decrement x
```

- Note: 1 (integer), not 1'b1 (bit)
    - Automatically resized

Comparison of Unsigned Integers

In some applications, it may be necessary to compare two unsigned binary integers for equality or inequality.

- XNOR gate: equality of two bits
  - Apply bitwise to two unsigned numbers

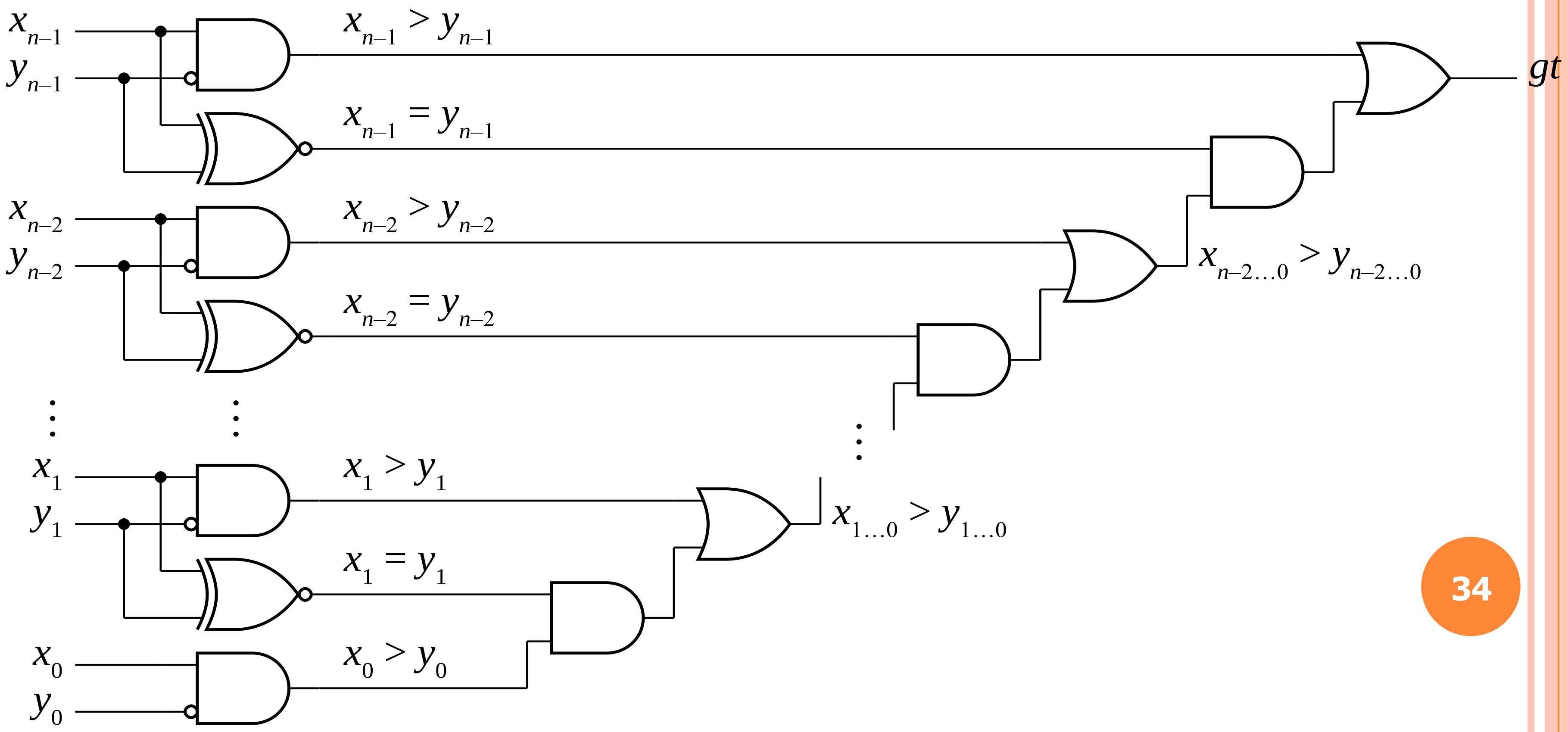


- In Verilog, `x == y` gives a bit result
  - 1'b0 for false, 1'b1 for true

```
assign eq = x == y;
```

Inequality Comparison

- Comparing two unsigned binary integers for inequality (greater than or less than comparison) using Magnitude comparator.
- To test whether a number x is greater than another number y, we can start by comparing the most significant bits, x_{n-1} and y_{n-1}. If $x_{n-1} > y_{n-1}$, we know immediately that x>y. Similarly, if $x_{n-1} < y_{n-1}$, we know immediately that x<y.



Develop a Verilog model for a thermostat that has two 8-bit unsigned binary inputs representing the target temperature and the actual temperature in degrees Fahrenheit (˚F). Assume that both temperatures are above freezing (32˚F). The detector has two outputs: one to turn a heater on when the actual temperature is more than 5˚F below target, and one to turn a cooler on when the actual temperature is more than 5˚F above target.

- Thermostat with target temperature
  - Heater or cooler on when actual temperature is more than 5° from target

```
module thermostat ( output       heater_on, cooler_on,
                    input  [7:0] target, actual );
  assign heater_on = actual < target - 5;
  assign cooler_on = actual > target + 5;
endmodule
```

Scaling by a Constant Power of 2

Scaling an unsigned integer by a given constant value that is a power of 2.

The value x represented by the n bits $x_{n-1}, x_{n-2}, \ldots, x_0$ is

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_0 2^0$$

If we multiply both sides by 2, we get

$$2x = x_{n-1}2^{n} + x_{n-2}2^{n-1} + \cdots + x_0 2^1 + (0)2^0$$

which is an n+1 bit number consisting of the bits of x, shifted left by one position, and a 0 bit appended as the least significant bit.

This operation is called a logical shift left by one position.

To scale by a factor of 2^k, we repeat the scaling-by-2 process k times.

$$2^k x = x_{n-1} 2^{k+n-1} + x_{n-2} 2^{k+n-2} + \cdots + x_0 2^k + (0) 2^{k-1} + \cdots + (0) 2^0$$

- This is *x* shifted left *k* places, with *k* bits of 0 added on the right
  - *logical shift left* by *k* places
  - e.g., $00010110_2 \times 2^3 = 00010110000_2$
- Truncate if result must fit in *n* bits
  - overflow if any truncated bit is not 0

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_0 2^0$$

Similarly, dividing by 2. If we divide both sides of Equation by 2 we get

$$x/2 = x_{n-1}2^{n-2} + x_{n-2}2^{n-3} + \ldots + x_1 2^0 + x_0 2^{-1}$$

Since 2^{-1} is the fraction ½, and we are dealing with integers only, then discard the last term in this equation. The result is an n-1 bit number consisting of the bits of x, except for the least significant bit, shifted right by one position.
This operation is called a logical shift right by one position.

To divide by 2^k, we shift the bits right by k positions, discarding the k least significant bits and appending k bits of 0 at the most significant end.

$$x/2^k = x_{n-1}2^{n-1-k} + x_{n-2}2^{n-2-k} + \cdots + x_k 2^0 + \cancel{x_{k-1}2^{-1}} \cancel{+ \cdots +} \cancel{x_0 2^{-k}}$$

- This is x shifted right k places, with k bits truncated on the right
  - *logical shift right* by k places
  - e.g., $01110110_2 / 2^3 = 01110_2$
- Fill on the left with k bits of 0 if result must fit in n bits

Scaling in Verilog

- Shift-left (<<) and shift-right (>>) operations
  - result is same size as operand

$s = 00010011_2 = 19_{10}$

$\Downarrow$

```
assign y = s << 2;
```

$\Downarrow$

$y = 01001100_2 = 76_{10}$

$s = 00010011_2 = 19_{10}$

$\Downarrow$

```
assign y = s >> 2;
```

$\Downarrow$

$y = 000100_2 = 4_{10}$

Unsigned Multiplication

A straightforward approach for multiplying x by y is to expand the product out as follows

$$xy = x\left(y_{n-1}2^{n-1} + y_{n-2}2^{n-2} + \cdots + y_0 2^0\right)$$

$$= y_{n-1}x2^{n-1} + y_{n-2}x2^{n-2} + \cdots + y_0 x2^0$$

- $y_i x\ 2^i$ is called a partial product
  - if $y_i = 0$, then $y_i x\ 2^i = 0$
  - if $y_i = 1$, then $y_i x\ 2^i$ is x shifted left by i
- Combinational array multiplier
  - AND gates form partial products
  - adders form full product

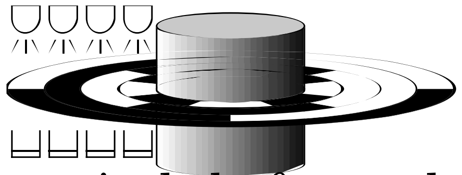
Other Unsigned Operations

- Division, remainder
  - More complicated than multiplication
  - Large circuit area, power
- Complicated operations are often performed sequentially
  - in a sequence of steps, one per clock cycle
  - cost/performance/power trade-off

GRAY CODES

The binary code natural code to use when we need to perform arithmetic operations. However, it has some disadvantages in other applications. Consider a scenario in which we are to design a system that uses a binary code to represent the angular position of a rotating shaft. A common way to measure the position is with a shaft encoder, illustrated in Figure.

- The disk attached to the shaft has a number of concentric bands, each of which has opaque parts and transparent parts. For each band, there is a light emitter and a detector. The detector output is 1 when the light shines through the transparent part of the band and 0 when the light is obscured by the opaque part of the band. The collection of four decoder outputs forms a binary code for the angular position of the shaft.


An optical shaft encoder.

- The pattern of transparency and opacity in the bands on the disk is shown in Figure, and corresponds to a 4-bit Gray code, in which adjacent code words differ by only one bit.
- A complete rotation is divided into 16 segments, and between any two adjacent segments, exactly one band changes between transparent and opaque. This prevents any minor error in positioning of the detectors from causing incorrect position codes.

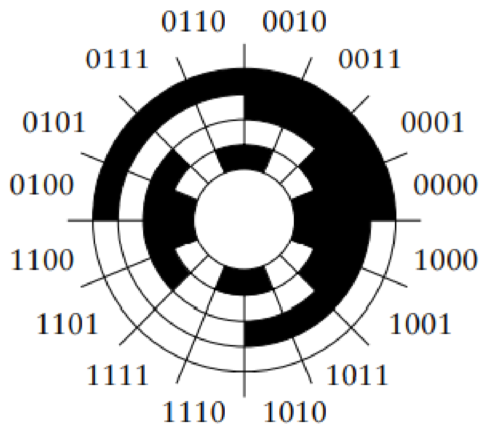


FIGURE 3.15 Gray code pattern on a shaft-encoder disk.

The 4-bit Gray code is listed, along with the corresponding decimal and unsigned binary codes, in Table.

DECIMAL	UNSIGNED BINARY	GRAY CODE
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

The code we have used here is generated by the following rules, which allow us to generate an n-bit Gray code:

- A 1-bit Gray code has the two code words 0 and 1.
- The first 2^{n-1} code words of an n-bit Gray code consist of the code words of an (n-1)-bit Gray code, in order, each with a 0 bit appended as the left-most bit.
- The last 2^{n-1} code words of an n-bit Gray code consist of the code words of an (n-1)-bit Gray code, in reverse order, each with a 1 bit appended as the left-most bit.

- Develop a Verilog model of a code converter to convert the 4-bit Gray code to a 4-bit unsigned binary integer.

```
module gray_converter ( output reg [3:0] numeric_value,
                        input      [3:0] gray_value );

  always @*
    case (gray_value)
      4'b0000: numeric_value = 4'b0000;
      4'b0001: numeric_value = 4'b0001;
      4'b0011: numeric_value = 4'b0010;
      4'b0010: numeric_value = 4'b0011;
      4'b0110: numeric_value = 4'b0100;
      4'b0111: numeric_value = 4'b0101;
      4'b0101: numeric_value = 4'b0110;
      4'b0100: numeric_value = 4'b0111;
      4'b1100: numeric_value = 4'b1000;
      4'b1101: numeric_value = 4'b1001;
      4'b1111: numeric_value = 4'b1010;
      4'b1110: numeric_value = 4'b1011;
      4'b1010: numeric_value = 4'b1100;
      4'b1011: numeric_value = 4'b1101;
      4'b1001: numeric_value = 4'b1101;
      4'b1000: numeric_value = 4'b1111;
    endcase

endmodule
```

Signed Integers:

CODING SIGNED INTEGERS

- The encoding used in digital systems for signed integers is called 2s complement. It is a special case of radix complement representation in which the radix is 2.
- A signed number is represented in 2s-complement form as a weighted sum of powers of two, in a similar way to unsigned binary representation.

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_0 2^0$$

- Positive and negative numbers (and 0)
- *n*-bit *signed magnitude* code
  - 1 bit for sign: $0 \Rightarrow +$, $1 \Rightarrow -$
  - $n - 1$ bits for magnitude
- Signed-magnitude rarely used for integers now
  - circuits are too complex
- Use *2s-complement* binary code

- Most-negative number
  - $1000\ldots0 = -2^{n-1}$
- Most-positive number
  - $0111\ldots1 = +2^{n-1} - 1$
- $x_{n-1} = 1 \Rightarrow$ negative,
  $x_{n-1} = 0 \Rightarrow$ non-negative
  - Since

$$2^{n-2} + \cdots + 2^0 = 2^{n-1} - 1$$

2s-Complement Examples

EXAMPLE 3.14 What values are represented by the 8-bit 2s-complement numbers 00110101 and 10110101?

SOLUTION The first number is

$$1 \times 2^5 + 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^0 = 32 + 16 + 4 + 1 = 53$$

The second number is

$$-1 \times 2^7 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^0 = -128 + 32 + 16 + 4 + 1 = -75$$

Signed Integers in Verilog

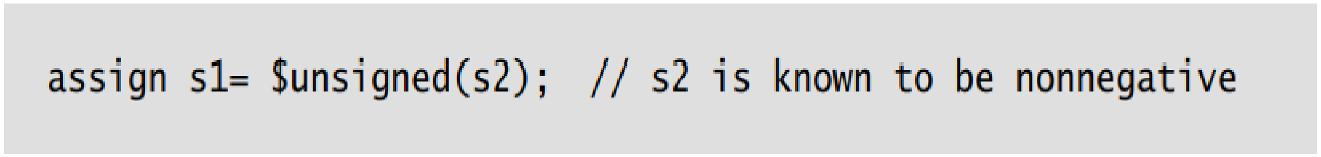
- Use signed vectors

```
wire signed [ 7:0] a;
reg  signed [13:0] b;
```

- Can convert between signed and unsigned interpretations

```
wire        [11:0] s1;
wire signed [11:0] s2;
...
assign s2 = $signed(s1);  // s1 is known to be
                          // less than 2**11

```

- Similarly, if we want to interpret values declared signed as representing unsigned values, we use the $unsigned conversion operation, for example:



```
assign s1= $unsigned(s2);  // s2 is known to be nonnegative
```

Octal and Hex Signed Integers

- Don't think of signed octal or hex
  - Just treat octal or hex as shorthand for a vector of bits
- E.g., 844_{10} is 001101001100
  - In hex: 0011 0100 1100 ⇒ 34C
- E.g., -42_{10} is 1111010110
  - In octal: 1 111 010 110 ⇒ 1726 (10 bits)

Example The 12-bit 2s-complement representation of 844_{10} is 001101001100. Express the bit vector in hexadecimal.

Solution: Dividing into groups of four bits, we get 0011 0100 1100. Substituting hexadecimal digits for the 4-bit groups gives $34C_{16}$.

- Example: The 10-bit 2s-complement representation of -42 is 1111010110. Express the bit vector in octal.

Solution: Dividing into groups of three bits, we get 1 111 010 110. Substituting octal digits for the 3-bit groups gives 1726_8. When reading this octal number, we need to understand that it represents 10 bits. The right-most three digits represent 9 bits, and the left-most digit represents just one bit, the sign bit. Since the sign bit is 1, the number is negative, even though the octal number does not include a - sign.

Resizing Signed Integers

- To extend a non-negative number
  - Add leading 0 bits
  - e.g., 53_{10} = 00110101 = 000000110101
- To truncate a non-negative number
  - Discard leftmost bits, provided
    - discarded bits are all 0
    - sign bit of result is 0
  - E.g., 41_{10} is 00101001
    - Truncating to 6 bits: 101001 — error!

Resizing Signed Integers

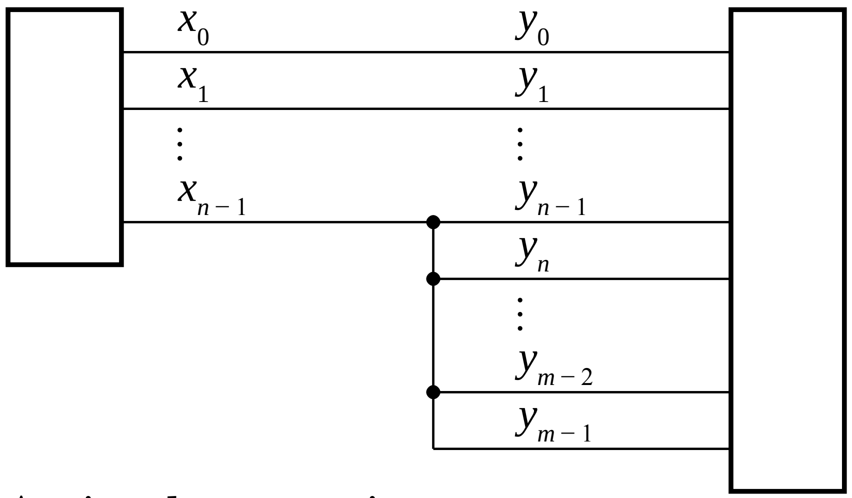
- To extend a negative number
  - Add leading 1 bits
    - See textbook for proof
  - e.g., -75_{10} = 10110101 = 111110110101
- To truncate a negative number
  - Discard leftmost bits, provided
    - discarded bits are all 1
    - sign bit of result is 1

Resizing Signed Integers

- for a 2s-complement signed integer, extending to a greater length involves replicating the sign bit to the left. This is called *sign extension, and preserves the numeric value, be it positive or negative.*

  - Truncate by discarding leading bits

- We can truncate by discarding the left-most bits, provided all of the discarded bits and the resulting sign bit are the same as the original sign bit. The circuit implementation for truncation from *m bits to n bits is the same as for truncation of an unsigned* value, shown



```
wire signed [ 7:0] x;
wire signed [15:0] y;
...
assign y = {{8{x[7]}}, x};
assign y = x;
...
assign x = y;
```

An implementation
of sign extension in a circuit.

Signed Negation

- Complement and add 1
  - Note that $\overline{x_i} = 1 - x_i$

$$\overline{x} + 1 = -(1 - x_{n-1})2^{n-1} + (1 - x_{n-2})2^{n-2} + \cdots + (1 - x_0)2^0 + 1$$
$$= -2^{n-1} + x_{n-1}2^{n-1} + 2^{n-2} - x_{n-2}2^{n-2} + \cdots + 2^0 - x_0 2^0 + 1$$
$$= -(-x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_0 2^0)$$
$$-2^{n-1} + (2^{n-2} + \cdots + 2^0) + 1$$
$$= -x - 2^{n-1} + 2^{n-1} = -x$$

- E.g., 43 is 00101011
  so –43 is 11010100 + 1 = 11010101

Signed Negation

- What about negating -2^{n-1}?
  - 1000…00 ⇒ 0111…11 + 1 = 1000…00
  - Result is -2^{n-1}!
- Recall range of n-bit numbers is not symmetric
  - Either check for overflow, extend by one bit, or ensure this case can't arise
- In Verilog: use – operator
  - E.g., `assign y = -x;`

Signed Addition

$$x = -x_{n-1}2^{n-1} + x_{n-2\ldots0} \qquad y = -y_{n-1}2^{n-1} + y_{n-2\ldots0}$$

$$x + y = -(x_{n-1} + y_{n-1})2^{n-1} + \underbrace{x_{n-2\ldots0} + y_{n-2\ldots0}}_{\text{yields } c_{n-1}}$$

- Perform addition as for unsigned
  - Overflow if c_{n-1} differs from c_n
  - See textbook for case analysis
- Can use the same circuit for signed and unsigned addition

Signed Addition Examples

	$\boxed{0}$	$\boxed{0}$	0	0	0	0	0	0	
72:		0	1	0	0	1	0	0	0
49:		0	0	1	1	0	0	0	1
121:		0	1	1	1	1	0	0	1

no overflow

	$\boxed{1}$	$\boxed{1}$	0	0	0	0	0	0	
–63:		1	1	0	0	0	0	0	1
–32:		1	1	1	0	0	0	0	0
–95:		1	0	1	0	0	0	0	1

no overflow

	$\boxed{0}$	$\boxed{0}$	0	0	0	0	0	0	
–42:		1	1	0	1	0	1	1	0
8:		0	0	0	0	1	0	0	0
–34:		1	1	0	1	1	1	1	0

no overflow

	$\boxed{0}$	$\boxed{1}$	0	0	1	0	0	0	
72:		0	1	0	0	1	0	0	0
105:		0	1	1	0	1	0	0	1
		1	0	1	1	0	0	0	1

positive overflow

	$\boxed{1}$	$\boxed{0}$	0	0	0	0	0	0	
–63:		1	1	0	0	0	0	0	1
–96:		1	0	1	0	0	0	0	0
		0	1	1	0	0	0	0	1

negative overflow

	$\boxed{1}$	$\boxed{1}$	1	1	1	0	0	0	
42:		0	0	1	0	1	0	1	0
–8:		1	1	1	1	1	0	0	0
34:		0	0	1	0	0	0	1	0

no overflow

Signed Addition in Verilog

- Result of + is same size as operands

```
wire signed [11:0] v1, v2;
wire signed [12:0] sum;
...
assign sum = {v1[11], v1} + {v2[11], v2};
...
assign sum = v1 + v2; // implicit sign extension
```

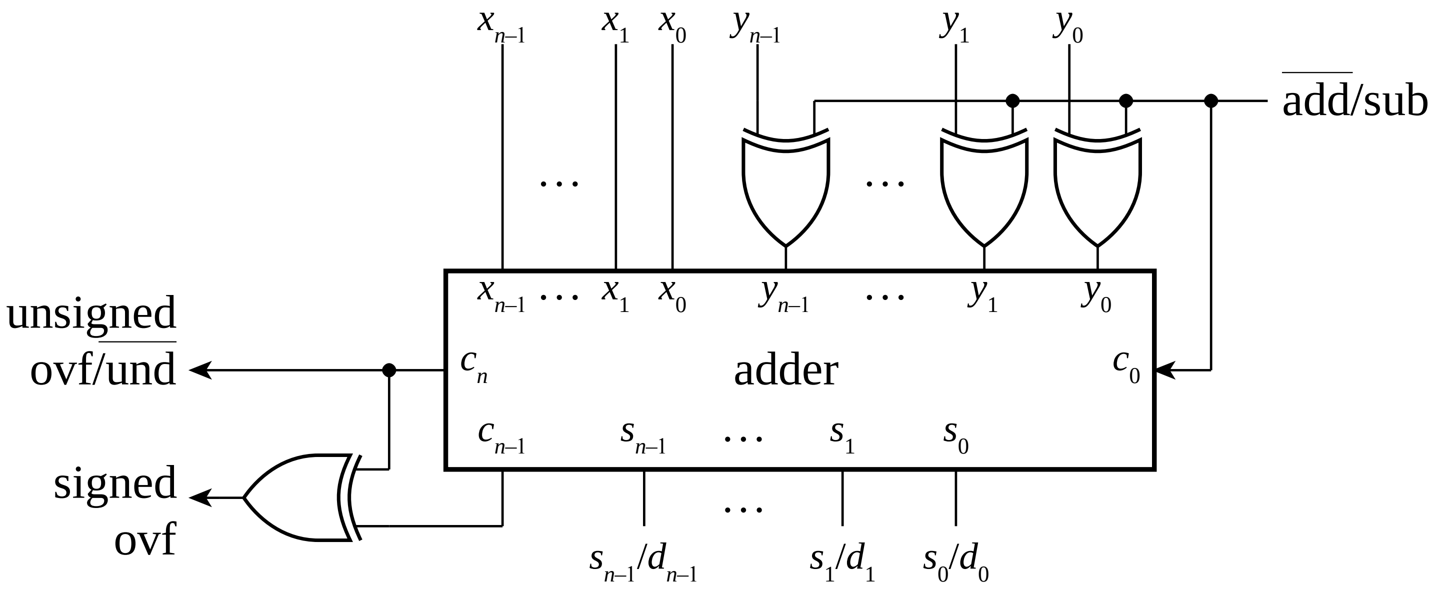
- To check overflow, compare signs

```
wire signed [7:0] x, y, z;
wire              ovf;
...
assign z   = x + y;
assign ovf = ~x[7] & ~y[7] & z[7] | x[7] & y[7] & ~z[7];
```

Signed Subtraction

$$x - y = x + (-y) = x + \overline{y} + 1$$

- Use a 2s-complement adder
  - Complement y and set $c_0 = 1$



OTHER SIGNED OPERATIONS

- Increment, decrement
  - same as unsigned
- Comparison
  - =, same as unsigned
  - >, compare sign bits using $\overline{x_{n-1}} \cdot y_{n-1}$
- Multiplication
  - Complicated by the need to sign extend partial products
  - Refer to Further Reading

Scaling Signed Integers

- Multiplying by 2^k
  - logical left shift (as for unsigned)
  - truncate result using 2s-complement rules
- Dividing by 2^k
  - *arithmetic right shift*
  - discard *k* bits from the right, and replicate sign bit *k* times on the left
  - e.g., s = "11110011" -- –13
    shift_right(s, 2) = "11111100" -- –13 / 2^2