



COMP311-1 Logic Circuit Design

Chap. 6

Instructor: Taigon Song (송대건)

2019 Fall

Levels of Design Abstraction

- Behavioral or algorithmic level
 - Highest level of abstraction. Almost as similar to C programming
- Dataflow level
 - How data flows between HW registers and how data is processed
- Gate level
 - AND, OR, INV...etc
- Switch level
 - Transistor level

6.1 Continuous Assignments

- Continuous assignments have the following characteristics
 - Left-hand side must always be a scalar or vector **net** (or a concatenation (연쇄) of scalar and vector nets)
 - Cannot be a scalar/vector register
 - Continuous assignments are always active
 - Expression evaluated as soon as one of the right-hand-side operands changes
 - Right-hand side can be registers or nets or function calls
 - Registers or nets can be scalars or vectors
 - Delay values can be specified for assignments in terms of time units.

Example 6-1 Examples of Continuous Assignment

```
// Continuous assign. out is a net. i1 and i2 are nets.
assign out = i1 & i2;

// Continuous assign for vector nets. addr is a 16-bit vector net
// addr1 and addr2 are 16-bit vector registers.
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

// Concatenation. Left-hand side is a concatenation of a scalar
// net and a vector net.
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

6.1 Continuous Assignments

6.1.1 Implicit (함축적인) Continuous Assignment

- A continuous assignment can be placed on a net when it is declared
 - There can be only one implicit declaration assignment per net because a net is declared only once

```
//Regular continuous assignment  
wire out;  
assign out = in1 & in2;
```

```
//Same effect is achieved by an implicit continuous assignment  
wire out = in1 & in2;
```

6.1 Continuous Assignments

6.1.2 Implicit Net Declaration

- If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name
 - I.e., the syntax below is also possible

```
// Continuous assign. out is a net.  
wire i1, i2;  
assign out = i1 & i2; //Note that out was not declared as a wire  
                      //but an implicit wire declaration for out  
                      //is done by the simulator
```

6.2 Delays

6.2.1 Regular Assignment Delay

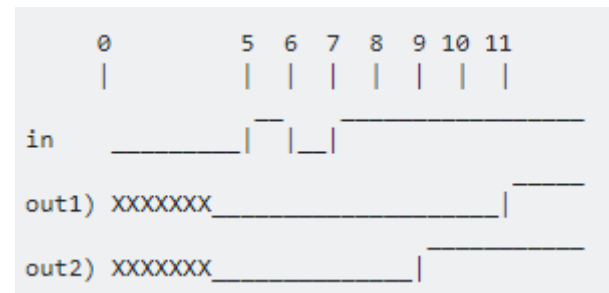
- Assign a delay value in a continuous assignment statement.
- Consider the example below
 - Any change in values of in1 or in2 will result in a delay of 10 time units before recomputation of the expression in1 & in2, and the result will be assigned out

```
assign #10 out = in1 & in2; // Delay in a continuous assign
```

- Note,
 - If in1 or in2 changes value again before 10 time units when the result propagates to out, the values of in1 and in2 at the time of recomputation are considered

- “inertial delay”
- This applies to gate delays too

```
assign #4 out = in;
initial
begin
    in = 0;
    #5 in = 1;
    #1 in = 0;
    #1 in = 1;
end
```



Source: <https://stackoverflow.com/questions/29781724/inertial-delay-in-verilog-hdl>

6.2 Delays

6.2.2 Implicit Continuous Assignment Delay

- This is possible

```
//implicit continuous assignment delay
wire #10 out = in1 & in2;

//same as
wire out;
assign #10 out = in1 & in2;
```

- This is also possible

```
//Net Delays
wire # 10 out;
assign out = in1 & in2;

//The above statement has the same effect as the following.
wire out;
assign #10 out = in1 & in2;
```

6.3 Expressions, Operators, and Operands

6.3.1 Expressions, 6.3.2 Operands

- Expressions: constructs that combine operators and operands to produce a result

```
// Examples of expressions. Combines operands and operators
a ^ b
addr1[20:17] + addr2[20:17]
in1 | in2
```

- Operands (피연산자: a, b, c...)
 - Including function calls (to be described later)

```
integer count, final_count;
final_count = count + 1; //count is an integer operand

real a, b, c;
c = a - b; //a and b are real operands

reg [15:0] reg1, reg2;
reg [3:0] reg_out;
reg_out = reg1[3:0] ^ reg2[3:0]; //reg1[3:0] and reg2[3:0] are
                                //part-select register operands

reg ret_value;
ret_value = calculate_parity(A, B); //calculate_parity is a
                                    //function type operand
```


6.3 Expressions, Operators, and Operands

6.3.3 Operators

- Operators (연산자)
 - Act on the operands to produce desired results

```
d1 && d2 // && is an operator on operands d1 and d2
!a[0] // ! is an operator on operand a[0]
B >> 1 // >> is an operator on operands B and 1
```

6.4 Operator Types

Table 6-1. Operator Types and Symbols

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
		logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two

Bitwise	~	bitwise negation	one
	&	bitwise and	two
		bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
		reduction or	one
	~	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one
Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

6.4 Operator Types

6.4.1 Arithmetic (산술) Operators

- multiply (*), divide (/), add (+), subtract (-), power (**), and modulus (%)
 - Take two operands

```
A = 4'b0011; B = 4'b0100; // A and B are register vectors
D = 6; E = 4; F=2// D and E are integers

A * B // Multiply A and B. Evaluates to 4'b1100
D / E // Divide D by E. Evaluates to 1. Truncates any fractional part.
A + B // Add A and B. Evaluates to 4'b0111
B - A // Subtract A from B. Evaluates to 4'b0001
F = E ** F; //E to the power F, yields 16
```

- Unknown bit of an operand will result the entire final expression as x
 - If an operand value is not known precisely, the result should be an unknown

```
in1 = 4'b101x;
in2 = 4'b1010;
sum = in1 + in2; // sum will be evaluated to the value 4'bx
```

- Modulus operator for remainders

```
13 % 3 // Evaluates to 1
16 % 4 // Evaluates to 0
-7 % 2 // Evaluates to -1, takes sign of the first operand
7 % -2 // Evaluates to +1, takes sign of the first operand
```

6.4 Operator Types

6.4.1 Arithmetic (산술) Operators

- Unary (단일) operators for positive/negative sign of the operand

```
-4 // Negative 4  
+5 // Positive 5
```

- Note, try to use integer or real for negative numbers
 - In fact, try to use these negative numbers in testbench only

```
//Advisable to use integer or real numbers  
-10 / 5// Evaluates to -2
```

```
//Do not use numbers of type <sss> '<base> <nnn>  
- 'd10 / 5// Is equivalent (2's complement of 10)/5 =  $2^{32} - 10$  / 5  
// where 32 is the default machine word width.  
// This evaluates to an incorrect and unexpected result
```

6.4 Operator Types

6.4.2 Logical Operators

- Logical operators are logical-and (&&), logical-or (||) and logical-not (!)
 - Always evaluate to a 1-bit value: 0 (false), 1 (true), or x (ambiguous)
 - If an operand is not equal to 0, it is equivalent to logical 1 (true)
 - If it is equal to 0, it is equivalent to logical 0 (false)
 - If x or z, it is equivalent to x (ambiguous) and is normally treated as a false condition

```
// Logical operations
A = 3; B = 0;
A && B // Evaluates to 0. Equivalent to (logical-1 && logical-0)
A || B // Evaluates to 1. Equivalent to (logical-1 || logical-0)
!A // Evaluates to 0. Equivalent to not(logical-1)
!B // Evaluates to 1. Equivalent to not(logical-0)

// Unknowns
A = 2'b0x; B = 2'b10;
A && B // Evaluates to x. Equivalent to (x && logical 1)

// Expressions
(a == 2) && (b == 3) // Evaluates to 1 if both a == 2 and b == 3 are
true.
// Evaluates to 0 if either is false.
```

- QnA) What should I do when I confuse operator precedence?
 - Use parentheses

6.4 Operator Types

6.4.3 Relational Operators

- greater-than (>), less-than (<), greater-than-or-equal-to (>=), and less-than-or-equal-to (<=)
- Note, if there are any unknown or z bits in the operands the expression takes a value x
 - These operators function exactly as the corresponding operators in the C programming language

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

A <= B // Evaluates to a logical 0
A > B // Evaluates to a logical 1
Y >= X // Evaluates to a logical 1
Y < Z // Evaluates to an x
```

6.4 Operator Types

6.4.4 Equality Operators

- logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==)
 - Logical 1 if true
 - Logical 0 if false

Table 6-2. Equality Operators

Expression	Description	Possible Logical Value
a == b	a equal to b, result unknown if x or z in a or b	0, 1, x
a != b	a not equal to b, result unknown if x or z in a or b	0, 1, x
a === b	a equal to b, including x and z	0, 1
a !== b	a not equal to b, including x and z	0, 1

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101
// Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx

A == B // Results in logical 0
X != Y // Results in logical 1
X == Z // Results in x
Z === M // Results in logical 1 (all bits match, including x and z)
Z === N // Results in logical 0 (least significant bit does not match)
M !== N // Results in logical 1
```

6.4 Operator Types

6.4.5 Bitwise Operators

- negation (~), and(&), or (|), xor (^), xnor (^~, ~^)
 - If one operand is shorter than the other, it will be bit-extended with zeros to match the length of the longer operand

```
// X = 4'b1010, Y = 4'b1101
// Z = 4'b10x1
```

```
~X      // Negation. Result is 4'b0101
X & Y    // Bitwise and. Result is 4'b1000
X | Y    // Bitwise or. Result is 4'b1111
X ^ Y    // Bitwise xor. Result is 4'b0111
X ^~ Y   // Bitwise xnor. Result is 4'b1000
X & Z    // Result is 4'b10x0
```

- Don't confuse between logical operators!

```
// X = 4'b1010, Y = 4'b0000
```

```
X | Y // bitwise operation. Result is 4'b1010
X || Y // logical operation. Equivalent to 1 || 0. Result is 1.
```

Table 6-3. Truth Tables for Bitwise Operators

bitwise and	0	1	x	bitwise or	0	1	x
0	0	0	0	0	0	1	x
1	0	1	x	1	1	1	1
x	0	x	x	x	x	1	x

bitwise xor	0	1	x	bitwise xnor	0	1	x
0	0	1	x	0	1	0	x
1	1	0	x	1	0	1	x
x	x	x	x	x	x	x	x

bitwise negation	result
0	1
1	0
x	x

6.4 Operator Types

6.4.6 Reduction Operators

- and (&), nand (~&), or (|), nor (~|), xor (^), and xnor (~^, ^~).
 - When used with only one operand
 - Perform a bitwise operation on a single vector operand
 - Yield a 1-bit result

```
// X = 4'b1010

&X //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
|X//Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
^X//Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0
//A reduction xor or xnor can be used for even or odd parity
//generation of a vector.
```

- May be confusing between logical (!, &&, ||), bitwise (~, &, |, ^), and reduction operators (&, |, ^)
 - Always remember (1) the number of operands and (2) the value of the results computed

6.4 Operator Types

6.4.7 Shift Operators

- right shift (\gg), left shift (\ll), arithmetic right shift (\ggg), and arithmetic left shift (\lll)
 - Shift a vector operand to the right or the left by a specified number of bits
 - When the bits are shifted, the vacant bit positions are filled with zeros
 - Shift operations do not wrap around
 - Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits
 - Normally effective with negative numbers

```
// X = 4'b1100
```

```
Y = X >> 1; //Y is 4'b0110. Shift right 1 bit. 0 filled in MSB  
position.
```

```
Y = X << 1; //Y is 4'b1000. Shift left 1 bit. 0 filled in LSB position.  
Y = X << 2; //Y is 4'b0000. Shift left 2 bits.
```

```
integer a, b, c; //Signed data types  
a = 0;  
b = -10; // 00111...10110 binary  
c = a + (b >>> 3); //Results in -2 decimal, due to arithmetic shift
```

6.4 Operator Types

6.4.8 Concatenation Operators

- Provides a mechanism to append multiple operands
 - Operands must be sized (Unsized operands are not allowed)

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110  
  
Y = {B , C} // Result Y is 4'b0010  
Y = {A , B , C , D , 3'b001} // Result Y is 11'b10010110001  
Y = {A , B[0], C[1]} // Result Y is 3'b101
```

- Repetitive concatenation also possible

```
reg A;  
reg [1:0] B, C;  
reg [2:0] D;  
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;  
  
Y = { 4{A} } // Result Y is 4'b1111  
Y = { 4{A} , 2{B} } // Result Y is 8'b11110000  
Y = { 4{A} , 2{B} , C } // Result Y is 8'b1111000010
```

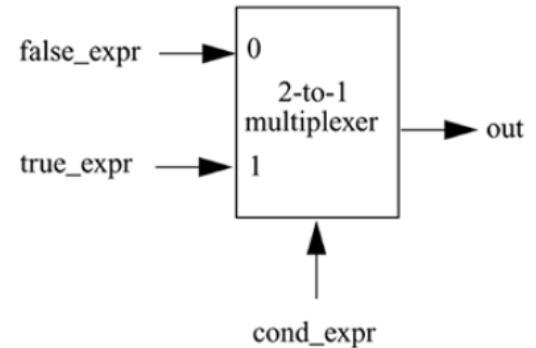
6.4 Operator Types

6.4.10 Conditional Operators

- Usage: `condition_expr ? true_expr : false_expr ;`

```
//model functionality of a tristate buffer
assign addr_bus = drive_enable ? addr_out : 36'bz;

//model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;
```



```
assign out = (A == 3) ? ( control ? x : y ) : ( control ? m : n) ;
```

6.4 Operator Types

6.4.11 Operator Precedence

- Usage: `condition_expr ? true_expr : false_expr ;`

Table 6-4. Operator Precedence

Operators	Operator Symbols	Precedence
Unary	+ - ! ~	Highest precedence
Multiply, Divide, Modulus	* / %	
Add, Subtract	+ -	
Shift	<< >>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction	&, ~& ^ ^~ , ~	
Logical	&& 	
Conditional	?:	Lowest precedence

6.5 Examples

6.5.1 4-to-1 Multiplexer

Example 6-2 4-to-1 Multiplexer, Using Logic Equations

```
// Module 4-to-1 multiplexer using data flow. logic equation
// Compare to gate-level model
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

//Logic equation for out
assign out =      (~s1 & ~s0 & i0) |
                  (~s1 & s0 & i1) |
                  (s1 & ~s0 & i2) |
                  (s1 & s0 & i3) ;

endmodule
```

Example 6-3 4-to-1 Multiplexer, Using Conditional Operators

```
// Module 4-to-1 multiplexer using data flow. Conditional operator.
// Compare to gate-level model
module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

// Use nested conditional operator
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;

endmodule
```

6.5 Examples

6.5.2 4-bit Full Adder

Example 6-4 4-bit Full Adder, Using Dataflow Operators

```
// Define a 4-bit full adder by using dataflow statements.
module fulladd4(sum, c_out, a, b, c_in);

// I/O port declarations
output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;

// Specify the function of a full adder
assign {c_out, sum} = a + b + c_in;

endmodule
```

Example 6-5 4-bit Full Adder with Carry Lookahead

```
module fulladd4(sum, c_out, a, b, c_in);
// Inputs and outputs
output [3:0] sum;
output c_out;
input [3:0] a,b;
input c_in;

// Internal wires
wire p0,g0, p1,g1, p2,g2, p3,g3;
wire c4, c3, c2, c1;

// compute the p for each stage
assign p0 = a[0] ^ b[0],
       p1 = a[1] ^ b[1],

       p2 = a[2] ^ b[2],
       p3 = a[3] ^ b[3];

// compute the g for each stage
assign g0 = a[0] & b[0],
       g1 = a[1] & b[1],
       g2 = a[2] & b[2],
       g3 = a[3] & b[3];

// compute the carry for each stage
// Note that c_in is equivalent c0 in the arithmetic equation for
// carry lookahead computation
assign c1 = g0 | (p0 & c_in),
       c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
       c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c_in),
       c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) |
           (p3 & p2 & p1 & p0 & c_in);

// Compute Sum
assign sum[0] = p0 ^ c_in,
       sum[1] = p1 ^ c1,
       sum[2] = p2 ^ c2,
       sum[3] = p3 ^ c3;

// Assign carry output
assign c_out = c4;

endmodule
```

Number systems

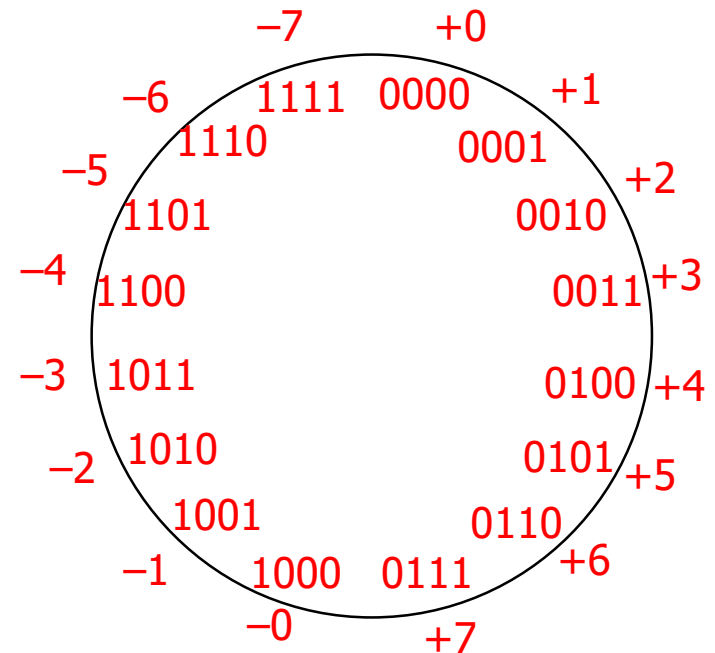
- Representation of positive numbers is the same in most systems
- Major differences are in how negative numbers are represented
- Representation of negative numbers come in three major schemes
 - sign and magnitude
 - 1s complement
 - 2s complement
- Assumptions
 - we'll assume a 4 bit machine word
 - 16 different values can be represented
 - roughly half are positive, half are negative
- Check
 - 덧셈? 뺄셈? 부호변환? 비교?

Sign and magnitude

- One bit dedicate to sign (positive or negative)
 - sign: 0 = positive (or zero), 1 = negative
- Rest represent the absolute value or magnitude
 - three low order bits: 0 (000) thru 7 (111)
- Range for n bits
 - $\pm 2^{n-1} - 1$ (two representations for 0)
- Cumbersome addition/subtraction
 - must compare magnitudes to determine sign of result

0 100 = + 4

1 100 = - 4



1s complement

- If N is a positive number, then the negative of N (its 1s complement or N') is $N' = (2^n - 1) - N$
 - example: 1s complement of 7

$$2^4 = 10000$$

$$1 = 00001$$

$$2^4 - 1 = \underline{1111}$$

$$7 = \underline{0111}$$

$$1000 = -7 \text{ in 1s complement form}$$

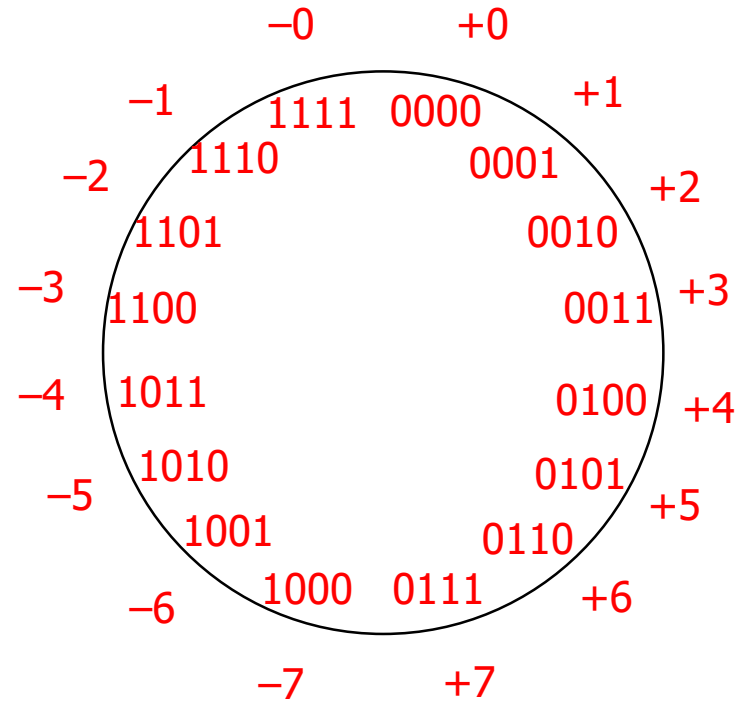
- shortcut: simply compute bit-wise complement (0111 \rightarrow 1000)

1s complement (cont'd)

- Subtraction implemented by 1s complement and then addition
- Two representations of 0
 - causes some complexities in addition
- High-order bit can act as sign bit

0 100 = + 4

1 011 = - 4

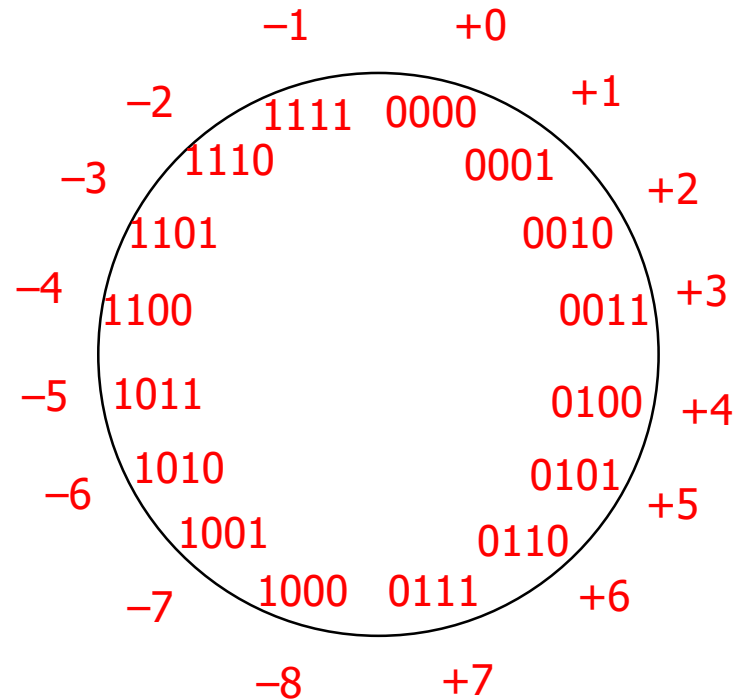


2s complement

- 1s complement with negative numbers shifted one position clockwise
 - only one representation for 0
 - one more negative number than positive numbers
 - high-order bit can act as sign bit

0 100 = + 4

1 100 = - 4



2s complement (cont'd)

- If N is a positive number, then the negative of N (its 2s complement or N^*) is $N^* = 2^n - N$

- example: 2s complement of 7

$$\begin{array}{rcl} & 2^4 & = 10000 \\ \text{subtract } 7 & = & \underline{0111} \end{array}$$

1001 = repr. of -7

- example: 2s complement of -7

$$\begin{array}{rcl} & 2^4 & = 10000 \\ \text{subtract } -7 & = & \underline{1001} \end{array}$$

0111 = repr. of 7

- shortcut: 2s complement = bit-wise complement + 1
 - 0111 \rightarrow 1000 + 1 \rightarrow 1001 (representation of -7)
 - 1001 \rightarrow 0110 + 1 \rightarrow 0111 (representation of 7)

2s complement addition and subtraction

- Simple addition and subtraction
 - simple scheme makes 2s complement the virtually unanimous choice for integer number systems in computers

$$\begin{array}{r} 4 \quad 0100 \\ + 3 \quad 0011 \\ \hline 7 \quad 0111 \end{array}$$

$$\begin{array}{r} -4 \quad 1100 \\ + (-3) \quad 1101 \\ \hline -7 \quad 11001 \end{array}$$

$$\begin{array}{r} 4 \quad 0100 \\ - 3 \quad 1101 \\ \hline 1 \quad 10001 \end{array}$$

$$\begin{array}{r} -4 \quad 1100 \\ + 3 \quad 0011 \\ \hline -1 \quad 1111 \end{array}$$

Why can the carry-out be ignored?

- Can't ignore it completely
 - needed to check for overflow (see next two slides)
- When there is no overflow, carry-out may be true but can be ignored

(-) $M + N$ when $|N| > |M|$:

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

ignoring carry-out is just like subtracting 2^n

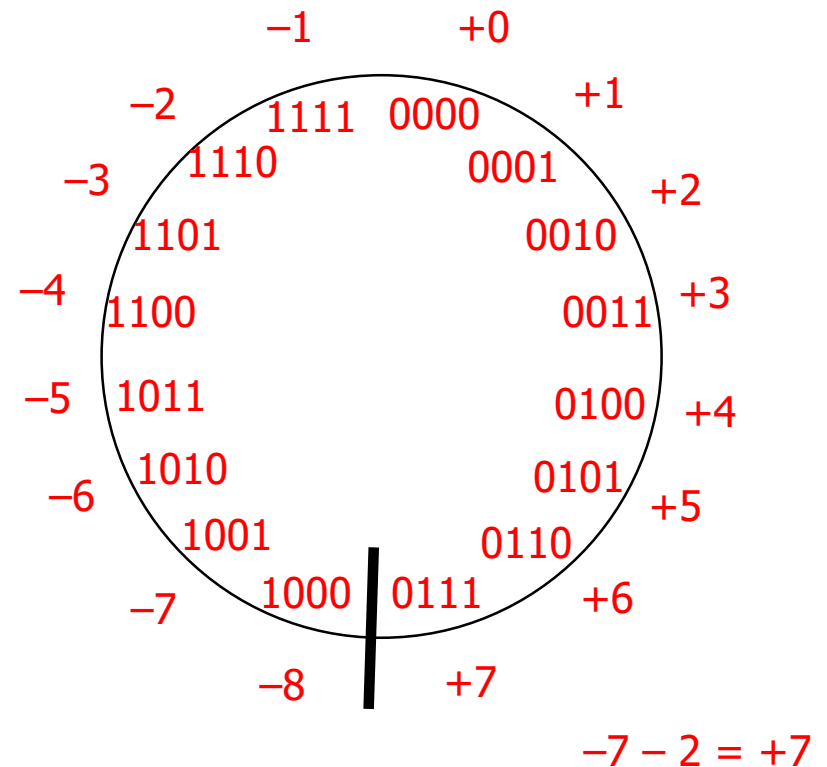
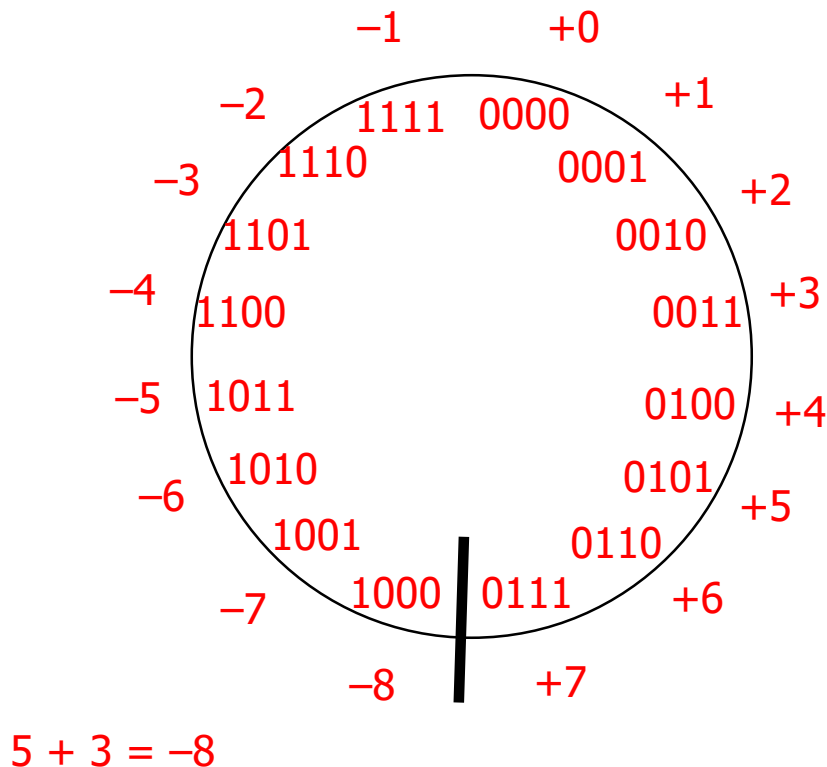
$$-M + -N \text{ where } N + M \leq 2^{(n-1)}$$

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

ignoring the carry, it is just the 2s complement representation for $-(M + N)$

Overflow in 2s complement addition/subtraction

- Overflow conditions
 - add two positive numbers to get a negative number
 - add two negative numbers to get a positive number



Overflow conditions

- Overflow when carry into sign bit position is not equal to carry-out

$$\begin{array}{r}
 5 \\
 \underline{-3} \\
 -8
 \end{array}
 \qquad
 \begin{array}{r}
 0\ 1\ 1\ 1 \\
 0\ 1\ 0\ 1 \\
 \underline{0\ 0\ 1\ 1} \\
 1\ 0\ 0\ 0
 \end{array}$$

overflow

$$\begin{array}{r}
 -7 \\
 \underline{-2} \\
 7
 \end{array}
 \qquad
 \begin{array}{r}
 1\ 0\ 0\ 0 \\
 1\ 0\ 0\ 1 \\
 \underline{1\ 1\ 1\ 0} \\
 1\ 0\ 1\ 1\ 1
 \end{array}$$

overflow

$$\begin{array}{r}
 5 \\
 \underline{+2} \\
 7
 \end{array}
 \qquad
 \begin{array}{r}
 0\ 0\ 0\ 0 \\
 0\ 1\ 0\ 1 \\
 \underline{0\ 0\ 1\ 0} \\
 0\ 1\ 1\ 1
 \end{array}$$

no overflow

$$\begin{array}{r}
 -3 \\
 \underline{-5} \\
 -8
 \end{array}
 \qquad
 \begin{array}{r}
 1\ 1\ 1\ 1 \\
 1\ 1\ 0\ 1 \\
 \underline{1\ 0\ 1\ 1} \\
 1\ 1\ 0\ 0\ 0
 \end{array}$$

no overflow

In-class Assignment

Design a 16-bit ALU

- Specifications
 - Two 16-bit input (a, b)
 - One 16-bit output (z)
 - One 2-bit select signal (sel)
 - Use +, -, *, / for calculation
 - Six modules in total (alu, add, sub, mul, div, mux41)
- Understand how inputs should be connected to the output
- Solution will not be released since this extends to another homework/project
 - Do not hesitate to ask if you don't understand the functionality

Homework #4

Answer the following questions using your tb.v file. All output should be out1234 (your #ID)

1. Does arithmetic operator really print xxxxxx as its output in certain conditions?
 - Provide an example, and show an example result via your .v and waveforms
 - How about `in1 = 5'b1101; in2 = 5'bx1100;`
 - Discuss your results
2. `in1 = 5'bx; in2 = -2; // in1 = -10; in2 = -2;`
 - What happens to `in1 && in2`?
 - What happens to `in1 || in2`?
 - What happens to `in1 & in2`?
 - What happens to `in1 | in2`?
 - Discuss your results
3. `in1 = 5'b100z; in2 = 5'b1100;`
 - What does `in1 < in2` print?
 - Discuss your results
4. `in1 = 5'b1xxz; in2 = 5'b1xxx;`
 - What happens to `'in1 === in2'`
 - What happens to `'in1 == in2'`
 - Discuss your results

Homework #4

Answer the following questions using your tb.v file. All output should be out1234 (your #ID)

5. `in1 = 5'b1001x;`
 - What happens to `&in1`?
 - What happens to `^in1`?
 - Discuss your results
6. Generate `out1234=6'b110011` by using `in1=5'b10011;`
 - Use any method you prefer
 - Use concatenation operator