Unit 1
Data Types and Combinational Logic

EL-GY 9463: INTRODUCTION TO HARDWARE DESIGN

PROFS. SUNDEEP RANGAN, SIDDHARTH GARG

Learning Objectives

- ❑ Represent signals in SystemVerilog of various types:
  - Logical vectors and signed and unsigned integers
- ❑ Implement arithmetic operations (additions, multiplications, bit operations)
- ❑ Identify cases for overflow and truncation
- ❑ Create simple combinational logic modules to implement functions
- ❑ Draw a block diagram representation of the module

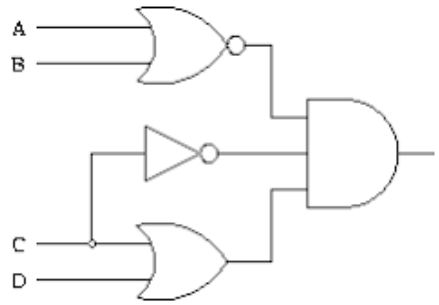
Outline

- Basic Data Types: Logic and Integers
- Truncation and Overflows
- Bit Shifts
- Simple Combinational Logic

Hardware Description Language

Hardware Description Language (HDL)

- Example: Verilog, SystemVerilog
- Describes hardware structure and behavior
- Items natively run in parallel
- Code is synthesized
- Code becomes to logic elements, registers, …
- Also become testbenches



Software

- Example: C, python
- Describes sequential behavior
- One instruction after another
- Code is compiled
- Becomes machine code for a processor

```
# This program says hello and asks for my name

print('Hello world!')
print('What is your name?') # ask for their name
myName = input()
print('It is good to meet you, ' + myName)
print('The length of your name is:')
print(len(myName))
print('What is your age?') # ask for their age
myAge = input()
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

System Verilog vs Verilog

- Verilog
  - Created in 1983-84
  - Standardized in 1995. Became mainstream RTL
  - Largely used for legacy designs

- System Verilog (this unit)
  - Developed in 2002
  - Extends Verilog
  - Better abstraction, parameterization
  - Significantly improved testbench
  - Overwhelming design choice today

Logic and Logic Vectors

```
logic x;          // Single 4-state bit
logic [15:0] y;   // 16-bit vector
```

- Logic: Basic data type in System Verilog
  - Value 0, 1, X or Z
  - X = unknown or unspecified
  - Z = not connected
  - We will discuss X and Z later
- Logic vector
  - A group of log bits
  - Each vector has a bit width.

Bitwise Operations

```
logic x[15:0];      // 16-bit vector
x[3]  // bit 3 of x (fourth bit)
x[7:0] // bits 0 through 7 of x (lower byte)
x[15:8] // bits 8 through 15 of x (upper byte)
```

- System Verilog has natural ways to pack and unpack bits
- Note bits are always counted from 0

Example Problem

Question 1. Bit packing

A designer wants a SystemVerilog 32-bit variable `instruction` with three fields:

- `opcode`: Bits [31:16], an operation code (opcode).
- `rega`: Bits [15:8], a first register address.
- `regb`: Bits [7:0], a second register address.

a. What data type should `instruction` have?
b. Write SystemVerilog code to extract the fields `opcode`, `rega`, and `regb` from `instruction` into separate variables. Make sure to declare the variable types.

Solution

Solution: For part (a), the variable `instruction` should be declared as a 32-bit logic vector:

```
logic [31:0] instruction;
```

For part (b), the fields can be extracted using bit slicing as follows:

```
logic [15:0] opcode;
logic [7:0]  rega;
logic [7:0]  regb;
opcode = instruction[31:16];
rega   = instruction[15:8];
regb   = instruction[7:0];
```

Integers

```
logic [15:0] y;   // 16-bit unsigned integer
logic signed [31:0] s; // 32-bit signed integer
```

- Integers can be regarded naturally as logic vectors
- Vector bit width will determine the range of the integer
- Must indicate if the value is signed or unsigned:
  - Unsigned b bits. Represents a value in $[0, 2^b)$
  - Signed b bits: Represents a value in $[-2^{b-1}, 2^{b-1})$
- Example with $b = 8$ bits
  - Unsigned range is 0, …, 255.
  - Signed range is -128, …, 127

Two Complement

- Signed values are represented with two complement
- Suppose x is represented by a b-bit signed integer q
  - Range $x \in [-2^{b-1}, 2^{b-1})$
  - $q[b-1]$: the "sign" bit
  - $q[b-2:0]$: the "lower" bits
- For $x \geq 0$:
  - Sign bit $q[b-1] = 0$
  - Lower bits: $q[b-2:0] = x$
- For $x < 0$:
  - Sign bit $q[b-1] = 1$
  - Lower bits: $q[b-2:0] = x + 2^b$

Sign bit $q[b-1]$

Lower bits $q[b-2:0]$

Example Problem

Question 2. Two's complement

Consider 8-bit two's-complement numbers. For each of the following decimal values, write the corresponding 8-bit two's-complement **binary** representation, meaning list the 8 bits for each number. Make sure to include leading zeros to show all 8 bits.

a. +13
b. -13
c. +64
d. -64
e. -1

Solution: The correct 8-bit two's-complement encodings are:

```
(a) +13  = 0000_1101
(b) -13  = 1111_0011
(c) +64  = 0100_0000
(d) -64  = 1100_0000
(e) -1   = 1111_1111
```

Two's Complement = Modulo Arithmetic

- Two's complement is equivalent to modulo arithmetic
- Modulo equality: Integers x and y are equal module 2^b:

$$x = y \; mod \; 2^b \iff x = y + k2^b \text{ for some } k$$

- Example:
  - $2 = 18 = -14 \; mod \; 16$
  - $2 \neq -2 \bmod 16$
- Given any x, its b-bit two's complement representation is the unique $q \in [0, 2^b)$ with:

$$x = q \bmod 2^b$$

- Example: Find the following $b = 4$ bits two's complement representations:
  - $x = 5 \Rightarrow q = 5 = (0101)_2$ (since x is already in range $[0,15)$)
  - $x = -5 \Rightarrow q = -5 + 16 = 11 = (1011)_2$

System Verilog Literals and Constants

- Literal: A value written directly in code (not stored in a storage element)
  - Constant. Does not change
- SV has a specific syntax: <size>'<base><digits>
  - <size> = number of bits. If omitted, takes a default value, typically 32-bits
  - <base> = b, o, d, h : binary, octal, decimal, hex
  - <digits> = value
- Examples:
  - 8'd13 → 8-bit decimal 13
  - 8'b0101_1010 → 8-bit binary
  - 4'hF → 4-bit hex (1111)
  - 16'd8 → 16-bit decimal 8
  - 8'b0101 → 8-bit binary 0101
  - -8'd5 → 5-bit two-complement decimal -5 (11111011)

Example Problem

Question 3. Literals to binary

Convert each of the following SystemVerilog literals into their 8-bit binary values. For negative values, give the 8-bit two's-complement representation.

a. 8'd13
b. 8'hF2
c. -5
d. 8'd-20
e. 8'h7F

Solution: The 8-bit binary values are:

a. `8'd13` → `0000_1101`
b. `8'hF2` → `1111_0010`
c. `-5` → `1111_1011` (two's complement)
d. `8'd-20` → `1110_1100` (two's complement)
e. `8'h7F` → `0111_1111`

Outline

- Basic Data Types: Logic and Integers
- Truncation and Overflows
- Bit Shifts
- Simple Combinational Logic

Truncation

```
logic signed [W-1:0] a = A;
```

- Occurs when a variable is assigned a value out of range
- In above code we need $A \in [-2^{-W-1}, 2^{W-1})$
- When A is outside this range, SystemVerilog will truncate:
  - Take the W LSBs of A
  - Sign extend if necessary
- Equivalent to $A \; mod \; 2^W$

Sample Problem

Question 4. Truncation

Truncate the following numbers to 4 bits in both unsigned and signed representations. Provide the decimal value of the truncated numbers in each case.

a. 9
b. 15
c. 23
d. -1
e. -12

Solution using Binary Representations

(a) 9

(b) 15

(c) 23

(d) -1

(e) -12

Solution: One method is to convert each decimal number to binary, truncate to 4 bits, and convert back to decimal.

a. Unsigned: `1001` = 9. Signed: `1001` = −7.
b. Unsigned: `1111` = 15. Signed: `1111` = −1.
c. Unsigned: `0111` = 7. Signed: `0111` = 7.
d. Unsigned: `1111` = 15. Signed: `1111` = −1.
e. Unsigned: `0100` = 4. Signed: `0100` = 4.

Solution using Modulo Arithmetic

(a) 9

(b) 15

(c) 23

(d) -1

(e) -12

You can also compute truncated values using modular arithmetic. For unsigned numbers, compute the value modulo $2^4 = 16$. For signed numbers, compute modulo 16 and adjust to the signed range $[-8, 7]$.

a. Unsigned: $9 \bmod 16 = 9$. Signed: $9 - 16 = -7$.
b. Unsigned: $15 \bmod 16 = 15$. Signed: $15 - 16 = -1$.
c. Unsigned: $23 \bmod 16 = 7$. Signed: 7 is in range.
d. Unsigned: $-1 \bmod 16 = 15$. Signed: $15 - 16 = -1$.
e. Unsigned: $-12 \bmod 16 = 4$. Signed: 4 is in range.

Simulating Truncation in Python

```
def truncate(x, n, signed=True):
    mask = (1 << n) - 1
    x = x & mask
    if signed:
        I = (x >= (1 << (n - 1)))
        x -= (1 << n)*I
    return x
```

- This code works for numpy arrays
- Mask the n LSB
- Then sign extend if necessary

Saturation

- Truncation results in values far different from original
  - Ex: Truncate 9 to 4 signed bits $\Rightarrow 9 - 16 = -7$
- Alternatively, can saturate
  - Clip to max or min values before assignment
  - Ex: Truncate 9 to 7
  - Provides less approximation error
  - But adds a bit of extra logic
- Example to right: $W_2 < W_1$
  - Want to assign $b = a$

```
logic signed [W1-1:0] a;
logic signed [W2-1:0] b;

// Assign with truncation
b = a; // lower W2 bits of a assigned to b

// Assign with saturation
if (a > (1 << (W2 - 1)) - 1) begin
    b = (1 << (W2 - 1)) - 1;
end else if (a < -(1 << (W2 - 1))) begin
    b = -(1 << (W2 - 1));
end else begin
    b = a;
end
```

Addition Overflow Rules

- Consider adding $c = a + b$
- If a and b are both signed or both unsigned:
  - No overflow requires $W(c) \geq \max\big(W(a), W(b)\big) + 1$
  - $W(x) =$ bit width of x

```
logic signed [15:0] a;
logic signed [15:0] b;
logic signed [16:0] c1;
logic signed [15:0] c2;
c1 = a + b; // no overflow
c2 = a + b; // potential overflow
```

- If a is signed and b is unsigned,
  - No overflow requires $W(c) \geq \max\big(W(a), W(b)\big) + 2$
  - Example if $a = 7$ (4 bits signed), $b = 15$ (4 bits unsigned)
  - Sum is $c = 22$ which needs 6 bits signed.

Sample Problem

Question 6. Addition with multiple numbers

Consider the addition:

```
logic signed [7:0] a, b, c, d;
logic signed [W-1:0] sum = a + b + c + d;
```

What value of `W` is required to ensure that no overflow occurs in the sum?

Solution: The partial sums `a + b` and `c + d` each require one extra bit, so they need $8 + 1 = 9$ bits. Adding these two 9-bit partial sums requires one more extra bit, so the final sum requires: `W = 9 + 1=10`.

Addition with Truncation

- Given a potential overflow, SV will:
  - Create a temporary variable with no overflow
  - Temporary variable has enough bits
  - Then it will truncate to lower bits
- Saturation must be implemented manually

```
logic signed [7:0] a;
logic signed [10:0] b;
logic signed [8:0] c;

c = a + b; // potential overflow

// SV implments with signed extension
// then truncation
logic signed [10:0] cfull;
cfull = $signed(a) + $signed(b);
c = cfull[8:0]; // Truncation
```

Multiplication Overflow

- Consider multiplying $c = a\ b$
  - a and b may be signed or unsigned
- To prevent overflow, we need:

$$W(c) \geq W(a)W(b)$$

- With overflow:
  - SV implements truncation

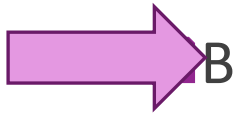
```
logic signed [8:0] a;    // 9-bit signed
logic signed [12:0] b;   // 13-bit signed
logic signed [21:0] c1;  // 22-bit signed
logic signed [20:0] c2;  // 21-bit signed

c1 = a * b; // No overflow
c2 = a * b; // Potential overflow
```

Summary

- Two integer types: $I(b)$ = signed integers $U(b)$ = unsigned
- Each has finite range
- Assignments outside that range result in overflow
- In case of overflow, system Verilog performs truncation
- Optionally, you can add code to explicitly saturate
  - Offers better performance but adds additional logic
- Arithmetic operations result in bitwidth growth
  - Addition of 2 numbers requires 1 extra bit
  - Multiplication of 2 numbers requires width = product of the two numbers

Outline

- ❑ Basic Data Types: Logic and Integers
- ❑ Truncation and Overflows
- ❑ Bit Shifts
- ❑ Simple Combinational Logic



Logical Bit Shift

- Bit shift operators:
  - $x \gg b$ and $x \ll b$ Right and left shift by b bits
  - Fills in with zeros
- Bit widths always stay the same.
  - Hence, bits may be lost
- Examples:
  - $(1011\ 0110) \gg 4 = (0000\ 1011)$
  - $(1011\ 0110) \ll 3 = (1011\ 0000)$

Logical Shift and Bit Widths

- Caution:
  - Logical bit shifts do not change width
  - Some bits may be discarded
- To avoid discarding:
  - Widen first, then shift

```
logic [15:0] x;   // 16-bit unsigned integer
logic [19:0] y1, y2;   // 20-bit unsigned integer

// Shift happens in 16-bits
// Upper bits are discared before assignment
y1 = x << 4;

// Widens first, then shifts
// No bits are discarded
y2 = x;
y2 = y2 << 4;

// Equaivalent widening
y2 = {x, 4'b000};
```

Arithmetic Shift

- Logical Shift Changes Negative Values to Positive
  - Since leading MSBs are filled with zeros
  - Example: Suppose $x = -21$ as an 8-bit signed number
  - Then $x \gg 2 = (0011{,}1011)_2 = 58$
- Arithmetic shift
  - Fills in leading MSBs with sign bit
  - Written in SV with >>>
  - Ex: $x >>> 2 = (1111{,}1011)_2 = -6$

```
// 8-bit signed integer
// Two's complement = (-21)_{10} = (11101011)_2
logic signed [7:0] x = -8'21;

// Logical shift
logic signed [7:0] y1;
y1 = x >>> 2; // y1 = (00111010)_2 = 58_{10}

// Arithmetic shift
logic signed [7:0] y2;
y2 = x >>> 2; // y2 = (11111010)_2 = -6_{10}
```

Python vs. SV

Operation	System Verilog syntax	Python syntax
Left shift (Logical or arithmetic)	`x << b`	`x << b`
Arithmetic right shift	`x >>> b`	`x >> b`
Logical right shift W = number of bits in destination	`x >> b`	`(x >> b) & ((1<<W)-1)`

- Note: All shifts in python are arithmetic
- To emulate logical right shift need a mask

Arithmetic Shift and Powers of Two

- Arithmetic shifts approximate multiplication by powers of two
- Left shift: $x \ll b = x2^b$
  - Assuming the MSBs are not discarded
  - Or the value is extended by b bits prior to left shift
- Right shift: $x >>> b \cong x\ 2^{-b} = \frac{x}{2^b}$
  - Only approximate since the LSBs will be discarded
  - So, there is a loss in precision

Example Problem

Question 8. Arithmetic shifts

Suppose `x` is the SystemVerilog variable:

```
logic signed [7:0] x;
```

Use arithmetic shifts to implement each of the following operations. For each case, choose an appropriate signed bit-width for `y` so that no bits are lost during left shifts.

a. `y = 4 * x`
b. `y = x / 16`
c. `y = 32 * x`

Solution

a. $y = 4x$ (left shift by 2)

```
logic signed [9:0] y;   // 8 bits + 2 extra bits
y = x;                  // widen first
y = y <<< 2;            // arithmetic left shift
```

Equivalent widening:

```
y = {x, 2'b00};
```

b. $y = x/16$ (right shift by 4)

```
logic signed [7:0] y;   // no widening needed
y = x >>> 4;            // arithmetic right shift
```

c. $y = 32x$ (left shift by 5)

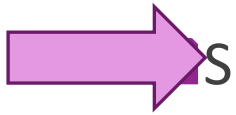
```
logic signed [12:0] y;  // 8 bits + 5 extra bits
y = x;
y = y <<< 5;
```

Equivalent widening:

```
y = {x, 5'b00000};
```

Outline

- ❑ Basic Data Types: Logic and Integers
- ❑ Truncation and Overflows
- ❑ Bit Shifts
- ❑ Simple Combinational Logic



Module and Combinational Logic

- Module: A named circuit block with some inputs and outputs
  - Fully self-contained
  - Takes some input signals and produces some outputs
  - May have sub-modules

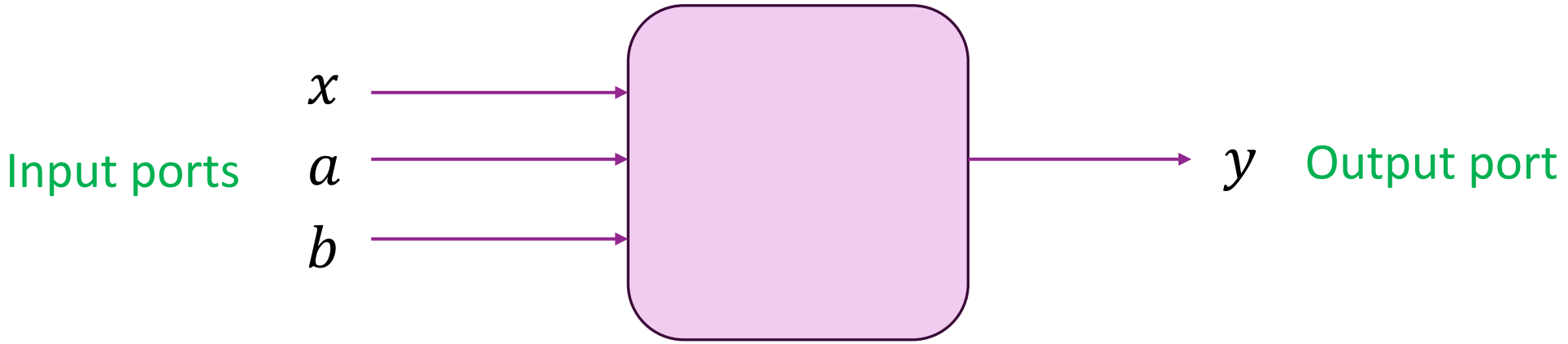
- Combinational logic: Outputs depend only on current inputs
  - No memory
  - No storage
  - We will contrast this type of module with sequential logic in the next unit

Simple Example: ReLU + Linear

Goal: Implement a combinational logic module for the function

$$y = \max\{ax + b, 0\}$$

- Linear function + ReLU
- Used widely in machine learning



SystemVerilog Description

- Each block is a module
  - Has inputs and outputs
  - Logical mapping from inputs to outputs

- Description is behavioral only
  - Does not say how it will be implemented
  - We discuss that later

```
module relu_lin (
  input logic signed [31:0] x,
  input logic signed [31:0] a,
  input logic signed [31:0] b,
  output logic signed [31:0] y
);
  always_comb begin
    logic signed [31:0] mult_out, add_out;
    mult_out = x * a;
    add_out = mult_out + b;
    y = (add_out > 0) ? add_out : 0;
  end
endmodule
```

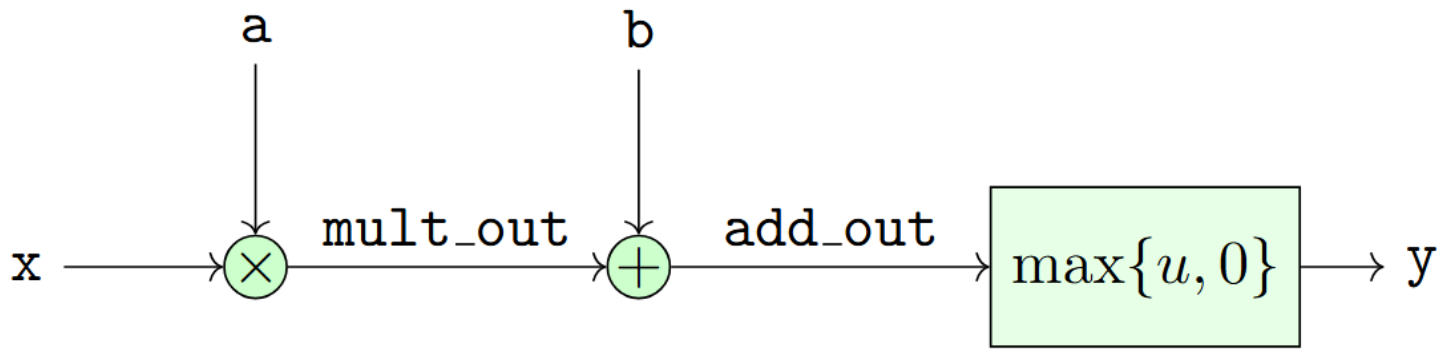
Module Components in SV

```
module relu_lin (
  input logic signed [31:0] x,
  input logic signed [31:0] a,
  input logic signed [31:0] b,
  output logic signed [31:0] y
);
  always_comb begin
    logic signed [31:0] mult_out, add_out;
    mult_out = x * a;
    add_out = mult_out + b;
    y = (add_out > 0) ? add_out : 0;
  end
endmodule
```

- Module name
- Ports:
  - Defines inputs and outputs
  - Each has a type (more detail later)
- Behavioral description
  - In this case, a combinational block
  - Sequence of operations
  - From input to output

Block Diagram

- Function of a module typically represented by a block diagram
- Graphical representation of a potential implementation
- Typically, diagram is at high-level
  - Adders, multipliers not low-level details components like gates



Problem

Question 9. Bit growth and arithmetic

Consider the function:

```
y = 354 + x * x / 8
```

where `x` is a signed 16-bit integer (`logic signed [15:0] x`).

a. Determine the minimum signed bit-width required for `y` so that the result never overflows for any valid 16-bit signed input `x`.
b. Write a SystemVerilog module that implements this function using purely combinational logic. Be sure to declare all intermediate signals with appropriate widths.

- Assigned as homework
- Note there is a conservative and more precise bit width analysis
  - Both will receive full credit