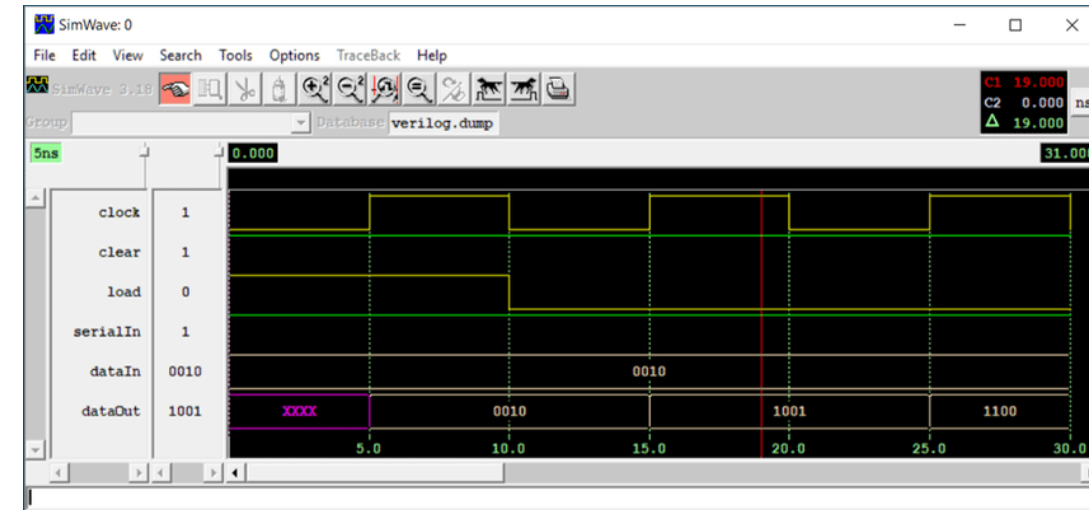
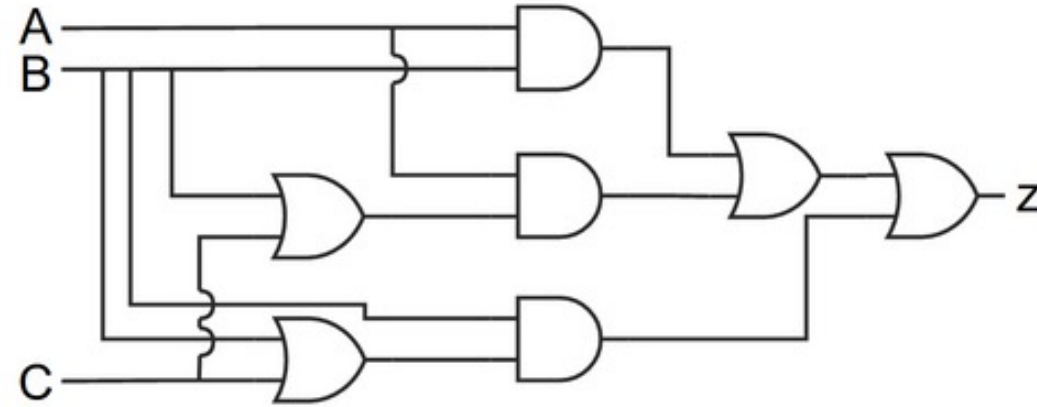


Introduction to Hardware Description Language (HDL)

What is Hardware description language (HDL)?

- Computer language that describes digital circuits.
- Used to simulate digital circuits.
- Used to make FPGA firmware.
 - = Synthesizing firmware
 - = Create digital circuits in FPGAs



Simulation HDL vs Synthesis HDL

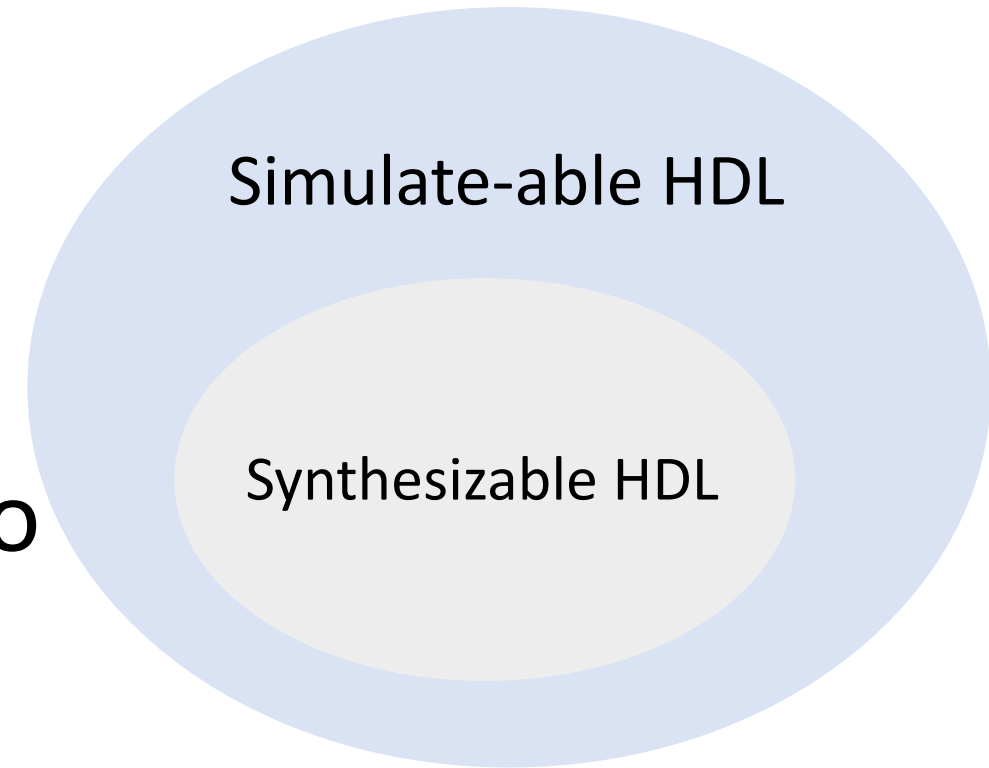
- Synthesis: Converts HDL to FPGA

components and connections.

- HDL must be physically realizable.

- Simulation: Additional syntax to do simulation easily on computer.

- Example: Read txt file.



Verilog vs VHDL

- There are two major HDL languages
(Like C++ and Java)

➤ VHDL: From U.S. Department of defense.

➤ Verilog: From company acquired by
Cadence

- VHDL and Verilog have versions, like
C++11.

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity half_adder is  
  Port (  
    A : in STD_LOGIC; -- First input bit  
    B : in STD_LOGIC; -- Second input bit  
    SUM : out STD_LOGIC; -- Sum output  
    COUT : out STD_LOGIC -- Carry output  
  );  
end half_adder;
```

```
architecture Behavioral of half_adder is  
begin  
  SUM <= A xor B; -- Sum is XOR of inputs  
  COUT <= A and B; -- Carry is AND of inputs  
end Behavioral;
```

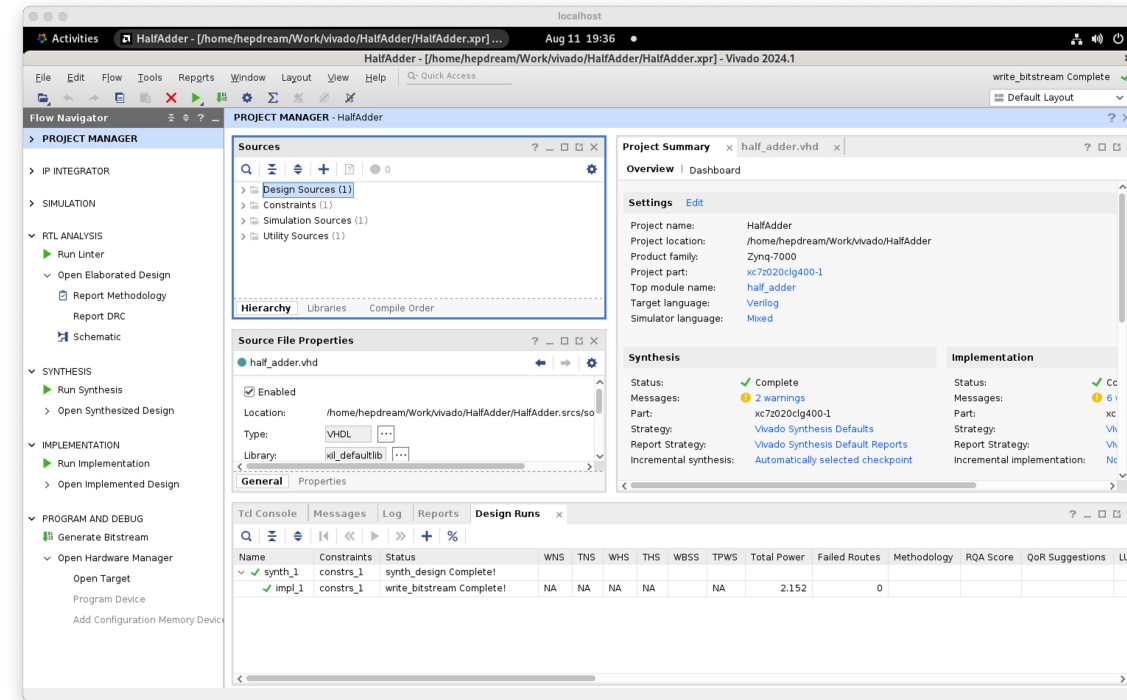
```
module half_adder (  
  input wire A, // First input bit  
  input wire B, // Second input bit  
  output wire SUM, // Sum output  
  output wire COUT // Carry output  
);  
  
  assign SUM = A ^ B; // XOR for sum  
  assign COUT = A & B; // AND for carry  
  
endmodule
```

Verilog vs VHDL

- VHDL is case insensitive. Verilog is case sensitive.
- VHDL is strongly typed. Verilog is weakly typed.
 - Strongly typed: Everything must be specifically defined.
- It is possible to use a Verilog “module” inside VHDL.
- It is possible to use a VHDL “module” inside Verilog.

Synthesizing and simulating HDL

- Programs are used to synthesize and simulate HDL.
- We will be using a program from AMD (company that acquired Xilinx) called Vivado.
- Vivado support Verilog-2001 and VHDL-2008.



HDL libraries

- Because we will be using AMD (Xilinx) FPGAs, we will need AMD HDL libraries.
 - UNISIM, XPM: Simulation library for AMD components “primitives”.
 - UNIMACRO: Simulation library for AMD macros.
- In Vivado, AMD libraries are automatically included.

Verilog and VHDL

- Will first be explaining about Verilog coding.
- VHDL also has similar/same concepts.
- Will also explain about VHDL coding.

Module

- HDL is written by creating a “module” (a function)
 - Has inputs and outputs.
- There is a “top” module that corresponds to main() in C++
 - The top module input / output should correspond to FPGA pins.



Structure of Verilog module

- Add library
- Module interface
- Body
 - Variable definition
 - Body logic
 - **Procedural block**

Can be swapped

*Body can also include input/output definition.

```
`include "folder/sub.v"
```

```
module example #(
    parameter nbit = 8
)
(
    input  wire [nbit-1:0] A,
    input  wire clk,
    output wire [nbit-1:0] B
);
```

```
wire [nbit-2:0] short_A;
reg [nbit-1:0] cnt;
```

```
assign short_A = A[nbit-2:0];
```

```
always @(posedge clk) begin
    cnt <= short_A + 1;
end
```

```
...
```

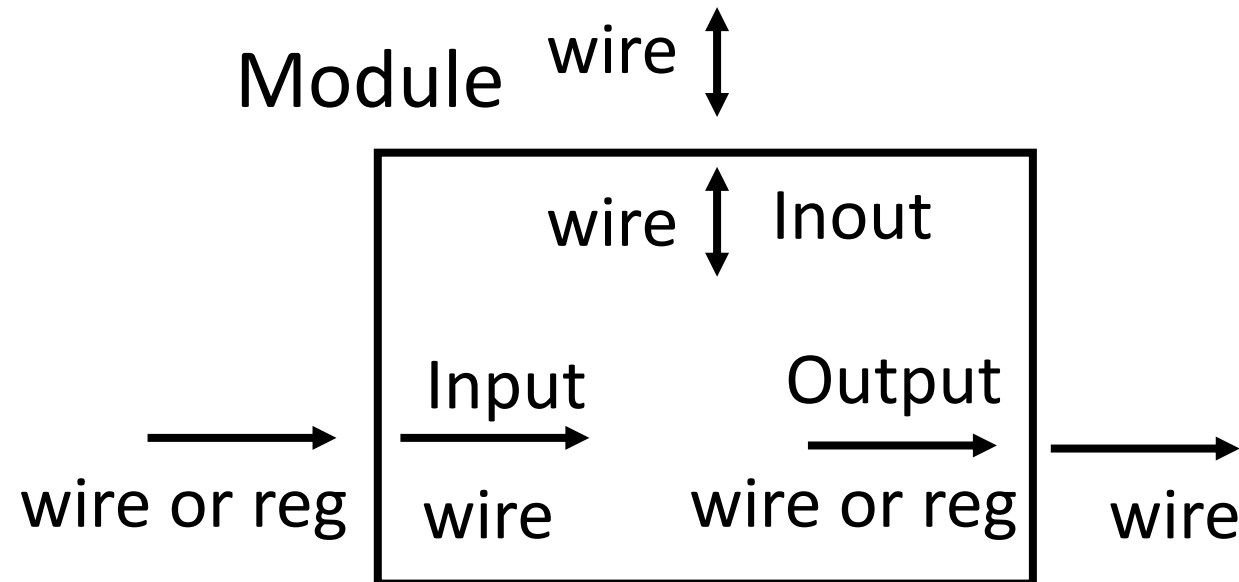
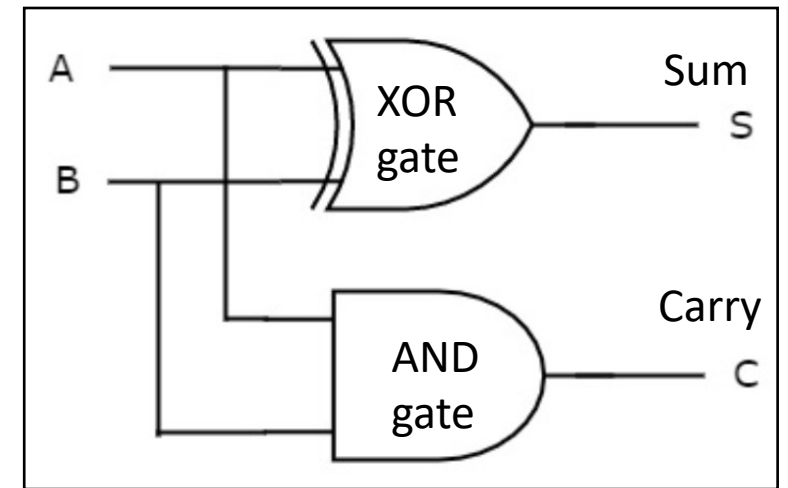
```
endmodule
```

Module syntax

- Need to define input & outputs ports

➤ Can define type: wire or reg

❖ wire just connection. reg saves values.



```
module half_adder (  
    input wire A,  
    input wire B,  
    output wire SUM,  
    output wire CARRY  
);
```

Module syntax

- Modules can also have parameters.

- Parameter values are static.
- Value cannot change for a made firmware.

- Input, output values are dynamic. (Values change).

```
module example #(
    parameter nbit = 8
)
(
    input  wire [nbit-1:0] A,
    input  wire clk,
    output wire [nbit-1:0] B
);
```

Module body

- **Body statements** are concurrent
 - All statements/blocks are assessed at same time.
 - Statements/blocks are continuously assessed.
- **Procedural block** (block of code)
 - Statements are assessed line by line in sequence.
 - **Always block:** Used for synthesis & simulation.
 - **Initial block:** Only used for simulation.

```
wire [nbit-2:0] short_A;  
reg [nbit-1:0] cnt;  
reg [nbit-1:0] cnt2;  
reg fake_clk;
```

```
assign short_A = A[nbit-2:0];
```

```
always @(posedge clk) begin  
    cnt = short_A + 1;  
    cnt2 = cnt + 1;  
end
```

```
initial begin  
    #10 fake_clk = 1'b1  
    #20 fake_clk = 1'b0  
end
```

After 10 time units
After 20 time units

Commenting

- `//` are comments
- `/* */` are multi-line comments

Note: Verilog sets undefined input and output types to wire.

```
module half_adder (  
    input A,    // First input bit  
    input B,    // Second input bit  
    output SUM, // Sum output  
    output CARRY // Carry output  
);  
  
/*  
    This is a multi-line comment.  
    The half adder produces:  
    - SUM = A XOR B  
    - CARRY = A AND B  
    Useful for basic arithmetic operations.  
*/
```

Keyword and identifiers

- Defining is typically done with keywords and values.

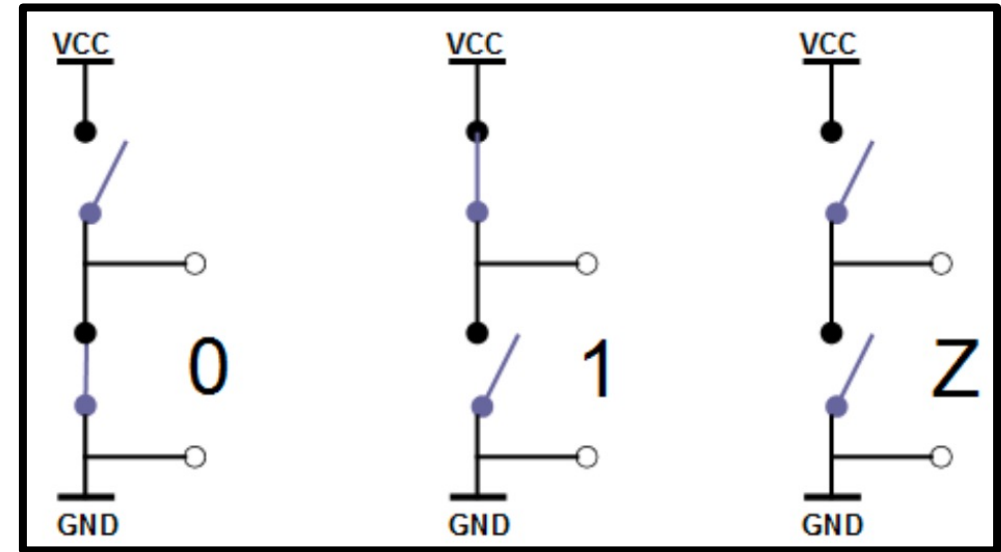
```
module add // module is keyword; add is identifier  
input clk; // input is keyword; clk is identifier  
reg cnt; // reg is keyword; cnt is identifier
```

- Keywords can be types.
- Other keywords: module, output, always, if, for, ...

Data types

- wire: Just a connection. Also known as “net”. Size 1 bit.
- reg: Stores values until overwritten. Size 1 bit. “register”
- Possible values for wire or reg.

Value	Represents
0	GND
1	High
Z	Floating, High impedance
X	Unknown



Arrays

- Typical arrays (Multiple bits)

- `wire [3:0] clk; // A four bit wire`

- `reg [7:0] cnt [0:3][0:3]; // 4x4 matrix, each element 8 bit`

- Accessing arrays.

```
clk_0 = clk[0];
```

```
cnt[0][1] = 8'b0000_0001;  
cnt[0][1][0] = 1'b1;
```

Representing numbers with bits

- Numbers are represented by bits in computers and FPGA

➤ $11 \text{ (decimal)} = 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0$
 $= 1011 \text{ (binary)}$

➤ Binary in Verilog: **4'b1011** (= 4 bits)

➤ Binary is long to write. Write with hexadecimal: **1'hB** (= 1 hex)

Denary/Decimal	Binary	Hexadecimal
Base 10 Number System	Base 2 Number System	Base 16 Number System
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

How to represent negative values with bits?

- Negative values are represented with two's complement

➤ Most significant bit (MSB) represents -2^{n-1} (n is total bits)

MSB LSB (Least significant bit)
 ↓ ↓
4b' 1011

➤ Other bits represents positive bits.

➤ $4b'1011 = 1 \times (-2^3) + 1 \times 2^1 + 1 \times 2^0 = -5$

➤ So 4 bits can represent numbers from -8 to 7 .

- Values with two's complement are called signed values.

Comparison between signed and unsigned

- What number does 4'b1011 represent?

➤ Is it 11? $= 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0$ "unsigned"

➤ Is it -5? $= 1 \times (-2^3) + 1 \times 2^1 + 1 \times 2^0$ "signed"

- Need to define if binary value is "unsigned" or "signed"

➤ `reg signed [3:0] a; // Binary will be signed.`

➤ `reg [3:0] b; // Binary will be unsigned.`

Decimal points (Two methods)

- Fixed point representation: Integer bits + Fractional bits

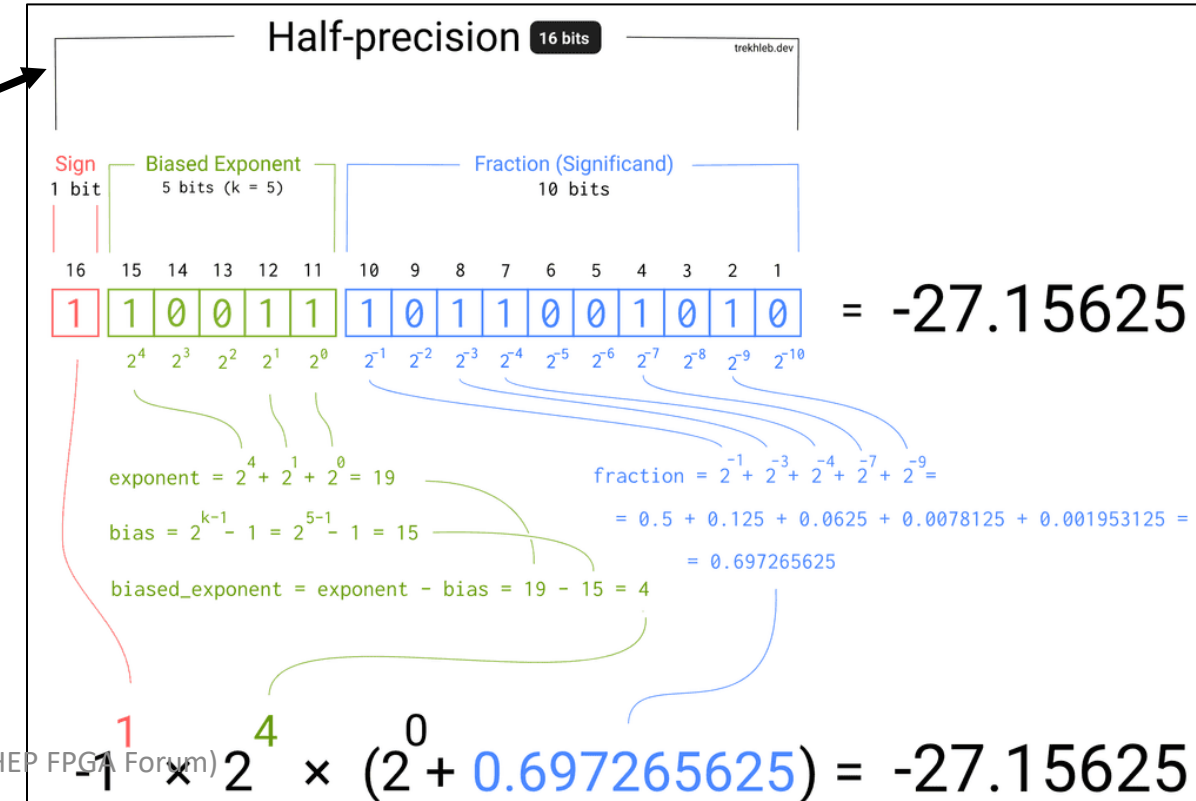
- Position of decimal point is fixed.

- Example: (binary) $101.11 = 2^2 + 2^0 + 2^{-1} + 2^{-2} = 5.75$

- Floating point representation:

- Decimal point floats.

- Used in computers



Decimal points in FPGAs

- Floating point arithmetic is difficult to implement with digital gates. Uses lots of resources. (Don't recommend)
- However fixed point (+, −, ×) calculation is easy.

Binary addition	Unsigned	Signed	Fixed point (two bit fraction)
4'b0010	2	2	0.5
+ 4'b1011	11	-5	2.75
= 4'b1101	13	-3	3.25

- Unsigned, Signed, Fixed point (+, −, ×) are identical!

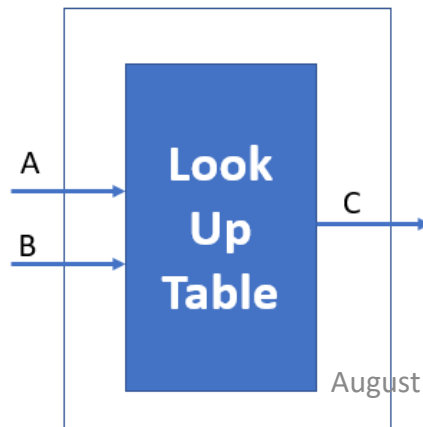
Operators

- There are many operators.

➤ Most can be used in synthesis.

➤ Divide and modulus are difficult to synthesize. (Don't recommend)

❖ Could use look up tables instead.



Verilog Operator	Name
[]	bit-select or part-select
()	parenthesis
!	logical negation
~	negation
&	reduction AND
	reduction OR
~&	reduction NAND
~	reduction NOR
^	reduction XOR
~^ or ^~	reduction XNOR
+	unary (sign) plus
-	unary (sign) minus
{ }	concatenation
{ { } }	replication
*	multiply
/	divide
%	modulus
+	binary plus
-	binary minus
<<	shift left
>>	shift right
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	case equality
!=	case inequality
&	bit-wise AND
^	bit-wise XOR
	bit-wise OR
&&	logical AND
	logical OR
?:	conditional

Break time

- How much did you understand? www.kahoot.it

Assigning values

- For body logic

- `assign var_a = 4'b1100;`

- For procedural block

- **Non-blocking** assignment: `var_a <= 4'b1100;`

- **Blocking** assignment: `var_a = 4'b1100;`

- First understand “synchronous logic”, “always block”, and “concurrency”

Assigning values

- For body logic

- `assign var_a = 4'b1100;`

- For procedural block

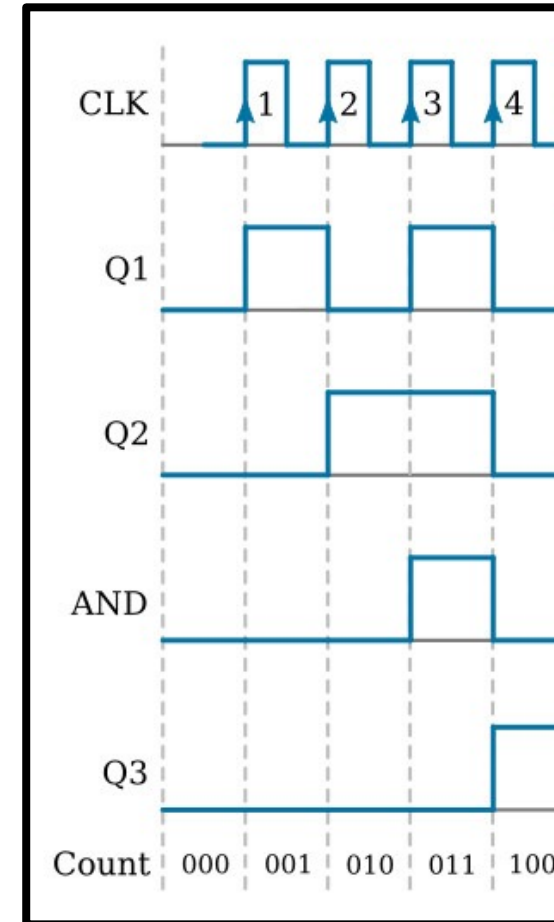
- **Non-blocking** assignment: `var_a <= 4'b1100;`

- **Blocking** assignment: `var_a = 4'b1100;`

- First understand “synchronous logic”, “always block”, and “concurrency”

Synchronous logic (= clocked process)

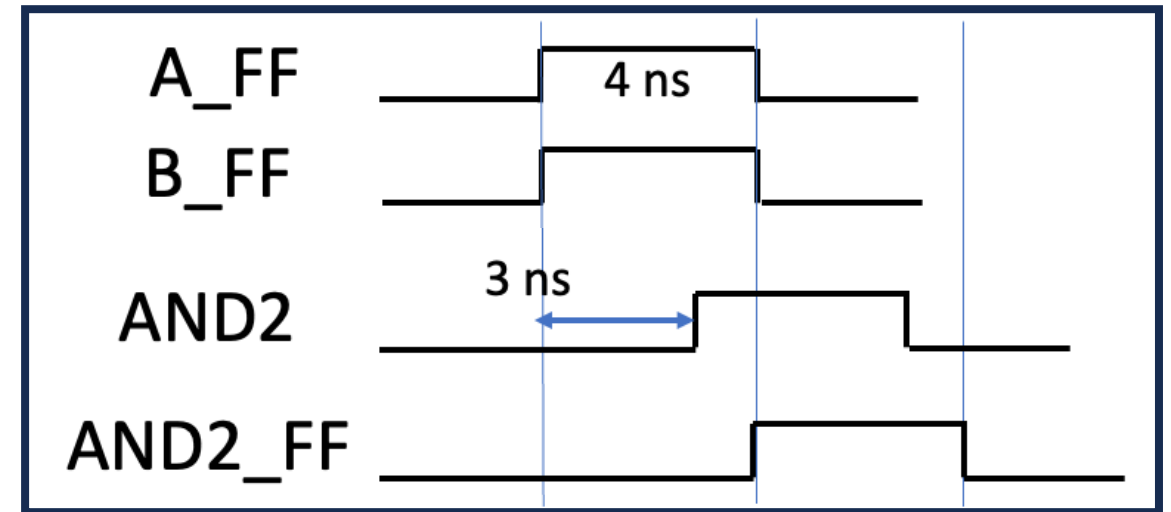
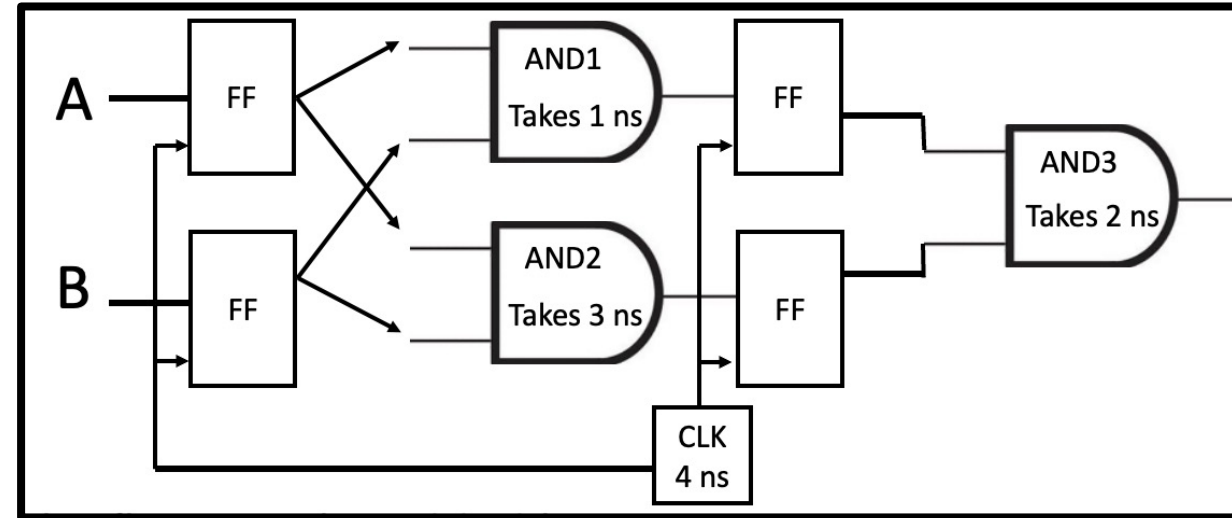
- Logic that changes at intervals of time
 - Example: Logic changes at rising edge of clock.
 - Logic has synchronized timing (through flip-flops).
- Opposite is asynchronous logic (Logic changes as soon as possible)



Synchronous logic (= clocked process)

FF is flip-flop

- Implemented with flip-flops
- Focus on AND2
 1. A_FF, B_FF changes.
 2. AND2 takes 3ns to work
 3. AND2_FF changes.
- There are multiple steps.
- How can Verilog model this?



Always block

- Statements assessed line by line in sequence.
- There is **sensitivity list** and **statements**.
- **Sensitivity list** has two purposes.
 - Used for simulation. Tells when to evaluate statements. If variable changes, evaluate.
 - Used to indicate synchronous logic for synthesis. (posedge → positive edge)

```
always @(posedge clk)
begin

if (reset) begin
    // set things
end else begin
    // do things
end


end
```

Always block (reset)

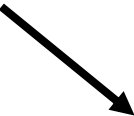
- Reset signal could be synchronous or asynchronous.

➤ Synchronous: reset accepted only at clock edge.

➤ Asynchronous: reset always accepted



```
always @(posedge clk) begin
  if (reset) begin
    // set things
  end else begin
    // do things
  end
end
```



```
always @(posedge clk or
posedge reset)
begin
  if (reset) begin
    // set things
  end else begin
    // do things
  end
end
```

Evaluation of blocks

- Evaluation is done in steps.

1. (Mainly for simulation) Event occurs to start evaluation. Event is when variable in **sensitivity list** changes.

```
always @(A)
begin

  B <= A;
  C <= B;

end
```

“new” B **will have** value of “current” A
“new” C **will have** value of “current” B

2. **Schedule** to change value for statements.

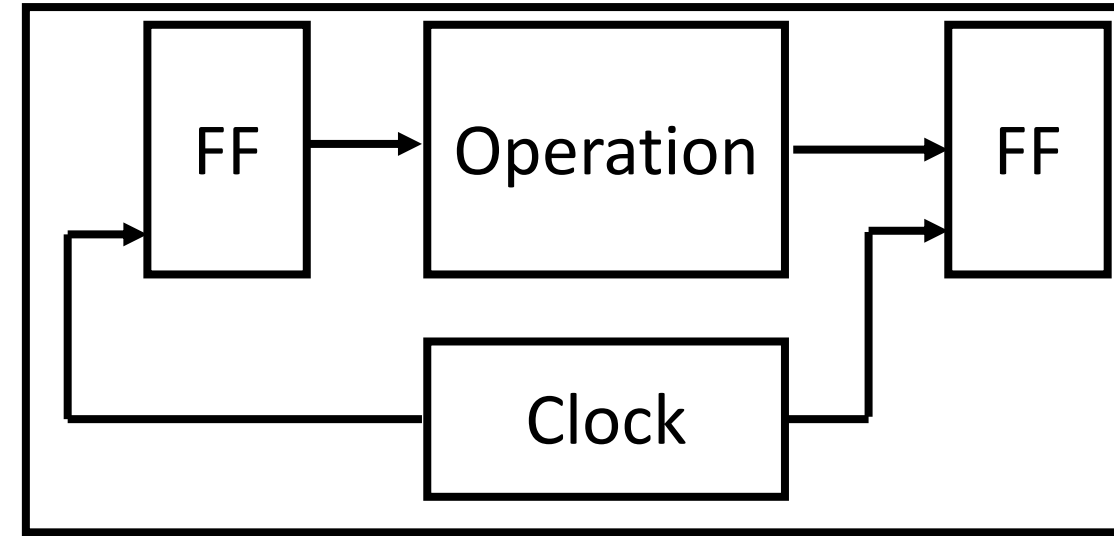
❖ At this step, values do not change.

“new” B and “current” B
are different.

3. Values are changed.

Synchronous logic and block evaluation relation

- Synchronous logic and block evaluation are similar



Synchronous logic	Block evaluation
Input flip-flop changes.	Sensitivity list event.
Operation takes time	Schedule to change value.
Output flip-flop changes.	Value changes.

```
always @(posedge clk) begin  
  A <= B + B;  
end
```

- always block can create synchronous logic.

Evaluation of statements in blocks.

- Non-blocking assignment: \leq

- Assignment is done with scheduling and then changing value.

```
always @(A)
```

```
begin
```

```
B = A;
```

```
C  $\leq$  B;
```

```
end
```

- Blocking assignment: $=$

- Value is changed immediately.

B **is** value of “current” A
“new” C **will have** value of B

How are non-blocking, blocking assignment used?

- Generally used in synchronous logic (Clocked process)

- General logic is written with `<=` to model flip-flops.

- When we want to make a nickname, we can use `=` (B is a nickname for `A[2:0]`)

- When we want to write a operation in multiple lines, we can use `=`

```
always @(posedge clk)
begin

    B = A[2:0];
    C <= B+1;

    // D <= E + F + G + H;
    D_1 = E + F;
    D_2 = G+H;
    D <= D_1 + D_2;

end
```

Always block (last statement wins)

- Statements are assessed line by line in sequence.
- Last statement will overwrite previous statement.

```
always @(posedge clk)
begin

A <= 1 + 2;
...
...
A <= 2 + 2;

end
```

In body logic, assign multiple times causes error.

- Body logic assess statements at same time.
- So assigning a variable multiple time causes an error.
 - Can't not know what variable should be.


```
`include "folder/sub.v"
```

```
module example #(
    parameter nbit = 8
)
(
    input  wire [nbit-1:0] A,
    input  wire clk,
    output wire [nbit-1:0] B
);
```

```
wire short_A;
reg [nbit-1:0] cnt;
```

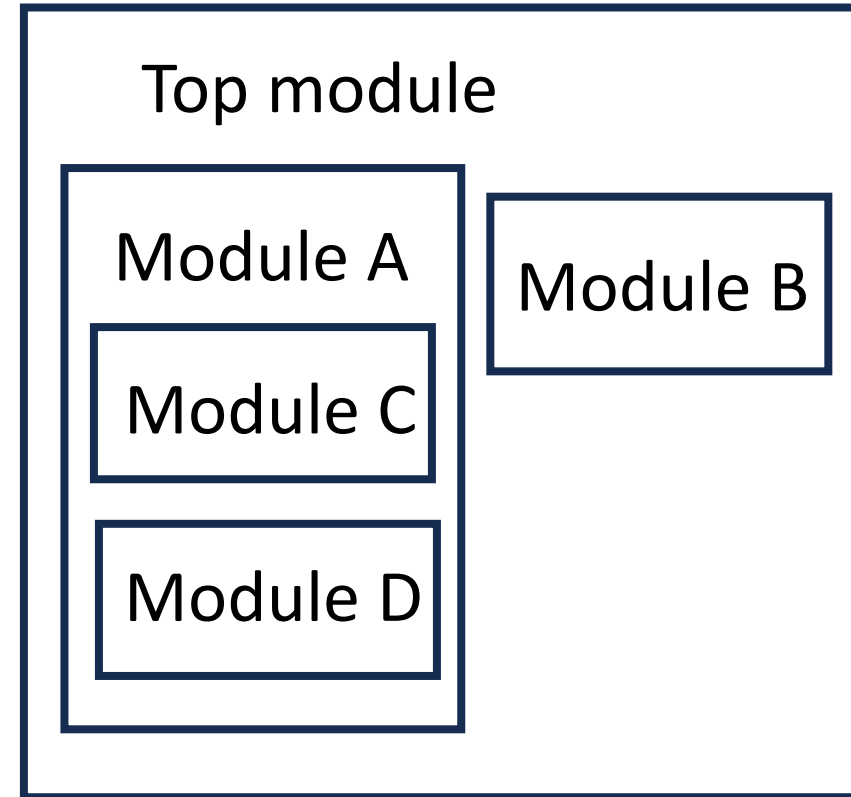
```
assign short_A = A[nbit-2:0];
...
...
assign short_A = A[nbit:2];

endmodule
```



Using modules inside other modules

- Verilog is structured with multiple modules.
 - Instead of writing long top module, multiple modules are written.
- There is code to define a module.
- There is code to use(“instantiate”) a module.



Using modules inside other modules

- There is code to define a module. →
- There is code to use(“instantiate”) a module.
 - Can make multiple copies of the module with instantiation.

```
module find_higgs (  
    input wire mu_p,  
    input wire mu_m,  
    output wire higgs);
```

```
find_higgs hunter1 (  
    .mu_p (first_mu),  
    .mu_n (second_mu),  
    .higgs (higgs1) );
```

```
find_higgs hunter2 (  
    .mu_p (third_mu),  
    .mu_n (fourth_mu),  
    .higgs (higgs2) );
```

Instantiation with setting parameters

Default is 8.

- A module can be defined to have a static parameter. (Can't change for a made firmware)
- Can instantiate with certain parameter.

```
module example #(
    parameter nbit = 8 )
(
    input  wire [nbit-1:0] A,
    input  wire clk,
    output wire [nbit-1:0] B
);
```

```
example example_inst #(
    .nbit (5) )
(
    .A (bottom),
    .clk (clk40mhz),
    .B (charm)
);
```

How to make many copies of a module/code

- Generate block: Can replicates design multiple times or

conditionally.

➤ Generate block is static.

➤ Copy and paste

multiple time. Can't

dynamically change how

many times.

Called as unrolling a for loop.

```
genvar i; // loop index
generate
  for (i=0; i < 5; i = i + 1) begin
    find_higgs hunter(mu_p[i], mu_n[i], higg[i]);
  end
endgenerate
```

```
module #(parameter c) my_design(input a, output b);

generate
  if (c) begin
    find_higgs hunter(a, b);
  end else begin
    find_z hunter(a,b);
  end
endgenerate
```


For loops

- For loops can be placed in generate block and also in always block.

- For loops in always block can be dynamic, but requires lot's of resources. (Not recommended)

```
always @(posedge clk) begin  
  
    integer i; // loop index  
  
    for (i = 5; i > 0; i = i - 1) begin  
        data_out[i] <= data_out[i-1];  
    end  
  
end
```

Need to create circuits for all possibilities.

If/else

- If/else can be placed in generate blocks and always blocks.

- Recommended to consider all possibilities.

➤ Recommended to write “else”.

```
always @(posedge clk) begin

    if (a == 1) begin
        \\ statements
    end else if (a < 5) begin
        \\ statements
    end else begin
        \\ statements
    end
end
```

case

- Case can be placed in generate block and always block.
- Commonly used in “Finite State Machines” (FSM) to consider cases of different states.

```
case (COUNTER)
  2'b00 : begin
    // statements
  end
  2'b01 : begin
    // statements
  end
  default: begin
    // statements
  end
endcase;
```

case vs if/else

- “if/else” conditions can have priority, while case conditions do not.

➤ `a == 1` has high priority.

- Priority conditions require more resources.

```
always @(posedge clk) begin
    if (a == 1) begin
        \\ statements
    end else if (a < 5) begin
        \\ statements
    end else begin
        \\ statements
    end
end
```

- Case can use less resources.

Break time

- How much did you understand? www.kahoot.it

Libraries in VHDL

- VHDL tries to be explicit.
- Need to explicitly write what libraries will be used.
 - Verilog has a default library built in.
- General libraries for VHDL:

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.NUMERIC_STD.ALL;
```

Verilog vs VHDL

```
module example #(
    parameter nbit = 8
)
(
    input wire [nbit-1:0] A,
    input wire clk,
    output wire [nbit-1:0] B
);
```

```
wire [nbit-2:0] short_A;
reg [nbit-1:0] cnt;
```

```
assign short_A = A[nbit-2:0];
```

```
always @(posedge clk) begin
    cnt <= short_A + 1;
end
```

```
endmodule
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

```
entity example is
    generic (nbit : integer := 8 );
    port ( A : in std_logic_vector (nbit-1 downto 0);
          clk : in std_logic;
          B : out std_logic_vector(nbit-1 downto 0) );
end example;
```

```
architecture behavior of example is
    signal short_A : std_logic_vector(nbit-2 downto 0);
    signal cnt : unsigned(nbit-1 downto 0);
```

```
begin
```

```
    short_A <= A(nbit-2 downto 0);
```

```
    process (clk) begin
```

```
        if rising_edge(clk) then
```

```
            cnt <= unsigned(short_A) + 1;
```

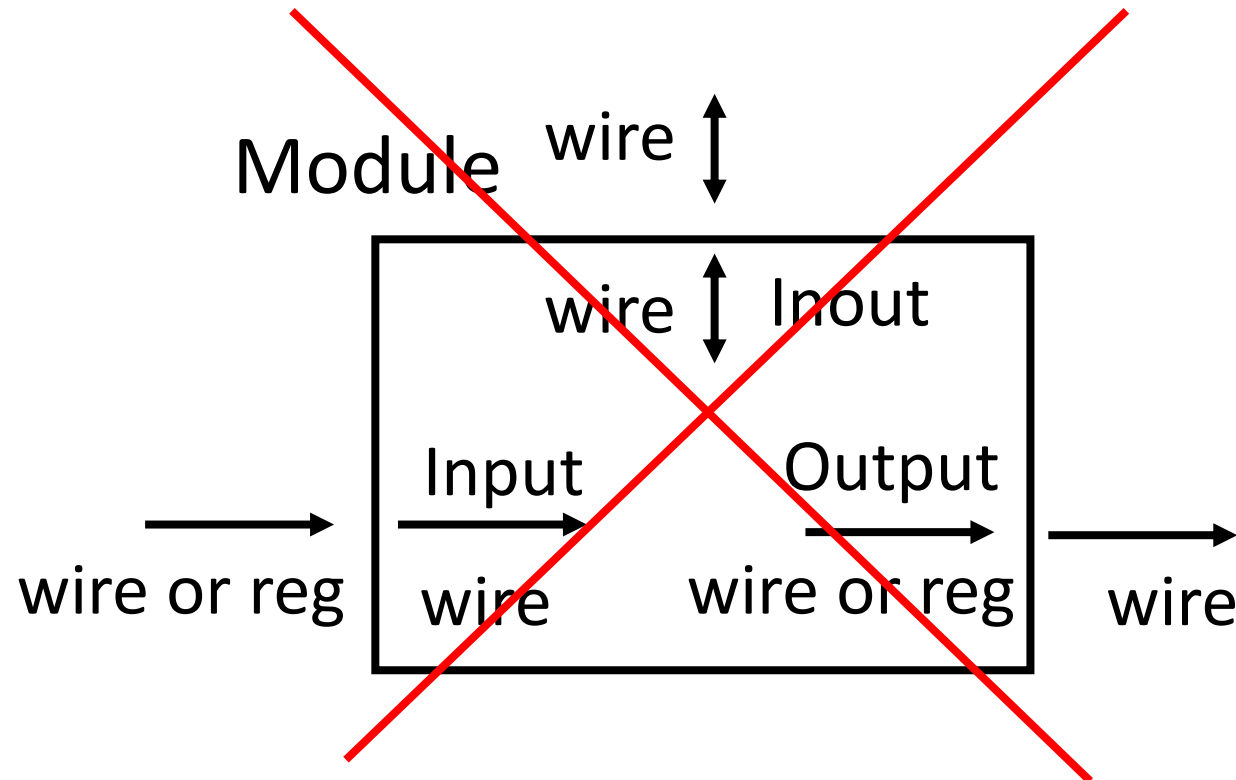
```
        end if;
```

```
    end process;
```

```
end behavior;
```

VHDL signal

- VHDL “signal” is same as Verilog wire and Verilog reg.
 - Everything is just a signal.



Defining a signal

```
signal A : std_logic := '0';  
signal cnt : std_logic_vector(nbit-1 downto 0) := (others=> '0');
```

- Keyword is **signal**.
- Define **name**.
- Define **type**.
- Possible to define initial value.

VHDL doesn't have initial block

Similar to always block



- VHDL doesn't need initial block. It is just a process.

```
always @(posedge clk) begin
  cnt = short_A + 1;
  cnt2 = cnt + 1;
end
```

```
initial begin
  #10 fake_clk = 1'b1
  #20 fake_clk = 1'b0
end
```

```
process (clk) begin
  if rising_edge(clk) then
    cnt <= unsigned(short_A) + 1;
    cnt2 <= unsigned(cnt) + 1;
  end if;
end process;
```

```
process begin
  wait for 10 ns;
  fake_clk <= '1';
  wait for 20 ns;
  fake_clk <= '0';
end process;
```

VHDL commenting

- `--` indicates comment
- `/* */` is multi line comment. (Available since VHDL-2008)

```
cnt <= unsigned(short_A) + 1; -- increment by 1
```

```
/* this is  
a multi-line  
comment */
```

VHDL data types

- `std_logic`: 1 bit being either 0, 1, Z, X
- `std_logic_vector`: multi-bit `std_logic`
 - Top module port must be `std_logic` or `std_logic_vector`
 - Inner modules can have any type for port
 - Can not do math.

VHDL data types

- unsigned: Indicates bits are unsigned. Can do math.
- signed: Indicates bits are signed. Can do math.
- Type casting: VHDL uses casting a lot.
 - VHDL is strongly typed.
 - Changing from one type to another.

Type casting

- VHDL is a strong typed language.

- Need to convert types.

- There are functions that **convert type**.

```
my_u_val <= unsigned(my_slv_val)
my_slv_val <= std_logic_vector(my_u_val)
my_s_val <= signed(my_slv_val)
my_slv_val <= std_logic_vector(my_s_val)
```

Arrays: Need to **define type** for array

Verilog

```
reg [7:0] cnt [0:3][0:3]; // 4x4 matrix, each element 8 bit  
  
cnt[0][1] = 8'b0000_0001;  
cnt[0][1][0] = 1'b1;
```

VHDL

```
type byte_t is std_logic_vector(7 downto 0);  
type matrix_t is array (0 to 3, 0 to 3) of byte_t;  
signal cnt : matrix_t := (others => (others => (others => '0')));  
  
cnt(0, 1) <= b"0000_0001"; -- Multi bit uses ""  
cnt(0, 1)(0) <= '1'; -- Single bit uses ''
```

Assigning values

- Non-blocking assignment: `var_a <= b"1100";`
- Blocking assignment:
 - Need to define "variable". Can only be used in "process"

Similar to always block



```
process(clk)
  variable temp : unsigned(7 downto 0); -- variable declaration
begin
  if rising_edge(clk) then
    temp := unsigned(din) + 1; -- immediate update
  end if;
end process;
```


Process is equivalent to always block

- **Sensitivity list** concept is the same as Verilog.
- **For synchronous logic**, which evaluates statements at clock edge, write "if rising_edge(clk)"

Verilog

```
always @(posedge clk) begin
    cnt <= short_A + 1;
end
```

Sensitivity list

VHDL

```
process (clk) begin
    if rising_edge(clk) then
        cnt <= unsigned(short_A) + 1;
    end if;
end process;
```

Instantiation of modules

- Generally need to **define module** (= Component)
- Then can **instantiate component**.

```
architecture behavior of top is
    -- Component(Module) declaration
    component find_higgs
    port (  mu_p      : in  std_logic;
           mu_m      : in  std_logic;
           higgs: out std_logic );
    end component;

begin
    -- Component instantiation
    hunter1 : find_higgs
        port map ( mu_p  => first_mu,
                   mu_n  => second_mu,
                   higgs => higgs1 );
end architecture;
```

Instantiation of modules

- Also possible to just instantiate component.

```
architecture behavior of top is
    -- No component declaration
begin
    -- Component instantiation
    hunter1 : work.find_higgs
        port map ( mu_p    => first_mu,
                   mu_n    => second_mu,
                   higgs   => higgs1 );
end architecture;
```

Instantiation with setting parameters

- Verilog parameter

= VHDL **generic**

- In declaration

“generic”

- In instantiation

“generic map”

architecture rtl of top is

component example

```
generic ( nbit : integer := 8 );
```

```
port ( A : in std_logic_vector(nbit-1:0);
```

```
      clk : in std_logic;
```

```
      B : out std_logic_vector(nbit-1 downto 0) );
```

```
end component;
```

begin

```
example_inst : example
```

```
generic map ( nbit => 5 )
```

```
port map (
```

```
    A => bottom,
```

```
    rst => clk40mhz,
```

```
    count => charm );
```

```
end architecture;
```

Making copies of components/code

- Can also use
for generate

and if generate

to replicate

design.

```
architecture behavior of top is
begin
    for i in 0 to 5 generate
        hunter : entity find_higgs
            port map (mu_p(i), mu_n(i), higg(i) );
        end generate;
    end architecture;
```

```
entity my_design is
    generic ( C : boolean := true );
end entity;
architecture behavior of my_design is
begin
    if (C) generate
        work.find_higgs(a,b)
    else generate
        work.find_z(a,b)
    end generate;
end architecture;
```

“For” and “if” in process

- For and if can also be used in process.
- Syntax is slightly different with generate case.

```
process(clk) begin
    if rising_edge(clk) then
        for i in 0 to 5 loop
            data_out(i) <= data_out(i-1);
        end loop;
    end if;
end process;
```

```
process(clk) begin
    if rising_edge(clk) then
        if a = 1 then
            -- statements
        elsif a < 5 then
            -- statements
        else
            -- statements
        end if;
    end if;
end process;
```

case

- There is also case for both process and generate.

```
case counter is
  when "00" =>
    -- statements
  when "01" =>
    -- statements
  when others =>
    -- statements
end case;
```

```
case c generate
  when '0' =>
    -- statements
  when others =>
    -- statements
end generate;
```

- How much did you understand? www.kahoot.it