• **HDL**: describes hardware behavior and their structure in textual form
  - To describe the circuits by syntax and sentences
  - As opposed to circuit described by **schematics**

• **Different levels of description**
  - Gate-level
  - Data-level
  - Behavioral-level

• **Widely used HDLs**
  - Verilog – Similar to C
  - SystemVerilog – Similar to C++
  - VHDL – Similar to PASCAL
Types of modeling

- **Behavioral**
  - Models describe what a module does.
  - Use of assignment statements, loops, if, else kind of statements

- **Structural**
  - Describes the structure of the hardware components
  - Interconnections of primitive gates (AND, OR, NAND, NOR, etc.) and other modules

```plaintext
Counter

if (rst)
  cnt = 0;
else
  cnt = cnt+1;
```

A → B → S

A → B → C
## Behavioral vs. Structural

### Behavioral

```verilog
module cter (  
    input  rst, clock,  
    output reg [1:0] count  
);  
always@(posedge clock)  
begin  
    if (rst) count = 0;  
    else count = count +1;  
end  
endmodule
```

### Structural

```verilog
module cter ( rst, clock, count );  
output [1:0] count;  
input rst, clock;  
wire   N5, n1, n4, n5, n6;  
    FFD U0 (.D(N5), .CP(clock),  
        .Q(count[0]), .QN(n6));  
    FFD U1 (.D(n1), .CP(clock),  
        .Q(count[1]), .QN(n5));  
    MUX21 U2 (.A(N5), .B(n4),  
        .S(n5), .Z(n1 ));  
    NR U3 (.A(n6), .B(rst), .Z(n4));  
    NR U4 (.A(count[0]), .B(rst),  
        .Z(N5));  
endmodule
```
The two major purposes of HDLs are logic simulation and synthesis
- During simulation, inputs are applied to a module, and the outputs are checked to verify that the module operates correctly.
- During synthesis, the textual description of a module is transformed into logic gates.

HDL code is divided into synthesizable modules and a test bench
- The synthesizable modules describe the hardware.
- The test bench checks whether the output results are correct (only for simulation and cannot be synthesized).

Test bench (verifies the circuit)

Synthesizable HDL (describes the circuit)
Outlines

- Data type representation
- Structures and Hierarchy
- Operators
- Control statements
- Generate blocks
Logical Values

• A bit can have any of these values
  o 0 representing logic low (false)
  o 1 representing logic high (true)
  o X representing either 0, 1, or Z
  o Z representing high impedance for tri-state (unconnected inputs are set to Z)

• Logic operations

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Number Representation

\[
<size>'<base~format><number>
\]

- **<size>**
  - number of bits (optional)

- **<base format>**
  - It is a single character ' followed by one of the following characters b, d, o and h, which stand for binary, decimal, octal and hex, respectively.

- **<number>**
  - Contains digits which are legal for the <base format>
  - Underscore can be used for readability
Number Representation

<size>'<base format><number>

Unsized numbers (at least 32 bit)
549  // decimal number
'h8F_F  // hex number
'o765  // octal number

Size numbers
4'b11  // 4-bit binary number 0011
3'b10x // 3-bit binary number with LSM bit unknown
8'hz  // 8-bit binary high-impedance number
4'hz1  // 4'bzzz1
5'd3  // 5-bit decimal number

Signed numbers
-8'd6  // 8-bit two's complement of 6 (-6)
4'shF  // 4-bit number ‘1111’ to be interpreted as
       // 2’s complement number
Data Types: Reg and Net (wire)

- **A reg (reg) stores value from one assignment to the next (data storage element)**
  - Default value is X; default range is one bit
  - By default are unsigned, but can be declare signed, using keyword signed
  - Maybe a flip-flop, a latch, or combinatorial circuit

- **Nets (wire) correspond to physical wires that connect instances**
  - Nets do not store values; have to be continuously driven
  - The default range is one bit; by default are unsigned

- **Other types**
  - **Integer (integer)**
    - Convenient to counting purposes
    - At least 32-bit wide
    - Useful for loop
  - **Real (real) – simulation only (why?)**
    - Can be specified in decimal and scientific notation
    - For floating point arithmetic, you have to develop an architecture based on IEEE754 standard. There are many research articles available related to FP implementation...
Vectors and Arrays

- **Verilog vectors: known as BUS in hardware**
  - `<data type> [left range : right range] <Variable name>`
- **Single element that is n-bits wide**
  - `reg [0:7] A, B; //Two 8-bit reg with MSB as the 0th bit`
  - `wire [3:0] Data; //4-bit wide wire MSB as the 4th bit`
- **Vector part select (access)**
  - `Data[2:0] // Three LSB of vector Data`
- **Verilog arrays: range follows the name**
  - `<datatype> <array name> [<array indices>]`
  - `reg B [15:0]; // array of 16 reg elements`
- **Array of vectors: model the memory**
  - `<data type> [<vector indices>]<array name>[<array indices>]`
  - `reg [15:0] mem [1023:0]; // array of vectors`
More about Memory

- On FPGA, memory maps to BRAM
- Everything must be decided at compile time – your hardware cannot be changed while running!
  - Adding one more piece of memory after the circuit is built?

```c
int mem[var];
int mem* = malloc(var * sizeof(int));
```

```vhdl
reg [0:7] mem [var:0];
```

8-bit element

8-bit element

8-bit element

...
Hierarchical HDL structures are achieved by defining modules and instantiating modules.

- Instantiation is the process of "calling" a module.
Module Ports

- Module header starts with module keyword, contains the I/O ports
- Port declarations begins with output, input or inout follow by bus indices
  - Provide the interface by which a module can communicate with the environment

```verilog
module S1 (a, b, c, d, e);
input [1:0] a, b;
input c;
output reg [1:0] d;
output e;
//Verilog 1995 Style
endmodule

module S2 (input [1:0] a, b,
           input c,
           output reg [1:0] d,
           output e);
//ANSI C Style
endmodule
```
Operators

- **Relational operators** (> , < , >= , <=)
  - Mainly use in expression (e.g. if sentences)
  - Returns a logical value (1/true 0/false)
- **Arithmetic operators**
  - Binary operators (+, -, *, /, %)
    - Takes 2 operators
    - Note: *, /, % may not be synthesizable! Especially division and modulo are very complicated in hardware
      - On FPGA it’s better – they use DSPs
  - Unary operators (+/-)
    - Specify the sign of the operand
- **Logical operators** (&&, ||)
- **Bitwise and Reduction operations** (&, |, ^, ~&, ~|)
- **Shift and other operator** (>>, <<)
Loop Statements (for)

- A **for** loop is used to **replicate** hardware logic in Verilog
  - I.e., the loop will essentially be unrolled
  - Again, everything (loop boundary) must be known at compile time!

```verilog
integer i;
always @(posedge clk) begin
    for (i = 0; i < 7; i = i + 1) begin
        memory[i] <= memory[i+1];
        memory[7] <= memory[0];
    end
end
```

```verilog
always @(posedge clk) begin
    memory[0] <= memory[1];
    memory[1] <= memory[2];
    memory[2] <= memory[3];
    memory[3] <= memory[4];
    memory[4] <= memory[5];
    memory[5] <= memory[6];
    memory[6] <= memory[7];
    memory[7] <= memory[0];
end
```
Generate blocks

- Allow to generate Verilog code at elaboration (compilation) time
  - Provides the ability for the design to be built based on Verilog parameters
  - Required the keywords `generate` – `endgenerate`
  - Generate instantiations can be
    - Module instantiations
    - Continuous assignments
    - initial / always blocks

```verilog
module top( input [0:3] in1,
            output [0:3] out1);
    // genvar control the loop
    genvar I;
    generate
     for( I = 0; I <= 3; I = I+1 ) begin
      sub U1(in1[I], out1[I]);
     end
    endgenerate
endmodule
```
Generate blocks

- **Allow to generate Verilog code at elaboration (compilation) time**
  - Provides the ability for the design to be built based on Verilog parameters
  - Required the keywords `generate – endgenerate`
  - Generate instantiations can be
    - Module instantiations
    - Continuous assignments
    - `initial / always` blocks

```verilog
module top #(parameter POS=0)
    (input in, clk, output reg out);

generate
    if(POS==1)
    always @(posedge clk)
        out = in;
    else
    always @(negedge clk)
        out = in;
endgenerate
endmodule
```
Finite State Machine (FSM)

Abstraction

Inputs \((X)\)  
\[ Y = Y(X, S_n) \]

Outputs \((Y)\)

Combinational Logic

Clocked Storage Elements

Present State: \(S_n\)

Next State \(S_{n+1}\)

\[ S_{n+1} = f(S_n, X) \]

Clock
Finite State Machine (FSM)

State Changes

Clock

$X_t \rightarrow \text{Combinational Logic} \rightarrow Y_t \rightarrow X_{t+1} \rightarrow \text{Combinational Logic} \rightarrow Y_{t+1}$

$S_n \rightarrow D_{CQ} \rightarrow U \rightarrow S_{n+1}$

$Y = S_{n-1} \rightarrow S_n \rightarrow S_{n+1}$

Time
• A level-to-pulse converter produces a single-cycle pulse each time its input goes high.
  o In other words, it’s a synchronous rising-edge detector.

• Sample uses:
  o Buttons and switches pressed by humans for arbitrary periods of time
  o Single-cycle enable signals for counters

Whenever input L goes from low to high...

CLK

Level to Pulse Converter

...output P produces a single pulse, one clock period wide.
FSM Design Example

Current State | In | Next State | Out
---|---|---|---
$S_1$ | $S_0$ | $L$ | $S_1^+$ | $S_0^+$ | $P$
0 | 0 | 0 | 0 | 0 | 0
0 | 0 | 1 | 0 | 1 | 0
0 | 1 | 0 | 0 | 0 | 1
0 | 1 | 1 | 1 | 1 | 1
1 | 1 | 0 | 0 | 0 | 0
1 | 1 | 1 | 1 | 1 | 0
Another Example

PE level parallelism

• When to load?
• Where to load?

Buffer 1

• When to write back?
• Where to write back?

Buffer 2

Result
Typical Flow using HLS

- Behavioral-level description (Python, PyTorch, Tensorflow)
- Naïve C/C++ (called golden-C) (no pointers, no recursive, no malloc)
- Optimized C/C++ for HLS tool (with optimization techniques)
Summary

• HDL is hardware description language – describes the “actual” hardware
  o Everything must be fixed at compile time: you cannot create a circuit on-the-fly

• Datatypes, operators, hierarchy, for loops, generate blocks, and FSM
  o HLS tool can take care most of them! :D

• Next lecture: HLS Tutorial
  o Virtual via Bluejeans
  o Prepare your environment and laptop! 😊