Traineeships in Advanced Computing for High Energy Physics (TAC-HEP)

FPGA module training

Week-4

Lecture-7: February 18th 2025

Varun Sharma

University of Wisconsin – Madison, USA

Content

- Hardware Description Languages
  - Verilog

Connecting to cmstrigger02

- Connect to cmstrigger02 machine:
  - ssh -X -Y <username>@cmstrigger02.hep.wisc.edu

- OR
- First sign-in to 'login' machine & then connect to 'cmstrigger02' machine - All of you should have access
  - ssh -X -Y <username>@login.hep.wisc.edu
  - ssh cmstrigger02
  - mkdir /nfs_scratch/`whoami` (If directory exist, go to next bullet)
  - cd /nfs_scratch/`whoami`

TAC-HEP 2025

Verilog HDL

Verilog HDL

Initially was created to simplify design simulation & verification

Verilog: Verification + Logic

Increasing logic complexity ➜ added support for synthesis

- Used to model a digital system at many levels of abstraction
- Complexity can range from a simple gate to a complete electronic digital system

Difference from programming languages

Verilog

C/C++

Variable declaration

```
wire [7:0]   sum;
```

```
int   sum;
```

Procedural block

```
if (a == 8'b0) begin
        …
end
else begin
        …
end
```

```
if (a == 0) {
        …
}
else {
        …
}
```

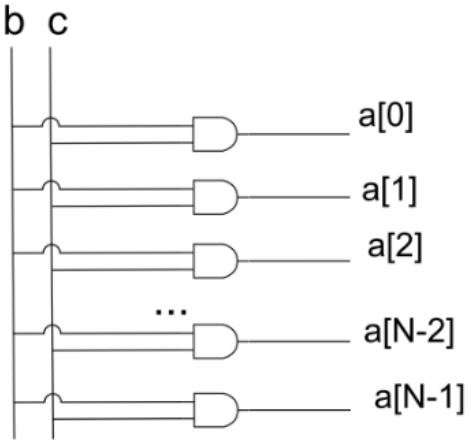
Difference from programming languages

Verilog

For Statement

```
wire [N-1:0] a;
generate
  for (i=0; i<N; i=i+1)
begin
      assign a[i] = b & c;
  end
endgenerate
```

Describes Hardware



C/C++

```
for (i=0; i<N; i++) {
      a[i] = b + c;
}
```

```
a[0] = b+c;
a[1] = b+c;
a[2] = b+c;
...
a[N-2] = b+c;
a[N-1] = b+c;
```

Describes Sequence of Instructions

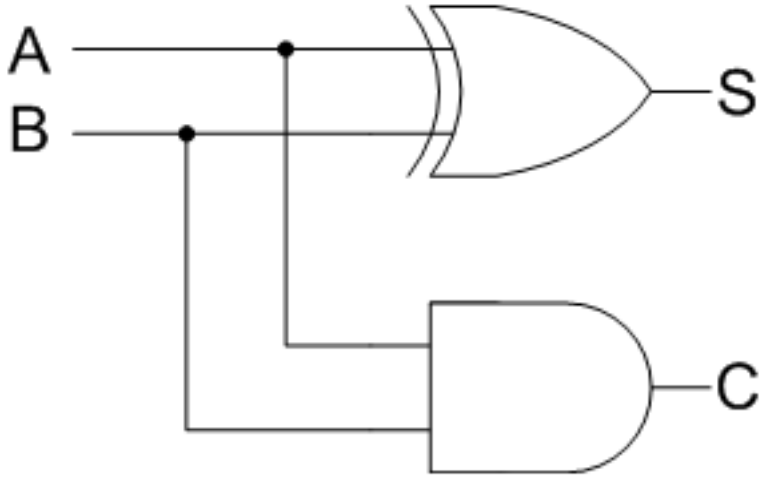
Types of modeling

Behavioral (High Level)

- Describes what the circuit should do, rather than how it is implemented
- Assignment statements, control statements

Structural (Gate Level)

- Describes the structure of the hardware components
- Interconnections of logic gates & modules



Behavioral vs Structural

Behavioral modeling: 4-to-1 MUX

```
module mux4x1 (input [1:0] sel, input [3:0] d,
output reg y);
   always @(*) begin
      case (sel)
         2'b00: y = d[0];
         2'b01: y = d[1];
         2'b10: y = d[2];
         2'b11: y = d[3];
         default: y = 0;
      endcase
   end
endmodule
```

Structural modeling: Half Adder

```
module half_adder (input a, input b,
output sum, output carry);
   xor(sum, a, b);  // Sum = A ⊕ B
   and(carry, a, b); // Carry = A & B
endmodule
```

Terminology

- **Register Transfer Level (RTL):** A type of behavioral modeling, for the purpose of synthesis
  - Describes how data moves between registers using combinational logics
- **Synthesis:** Translating HDL to a circuit (hardware logic) & then optimizing the represented
- **RTL Synthesis:** Translating an RTL model of hardware into an optimized technology-specific gate level implementation using synthesis tools like Xilinx Vivado

Comparison RTL & RTL Synthesis

RTL code (before synthesis)

```
module and_gate (input A, input B, output reg Y);
   always @(*) begin
      Y = A & B;  // RTL level logic
   end
endmodule
```

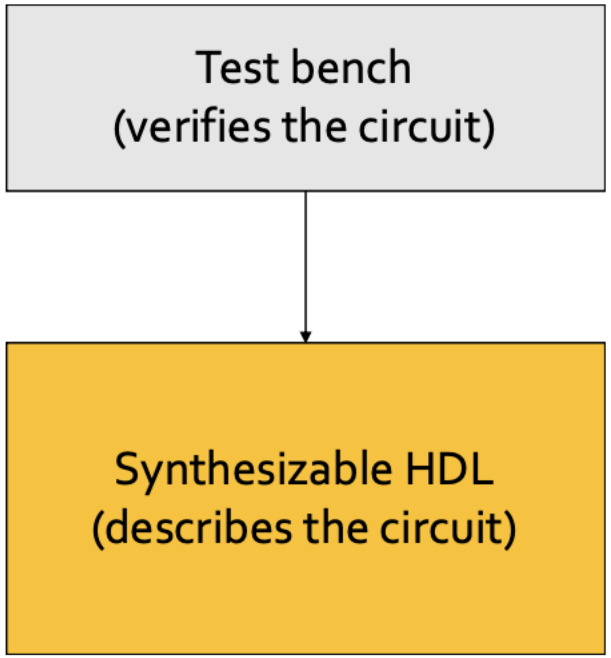
Synthesized Gate-Level Output

```
module and_gate (input A, input B, output Y);
   and U1 (Y, A, B);  // Replaced with actual AND
gate
endmodule
```

Simulation and Synthesis

The two major purposes of HDLs are logic simulation and synthesis

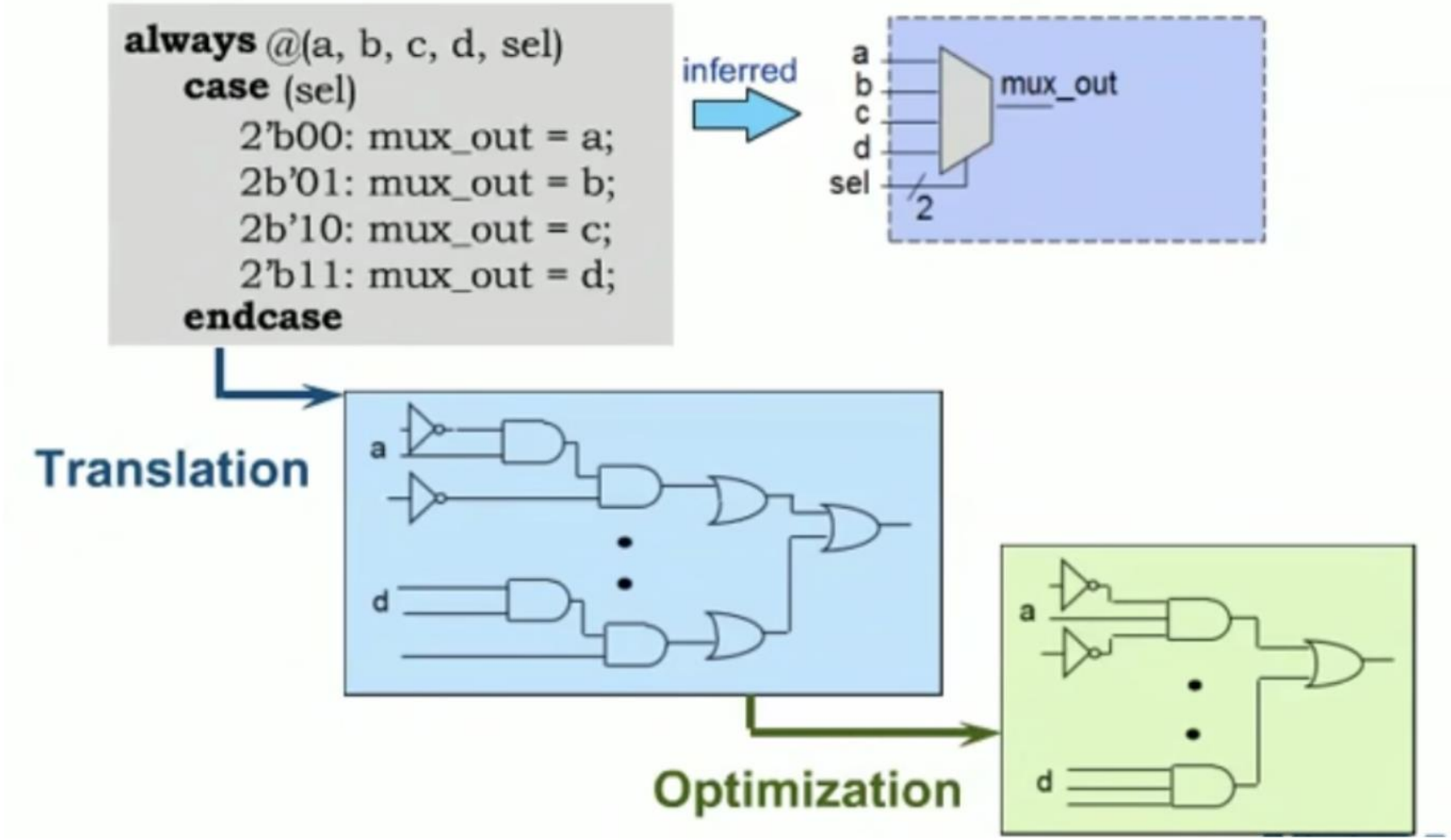
- Not all of the Verilog commands can be synthesized into hardware
- During **simulation:** inputs are applied to a module, and the outputs are checked to verify that module operates correctly
- During **synthesis:** textual description of a module is transformed into logic gates



HDL code is divided into synthesizable modules and a test bench

- **Synthesizable:** describes hardware
- **Test Bench:** Checks whether the output result is correct (only simulation)

Synthesis


always @(a, b, c, d, sel)
case (sel)
2'b00: mux_out = a;
2b'01: mux_out = b;
2b'10: mux_out = c;
2'b11: mux_out = d;
endcase
inferred
a
b
c
d
sel
2
mux_out
Translation
a
d
Optimization
a
d

TAC-HEP 2025

Verilog Syntax

Logical Values

A bit can have any of these values:

- **0** representing logic low (false)
- **1** representing logic high (true)
- **X** representing unknown value, 0, 1, or Z
- **Z** representing high impedance for tri-state (unconnected inputs are set to Z)

X & Z are case in-sensitive

Logic operations

&	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

AND operation

|	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

OR operation

Lexical elements

- Case sensitive: keywords are lower case
- Semicolons (;) are line terminators
- Comments:
  - One line //…..
  - Mutli-line comments start with /* and end with */
- System task & functions start with a dollar sign
  - Example: $display, $time, $signed

Lexical elements

- Variable names have to start with an alphabetic character or underscore (_) followed by alphanumeric or underscore characters

```
Count
COUNT //distinct from Count
_R2_D2
R56_68
FIVE$
```

- Escaped identifiers (\)
  - Permit non alphanumeric characters in Verilog names
  - The escaped name includes all the characters following the backslash until the first white space character

```
\7400
\.*.$
\{*****}
\OutGate // same as OutGate
```

```
wire \fo+o=a ;  // Declare the varaible fo+o=a=a
wire \fo+o =a ; // Assign a to wire fo+o
```

Compile Directives

- When compilers, remains in effect through the entire compilation process (which could span multiple files) until a different compiler directive specifies otherwise

- Certain identifiers that start with `` ` `` **(backquote)** character are compiler directives

  - `` `define, `undef ``
  - `` `ifdef, `else, `endif ``
  - `` `default_nettype ``
  - `` `include ``
  - `` `resetall ``
  - `` `timescale ``
  - `` `unconnected_drive, `nounconected_drive ``
  - `` `celldefine, `endcelldefine ``

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
```

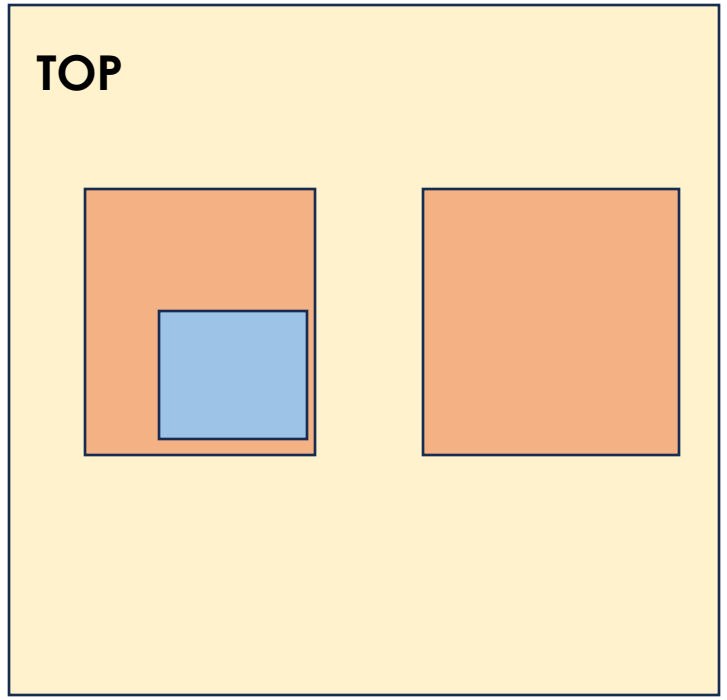
Basic components of Verilog

- Module
- Ports
- Data Types
- Operators
- Always Block
- Initial & Test bench
- Control Statements

Verilog Design

A digital system is built using hierarchical modules,

- At each level of hierarchy, different modules are connected to work together



Hierarchy:

- Design: multiple modules
- Module can contain sub-modules, forming a hierarchy
- Top module integrates all the sub-modules

Same level of hierarchy

- Modules can be **connected** and interact without one being inside another
- These modules communicate through **ports**

Module

Basic building block in Verilog, similar to a function in programming

- Defines the circuit & its behaviour

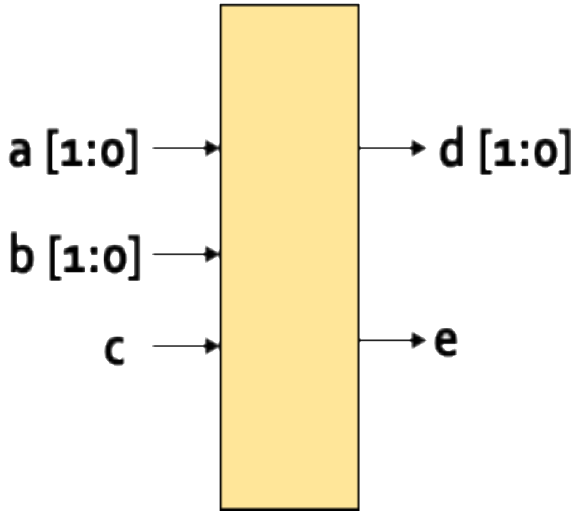
```
module <module name> #(<param list>) (<port list>)
  <Declarations>:
        reg, wire, parameters
        input, output, inout
  <Instantiations>
  <Data flow statements>
  <Behavioral blocks>
  <task and functions>
endmodule
```

```
module HalfAdder (A, B, Sum, Carry);
   input A, B;
   output Sum, Carry;

   assign #2 Sum = A ^ B;
   assign #5 Carry = A & B;
endmodule
```

Module Ports

```
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
```



Ports

- Input (input): Takes external signals
- Output (output): Produces results
- Bidirectional (inout): Can act as input and output

```
module full_adder (input a, input b, input cin, output
sum, output cout);
    assign sum = a ^ b ^ cin;  // XOR for sum
    assign cout = (a & b)  |  (b & cin)  |  (a & cin); //
Carry out
endmodule
```

Data Types

- **Nets** (wire)
  - Represents a physical connection b/w structural elements.
  - `wire' is used for combinational logic
  - Continously reflects the value assigned to it **(default z)**

```
wire y;
assign y = a & b; // y changes automatically when a or b changes
```

- **Registers** (reg)
  - Represents an abstract data storage element
  - Assigned values only within an always statement or an initial statement
  - Value is saved from one assignment to the next **(default x)**

```
reg q;
always @(posedge clk)
    q <= d; // q changes only on the clock edge
```

Vector and Arrays

Verilog vectors: known as **BUS** in hardware

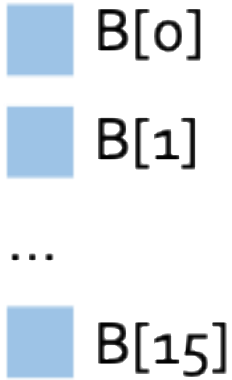
- <data type> [left range : right range] <Variable name>

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)

- A[5] // bit # 5 of vector A
- 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
- Reg [0:3] B [0:63]: //B is an array of sixty four 4-bit registers

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 wires

- **Slicing**
  - Wire [31:0] bus; //declare 4 byte bus
  - bus[7:0]; //lowest byte of bus
- **Concatenation**
  - {bus[7:0], bus[15:8], bus[23:16], bus[31:24]}
- **Replication**
  - {8{4'b1010}}; // replicate binary value 4'b1010 8 times

More above memory

On FPGA, memory maps to BRAM

Everything must be decided at compile time – your hardware cannot be changed while running!

- Cannot add one more piece of memory after the circuit is built!

```
int mem[var];
```

```
int mem* = malloc(var * sizeof(int));
```

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

Operators

Operator Type	Symbols	Example
Bitwise	~ & | ^	4'b1010 & 4'b0100
Logical	! && ||	4'b1010 && 4'b0100
Reduction	& ~& | ~| ^ ~^	|4'b0001
Arithmetic	+ - * / ** %	4'b1110 - 1
Relational	> < >= <= == != === !===	4'd5 < 4'd3
Shift	>> << >>> <<<	4'0110 << 1
Concatenation	{ , }	{2'b10, 2'b01}
Replication	{n{m}}	{8{1'b1}}
Conditional	? :	empty ? 1'b1 : 1'b0

```
wire result;
assign result = (a > b) ? a : b; // Assigns max(a, b) to result
```

Always Block

Used for

- **Combinational logic (always @(*))**

```
always @(*) begin
    sum = a + b;  // Executes when a or b changes
end
```

- **Sequential logic**

```
always @(posedge clk) begin
    q <= d;  // q updates at every positive clock edge
end
```

Blocking vs non-blocking

Blocking VS non-blocking assignments

- = blocking
- <= Non blocking, used only in always block

Blocking

Statement inside always block are executed **sequentially**

```
always @(*) begin
  b = a;
  c = b;
  d = b;
end
```

```
always @(*) begin
  a = b;
  b = c;
  c = a;
end
```

Non-Blocking

Statement inside always block are executed in **parallel**

```
always @(posedge clk) begin
  b <= a;
  c <= b;
  d <= b;
end
```

Intial & TestBench

Initial: Executes once, used in test benches for simulation

```
module testbench;
    reg a, b;
    wire y;

    and_gate uut (a, b, y); // Instantiate AND gate

    initial begin
        a = 0; b = 0; #10;
        a = 0; b = 1; #10;
        a = 1; b = 0; #10;
        a = 1; b = 1; #10;
        $finish;
    end
endmodule
```

Control statements

- if-else, case, for, while, repeat for behavioral modeling

Example: if-else

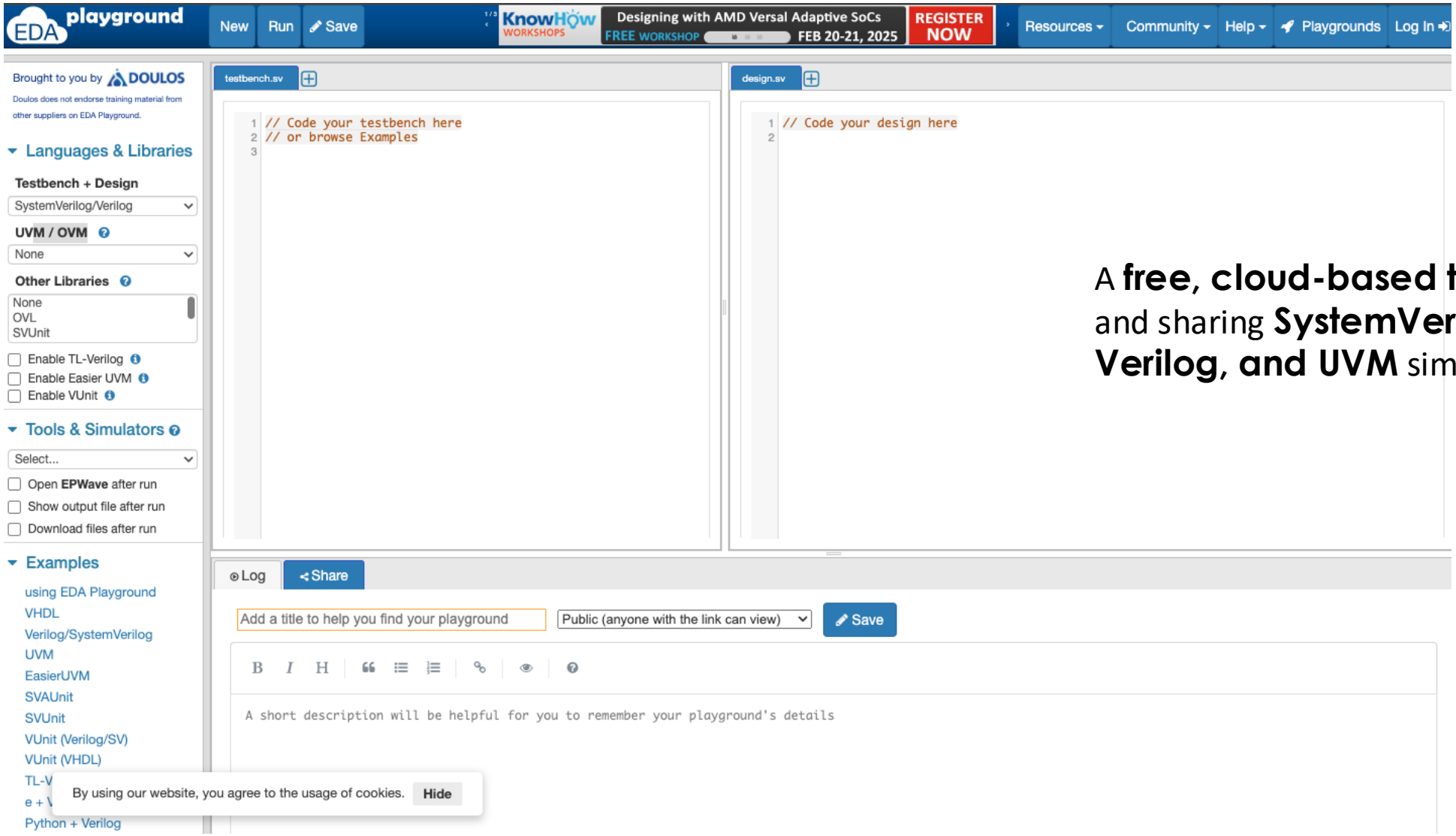
```
always @(*) begin
    if (a == 1)
        y = b;
    else
        y = c;
end
```

Example: case

```
always @(*) begin
    case (sel)
        2'b00: y = a;
        2'b01: y = b;
        2'b10: y = c;
        default: y = d;
    endcase
end
```

Verilog Playground

https://edaplayground.com/



A **free, cloud-based tool** for running and sharing **SystemVerilog, VHDL, Verilog, and UVM** simulations online.

Assignments

- Assignment-1 (13-02-2025)
- Assignment-2 (18-02-2025)

Uploaded to cernbox: https://cernbox.cern.ch/s/gmUqRDHTxDLqx4M

Submit in 2 weeks from date of assignment

TAC-HEP 2025

Questions?

Acknowledgements:
- Some of these slides are from Isobel Ojalvo

Jargons

- **ICs - Integrated chip:** assembly of hundreds of millions of transistors on a minor chip
- **PCB:** Printed Circuit Board
- **LUT - Look Up Table aka 'logic'** - generic functions on small bitwidth inputs. Combine many to build the algorithm
- **FF - Flip Flops** - control the flow of data with the clock pulse. Used to build the pipeline and achieve high throughput
- **DSP - Digital Signal Processor** - performs multiplication and other arithmetic in the FPGA
- **BRAM - Block RAM** - hardened RAM resource. More efficient memories than using LUTs for more than a few elements
- **PCIe or PCI-E - Peripheral Component Interconnect Express**: is a serial expansion bus standard for connecting a computer to one or more peripheral devices
- **InfiniBand** is a computer networking communications standard used in high-performance computing that features very high throughput and very low latency
- **HLS** - High Level Synthesis - compiler for C, C++, SystemC into FPGA IP cores
- **HDL** - Hardware Description Language - low level language for describing circuits
- **RTL** - Register Transfer Level - the very low level description of the function and connection of logic gates
- **FIFO** – First In First Out memory
- **Latency** - time between starting processing and receiving the result
  - Measured in clock cycles or seconds
- **II - Initiation Interval** - time from accepting first input to accepting next input