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

FPGA module training

Week-4

Lecture-8: February 20th 2025

Varun Sharma

University of Wisconsin – Madison, USA

Content

- Hardware Description Languages
  - Verilog

Basic mapping rules from C/C++ to RTL

RTL Components
Modules
I/O Ports
Functional units (adder, multiplier)
Wires or registers
Memory
Control logics (Finite State Machine)

Basic mapping rules from C/C++ to RTL

C Constructs		RTL Components
Functions	→	Modules
Arguments	→	I/O Ports
Operators (+, *)	→	Functional units (adder, multiplier)
Scalars	→	Wires or registers
Arrays	→	Memory
Control flows	→	Control logics (Finite State Machine)

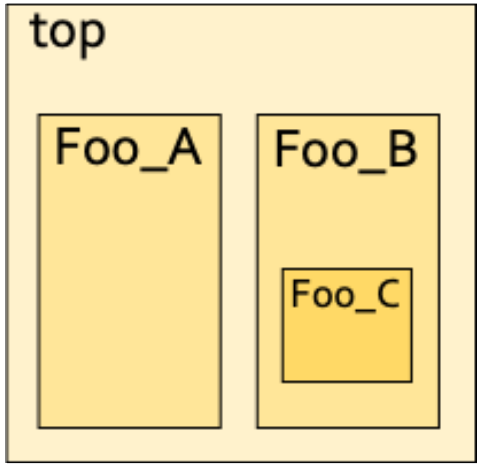
Basic mapping rules from C/C++ to RTL

C Constructs		RTL Components
Functions	→	Modules
Arguments	→	I/O Ports
Operators (+, *)	→	Functional units (adder, multiplier)
Scalars	→	Wires or registers
Arrays	→	Memory
Control flows	→	Control logics (Finite State Machine)

C Source Code

```
void Foo_C() {...}
void Foo_A() {...}
Void Foo_B() {
    Foo_C();
}

void main() {
    Foo_A();
    Foo_B();
    ...
    Foo_B();
}
```

RTL Hierarchy



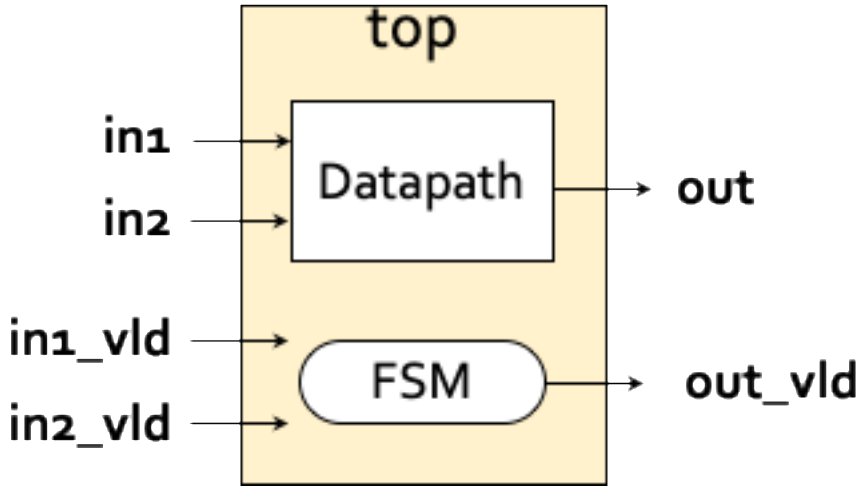
Resource sharing: Only one instance of Foo_B written to hardware

Basic mapping rules from C/C++ to RTL

C Constructs		RTL Components
Functions	→	Modules
Arguments	→	I/O Ports
Operators (+, *)	→	Functional units (adder, multiplier)
Scalars	→	Wires or registers
Arrays	→	Memory
Control flows	→	Control logics (Finite State Machine)

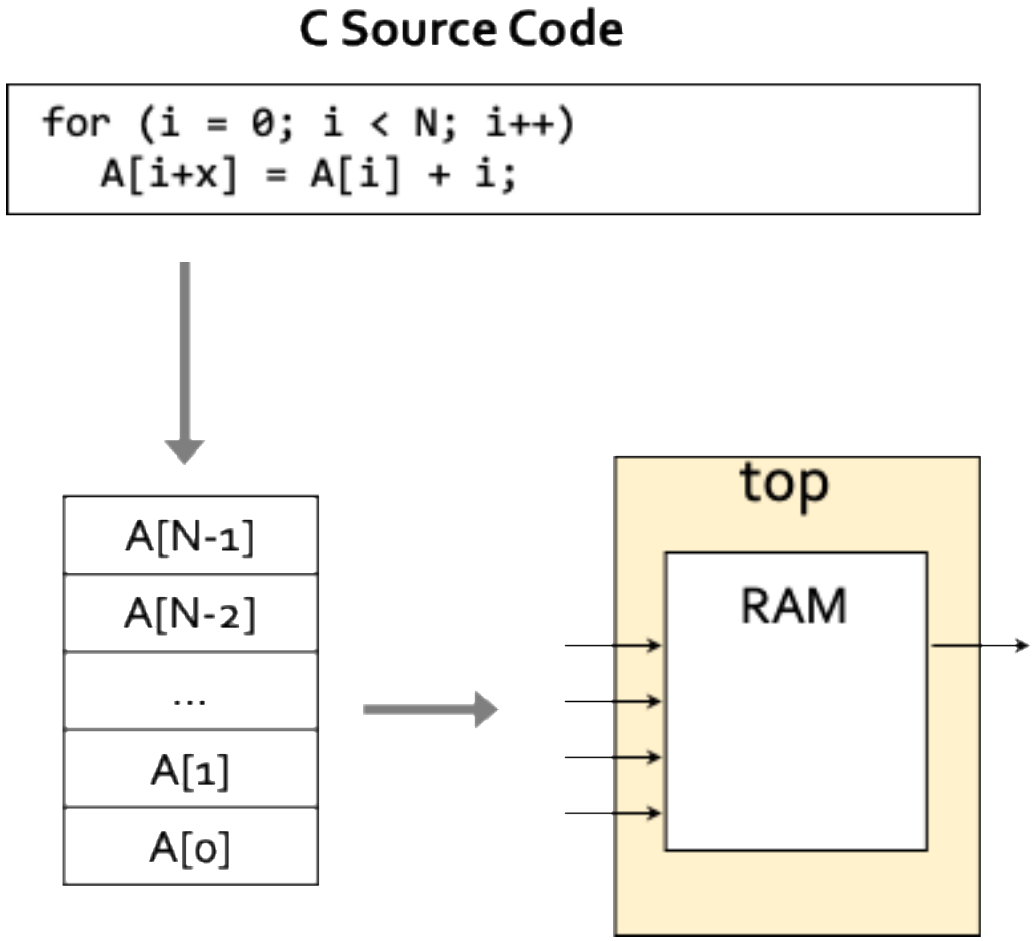
C Source Code

```
void top(int* in1, int* in2, int* out) {
    *out = *in1 + *in2;
}
```



Basic mapping rules from C/C++ to RTL

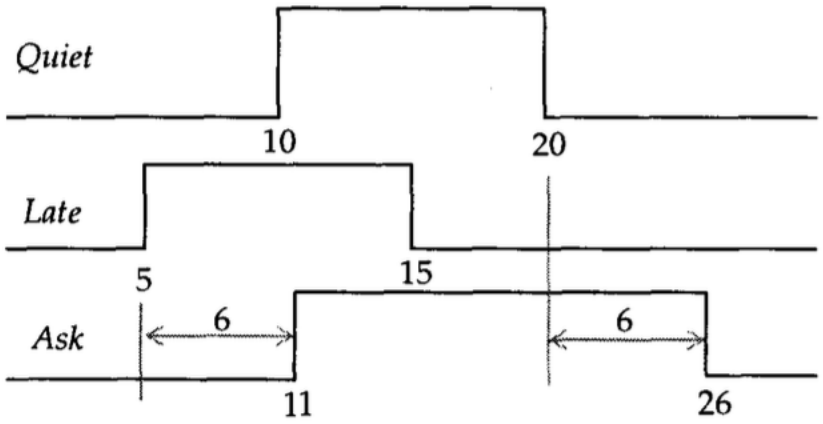
C Constructs		RTL Components
Functions	→	Modules
Arguments	→	I/O Ports
Operators (+, *)	→	Functional units (adder, multiplier)
Scalars	→	Wires or registers
Arrays	→	Memory
Control flows	→	Control logics (Finite State Machine)



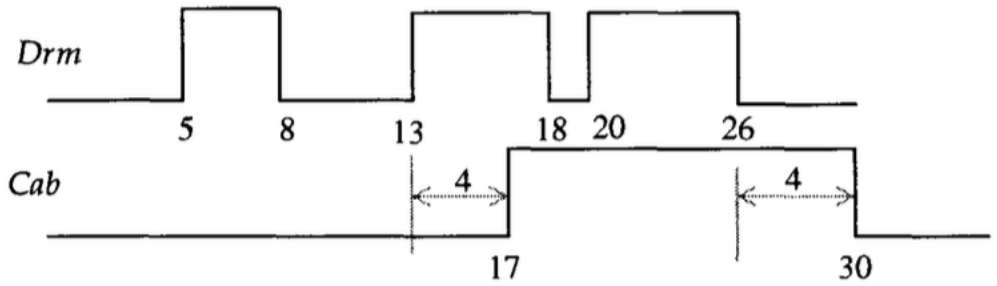
Delay

- A delay can be explicitly specified in a continuous assignment
  - **assign #6** Ask = Quiet | Late;

  - The delay specified #6 is the delay b/w RHS and LHS
  - Eg: if a change of value occurs on Late at time 5, then the expression on the RHS of the assignment is evaluated at time 5 and Ask will be assigned a new value at time 11.



assign #4 cab = drm



Verilog Statements

- **Procedural:**
  - Evaluated **sequentially**
  - **initial** block
    - Model a block of activity that is executed at the beginning
  - **always** blocks
    - Model a block of activity that is repeated continuously

```
always @ (x, y)
    begin
s = x^y;
c = x&y;  end
```

- **Concurrent:**
  - Evaluated in **parallel**
  - Order **NOT** important
  - Continuous assigment

```
assign  s=x^y;
assign  c=x&y;
assign  out=x|y;
```

Initial block

- Executes **only once**, at time **t = 0**
- Mostly used in **testbenches** for simulation (not synthesized in hardware)

```
module test;
   reg [3:0] a;

   initial begin
      a = 4'b0001;  // Assign 0001 at time 0
      #10 a = 4'b0010;  // After 10 time units, assign 0010
      #20 a = 4'b0100;  // After another 20 time units, assign 0100
   end
endmodule
```

Always Block

- All procedural statements must be within **always** (or **initial**) block
- Used for both **combinational** and **sequential** logic inference
  - Model an activity that is repeated continuously

- @ can control the execution
  - posedge or negedge: make sensitive to edge
  - @* / @(*) are sensitive to any signal that may read in the statement group
  - Use ","/ or multiple statements

Always Block

```
module M1 (input B, C, clk, rst, output reg X, Y,Z);
// controlled by any value change in B or C
always @ (B or C)
  X = B & C;

// Controlled by positive edge of clk
always @(posedge clk)
  Y = B & C;

// Controlled by negative edge of clk or rst
always @(negedge clk, negedge rst)
  if (!rst) Z = B & C;
  else      Z = B & C;
endmodule
```

Blocking/Non-blocking assignment

Blocking assignment (= operator)

- The whole statement is done before control passes on to the next statement.
- Similar to traditional programming languages
- Used in **combinational** logics

```
always @(posedge clk)
begin
   a = a + 1;
   b = a + 1;
end
```

Non-blocking assignment (<= operator)

- Executes **in parallel**, meaning it does **not** update the value immediately.
- The right-hand side (RHS) is evaluated at the start of the time step, and all updates happen **simultaneously** at the end.
- Used for **synchronous (sequential) logic**.

```
always @(posedge clk)
begin
   a <= a + 1;
   b <= a + 1;
end
```

Can we mix two?

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

Mistake to avoid: Using blocking (=) for sequential logic can cause unexpected behavior

Can we mix two?

```
always @(posedge clk) begin
    a = b;   // Immediate update
    c <= a;  // 'c' gets the OLD value of 'a'
end
```

Mistake to avoid: Using blocking (=) for sequential logic can cause unexpected behavior

Can we mix two?

```
always @(posedge clk) begin
    a = b;   // Immediate update
    c <= a;  // 'c' gets the OLD value of 'a'
end
```

Mistake to avoid: Using blocking (=) for sequential logic can cause unexpected behavior

- **Blocking (=)**: Executes **immediately**, used for **combinational logic**.
- **Non-Blocking (<=)**: Executes **in parallel**, used for **sequential logic** (flip-flops).
- Use **blocking (=) inside always @(*)** and **non-blocking (<=) inside always @(posedge clk)**.

Conditional statements (if … else)

```
if  (<expression>)
// statement1
else if (<expression>)
// statement2
else
// statement3
```

```
always @ (WRITE or STATUS)
begin
  if (!WRITE)
  begin
    out = oldvalue;
  end
  else if (!STATUS)
  begin
    q = newstatus;
  end
end
```

Conditional statements (case)

Case: used for switching between multiple selection
Casex treats Z and X as don't care
Casez treats Z as don't care

```
case (<expression>)
  <alternative 1> : <statement 1>;
  <alternative 2> : <statement 2>;
   default        : <default statement>;
endcase
```

```
always @(s, a, b, c, d)
case (s)
  2'b00: out = a;
  2'b01: out = b;
  2'b10: out = c;
  2'b11: out = d;
endcase
```

```
always @*
casez (state)
// 3'b11z, 3'b1zz,... match
3'b1??
3'b1??: fsm = 0;
3'b01?: fsm = 1;
endcase
```

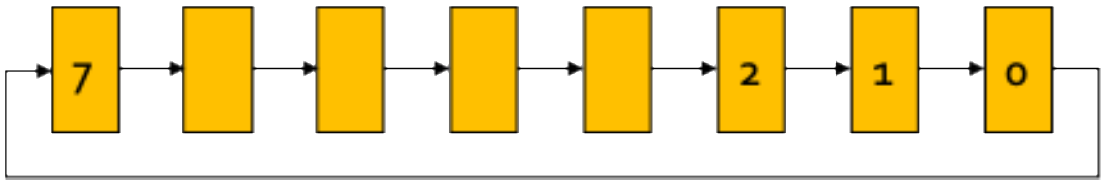
```
always @*
casex (state)
/*
during comparison : 3'b01z,
3'b01x, 3b'011 ... match case
3'b01x
*/
3'b01x:  fsm = 0 ;
3'b0xx:  fsm = 1 ;
default: fsm = 1 ;
endcase
```

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!**

```
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
```




```
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
```

Task and function

- Task & Function serve the same purpose on Verilog as subroutines do in C
- Reusable blocks of code with procedural blocks

Task:

- Declare with task & endtask
- May have **ZERO** or more arguments of type input, output, inout

- **DO NOT** return with a value, can pass value through output & inout argument
- **Used for complex operations**
- Can contain delays, control or timing statements
- **Usage: I/O operations, sequential logics**

Function:

- Declare with function & endfunction
- Must have at least one input

- Always return a single value
- **CANNOT** have output or inout arguments
- **Used for simple computations**
- Can't have delays or timing controls
- Can call only functions
- **Usage: Arithmetic operations**

Task and function

Task

```
module Has_Task;
  parameter MAXBITS = 8;

  task Reverse_Bits;
    input [MAXBITS-1 : 0] Din;
    output [MAXBITS-1 : 0] Dout;
    integer K;

    begin
      for (K = 0; K < MAXBITS; K = K + 1)
        Dout [MAXBITS-K] = Din [K];
    end
  endtask
  . . .
endmodule
```

Function

```
module Function_Example;
  parameter MAXBITS = 8;

  function [MAXBITS-1 : 0] Reverse_Bits;
    input [MAXBITS-1 : 0] Din;
    integer K;
    begin
      for (K = 0; K < MAXBITS; K = K + 1)
        Reverse_Bits [MAXBITS-K-1] = Din [K];
    end
  endfunction
  . . .
endmodule
```

Generate Blocks

- Used to create **repetitive** or **conditional hardware structures** at **compile time**
- Advantageous in designing parameterized, scalable, and reusable hardware.

- 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

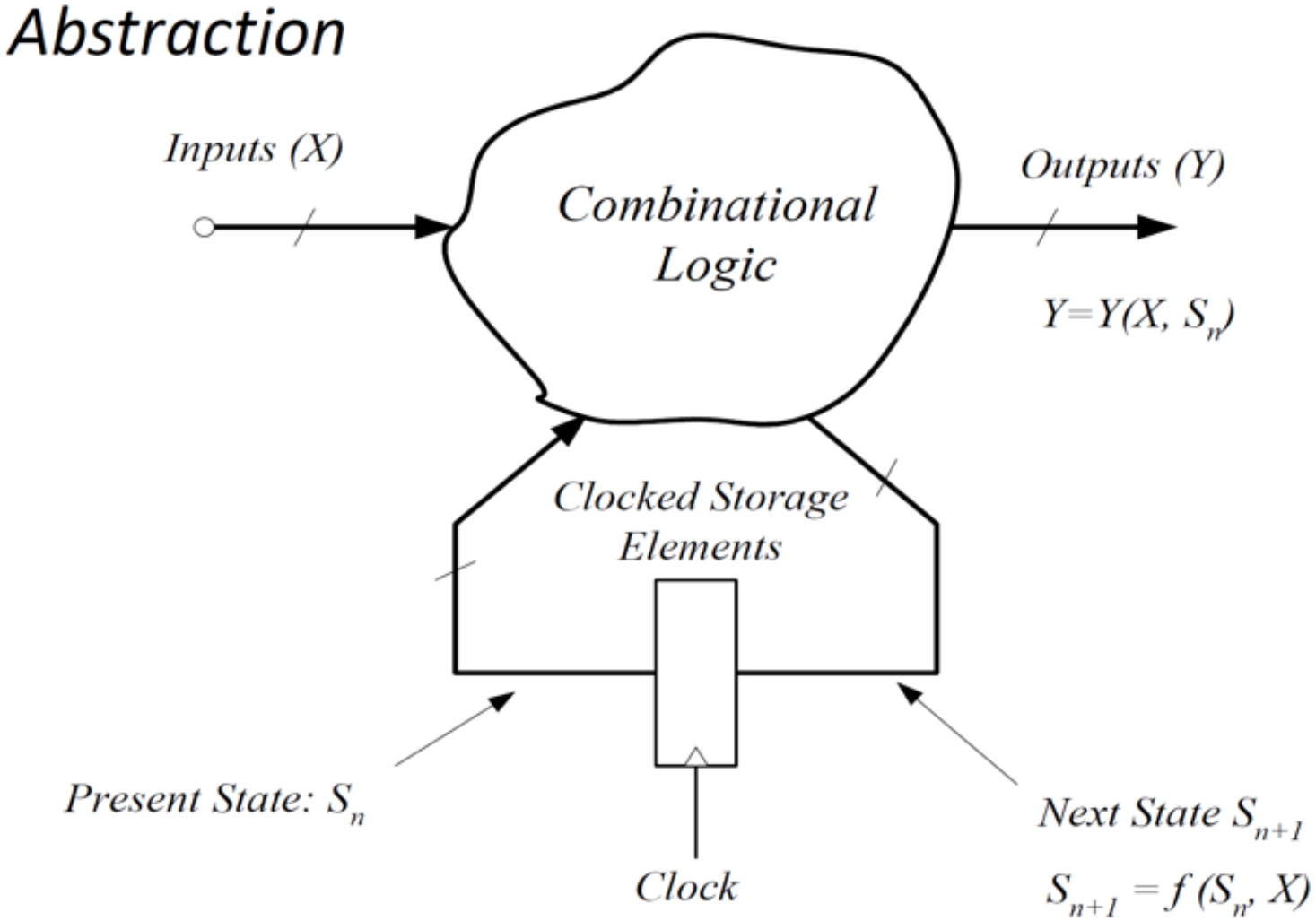
Generate Blocks

```
module generate_example (
    input wire clk,
    input wire [3:0] d,
    output reg [3:0] q
);
    genvar i;  // Generate loop variable

    generate
        for (i = 0; i < 4; i = i + 1) begin : gen_ff  // Label: gen_ff
            always @(posedge clk)
                q[i] <= d[i];  // Create 4 flip-flops
        end
    endgenerate
endmodule
```

Conditional Instantiation

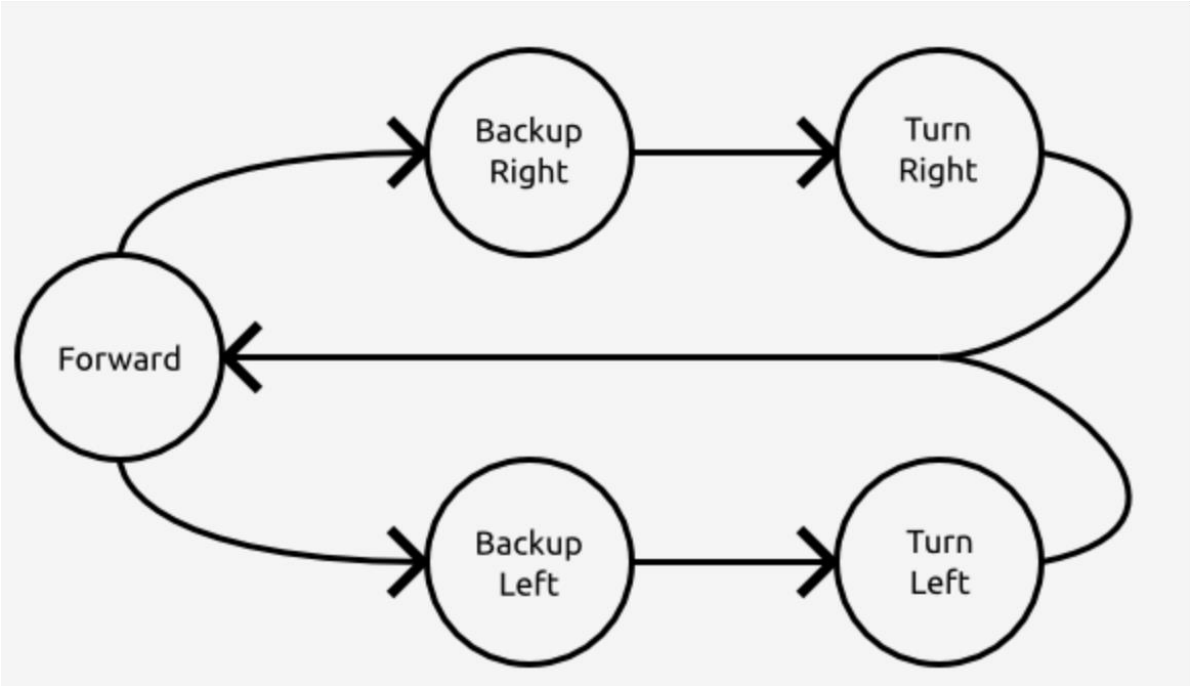
```
module conditional_generate (
    input wire clk, d,
    output reg q
);
    parameter USE_FF = 1;  // Set to 0 to disable flip-flop

    generate
        if (USE_FF) begin
            always @(posedge clk)
                q <= d;  // Flip-flop instantiated if USE_FF = 1
        end
    endgenerate
endmodule
```

Finite State Machines

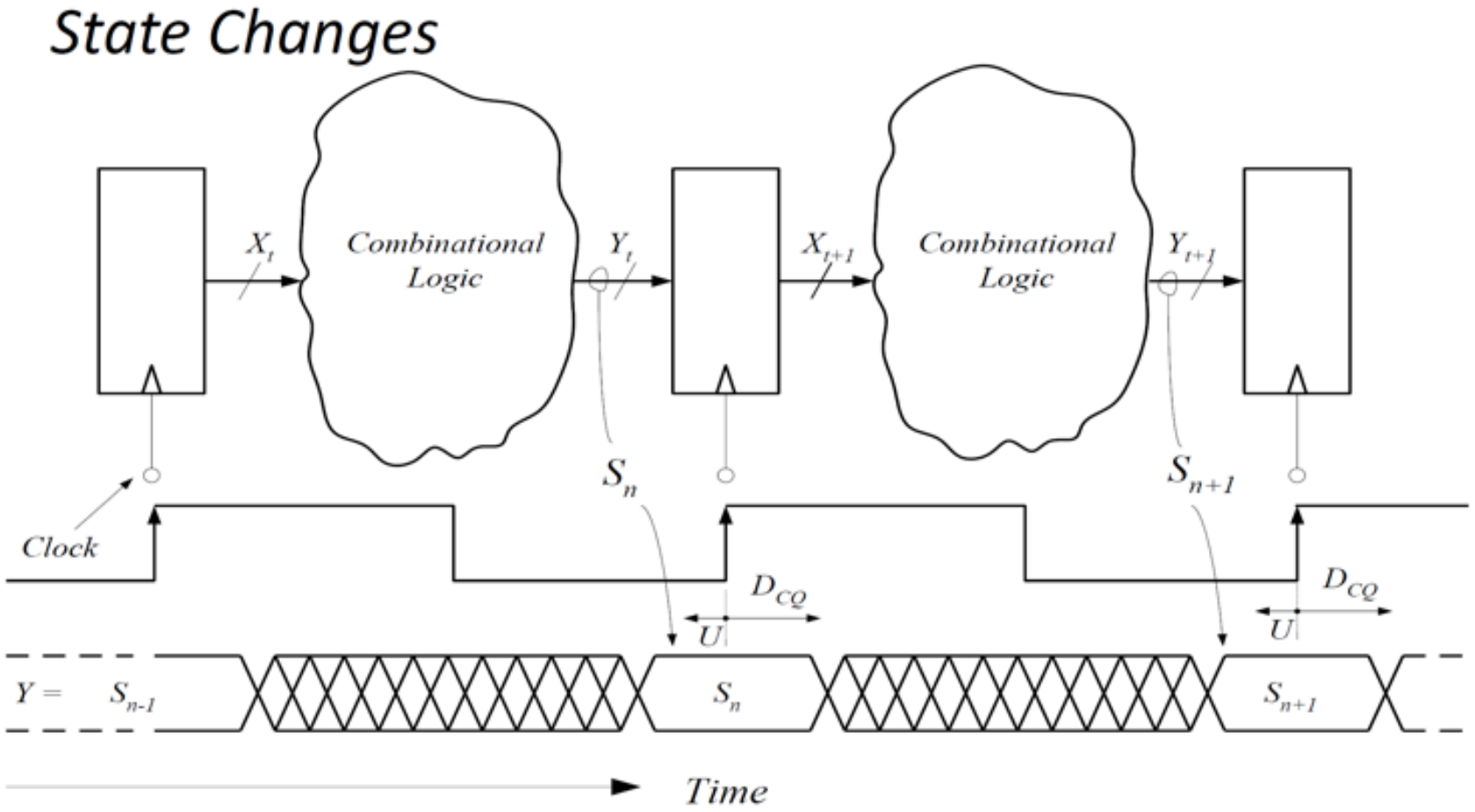


FSM: Robot Examples

- The robot can realize when it runs into something
  - For some basic object avoidance, the robot backs up when it hits something then turns right if the left switch is pressed or turns right if the right switch is pressed
  - When the robot hasn't hit anything, it will just drive forward



Finite State Machine



Some Good Practices

- Value assignments
  - For correct simulation results, use non-blocking assignments within sequential Verilog always blocks
  - DO NOT mix blocking non blocking assignments
  - DO NOT make assignment to same variable from more than one always block

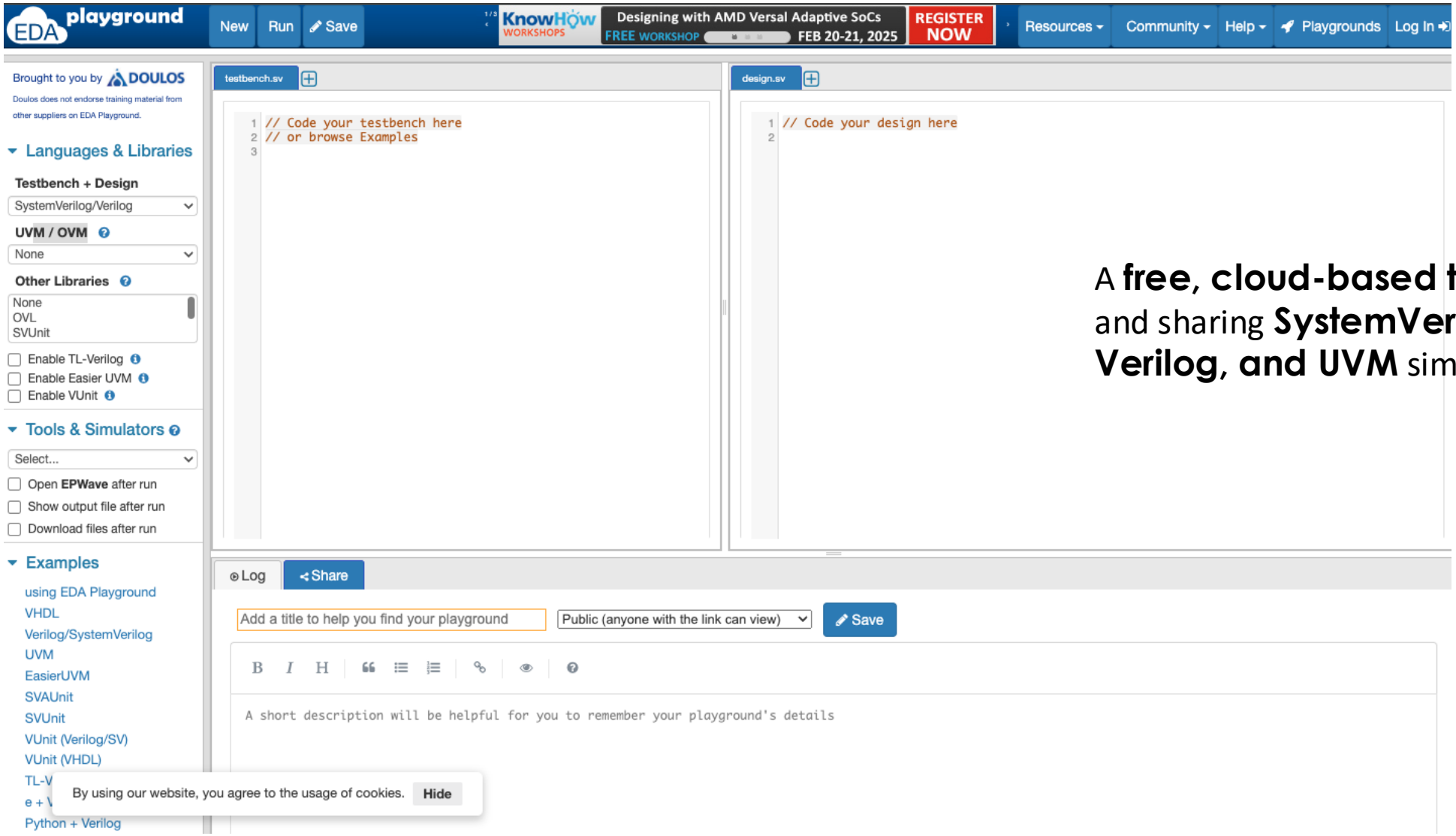
- Case statements
  - If your if statement contain more than three conditions, consider using case statement to improve the parallelism of your design and clarity of code
  - An incomplete case statement results in the creating of a latch

Some Good Practices

- Constant Definitions
  - Use the Verilog `` `define `` statement to define global constants
  - Keep the definitions in a separate file
- Guidelines for Identifiers
  - Ensure that the signal name conveys the meaning of the signal or the value of a variable without being verbose
- Never use high-impedance values in a conditional expression
- All loops should have finite pre-defined lenght

Verilog Playground

https://edaplayground.com/



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

Reminder: Assignments

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

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

Send via email: **varun.sharma@cern.ch**

Submit in 2 weeks from date of assignment

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

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