Agenda for Today

- Hardware Description Languages
- Implementing Combinational Logic (in Verilog)
- Implementing Sequential Logic (in Verilog)

- The Verilog slides constitute a tutorial. We may not cover all.
- All slides will be beneficial for your labs.
Why Specialized Languages for Hardware?

- HDLs enable easy description of hardware structures
  - Wires, gates, registers, flip-flops, clock, rising/falling edge, ...
  - Combinational and sequential logic elements

- HDLs enable seamless expression of parallelism inherent in hardware
  - All hardware logic operates concurrently

- Both of the above ease specification, simulation & synthesis
Hardware Design Using HDL
Structural (Gate-Level) HDL
Behavioral HDL
Recall: Two Main Styles of HDL Implementation

- **Structural (Gate-Level)**
  - The module body contains gate-level description of the circuit
  - Describe how modules are interconnected
  - Each module contains other modules (instances)
  - ... and interconnections between those modules
  - Describes a hierarchy of modules defined as gates

- **Behavioral**
  - The module body contains functional description of the circuit
  - Contains logical and mathematical operators
  - **Level of abstraction is higher than gate-level**
    - Many possible gate-level realizations of a behavioral description

- Many practical designs use a combination of both
Recall: What Happens with HDL Code?

- **Synthesis (i.e., Hardware Synthesis)**
  - Modern tools are able to map *synthesizable* HDL code into low-level *cell libraries* → *netlist describing gates and wires*
  - They can perform many optimizations
  - ... however they *can not* guarantee that a solution is optimal
    - Mainly due to *computationally expensive placement and routing algorithms*
    - Need to describe your circuit in HDL in a nice-to-synthesize way
  - Most common way of Digital Design these days

- **Simulation**
  - Allows the behavior of the circuit to be *verified without actually manufacturing the circuit*
  - Simulators can work on *structural* or *behavioral* HDL
  - Simulation is essential for functional and timing verification
Implementing Sequential Logic Using Verilog
Sequential = Combinational + Memory
Sequential Logic in Verilog

- We can describe hardware that has memory
  - *Flip-Flops, Latches, Finite State Machines*

- **Sequential Logic** state transition is triggered by a “CLOCK” signal
  - Latches are sensitive to level of the signal
  - Flip-flops are sensitive to the transitioning of signal

- Combinational HDL constructs are **not** sufficient to express sequential logic
  - We need **new constructs**:
    - *always*
    - *posedge/negedge*
The “always” Block

always @ (sensitivity list)
statement;

Whenever the event in the sensitivity list occurs, the statement is executed
Recall: The D Flip-Flop

- 1) state change on clock edge, 2) data available for full cycle

- When the clock is low, 1\textsuperscript{st} latch propagates D to the input of the 2\textsuperscript{nd} (Q unchanged)
- Only when the clock is high, 2\textsuperscript{nd} latch latches D (Q stores D)
  - At the rising edge of clock (clock going from 0->1), Q gets assigned D
Recall: The D Flip-Flop

1) state change on clock edge, 2) data available for full cycle

- At the rising edge of clock (clock going from 0->1), \( Q \) gets assigned \( D \)
- At all other times, \( Q \) is unchanged
Recall: The D Flip-Flop

- 1) state change on clock edge, 2) data available for full cycle

At the rising edge of clock (clock going from 0->1), \( Q \) gets assigned \( D \)

At all other times, \( Q \) is unchanged

We can use **D Flip-Flops** to implement the state register
Example: D Flip-Flop

```verilog
module flop(input clk,
            input [3:0] d,
            output reg [3:0] q);

    always @(posedge clk)
    q <= d; // pronounced “q gets d”

endmodule
```

- **posedge** defines a rising edge (transition from 0 to 1).
- Statement executed when the **clk** signal rises (**posedge of clk**)
- Once the **clk** signal rises: the value of **d** is copied to **q**
Example: D Flip-Flop

module flop(input clk,
            input [3:0] d,
            output reg [3:0] q);

    always @(posedge clk)
    q <= d; // pronounced “q gets d”
endmodule

- **assign** statement is **not** used within an always block
- **<=** describes a **non-blocking** assignment
  - We will see the difference between **blocking assignment** and **non-blocking** assignment soon
Example: D Flip-Flop

module flop(input clk, 
            input [3:0] d, 
            output reg [3:0] q);

always @ (posedge clk)
  q <= d; // pronounced “q gets d”

endmodule

- Assigned variables need to be declared as reg
- The name reg does not necessarily mean that the value is a register (It could be, but it does not have to be)
- We will see examples later
Asynchronous and Synchronous Reset

- **Reset** signals are used to *initialize* the hardware to a known state
  - Usually activated *at system start* (on power up)

- **Asynchronous Reset**
  - The reset signal is sampled *independent of the clock*
  - Reset gets the highest priority
  - Sensitive to *glitches*, may have *metastability* issues
    - Will be discussed in the Timing & Verification Lecture

- **Synchronous Reset**
  - The reset signal is sampled *with respect to the clock*
  - The reset *should be active long enough* to get sampled at the clock edge
  - Results in *completely synchronous circuit*
Recall: Asynchronous vs. Synchronous State Changes

- Sequential lock we saw is an asynchronous “machine”
  - State transitions occur when they occur
  - There is nothing that synchronizes when each state transition must occur

- Most modern computers are synchronous “machines”
  - State transitions take place after fixed units of time
  - Controlled in part by a clock, as we will see soon

- These are two different design paradigms, with tradeoffs
  - Synchronous control can be easier to get correct when the system consists of many components and many states
  - Asynchronous control can be more efficient (no clock overheads)

We will assume synchronous systems in most of this course
D Flip-Flop with Asynchronous Reset

```verilog
module flop_ar (input clk, reset, [3:0] d, output reg [3:0] q);

always @ (posedge clk, negedge reset)
begin
    if (reset == 0) q <= 0; // when reset
    else q <= d; // when clk
end
endmodule
```

- In this example: two events can trigger the process:
  - A *rising edge* on clk
  - A *falling edge* on reset
D Flip-Flop with Asynchronous Reset

```vhdl
module flop_ar (input clk, reset, [3:0] d, output reg [3:0] q);

always @ (posedge clk, negedge reset)
begin
  if (reset == 0) q <= 0;  // when reset
  else q <= d;            // when clk
end
endmodule
```

- For longer statements, a `begin-end` pair can be used
  - To improve readability
  - In this example, it was not necessary, but it is a good idea
D Flip-Flop with *Asynchronous* Reset

```verbatim
d Flip - Flop with Asynchronous Reset

module flop_ar (input clk, input reset, input [3:0] d, output reg [3:0] q);

always @ (posedge clk, negedge reset)
begin
    if (reset == 0) q <= 0; // when reset
    else q <= d; // when clk
end
endmodule
```

- First `reset` is checked: if `reset` is 0, `q` is set to 0.
  - This is an *asynchronous* reset as the reset can happen independently of the clock (on the negative edge of reset signal)
- If there is no reset, then regular assignment takes effect
D Flip-Flop with Synchronous Reset

```verilog
module flop_sr (input clk, reset, [3:0] d,
output reg [3:0] q);

always @ (posedge clk)
begin
  if (reset == '0') q <= 0;  // when reset
  else q <= d;            // when clk
end
endmodule
```

- The process is sensitive to only clock
  - Reset *happens only* when the *clock rises*. This is a *synchronous* reset
D Flip-Flop with Enable and Reset

```verilog
module flop_en_ar (input clk, input reset, input en, input [3:0] d, output reg [3:0] q);

always @ (posedge clk, negedge reset)
begin
  if (reset == '0') q <= 0; // when reset
  else if (en) q <= d;    // when en AND clk
end
endmodule
```

- A flip-flop with **enable** and **reset**
  - Note that the **en** signal is **not** in the **sensitivity list**
- **q** gets **d** only when **clk** is rising and **en** is **1**
Example: D Latch

```verilog
module latch (input clk, 
              input [3:0] d,
              output reg [3:0] q);

always @ (clk, d)
    if (clk) q <= d;  // latch is transparent when
                       // clock is 1
endmodule
```
Sequential statements are within an `always` block.

The sequential block is triggered with a change in the sensitivity list.

Signals assigned within an `always` must be declared as `reg`.

We use `<=` for (non-blocking) assignments and do not use `assign` within the always block.
Basics of `always` Blocks

```verilog
define module example (input clk, input [3:0] d, output reg [3:0] q);
wire [3:0] normal; // standard wire
reg [3:0] special; // assigned in always

always @(posedge clk)
special <= d; // first FF array

assign normal = ~special; // simple assignment

always @(posedge clk)
q <= normal; // second FF array
endmodule
```

You can have as many `always` blocks as needed.
Assignment to the same signal in different `always` blocks is not allowed!
Why Does an **always** Block Remember?

```verilog
module flop (input clk,
             input [3:0] d,
             output reg [3:0] q);

  always @(posedge clk)
  begin
    q <= d; // when clk rises copy d to q
  end
endmodule
```

- This statement describes what happens to signal `q`
- ... but what happens when the clock is not rising?
- The value of `q` is preserved (remembered)
An always Block Does NOT Always Remember

module comb (input inv,
            input [3:0] data,
            output reg [3:0] result);

always @ (inv, data)       // trigger with inv, data
    if (inv) result <= ~data; // result is inverted data
    else result <= data;      // result is data
endmodule

- This statement describes what happens to signal result
  - When inv is 1, result is ~data
  - When inv is not 1, result is data

- The circuit is combinational (no memory)
  - result is assigned a value whenever an input value changes & in all cases of the if .. else block
always Blocks for Combinational Circuits

- An always block defines combinational logic if:
  - All outputs are always (continuously) updated
  1. All right-hand side signals are in the sensitivity list
     - You can use always @* for short
  2. All left-hand side signals get assigned in every possible condition of if .. else and case blocks

- It is easy to make mistakes and unintentionally describe memorizing elements (latches)
  - Vivado will most likely warn you. Make sure you check the warning messages

- Always blocks allow powerful combinational logic statements
  - if .. else
  - case
Sequential or Combinational?

**Sequential**

```verilog
wire enable, data;
reg out_a, out_b;

always @ (*) begin
  out_a = 1'b0;
  if(enable) begin
    out_a = data;
    out_b = data;
  end
end
```

**Sequential**

```verilog
wire enable, data;
reg out_a, out_b;

always @ (data) begin
  out_a = 1'b0;
  if(enable) begin
    out_a = data;
    out_b = data;
  end
end
```

*No assignment for ~enable*
*Not in the sensitivity list*
The **always** Block is **NOT** Always Practical/Nice

```vhdl
reg [31:0] result;
wire [31:0] a, b, comb;
wire sel,

always @ (a, b, sel)  // trigger with a, b, sel
  if (sel) result <= a; // result is a
else result <= b; // result is b

assign comb = sel ? a : b;
```

- Both statements describe the **same** multiplexer
- In this case, the **always** block is more work
always Block for Case Statements (Handy!)

```verilog
module sevensegment (input [3:0] data,
                      output reg [6:0] segments);

always @ ( * ) // * is short for all signals
   case (data) // case statement
       4'd0: segments = 7'b111_1110; // when data is 0
       4'd1: segments = 7'b011_0000; // when data is 1
       4'd2: segments = 7'b110_1101;
       4'd3: segments = 7'b111_1001;
       4'd4: segments = 7'b011_0011;
       4'd5: segments = 7'b101_1011;
                         // etc etc
   default: segments = 7'b000_0000; // required
   endcase

endmodule
```
Summary: always Block

- **if .. else** can **only** be used in **always** blocks

- The **always** block is **combinational** only if all **regs** within the block are always assigned to a signal
  - Use the **default** case to make sure you do not forget an unimplemented case, which may otherwise result in a latch

- Use **casex** statement to be able to check for don’t cares
Non-Blocking and Blocking Assignments

Non-blocking ($\leq$)

```verilog
always @ (a)
begin
  a <= 2'b01;
  b <= a;
// all assignments are made here
// b is not (yet) 2'b01
end
```

- All assignments are made at the end of the block
- All assignments are made in parallel, process flow is not-blocked

Blocking ($=$)

```verilog
always @ (a)
begin
  a = 2'b01;
  b = a;
// b is now 2'b01 as well
end
```

- Each assignment is made immediately
- Process waits until the first assignment is complete, it blocks progress
- Similar to sequential programs
Why Use (Non)-Blocking Statements

- Non-blocking statements allow operating on “old” values
  - Enable easy sequential logic descriptions

- Blocking statements allow a sequence of operations
  - Allow operating on immediately updated values
  - More like a “software” programming language

- If the sensitivity list is correct, a block with non-blocking statements will eventually evaluate to the same result as the same block with blocking statements
  - This may require some additional iterations
Example: Blocking Assignment

- Assume all inputs are initially ‘0’

```verbatim
always @( * )
begin
  p    = a ^ b ;  // p    = 0  1
  g    = a & b ;   // g    = 0  0
  s    = p ^ cin ; // s    = 0  1
  cout = g | (p & cin) ; // cout = 0  0
end
```

- If a changes to ‘1’
  - All values are updated in order
Assume all inputs are initially ‘0’

always @ (* )
begin
    p <= a ^ b ;  // p = 0 1
    g <= a & b ;  // g = 0 0
    s <= p ^ cin ;  // s = 0 0
    cout <= g | (p & cin) ;  // cout = 0 0
end

If a changes to ‘1’
- All assignments are concurrent
- When s is being assigned, p is still 0
The Same Example: Non-Blocking Assignment

- After the first iteration, \( p \) has changed to ‘1’ as well

```
always @ ( * )
begin
  p    <= a ^ b ;   // p    = 1 1
  g    <= a & b ;   // g    = 0 0
  s    <= p ^ cin ; // s    = 0 1
  cout <= g | (p & cin) ; // cout = 0 0
end
```

- Since there is a change in \( p \), the process triggers again
- This time \( s \) is calculated with \( p=1 \)
Rules for Signal Assignment

- Use `always @(posedge clk)` and non-blocking assignments (`<=`) to model synchronous sequential logic

```vhdl
always @ (posedge clk)
q <= d; // non-blocking
```

- Use continuous assignments (`assign`) to model simple combinational logic

```vhdl
assign y = a & b;
```
Rules for Signal Assignment (Cont.)

- Use **always @ (*)** and **blocking assignments (==)** to model more **complicated combinational logic**

- You **cannot** make assignments to the **same** signal in more than one **always** block or in a **continuous assignment**

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

```
always @ (*)
  a = b;
assign a = c;
```
Recall: Finite State Machines (FSMs)

- Each FSM consists of three separate parts:
  - next state logic
  - state register
  - output logic

At the beginning of the clock cycle, next state is latched into the state register.
Recall: Finite State Machines (FSMs) Consist of:

- **Sequential Circuits**
  - State register(s)
    - Store the current state and
    - Load the next state at the clock edge

- **Combinational Circuits**
  - Next state logic
    - Determines what the next state will be
  - Output logic
    - Generates the outputs
FSM Example 1: Divide the Clock Frequency by 3

The output $Y$ is HIGH for one clock cycle out of every 3. In other words, the output divides the frequency of the clock by 3.
Implementing FSM Example 1: Definitions

```verilog
module divideby3FSM (input clk,
    input reset,
    output q);

reg [1:0] state, nextstate;

parameter S0 = 2'b00;
parameter S1 = 2'b01;
parameter S2 = 2'b10;
```

- We define `state` and `nextstate` as 2-bit `reg`
- The parameter descriptions are `optional`, it makes reading easier
Implementing FSM Example 1: State Register

This part defines the state register (memorizing process)
- Sensitive to only clk, reset
- In this example, reset is active when it is ‘1’ (active-high)
// next state logic

```vhdl
always @ (*)
  case (state)
    S0:    nextstate = S1;
    S1:    nextstate = S2;
    S2:    nextstate = S0;
    default: nextstate = S0;
  endcase
```

Implementing FSM Example 1: Output Logic

// output logic
assign q = (state == S0);

- In this example, output depends only on state
  - **Moore type FSM**
module divideby3FSM (input clk, input reset, output q);
  reg [1:0] state, nextstate;

  parameter S0 = 2'b00; parameter S1 = 2'b01; parameter S2 = 2'b10;

  always @(posedge clk, posedge reset) // state register
    if (reset) state <= S0;
    else    state <= nextstate;

  always @(*) // next state logic
    case (state)
      S0:    nextstate = S1;
      S1:    nextstate = S2;
      S2:    nextstate = S0;
      default: nextstate = S0;
    endcase

  assign q = (state == S0); // output logic
endmodule
FSM Example 2: Recall the Smiling Snail

- Alyssa P. Hacker has a snail that crawls down a paper tape with 1’s and 0’s on it
- The snail smiles whenever the last four digits it has crawled over are 1101
- Design Moore and Mealy FSMs of the snail’s brain
module SmilingSnail (input clk,
    input reset,
    input number,
    output smile);

reg [1:0] state, nextstate;

parameter S0 = 2'b00;
parameter S1 = 2'b01;
parameter S2 = 2'b10;
parameter S3 = 2'b11;
Implementing FSM Example 2: State Register

```
// state register
always @ (posedge clk, posedge reset)
  if (reset) state <= S0;
  else state <= nextstate;
```

- This part defines the **state register** (memorizing process)
- Sensitive to only **clk, reset**
- In this example **reset** is active when ‘1’ (active-high)
Implementing FSM Example 2: Next State Logic

// next state logic
always @ (*)
    case (state)
        S0: if (number) nextstate = S1;
            else nextstate = S0;
        S1: if (number) nextstate = S2;
            else nextstate = S0;
        S2: if (number) nextstate = S2;
            else nextstate = S3;
        S3: if (number) nextstate = S1;
            else nextstate = S0;
        default: nextstate = S0;
    endcase
Implementing FSM Example 2: Output Logic

// output logic
assign smile = (number & state == S3);

- In this example, output depends on state and input
  - Mealy type FSM

- We used a simple combinational assignment
module SmilingSnail (input clk,  
  input reset,  
  input number,  
  output smile);

reg [1:0] state, nextstate;

parameter S0 = 2'b00;  
parameter S1 = 2'b01;  
parameter S2 = 2'b10;  
parameter S3 = 2'b11;

// state register
always @ (posedge clk, posedge reset) 
  if (reset) state <= S0;  
  else state <= nextstate;

always @ (*) // next state logic 
  case (state) 
    S0: if (number) 
      nextstate = S1; 
      else nextstate = S0; 
    S1: if (number) 
      nextstate = S2; 
      else nextstate = S0; 
    S2: if (number) 
      nextstate = S2; 
      else nextstate = S3; 
    S3: if (number) 
      nextstate = S1; 
      else nextstate = S0; 
  default: nextstate = S0;
endcase  

// output logic
assign smile = (number & state==S3);
endmodule
What Did We Learn?

- Basics of describing sequential circuits in Verilog

- The **always** statement
  - Needed for describing memorizing elements (**flip-flops, latches**)
  - Can also be used to describe **combinational circuits**

- **Blocking** vs **Non-blocking** statements
  - `=` assigns the value **immediately**
  - `<=` assigns the value **at the end of the block**

- **Describing FSMs in Verilog**
  - Next state logic
  - State assignment
  - Output logic
Next Lecture:
Timing and Verification