Topics

1. Programmable logic
2. Design Flow
3. Verilog -- Hardware definition languages

module fulladd(output reg[3:0] sum, output reg c_out, input [3:0] a, b, input c_in);
...
endmodule
1. Programmable logic

- A logic element (LE) consist of a k-input look up table (LUT) and a flip-flop
A computing circuit must be mapped into LEs

- Tools will take care of compiling your digital design
Field Programmable Gate Arrays (FPGAs)

A configurable fabric of interconnected LEs that are packed into groups of programmable array blocks.

[Maxfield’ 04]
Programmable interconnects

![Diagram of programmable interconnects](image-url)
Configuring programmable logic

- The programming bits configures (1) the functionality of the logic elements, and (2) the exact wiring between the logic elements

[Maxfield’ 04]
Lab device: Altera’s Cyclone II device

- Two dimensional array of Logic Array Blocks (LABs), with 16 Logic Elements (LEs) in each LAB.
- EP2C35 (in DE2 board) has 60 columns and 45 rows for a total of 33216 LEs. 105 M4K blocks and 35 embedded multipliers.
- Chip has 105 memory blocks (M4K) and 35 multipliers (18x18)
- Contains 4 PLLs to generate clock frequencies
FPGA Cyclone II organization

- Local interconnects transfer signals between LEs in the same LAB and are driven by column and row interconnects and LE outputs within the same LAB.
- Neighboring LABs, PLLs, M4K RAM and multipliers from the left and right can also drive an LAB’s local interconnect directly.
- Larger communication range can be achieved through R4, R16 and R24 links.
FPGA Cyclone II organization

- Multitrack interconnect consists of row (directlink, R4, R24) and column (register chain, C4, C16)
- R4/C4 interconnects spans 4 blocks (right, left / top, down)
- R24/C16 spans 24/16 blocks and connects to R4/C4 interconnects
- R4/C4 can drive each other to extend their range
2. Chip design with CAD flow

- Design Entry
- Synthesis
- Functional Simulation
  - Design correct?
    - No
    - Fitting
      - Timing Analysis and Simulation
        - No
        - Timing requirements met?
          - No
          - Programming and Configuration
        - Yes
        - Yes
    - Yes
Synthesis

- Maps your design into a network of 4-input LE
- Packs the LEs into logic array blocks (LABs) of at most 15 LEs
Functional simulation

- Used to verify correct functionality of the design with no regard to the actual propagation delays.
  - **Quartus waveform editor** is a tool based on a GUI.
  - Enables you to create waveforms easy (in binary, decimal, hexadecimal).
  - Tutorial available on class webpage
  - Will also use MentorGraphicsModelsim for advanced simulation
Presynthesis vs. postsynthesis simulation

(a) Pre-synthesis Verilog functional simulation

The example below is zero-delay as no delays are specified.
```
always @*
begin
  C = ~B;
  Y = C & A;
end
```

(b) Post-synthesis simulation of Implementation X

Synthesis to technology ‘X’
Timing (and glitches!) depend on implementation technology

(c) Post-synthesis simulation of Implementation Y

[Example from Thornton & Reese]
Fitting (placement and routing)

Fitting objectives:
• reduce the routing resources to fit into the device
• meet timing requirement
Timing simulation (post fitting)

- Timing information from placement and routing is imported into the simulation and used for more accurate simulations.
- The path with the largest delay between flip-flops determines your computer clock frequency.
Programming the FPGA

Configuration memory determines the programmability of the logic blocks and interconnects.
3. Introduction to Verilog

- Differences between hard definition language (HDL) and software languages: Concurrency, propagation of time, signal dependency or sensitivity
- Verilog is case sensitive and syntax is similar to C
- Comments are designated by // to the end of a line or by /* to */ across several lines.
- Many online textbooks available from Brown library
  - Chapter 4 of your textbook
  - Introduction to Logic Synthesis using Verilog HDL
  - Verilog Digital System Design
  - Verilog Quickstart
  - The Verilog Hardware Description Language
• All port declarations (input, output, inout) are implicitly declared as wire.

• If an output should hold its value, it must be declared are reg

module FA(input A, input B, input Cin, output S, output Cout);
... 
endmodule
Gate-level modeling

module FA(A, B, Cin, S, Cout);
input A, B, Cin;
output S, Cout;
wire s1, c1, c2;
xor g1(s1, A, B);
xor g2(S, s1, Cin);
and g3(c1, s1, Cin);
and g4(c2, A, B);
or g5(Cout, c1, c2);
endmodule

• describes the topology of a circuit
• All instances are executed concurrently just as in hardware
• Instance name is not necessary
• The first terminal in the list of terminals is an output and the other terminals are inputs

[from Tools ➔ Netlist Viewers ➔ RTL Viewer] [from Tools ➔ Netlist Viewers ➔ after technology mapping]
Data types: vectors

• A net or register can be declared as vector.
  - wire [7:0] bus;
  - wire [31:0] busA, busB, busC;

• It is possible to access bits or parts of vectors
  - busA[7]
  - bus[2:0]
  - virt_addr[0:2]

• Specifying values for signals

<table>
<thead>
<tr>
<th>#bits</th>
<th>&lt;size&gt;’&lt;base format&gt;&lt;number&gt;</th>
<th>value</th>
</tr>
</thead>
</table>

Examples:
- 4’b1111
- 12’habc
- 16’d235
- 12’h13x
- -6’d3
- 12’b1111_0000_1010

d or D for decimal

h or H for hexadecimal

b or B for binary

o or O for octal

X or x: don’t care

Z or z: high impedance

_: used for readability
Modules with input / output vectors

module fulladd4(output [3:0] sum, 
               output c_out, 
               input [3:0] a, b, 
               input c_in);
...
...
endmodule

OR

module fulladd4(sum, c_out, a, b, input c_in);
Output [3:0] sum
Output c_out;
input [3:0] a, b;
input c_in;
...
...
endmodule
Instantiation an array of gates

wire [7:0] OUT, IN1, IN2;

// array of gates instantiations
nand n_gate [7:0] (OUT, IN1, IN2);

// which is equivalent to the following
nand n_gate0 (OUT[0], IN1[0], IN2[0]);
nand n_gate1 (OUT[1], IN1[1], IN2[1]);
nand n_gate2 (OUT[2], IN1[2], IN2[2]);
nand n_gate3 (OUT[3], IN1[3], IN2[3]);
nand n_gate4 (OUT[4], IN1[4], IN2[4]);
nand n_gate5 (OUT[5], IN1[5], IN2[5]);
nand n_gate6 (OUT[6], IN1[6], IN2[6]);
nand n_gate7 (OUT[7], IN1[7], IN2[7]);
module instantiation

module fulladd4(A, B, Cin, S, Cout);
input [3:0] A, B;
input Cin;
output [3:0] S;
output Cout;
wire C1, C2, C3;
FA f1(A[0], B[0], Cin, S[0], C1);
FA f2(A[1], B[1], C1, S[1], C2);
FA f3(A[2], B[2], C2, S[2], C3);
FA f4(A[3], B[3], C3, S[3], Cout);
endmodule
Alternative form of module instantiation

```
module FA(A, B, Cin, S, Cout);
input A, B, Cin;
output S, Cout;
...
endmodule
```

wire a1, a2, a3, a4, a5

```
FA f1(a1, a2, a3, a4, a5);
```

OR

```
FA1 f1(.Cout(a5), .S(a4), .B(a2), .A(a1), .Cin(a3));
``
Quartus II built-in modules (megafunctions)

- Use megafunctions instead of coding your own logic to save valuable design time.
- Megafunctions include the library of parameterized modules (LPM) and Altera device-specific megafunctions → use when possible.
Dataflow modeling

• Module is designed by specifying the data flow, where the designer is aware of how data flows between hardware registers and how the data is processed in the design.

• The continuous assignment is one of the main constructs used in dataflow modeling:
  - assign out = i1 & i2;
  - assign addr[15:0] = addr1[15:0] ^ addr2[15:0];
  - assign {c_out, sum[3:0]}=a[3:0]+b[3:0]+c_in;

• A continuous assignment is always active and the assignment expression is evaluated as soon as one of the right-hand-side variables change.

• Assign statements describe hardware that operates concurrently – ordering does not matter.

• Left-hand side must be a scalar or vector net. Right-hand side operands can be wires, (registers, integers, and real)
Operator types in dataflow expressions

- Operators are similar to C except that there are no ++ or –

- **Arithmetic:** *, /, +, -, % and **
- **Logical:** !, && and ||
- **Relational:** >, <, >= and <=
- **Equality:** ==, !=, === and !==
- **Bitwise:** ~, &, |, ^ and ^~
- **Reduction:** &, ~&, |, ~|, ^ and ^~
- **Shift:** << and >>
- **Concatenation:** { }
- **Replication:** {{}}
- **Conditional:** ?: 
Examples of 2x1 MUX and 4x1 MUX

module mux2to1(s, a, b, y);
output y;
input s, a, b;
assign y = (b & s) | (a & ~s);
// OR THIS WAY
assign y = s ? b : a;
endmodule

module mux4to1(out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3;
output s1, s0;
assign out = (~s1 & ~s0 & i0) |
            (~s1 & s0 & i1) |
            (s1 & ~s0 & i2) |
            (s1 & s0 & i3);
// OR THIS WAY
assign out = s1 ? (s0 ? i3:i2) : (s0 ? i1:i0);
endmodule
Difference between HLL and Verilog assign

(a) assignment statement
ordering does matter in an HLL

\[
\begin{align*}
    a &= 1; \\
    b &= 0; \\
    s &= 0; \\
    na &= 0; \\
    nb &= 0; \\
    y &= na \lor nb; \\
    nb &= b \land s; \\
    na &= a \land \neg s; \\
    y &= na \lor nb;
\end{align*}
\]

Final y value is 0.

(b) assign statement
ordering does not matter in Verilog

\[
\begin{align*}
    \text{wire } na, nb; \\
    \text{assign } y = na \lor nb; \\
    \text{assign } nb = b \land s; \\
    \text{assign } na = a \land \neg s;
\end{align*}
\]

[Example from Thornton & Reese]
Difference between HLL and Verilog assign

(a) assignment statements in an HLL can target the same variable

\[
\begin{align*}
    a &= 1; \\
    b &= 0; \\
    s &= 0; \\
    na &= 0; \\
    nb &= 0; \\
    na &= b \& s; \\
    na &= a \& \neg s;
\end{align*}
\]

The \textit{na} variable is assigned twice; the final value of \textit{na} is the last assignment.

(b) illegal use of `assign` statements

\[
\begin{align*}
    \text{wire na;} \\
    \text{assign na} &= b \& s; \\
    \text{assign na} &= a \& \neg s;
\end{align*}
\]

Gate outputs are shorted together!

can only work with tri-state drivers

[Example from Thornton & Reese]
Example of a dataflow 4-bit adder

(a) Four-bit adder with no carry-in or carry-out

```verilog
// 4-bit adder
// no carry-in, carry-out
module add4bit (a, b, s);

input [3:0] a, b;
output [3:0] s;

assign s = a + b;
endmodule
```

(b) Four-bit adder with carry-in, carry-out

```verilog
// 4-bit adder with carry-in, carry-out
module add4bit (ci, a, b, s, co);

input ci;
input [3:0] a, b;
output [3:0] s;
output co;

wire [4:0] y;

// do 5-bit sum so that we have access to carry out
assign y = {1'b0, a} + {1'b0, b} + {4'b0, ci};
assign s = y[3:0]; // four-bit output
assign co = y[4]; // carry-out
endmodule
```

[Example from Thornton & Reese]
Behavioral modeling

- Design is expressed in algorithmic level, which frees designers from thinking in terms of logic gates or data flow.

- All algorithmic or procedural statements in Verilog can appear only inside two statements: `always` and `initial`.

- Each `always` and `initial` statement represents a separate activity flow in Verilog. Remember that activity flows in Verilog run in parallel.

- You can have multiple `initial` and `always` statements but you cannot nest them.

```verilog
reg a, b, c;
initial a=1'b0;
always @* begin
  b = a ^ 1'b1;
  c = a + b;
end
```
always blocks

- The always block executes its statements when the events in its sensitivity list occur.

- Constructs like if, if-else, case, and looping are only allowed inside always blocks.

- always blocks can implement combinational or sequential logic.

- Multiple statements must be grouped using begin and end.

- LHS variables assigned within an always block must be declared as a reg or integer type \( \rightarrow \) does not necessarily imply a register or sequential logic.

- Multiple always blocks can appear in a module.

```verilog
module mux2to1(s,a,b,y);
input s,a,b;
output y;

reg y;

//use boolean ops
always @(a or b or s)
begin
    y = (b & s) | (a & ~s);
end

endmodule
```
Data types

• A `reg` is a Verilog variable type and does not necessarily imply a physical register. It is unsigned by default.
  - `reg clock;`
  - `reg [0:40] virt_address;`

• Register arrays or memories. Used to model register files, RAMs and ROMs. Modeled in Verilog as a one-dimensional array of registers. Examples.
  - `reg mem1bit[0:1023];`
  - `reg [7:0] membyte[0:1023];`
  - `accessing: membyte[511];`

• Parameters. Define constants and cannot be used as variables.
  - `parameter port_id=5;`

• integers (signed)
  - `integer i`
  - `integer count[0:7];`
  - `integer matrix[4:0][0:255];`
Sensitivity list of events

• An event is the change in the value on a register or a net. Events can be utilized to trigger the execution of a statement of a block of statements.

• The @ symbol is used to specify an event control.

• For combinational logic, any net that appears on the right side of an “=” operator in the always block should be included in the event list or just @*

• For sequential, Statements can be executed on changes in signal value or at a positive (posedge) or negative (negedge) transition of the signal.

```module mux2to1(s,a,b,y);
    input s,a,b;
    output y;
    reg y, na, nb;
    //use intermediates
    //and implicit event
    //list
    always @*
    begin
        nb = b & s;
        na = a & ~s;
        y = na | nb;
    end
endmodule```
Conditional statements

- Very similar to C
- Can always appear inside always and initial blocks

```verilog
reg [1:0] alu_control;

always @*
begin
    if(alu_control == 0)
        y = x + z;
    else if (alu_control == 1)
        y = x - z;
    else if (alu_control == 2)
        y = x * z;
    else
        y = x;
end
```

```verilog
always @*
begin
    case (alu_control)
        2’d0 : y = x + z;
        2’d1 : y = x - z;
        2’d2 : y = x * z;
        default: y=x;
    endcase
end
```
Importance of order

- With **blocking assignments**, the same variable can be assigned multiple times in an always block; the last assignment takes precedence.

[Example from Thornton & Reese]
Loops in Verilog

for (count = 0; count < 128; count = count + 1)
begin
  .
  .
end

count = 0;
while (count < 128)
begin
  .
  .
  count = count +1;
end

repeat(128)
begin
  .
  .
end

• Compiler will always unroll all loops fully before synthesis!
module trial(A, out);
input [15:0] A;
output reg [15:0] out;
integer count;

always @*
begin
  for(count = 0; count <= 16'd15; count = count+1)
  begin
    if (count % 2 == 0) out[count] = A[count];
    else out[count] = ~A[count];
  end
end
endmodule
always statements

• The “=” operator when used in an always block is called a blocking assignment

• If there is some logic path through the always block that does not assign a value to a variable then a latch is inferred → be careful.

• The logic synthesized assumed the blocking assignments are evaluated in order. This means that the order in which assignments are written in an always blocks affects the logic that is synthesized.
Edge-triggered Flip flops (D-FF)

- The @ symbol is used to specify an event control.
- Statements can be executed on changes in signal value or at a positive (posedge) or negative (negedge) transition of the signal.
- Use non-blocking assignments (<=)
- For nonblocking assignments within an always block, all RHS expressions are evaluated, and are only assigned to the LHS at the edge of the clock.

[Thornton & Reese]
module pipe(clk, w1, w2, w3, w4, i1, i2, i3, i4, z);
input clk;
output [17:0] z;
input [7:0] w1, w2, w3, w4, i1, i2, i3, i4;
reg [63:0] p1;
reg [33:0] p2;
always @(posedge clk)
begin
  p1[15:0] <= w1*i1;
p1[31:16] <= w2*i2;
p1[47:32] <= w3*i3;
p1[63:48] <= w4*i4;
  p2[16:0] <= p1[31:16]+p1[15:0];
end
assign z=p2[16:0]+p2[33:17];
endmodule
Blocking vs. non-blocking statements

- Use `always @(*)` and blocking assignments (`=`) to model combinational logic (if else, loops, etc).
- Use `always @(posedge clk)` and non-blocking assignments (`<=`) to model sequential elements.
- Advice: do not mix blocking and non-blocking assignments in the same always block.
module trial(clk, rst, gray_output);
    input clk, rst;
    output reg [2:0] gray_output;
    reg [2:0] counter;

    always @(posedge clk)
        if(rst) counter <= 3'b000;
        else counter <= counter + 1;

    always @*
        case(counter)
            3'b000: gray_output = 3'b000;
            3'b001: gray_output = 3'b001;
            3'b010: gray_output = 3'b011;
            3'b011: gray_output = 3'b010;
            3'b100: gray_output = 3'b110;
            3'b101: gray_output = 3'b111;
            3'b110: gray_output = 3'b101;
            3'b111: gray_output = 3'b100;
            default: gray_output = 3'b000;
        endcase
    endmodule

[from Tools -> Netlist Viewers -> RTL Viewer]
Initialization of latches and flipflops

- An initial is used to initialize variables (i.e., FFs and latches).
- Multiple initializations should be grouped using begin and end. If there is one statement then grouping is not necessary.
- Multiple initial blocks can be used.

```verilog
reg x, y, m;
initial m=1'b0;
initial begin
  x = 1'b0;
  y = 1'b1;
end
```

In procedural statements (initial, always) LHS must be of type registers (and its derivatives)
Avoiding pitfalls

- **Combinational logic:**
  
  - Use continuous assign statements to model simple combinational logic.
  
  - **Use** `always @(*)` and blocking assignments (`=`) to model more complicated combinational logic (if else, loops, etc)

  - For logic pathways arising from if-else branching or other logic constructs, then assign every output a default value at the beginning of the block → avoids wrong latches.

  
  - Avoid combinational loops

  ![Diagram of combinational logic and sequential elements](image)
Avoiding pitfalls

• **Sequential logic:**
  - Use non-blocking assignments ($<=$) in always blocks that are meant to represent sequential logic
  - Use `posedge` sensitivity to ensure DFF

• Do not make assignments to the same signal in more than one `always` statement or continuous `assign` statement.

• Avoid mixing blocking and non-blocking assignments in the same always block.

[Thornton/ Reese & Harris]
Advanced simulation and debugging

- Verilog can be used to create **testbenches** → more flexible than waveform editing.
- Testbenches are just scripts for simulation but are not synthesizable.
- The testbench instantiates your design module and connect its input for stimulus and monitor its outputs.
- New verilog commands to enable precise timing simulation and monitoring of outputs.
- We will use industry-standard MentorGraphics ModelSim for simulation.
- Timing information (after fitting) can be given to ModelSim as well.
Additional Verilog constructs for simulation

- Specify delta time using 
- Use `timescale` to specify unit of time and precision
- Use `display` to print a signal at a particular time (same format as `printf`)
- Use `monitor` to display signals when they change (same format as `printf`)
- `$random` generates a random number.

- `$time` is the current time.
- `$finish` ends simulation (e.g., if a particular condition is met
- Use `$fopen()`, `$fclose()`, `$fwrite()`, `$fscanf()` to access files (e.g., reading data for simulation or writing results). Syntax is similar to equivalent C functions.
- Lab tutorial available on class website.

```verilog
`timescale 10ns/1ns
module tb;
    reg [7:0] a;
    reg [7:0] b;
    wire [7:0] c;
    add_module add1(a, b, c);
    initial
        begin
            #5 b = 20;
            #10 b = 50;
            $monitor("%d", c);
        end
    always
        #10 a=$random;
    always
        #10 if(c %2 == 0)
            $display("even\n");
endmodule
```
SignalTapII for in-system debugging

- Enables debugging on FPGA
- Insert probes and additional HW to capture internal signals and relay them over JTAG USB in the form of waveform displays
- Very valuable for identifying bugs after implementation → Tutorial available on website
- Make sure to disable/remove signalTap after you are done with debugging
- Lab tutorial available on class website.