EN164: Design of Computing Systems
Topic 02: Lab Foundations

Professor Sherief Reda
http://scale.engin.brown.edu
Electrical Sciences and Computer Engineering
School of Engineering
Brown University
Spring 2012
A logic element 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

[Figure form Cong FPGA’ 01]
A configurable fabric of interconnected LEs that are packed into groups of programmable array blocks.
Programmable interconnects

Switch box
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.
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
Design automation with CAD flow

1. Design Entry
2. Synthesis
3. Functional Simulation
   - Design correct? (Yes/No)
4. Fitting
5. Timing Analysis and Simulation
   - Timing requirements met? (Yes/No)
6. Programming and Configuration
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
Fitting (placement and routing)

Fitting objectives:
• reduce the routing resources to fit into the device
• meet timing requirement
- The path with the largest delay between flip-flops determines your computer clock frequency
Programming the FPGA

Configuration data in

Configuration data out

= I/O pin/pad

= SRAM cell

Configuration memory determines the programmability of the logic blocks and interconnects
Summary of key concepts

- Logic elements (LEs) and Logic Array Blocks (LABs)
- Field Programmable Gate Arrays
- CAD Tools
- Synthesis, fitting, simulation and programming
- *Ultimate objective of design tools is to make hardware design as easy as software design*
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
  • Introduction to Logic Synthesis using Verilog HDL
  • Verilog Digital System Design
  • Verilog Quickstart
  • The Verilog Hardware Description Language
Verilog modules

The functionality of each module can be defined with three modeling levels:

1. *Structural or gate level*
2. *Dataflow level*
3. *Behavioral or algorithmic level*

Verilog allows different levels of abstraction to be mixed in the same module.
1. Structural modeling -- Data types: bits

- Nets represent connections between hardware elements. They are continuously driven by output of connected devices. They are declared using the keyword `wire`.
  - `wire s1;`
  - `wire c1, c2;`
  - `wire d=0;`
Modules and ports

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

• All port declarations (input, output, inout) are implicitly declared as wire.
• If an output should hold its value, it must be declared as reg

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

wire Z, Z1, OUT, OUT1, OUT2, IN1, IN2;

and a1(OUT1, IN1, IN2);
nand na1(OUT2, IN1, IN2);
xor x1(OUT, OUT1, OUT2);
not (Z, OUT);
buf final (Z1, Z);

• Essentially 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
• Not the most interesting modeling technique for our class
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

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

- A net or register can be declared as *vectors*. Example of declarations:
  - wire a;
  - 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 wires and variables

- Number specification.

Example:
- $4'b1111$
- $12'habc$
- $16'd235$
- $12'h13x$
- $-6'd3$
- $12'b1111_0000_1010$

Specifications:
- `<size>` specifies the number of bits in the number
- `<base format>` specifies the base format:
  - `d` or `D` for decimal
  - `h` or `H` for hexadecimal
  - `b` or `B` for binary
  - `o` or `O` for octal
  - `X` or `x` indicates don't care
  - `Z` or `z` indicates high impedance
  - `_` is used for readability

Number depends on the base.
Modules with input / output vectors

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

... ...
endmodule

OR

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

[from Tools ➔ Netlist Viewers ➔ RTL Viewer]
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));
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]);
2. 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**: { }
2. 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

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

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

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

The `na` variable is assigned twice; the final value of `na` is the last assignment.

(b) illegal use of `assign` statements

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

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

```
// 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

```
// 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]
3. 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 can’t nest them.

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

• A `reg` is a Verilog variable type and does not necessarily imply a physical register. Think of it as a variable or place holder. 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;`
Data types

• integers (signed) and reals. They are type of reg.
  ▪ real delta;
  ▪ integer i;
  ▪ initial
  ▪ begin
    ▪ delta = 4e10;
    ▪ i = 4;
  ▪ end

• Arrays of integers and real.
  ▪ integer count[0:7];
  ▪ integer matrix[4:0][0:255];

• Strings can be stored in reg. The width of the register variables must be large enough to hold the string.
  ▪ reg [8*19:1] string_value;
  ▪ initial
    ▪ string_value = “Hello Verilog World”;
initial statements

- An initial block start at time 0, executes exactly once and then never again.
- If there are multiple initial blocks, each blocks starts to execute concurrently at time 0 and each blocks finish execution independently of the others.
- Multiple behavioral statements must be grouped using begin and end. If there is one statement then grouping is not necessary.

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

```vhdl
reg x, y, m;
initial m=1'b0;
begin
  x = 1'b0;
  y = 1'b1;
end
```
always statements

• The always statement starts at time 0 and executes the statements in the always block when the events in its sensitivity list occur.

• Powerful constructs like if, if-else, case, and looping are only allowed inside always blocks.

• always statements can be used to implement both combinational or sequential logic.

• Multiple behavioral statements must be grouped using begin and end.

• Multiple always statement can appear in a module.

```vhdl
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
```
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.

- [for sequential – ignore for now] Statements can be executed on changes in signal value or at a positive (posedge) or negative (negedge) transition of the signal.
always statements

- Any net that is assigned within an always block must be declared as a \texttt{reg} type; this does not imply that this net is driven by a register or sequential logic.
- 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 the output net then a latch is inferred.
- The logic synthesized assumed the blocking assignments are evaluated sequentially. This means that the order in which assignments are written in an always blocks affects the logic that is synthesized.
always statements

- Because of the sequential nature of an always block, the same net can be assigned multiple times in an always block; the last assignment takes precedence.

[Example from Thornton & Reese]
Conditional statements

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

```vhdl
if(x)
begin
  y = 1'b1;
  z = 1'b0;
end
expression
if (count < 10)
  count = count + 1;
else
  count = 0;
.
```

```vhdl
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;
.
```

```vhdl
reg [1:0] alu_control;
.. case (alu_control)
  2'd0 : y = x + z;
  2'd1 : y = x - z;
  2'd2 : y = x * z;
  default: y=x;
endcase
..```
module mux4x1(out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;

always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
  case({s1, s0})
    2'd0: out = i0;
    2'd1: out = i1;
    2'd2: out = i2;
    2'd3: out = i3;
  endcase
endmodule
Level sensitive latch (D-Latch)

• The Verilog implementation of a D-latch is an always block that makes a *nonblocking assignment* ("\(\leq\)") of \(d\) to \(q\) when the \(g\) input is nonzero.

• When \(g\) input is zero, then the *always* block does not make any assignment to \(q\), causing the synthesis tool to infer a latch on the \(q\) output as the \(q\) output must retain its last known \(d\) value when \(g\) was nonzero.

• Nonblocking assignments ("\(\leq\)") as opposed to blocking assignments ("\(=\)”) should be used in *always* blocks that are used to synthesize sequential logic.

[from Thornton & Reese]
Edge-triggered storage element (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.
- In general, edge-triggered storage elements are preferred to level-sensitive storage elements because of simpler timing requirements.
- The 1-bit edge-triggered storage elements provided by FPGA vendors are DFFs because of their simplicity and speed.

[Thornton & Reese]
module quest3(CLOCK_50, LEDR);
input CLOCK_50;
output reg [17:0] LEDR;
integer count;
always @(posedge CLOCK_50)
begin
    if(count == 50000000)
        begin
            LEDR[0] <= !LEDR[0];
            count <= 0;
        end
    else
        count <= count + 1;
end
endmodule
DFF chains

- Each nonblocking assignment synthesizes to a single DFF whose input happens to be the output of another nonblocking assignment.
- The ordering of these nonblocking assignments within an always block does not matter.

[Thornton & Reese]
Blocking vs. non-blocking statements

- Zero-delay blocking assignments are so named because the assignment of the right-hand side (RHS) to the left-hand side (LHS) is completed without any intervening Verilog code allowed to execute, i.e., the assignment blocks the execution of the other Verilog code.

- For nonblocking assignments within an always block, all RHS expressions are evaluated, and are only assigned to the LHS targets after the always block completes.

[Thornton & Reese]

(a) Blocking assignments - RHS values applied to LHS immediately

(b) Non-blocking assignments - all RHS values applied to LHS after always b
Avoid combinational loops

(a) A combinational loop

always @*
begin
  y = y + a;
end

Output oscillates; period is dependent upon adder delay

(b) Sequential element in feedback path

always @(posedge clk)
begin
  y <= y + a;
end

Output can only change on the active clock edge

[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

• It is sometimes easier/clearer to use counts and if then statements to create loops
module loop(CLOCK_50, A, out);
input CLOCK_50;
input [15:0] A;
output reg [15:0] out;

reg [15:0] r;
reg [4:0] count;

initial out = 16'd0;

always @(posedge CLOCK_50)
begin
    r <= A;
    for(count = 0; count <= 16'd15; count = count+1)
    begin
        if (count % 2 == 0) out[count] <= r[count];
        else out[count] <= ~r[count];
    end
end
endmodule
Loop synthesis with counts & if-then

module loop(CLOCK_50, A, out);
input CLOCK_50;
input [3:0] A;
output reg [6:0] out
reg [3:0] r, count, B;
initial
begin
  out = 4'b0;
  count = 4'd0;
end
always @(posedge CLOCK_50) r <= A;
always @(posedge CLOCK_50)
begin
  if (count <= 3)
  begin
    out <= out + r;
    count <= count + 1;
  end
end
endmodule
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

Timing (and glitches!) depend on implementation technology

Synthesis to technology ‘X’

(c) Post-synthesis simulation of Implementation Y

4x1 Memory
(4 locations, each location has 1 bit)

[Example from Thornton & Reese]
Guidelines to avoid frustration (1)

- **Combinational logic:**
  - Use continuous assign statements to model simple combinational logic
  - Use `always @(*)` and blocking assignments (=) to model more complicated combinational logic where the `always` statement is helpful
  - If an always block for combinational logic contains logic pathways due to if-else branching or other logic constructs, then assign every output a default value at the beginning of the block. This ensures that all outputs are assigned a value regardless of the path taken through the logic, avoiding inferred latches on outputs.

[Thornton/ Reese & Harris]
Guidelines to avoid frustration (2)

• **Sequential logic:**
  - Use non-blocking assignments (<=) in always blocks that are meant to represent sequential logic
  - Use `posedge` sensitivity to ensure DFF
  - Avoid using latches and use edge-triggered flip-flops.

• 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]