EECS 150 - Components and Design Techniques for Digital Systems

Lec 10 – Logic Synthesis
9-30-04

David Culler
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~culler
http://www-inst.eecs.berkeley.edu/~cs150

Recall: Design Flow

Design Entry

High-level Analysis

Technology Mapping

Low-level Analysis

Netlist: used between and internally for all steps.

Design Methodology in Detail

Design Specification

Design Partition

Design Entry Behavioral Modeling

Simulator/Functional Verification

Design Configuration

Pre-Synthesis Sign-Off

Synthesize and Map

Gate-level Net List

Post-synthesis Design Validation

Post-synthesis Timing Verification

Test Generation and Fault Simulation

Cell Placement/Scan

Interconnect/Routing

Verify Physical and Electrical Rules

Synthesize and Map

Gate-level Net List

Design Sign-Off

Design Specification

• Written statement of functionality, timing, area, power, testability, fault coverage, etc.

• Functional specification methods:
  – State Transition Graphs
  – Timing Charts
  – Algorithm State Machines (like flowcharts)
  – HDLs (Verilog and VHDL)

Design Partition

• Partition to form an Architecture
  – Interacting functional units
  – Control vs. datapath separation
  – Interconnection structures within datapath
  – Structural design descriptions
  – Components described by their behaviors
  – Register-transfer descriptions
  – Top-down design method exploiting hierarchy and reuse of design effort

Review

• Basic registers
  – Array of FlipFlips, MUXes & CL, Common control

• Simple, important FSMs
  – simple internal feedback
  – Ring counters, Pattern detectors
  – Binary Counters

• Universal Shift Register
• Using Counters to build controllers
  – Simplify control by controlling simpler FSM
### Design Methodology in Detail

#### Design Entry
- Primary modern method: hardware description language
  - Higher productivity than schematic entry
  - Inherently easy to document
  - Easier to debug and correct
  - Easy to change/extend and hence experiment with alternative architectures
- Synthesis tools map description into generic technology description
  - E.g., logic equations or gates that will subsequently be mapped into detailed target technology
  - Allows this stage to be technology independent (e.g., FPGA LUTs or ASIC standard cell libraries)
- Behavioral descriptions are how it is done in industry today

#### Simulation and Functional Verification
- Simulation vs. Formal Methods
- Test Plan Development
  - What functions are to be tested and how
  - Testbench Development
  - Testing of independent modules
  - Testing of composed modules
  - Test Execution and Model Verification
  - Errors in design
  - Errors in description syntax
  - Ensure that the design can be synthesized
- The model must be VERIFIED before the design methodology can proceed

#### Design Integration and Verification
- Integrate and test the individual components that have been independently verified
- Appropriate testbench development and integration
- Extremely important step and one that is often the source of the biggest problems
  - Individual modules thoroughly tested
  - Integration not as carefully tested
  - Bugs lurking in the interface behavior among modules!

#### Presynthesis Sign-off
- Demonstrate full functionality of the design
- Make sure that the behavior specification meets the design specification
  - Does the demonstrated input/output behavior of the HDL description represent that which is expected from the original design specification
- Sign-off only when all functional errors have been eliminated

#### Gate-Level Synthesis and Technology Mapping
- Once all syntax and functional errors have been eliminated, synthesize the design from the behavior description
  - Optimized Boolean description
  - Map onto target technology
  - Minimize logic
  - Reduce area
  - Reduce power
  - Balance speed vs. other resources consumed
- Produces netlist of standard cells or database to configure target FPGA

#### Design Methodology in Detail
- Design Specification
- Design Partition
- Design Entry behavioral Modeling
- Simulation/Functional Verification
- Design Integration and Verification
- Pre-Synthesis Sign-Off
- Synthesize and Map
- Gate-level Net List
Logic Synthesis

- Verilog and VHDL started out as simulation languages, but quickly people wrote programs to automatically convert Verilog code into low-level circuit descriptions (netlists).

```
Verilog      Synthesis Tool  circuit netlist
HDL
```

- Synthesis converts Verilog (or other HDL) descriptions to implementation technology specific primitives:
  - For FPGAs: LUTs, flip-flops, and RAM blocks
  - For ASICs: standard cell gate and flip-flop libraries, and memory blocks.

Logic Synthesis – where EE and CS meet

Announcements/Reminders

- First mid term 10/7
- No notes
- Review session Monday 10/4 6-8pm in the lab
- Review materials will be on line
- Homework 4 due Friday
- Homework 5 out (short)

Why Logic Synthesis?

1. Automatically manages many details of the design process:
   - Fewer bugs
   - Improved productivity

2. Abstracts the design data (HDL description) from any particular implementation technology:
   - Designs can be re-synthesized targeting different chip technologies.
   - Ex: first implement in FPGA then later in ASIC.

3. In some cases, leads to a more optimal design than could be achieved by manual means (ex: logic optimization)

Why Not Logic Synthesis?

1. May lead to non-optimal designs in some cases.

Die Photos: Vertex vs. Pentium IV

- FPGA Vertex chip looks remarkably structured
  - Very dense, very regular structure
  - Lots of volume, low NRE, high silicon overhead

- Full Custom Pentium chip somewhat more random in structure
  - Large on-chip memories (caches) are visible

- Logic Synthesis essential for both

How does it work?

- A variety of general and ad-hoc (special case) methods:
  - Instantiation: maintains a library of primitive modules (AND, OR, etc.) and user defined modules.
  - "macro expansion" / substitution: a large set of language operators (+, -, Boolean operators, etc.) and constructs (if-else, case) expand into special circuits.
  - Inference: special patterns are detected in the language description and treated specially (ex: inferring memory blocks from variable declaration and read/write statements, FSM detection and generation from "always @(posedge clk)" blocks).
  - Logic optimization: Boolean operations are grouped and optimized with logic minimization techniques.
  - Structural reorganization: advanced techniques including sharing of operators, and retiming of circuits (moving FFs), and others.

   For ASICs: standard cell gate and flip-flop libraries, and memory blocks.

   Lots of volume, low NRE, high silicon overhead

   Large on-chip memories (caches) are visible

   Logic Synthesis essential for both

   Why Not Logic Synthesis?

   May lead to non-optimal designs in some cases.
Synthesis vs Compilation

Levels of Representation

- Compiler
  - Recognizes all possible constructs in a formally defined program language
  - Translates them to a machine language representation of execution process
- Synthesis
  - Recognizes a target dependent subset of a hardware description language
  - Maps to collection of concrete hardware resources
  - Iterative tool in the design flow

Mapping

<table>
<thead>
<tr>
<th>Verilog Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements (wire, latch, flipflop, ALU, MUX, ...)</td>
</tr>
<tr>
<td>Structures (module, gate,...)</td>
</tr>
<tr>
<td>Statements (procedural assignment, if, case,...)</td>
</tr>
<tr>
<td>Constants (net, register, parameter)</td>
</tr>
</tbody>
</table>

Simple Example

```verilog
module foo (a, b, s0, s1, f);
input [3:0] a;
input [3:0] b;
input s0, s1;
output [3:0] f;
wire f;
always @ (a or b or s0 or s1)
  if ('s0 & s1 || s0) f = a; else f = b;
endmodule
```

Unsupported Language Constructs

- Should expand if-else into 4-bit wide multiplexer and optimize the control logic.

Module Template

```verilog
/* Port declarations. followed by wire, reg, integer, task and function declarations */
// Instantiation and module instantiations
module (module_name, (port list));
/* Port declarations. followed by wire, reg, integer, task and function declarations */
// Instantiate gate primitive
assign result_signal_name = expression;
// always block always (if/and expression) begin
// Procedural assignments // if statements
begin // case, case, and case statements // while, repeat and for loops // user task and user function calls
end // Module instantiation endmodule (module_name, (port list));
// Instantiation of built-in gate primitive (gate_type_keyword (port list));
endmodule
```

Supported Verilog Constructs

- Net types:
  - wire, tri, supply1, supply2;
  - register types: reg, integer, time (64 bit reg); arrays of reg.
- Continuous assignments.
- Gate primitive and module instantiations.
- always blocks, user tasks, user functions.
- Inputs, outputs, and inout to a module.
- All operators
  - +, -, *, /, %, <<, >>, |, &&.
- Note: / and % are supported for compile-time constants and constant powers of 2.

- Procedural statements:
  - if-else, case, cases, cases, for, repeat, while, forever, begin, end, fork, join.
- Procedural assignments:
  - blocking assignments
  - non-blocking assignments
  - Note: && cannot be mixed with a || for the same register.
- Compiler directives: define, ifdef, else, endif, include, undef
- Miscellaneous:
  - Integer ranges and parameter ranges.
  - Local declarations to begin module block.
  - Variable indexing of bit vectors on the left and right sides of assignments.

Unsupported Language Constructs

- Simply ignored
  - delay, delay control, and drive strength.
  - scaled, vector.
  - initial block.
- Compiler directives (except for define, ifdef, else, endif, include, and undef, which are supported).
- Calls to system tasks and system functions (they are only for simulation).
Combinational Logic

CL can be generated using:

1. primitive gate instantiation:
   AND, OR, etc.
2. continuous assignment (assign keyword), example:
   Module adder_8 (cout, sum, a, b, cin);
   output cout;
   output [7:0] sum;
   input cin;
   input [1:0] a, b;
   assign [cout, sum] = a + b + cin;
   endmodule
3. Always block:
   always @(event_expression)
   begin
   // procedural assignment statements, if statements,
   // case statements, while, repeat, and for loops.
   end

Procedural Assignments

- Verilog has two types of assignments within always blocks:
  - Blocking procedural assignment "="
    - The RHS is executed and the assignment is completed before the next statement is executed. Example:
      Assume A holds the value 1 … A <= 2; B <= A; A is left with 2, B with 1.
  - Non-blocking procedural assignment "="
    - The RHS is executed and assignment takes place at the end of the current time step (not clock cycle). Example:
      Assume A holds the value 1 … A <= 2; B <= A; A is left with 2, B with 1.
  - The notion of the “current time step” is tricky in synthesis, so to guarantee that your simulation matches the behavior of the synthesized circuit, follow these rules:
    1. Use blocking assignments to model combinational logic within an always block.
    2. Use non-blocking assignments to implement sequential logic.
    3. Do not mix blocking and non-blocking assignments in the same always block.
    4. Do not make assignments to the same variable from more than one always block.

Register Data Type

- Reg declaration specifies size in bits
- Integer type – max size is 32 bits, synthesis may determine size by analysis
  - Wire [1:5] Brq, Rbu
  - Integer Arb
    - Arb = Brq + Rbu  “Arb is 6 bits”
- Variable of reg typ maps into wire, latch or flip-flop depending on context

Net Data Type

- Variable of NET type maps into a wire
- wire ▶ wire
- supply0 ▶ wire connected to logic-0
- supply1 ▶ wire connected to logic-1
- tri ▶ like a wire
- wor
- wand

Combinational logic always blocks

- Make sure all signals assigned in a combinational always block are explicitly assigned values every time that the always block executes. Otherwise latches will be generated to hold the last value for the signals not assigned values.

  - Example:
    - Set case value 2’d2 omitted.
    - Out is not updated when select line has 2’d2.
    - Latch is added by tool to hold the last value of out under this condition.

Operator

- Logical operators map into primitive logic gates
- Arithmetic operators map into adders, subtractors, ...
  - Unsigned 2’s complement
  - Watch out for *, %, and /
  - Switching operators generate comparators
  - Shifts by constant amount are just wire connections
  - No logic involved
  - Variable shift amounts a whole different story --- shifter
  - Conditional expression generates logic or MUX

Procedural Assignments

- Verilog has two types of assignments within always blocks:
  - Blocking procedural assignment "="
    - The RHS is executed and the assignment is completed before the next statement is executed. Example:
      Assume A holds the value 1 … A <= 2; B <= A; A is left with 2, B with 2.
  - Non-blocking procedural assignment "="
    - The RHS is executed and assignment takes place at the end of the current time step (not clock cycle). Example:
      Assume A holds the value 1 … A <= 2; B <= A; A is left with 2, B with 1.
  - The notion of the “current time step” is tricky in synthesis, so to guarantee that your simulation matches the behavior of the synthesized circuit, follow these rules:
    1. Use blocking assignments to model combinational logic within an always block.
    2. Use non-blocking assignments to implement sequential logic.
    3. Do not mix blocking and non-blocking assignments in the same always block.
    4. Do not make assignments to the same variable from more than one always block.
Fixes to the avoid creating latch

```verilog
module mux4to1 (out, a, b, c, d, sel);
output out;
input a, b, c, d;
input [1:0] sel;
reg out;
always @(sel or a or b or c or d)
begin
  case (sel)
    2'd0: out = a;
    2'd1: out = b;
    2'd2: out = c;
    2'd3: out = d;
  endcase
endmodule
```

Example (cont)

```verilog
module funnymux4to1 (out, a, b, c, d, sel);
output out;
input a, b, c, d;
input [1:0] sel;
reg out;
always @(sel or a or b or c or d)
begin
  case (sel)
    2'd0: out = a;
    2'd1: out = b;
    2'd2: out = c;
    default: out = 'bx;
  endcase
endmodule
```

Latch rule

- If a variable is not assigned in all possible executions of an always statement then a latch is inferred
  - E.g., when not assigned in all branches of an if or case
  - Even a variable declared locally within an always is inferred as a latch if incompletely assigned in a conditional statement

Assign before use ordering

```verilog
module on latch (clock, curState, nxtState);
input clock;
input curState;
output nxtState;
reg nxtState;
always @(clock or curState)
begin
  integer temp;
  if (clock) begin
    temp = CurState;
    nxtState = temp;
  end
endmodule
```

Combination Logic (cont.)

- Be careful with nested IF-ELSE. They can lead to “priority logic”
  - Example: 4-to-2 encoder

```verilog
always @ (clock or curState)
begin
  integer temp;
  if (clock) begin
    temp = CurState;
    nxtState = temp;
  end
endmodule
```
Sequential Logic

- Example: D flip-flop with synchronous set/reset:

```verilog
module dff(q, d, clk, rst);  // Definition of flip-flop
  input d, clk, rst;  // Inputs: data, clock, reset
  output q;  // Output: flip-flop output
  reg q;  // Register variable
  assign q = (posedge clk) ? d : q;  // Flip-flop logic
endmodule
```

We prefer synchronous set/reset, but how would you specify asynchronous preset/clear?

Finite State Machines

```verilog
module fsm(clk, rst, enable, data_in, data_out);
  input clk, rst, enable;  // Inputs: clock, reset, enable
  input [2:0] data_in;  // Data input
  output data_out;  // Data output
  reg [1:0] state, next_state;  // State registers
  reg data_out;  // Data output
  always @ (posedge clk)  // Event-driven style
    begin
      if (!rst) state <= idle;  // Initial state
      else state <= next_state;  // State transition
      if (enable) state <= next_state;  // Enable
    end
  endmodule
```

```
module dff(q, d, clk, rst);
  input d, clk, rst;  // Inputs: data, clock, reset
  output q;  // Output: flip-flop output
  reg q;  // Register variable
  assign q = (posedge clk) ? d : q;  // Flip-flop logic
endmodule
```

Procedural Assignment

- Target of proc. Assignment is synthesized into a wire, a flip-flop or a latch, depending on the context under which the assignment appears.
- A target cannot be assigned using a blocking assignment and a non-blocking assignment.

```verilog
module dff(q, d, clk, rst);
  input d, clk, rst;  // Inputs: data, clock, reset
  output q;  // Output: flip-flop output
  reg q;  // Register variable
  assign q = (posedge clk) ? d : q;  // Flip-flop logic
endmodule
```

```verilog
always @ (posedge clk)
  begin
    if (!rst)
      state <= idle;
    else state <= next_state;
  end
endmodule
```

Values x and z

- Assigning the value x to a variable tells synthesis to treat as don't-care
- Assigning z generates tristate gate
  - Z can be assigned to any variable in an assignment, but for synthesis its must occur under the control of a conditional statement

```
module threestate(rdy, inA, inB, sal);  // Three-state logic
  input rdy, inA, inB;  // Inputs
  output sal;  // Output
  reg sal;  // Register
  always @ (rdy or inA or inB)
    if (rdy) sal = 1'b0  // Tristate
    else sal = inA & inB  // Normal
endmodule
```

Postsynthesis Design Validation

- Does gate-level synthesized logic implement the same input-output function as the HDL behavioral description?

```
module threestate(rdy, inA, inB, sal);  // Three-state logic
  input rdy, inA, inB;  // Inputs
  output sal;  // Output
  reg sal;  // Register
  always @ (rdy or inA or inB)
    if (rdy) sal = 1'b0  // Tristate
    else sal = inA & inB  // Normal
endmodule
```
More Help

- Online documentation for Synplify Synthesis Tool:
  - Under “refs/links” and linked to today’s lecture on calendar
  - Online examples from Synplicity.
- Bhasker (same author as Verilog reference book) on reserve in the Engineering library.
- Trial and error with the synthesis tool.
  - Synplify will display the output of synthesis in schematic form for your inspection. Try different input and see what it produces.

Bottom line

- Have the hardware design clear in your mind when you write the verilog.
- Write the verilog to describe that HW
  - It is a Hardware Description Language not a Hardware Imagination Language.
- If you are very clear, the synthesis tools are likely to figure it out.