CSE140L: Components and Design Techniques for Digital Systems Lab

Tajana Simunic Rosing

Source: Vahid, Katz, Culler
Updates & Outline

• Grade distribution:
  – 70% Labs
    • 25% Lab 4
    • 20% Lab 3
    • 15% Lab 2
    • 10% Lab 1
  – 30% Final exam

• Office hrs: MTh 2-3pm finals week, by appointment

• Final exam – Monday, 3/16, 3-6pm
  – Everything covered in labs, lectures, book & handouts
  – Format:
    • Multiple choice and/or T/F questions on the assigned reading and labs
    • Problems to solve based on the lectures, labs and textbook
Lab 4 Demo

Demo time and logistics:
1. Each demo will last approximately 20 min
2. The TA will provide you with the test cases
3. Come early to set-up your SW/HW. We expect the setup to be completed by the time of the demo
4. Sign up on the time-slot sheet posted inside the lab.
5. If you can't make any of the posted times periods, email the TA in advance so you can arrange an appointment time to do your demo.
6. If you do not complete your demo by Sunday at 6:10 pm, 3/15, you will get a penalty of 10% per day late.
7. Remove all your files and reset the password of your computer by the end of your demo to get your grade.

Demo location: Lab room CSE 3219

Available times:
Friday 3/13: 10:00 am - 12:00 pm
Saturday 3/14: 5:00 pm - 6:20 pm
Sunday 3/15: 5:00 pm - 6:20 pm
CSE140L: Components and Design Techniques for Digital Systems Lab

Synthesis

Tajana Simunic Rosing

Source: Vahid, Katz, Culler
Verilog and VHDL started out as simulation languages, but soon programs were written to automatically convert Verilog code into low-level circuit descriptions (netlists).

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

1. Automatically manages many details of the design process:
   • Fewer bugs
   • Improves productivity

2. Abstracts the design data (HDL description) from any particular implementation technology
   • Designs can be re-synthesized targeting different chip technologies; E.g.: 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 (e.g.: logic optimization)
How Does It Work?

• 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 (e.g.,: 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
Operators

- Logical operators map into primitive logic gates
- Arithmetic operators map into adders, subtractors, ...
  - Unsigned 2s complement
  - Model carry: target is one-bit wider than source
- Relational 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

\[ Y = \sim X \ll 2 \]
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

```
61C

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $t0, 0($2)
lw $t1, 4($2)
sw$t1, 0($2)
sw$t0, 4($2)
```

Machine Language Program (MIPS)

Machine Interpretation

Control Signal Specification
Simple Example

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

• Should expand if-else into 4-bit wide multiplexer (a, b, f are 4-bit vectors) and optimize/minimize the control logic:
Procedural Assignments

Verilog has two types of assignments within always blocks:

• **Blocking** procedural assignment “=“
  – RHS is executed and assignment is completed before the next statement is executed; e.g.,
    Assume A holds the value 1 … A=2; B=A; A is left with 2, B with 2.

• **Non-blocking** procedural assignment “<={
  – RHS is executed and assignment takes place at the end of the current time step (not clock cycle); e.g.,
    Assume A holds the value 1 … A<=2; B<=A; A is left with 2, B with 1.
Supported Verilog Constructs

- Net types: wire, tri, supply1, supply0; 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-if, case, casex, casez, for, repeat, while, forever, begin, end, fork, join
- Procedural assignments: blocking assignments =, nonblocking assignments <= (Note: <= cannot be mixed with = for the same register).
- Compiler directives: `define, `ifdef, `else, `endif, `include, `undef
- Miscellaneous:
  - Integer ranges and parameter ranges
  - Local declarations to begin-end block
  - Variable indexing of bit vectors on the left and right sides of assignments
Unsupported Language Constructs

*Generate error and halt synthesis*

- Net types: trireg, wor, trior, wand, triand, tri0, tri1, and charge strength;
- Register type: real
- Built-in unidirectional and bidirectional switches, and pull-up, pull-down
- Procedural statements: assign (different from the “continuous assignment”), deassign, wait
- Named events and event triggers
- UDPs (user defined primitives) and specify blocks
- Force, release, and hierarchical net names (for simulation only)

*Simply ignored*

- Delay, delay control, and drive strength
- Scalared, vectored
- 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 [7:0] 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.
       // Task and function calls
   end
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:
  - Sel 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

```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'd3: out = d;
  endcase
end
endmodule
```
Combinational Logic Always Blocks (cont.)

- To avoid synthesizing a latch in this case, add the missing select line:
  
  ```
  2'd2: out = c;
  ```

- Or, in general, use the “default” case:

  ```
  default: out = foo;
  ```

- If you don’t care about the assignment in a case (for instance you know that it will never come up) then assign the value “x” to the variable; E.g.:

  ```
  default: out = 1'bx;
  ```

  The x is treated as a “don’t care” for synthesis and will simplify the logic.

  (The synthesis directive “full_case” will accomplish the same, but can lead to differences between simulation and synthesis.)
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
Encoder Example

- Nested IF-ELSE might lead to “priority logic”
  - Example: 4-to-2 encoder

```verilog
always @(x)
begin : encode
  if (x == 4'b0001) y = 2'b00;
  else if (x == 4'b0010) y = 2'b01;
  else if (x == 4'b0100) y = 2'b10;
  else if (x == 4'b1000) y = 2'b11;
  else y = 2'bxx;
end
```

- This style of cascaded logic may adversely affect the performance of the circuit
Encoder Example (cont.)

- To avoid “priority logic” use the case construct:

```verilog
always @(x)
begin : encode
  case (x)
    4'b0001: y = 2'b00;
    4'b0010: y = 2'b01;
    4'b0100: y = 2'b10;
    4'b1000: y = 2'b11;
    default: y = 2'bxx;
  endcase
end
```

- All cases are matched in parallel
- Note, you don’t need the “parallel case” directive (except under special circumstances, described later)
Encoder Example (cont.)

- Circuit would be simplified during synthesis to take advantage of constant values as follows and other Boolean equalities:

A similar simplification would be applied to the if-else version also
• If you can guarantee that only one 1 appears in the input (one hot encoding), then simpler logic can be generated:

```verilog
always @(x)
begin : encode
  if (x[0]) y = 2'b00;
  else if (x[1]) y = 2'b01;
  else if (x[2]) y = 2'b10;
  else if (x[3]) y = 2'b11;
  else y = 2'bxx;
end
```

• If the input applied has more than one 1, then this version functions as a “priority encoder” -- least significant 1 gets priority (the more significant 1’s are ignored); the circuit will be simplified when possible.
Encoder Example (cont.)

- Parallel version, assuming we can guarantee only one 1 in the input:
  
  ```vhdl
  always @(x)
  begin : encode
    casex (x) // synthesis parallel_case
    4'bxxxx: y = 2'b00;
    4'bxx1x: y = 2'b01;
    4'bxxlx: y = 2'b10;
    4'bxxxx: y = 2'b11;
    default: y = 2'bxx;
  endcase
  end
  ```

- Note now more than one case might match the input
- Therefore use “parallel case” directive: without it, synthesis adds appropriate matching logic to force priority
  - Semantics of case construct says that the cases are evaluated from top to bottom
  - *Only an issue for synthesis when more than one case could match input*
Encoder Example (cont.)

- Parallel version of “priority encoder”:

```verilog
class always @(x)
begin : encode
  case (x)
    4'bxxxx1: y = 2'b00;
    4'bxxlx: y = 2'b01;
    4'bxxlx: y = 2'b01;
    4'b1xxx: y = 2'b11;
    default: y = 2'bxx;
  endcase
end
```

- Note: “parallel case” directive is not used, synthesis adds appropriate matching logic to force priority
  - Just what we want for a priority encoder

- Behavior matches that of the if-else version presented earlier
Sequential Logic

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

```verilog
module dff(q, d, clk, set, rst);
    input d, clk, set, rst;
    output q;
    reg q;
    always @(posedge clk)
        if (reset)
            q <= 0;
        else if (set)
            q <= 1;
        else
            q <= d;
endmodule
```

- “@ (posedge clk)” key to flip-flop generation
- Note in this case, priority logic is appropriate
- For Xilinx Virtex FPGAs, the tool infers a native flip-flop
  - No extra logic needed for set/reset

We prefer *synchronous* set/reset, but how would you specify *asynchronous* preset/clear?
module FSM1(clk,rst, enable, data_in, data_out);
input clk, rst, enable;
input data_in;
output data_out;

/* Defined state encoding;
this style preferred over 'defines */
parameter default=2'bxx;
parameter idle=2'b00;
parameter read=2'b01;
parameter write=2'b10;
reg data_out;
reg [1:0] state, next_state;

/* always block for sequential logic */
always @(posedge clk)
    if (rst) state <= idle;
    else state <= next_state;

• Style guidelines (some of these are to get the right result, and some just for readability)
  – Must have reset
  – Use separate always blocks for sequential and combination logic parts
  – Represent states with defined labels or enumerated types
FSMs (cont.)

/* always block for CL */
always @(state or enable or data_in)
begin
case (state)
/* For each state def output and next */ •
  idle : begin
    data_out = 1'b0;
    if (enable)
      next_state = read;
    else next_state = idle;
  end
read : begin ... end
write : begin ... end

  default : begin
    next_state = default;
    data_out = 1'bx;
  end
endcase
end
endmodule

• Use CASE statement in an always to implement next state and output logic
• Always use default case and assert the state variable and output to ‘bx:
  • Avoids implied latches
  • Allows use of don’t cares leading to simplified logic
• “FSM compiler” within synthesis tool can re-encode your states; Process is controlled by using a synthesis attribute (passed in a comment).
More information

• Online documentation for Synplify Synthesis Tool:
  – Under Documents/General Documentation, see Synplify Web Site/Literature:
  – Online examples from Synplicity
• Bhasker is a good synthesis reference