

CSE372

Digital Systems Organization and Design Lab

Prof. Milo Martin

Unit 1: Synthesizable Verilog

Lab 1 - ALU (Arithmetic/Logical Unit)

- Task: design an ALU for a P37X CPU
- Ten operations:
 - Addition, subtraction
 - Multiplication
 - And, or, not, xor
 - Shift left, logical shift right, arithmetic shift right
- The different adder implementations
 - Ripple-carry
 - Two carry-select adders
- Pay close attention in CSE371 lectures on this!

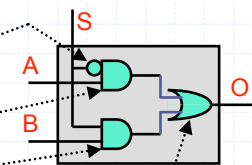
Hardware Description Languages (HDLs)

- Textual representation of a digital logic design
 - Easier to edit and revise than schematics
 - However, you still need to *think* in terms of schematics (pictures)
- HDLs are not “programming languages”
 - No, really. Even if they look like it, they are not.
 - One of the most difficult conceptual leaps of this course
- Similar development chain
 - Compiler: source code → assembly code → binary machine code
 - Synthesis tool: HDL source → gate-level specification → hardware

Hardware Description Languages (HDLs)

- Write “code” to describe hardware
 - Specify wires, gates, modules
 - Also hierarchical
- Pro: easier to edit and create; Con: more abstract

```
module mux2to1(S, A, B, O);  
    input S, A, B;  
    output O;  
    wire S_, AnS_, BnS;  
  
    not (S_, S);  
    and (AnS_, A, S_);  
    and (BnS_, B, S);  
    or (O, AnS_, BnS);  
endmodule
```



Verilog HDL

- **Verilog**
 - One of two commonly-used HDLs
 - Verilog is a (surprisingly) big language
 - Lots of features for synthesis and simulation of hardware
- **We're going to learn a focused *subset* of Verilog**
 - Focus on synthesizable constructs
 - Focus on avoiding subtle synthesis errors
 - Use as an educational tool
 - Initially restrict some features to "build up" primitives
 - **Rule: if you haven't seen it in lecture, you can't use it**
 - **Ask me if you have any questions**

HDL History

- 1970s: First HDLs
- Late 1970s: VHDL
 - VHDL = VHSIC HDL = Very High Speed Integrated Circuit HDL
 - VHDL inspired by programming languages of the day (Ada)
- 1980s:
 - Verilog first introduced
 - Verilog inspired by the C programming language
 - VHDL standardized
- 1990s:
 - Verilog standardized (Verilog-1995 standard)
- 2000s:
 - Continued evolution (Verilog-2001 standard)
- Both VHDL and Verilog evolving, still in use today

Two Roles of HDL and Related Tools

- **#1: Specifying digital logic**
 - Specify the logic that appears in final design
 - Either
 - Translated automatically (called *synthesis*) or
 - Optimized manually (automatically checked for equivalence)
- **#2: Simulating and testing a design**
 - High-speed simulation is crucial for large designs
 - Many HDL *interpreters* optimized for speed
 - Testbench: code to test design, but not part of final design

Synthesis vs Simulation

- **HDLs have features for *both* synthesis and simulation**
 - E.g., simulation-only operations for error messages, reading files
 - Obviously, these can be simulated, but not synthesized into circuits
 - Also has constructs such as for-loops, while-loops, etc.
 - These are either un-synthesizable or (worse) synthesize poorly
- Trends: a moving target
 - Good: better synthesis tools for higher-level constructs
 - Bad: harder than ever to know what is synthesizable or not

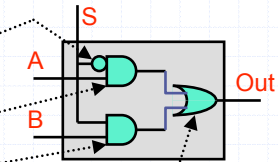
Structural vs Behavioral HDL Constructs

- **Structural** constructs specify actual hardware structures
 - Low-level, direct correspondence to hardware
 - Primitive gates (e.g., and, or, not)
 - Hierarchical structures via modules
 - Analogous to programming software in assembly
- **"Behavioral"** constructs specify an operation on bits
 - High-level, more abstract
 - Specified via equations, e.g., `out = (a & b) | c`
 - Statements, e.g., if-then-else
 - Analogous to programming software in C
- **Not all behavioral constructs are synthesizable**
 - Even higher-level, synthesize poorly or not at all (e.g., loops)
 - Perhaps analogous to programming in Perl, Python, Matlab, SQL

Verilog Structural Example

Structural

```
module mux2to1(S, A, B, Out);  
  input S, A, B;  
  output Out;  
  wire S_, AnS_, BnS_;  
  not (S_, S);  
  and (AnS_, A, S_);  
  and (BnS_, B, S);  
  or (Out, AnS_, BnS);  
endmodule
```



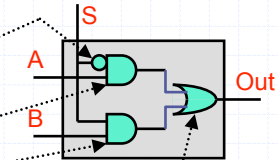
Verilog Structural Primitives

- Gate-level
 - One-output boolean operators: `and`, `or`, `xor`, `nand`, `nor`, `xnor`
 - E.g., `C = A+B`
`or (C, A, B);`
 - E.g., `C = A+B+D`
`or (C, A, B, D);`
 - One-input operators: `not`, `buf`
 - E.g., `A = not Z`
`not (A, Z);`
 - E.g., `A = not Z, B = not Z`
`not (A, B, Z);`
 - Buf just replicates signals (can increase drive strength)
- Transistor-level primitives too
 - Will not use

Verilog "Behavioral" Example

"Behavioral" (Synthesizable)

```
module mux2to1(S, A, B, Out);  
  input S, A, B;  
  output Out;  
  wire S_, AnS_, BnS_;  
  assign S_ = ~S;  
  assign AnS_ = A & S_;  
  assign BnS_ = B & S;  
  assign Out = AnS_ | BnS;  
endmodule
```



Wire Assignment

- Wire assignment: “continuous assignment”
 - Connect combinational logic block or other wire to wire input
 - **Order of statements not important**, executed totally in parallel
 - When right-hand-side changes, it is re-evaluated and re-assigned
 - Designated by the keyword **assign**

```
wire c;  
assign c = a | b;
```

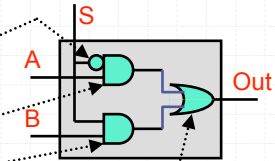
- Can be combined with declaration

```
wire c = a | b; // same thing
```
- Basic operators
 - Not: ~ Or: | And: & Xor: ^
 - Can be combined: (a & b) | (c ^ d)

Verilog “Behavioral” Example

“Behavioral” (Synthesizable)

```
module mux2to1(S, A, B, Out);  
  input S, A, B;  
  output Out;  
  wire S_, AnS_, BnS;  
  assign S_ = ~S;  
  assign AnS_ = A & S_  
  assign BnS = B & S;  
  assign Out = AnS_ | BnS;  
endmodule
```



```
module mux2to1(S, A, B, Out);  
  input S, A, B;  
  output Out;  
  assign Out = (~S & A) | (S & B);  
endmodule
```

Wires are not C-like Variables

- Order of assignment doesn’t matter
- Can’t reuse a wire

```
assign temp = a & b;  
assign c = temp;  
assign temp = a | b;  
assign d = temp;
```
- Can’t assign a wire multiple values
 - Actually, you can; but don’t do it.

Aside: Non-binary Hardware Values

- A hardware signal can have any of four values
 - **0, 1**
 - **x**: don’t know, don’t care
 - **z**: high-impedance (no current flowing)
- Uses for “x”
 - Tells synthesis tool you don’t care
 - Synthesis tool makes the most convenient circuit (fast, small)
 - Use with care, leads to synthesis dependent operation
- Uses for “z”
 - Tri-state devices drive a zero, one, or “off” (z)
 - Many tri-states drive the same wire, all but one must be “z”
 - Makes some circuits very fast
 - Example: multiplexer
 - This is why Verilog allows multiple assignments to same wire

Vectors of Wires

- Wire vectors:

```
wire [7:0] W1;    // 8 bits, w1[7] is MSB
wire [0:7] W2;    // 8 bits, w2[0] is MSB
```

- Also called "arrays" or "busses"

- Operations

- Bit select: `W1[3]`
- Range select: `W1[3:2]`
- Concatenate: `{<expr>[,<expr>]*}`

```
vec = {x, y, z};
{carry, sum} = vec[0:1];
```
- e.g., swap high and low-order bytes of 16-bit vector

```
wire [15:0] w1, w2;
assign w2 = {w1[7:0], w1[15:8]}
```

Vector Constants

- Constants:

- assign `x = 3'b011`
- The "3" is the number of bits
- The "b" means "binary" - "h" for hex, "d" for decimal
- The "011" are the digits (in binary in this case)

Repeated Signals

- Concatenation

```
assign vec = {x, y, z};
```

- Can also repeat a signal n times

```
assign vec = {16{x}}; // 16 copies of x
```

- Example uses (what does this do?):

```
wire [7:0] out;
wire [3:0] A;
assign out = {{4{0}}, A[3:0]};
```

- What about this?

```
assign out = {{4{A[3]}}, A[3:0]};
```

Operator List

- Operators similar to C or Java

- On wires:

- & (and), | (or), ~ (not), ^ (xor)

- On vectors:

- &, |, ~, ^ (bit-wise operation on all wires in vector)
 - E.g., `assign vec1 = vec2 & vec3;`
- &, |, ^ (reduction on the vector)
 - E.g., `assign wire1 = | vec1;`
- Even ==, != (comparisons), +, -, * (arithmetic), <<, >> (shifts)
 - But you can't use these, yet. Can you guess why?
 - Note: use with care, assume unsigned numbers
 - Verilog 2001: signed vs unsigned vectors, >>> operator

- Can be arbitrarily nested: `(a & ~b) | c`

Conditional Operator

- Verilog supports the ?: conditional operator
 - Almost never useful in C (in *my* opinion)
 - Much more useful (and common) in Verilog

- Examples:

```
assign out = S ? B : A;
```

```
assign out = (sel == 2'b00) ? a :  
             (sel == 2'b01) ? b :  
             (sel == 2'b10) ? c :  
             (sel == 2'b11) ? d : 1'b0;
```

- What do these do?

Hierarchical Design using Modules

- Interface specification

```
module mux2to1(S, A, B, O);  
    input S, A, B;  
    output O;
```

- Can also have `inout`: bidirectional wire (we will not need)

- Alternative: Verilog 2001 interface specification

```
module mux2to1(input S, A, B, output O);
```

- Declarations

- Internal wires, i.e., "local" variables
- Wires also known as "nets" or "signals"

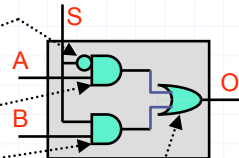
```
wire S_, AnS_, BnS;
```

- Implementation: primitive and module instantiations

```
and (AnS_, A, S_);
```

Verilog Module Example

```
module mux2to1(S, A, B, O);  
    input S, A, B;  
    output O;  
    wire S_, AnS_, BnS;  
  
    not (S_, S);  
    and (AnS_, A, S_);  
    and (BnS, B, S);  
    or (O, AnS_, BnS);  
endmodule
```



- Instantiation: `mux2to1 mux0 (cond, in1, in2, out);`
- Operators and expressions can be used with modules
 - `mux2to1 mux0 (cond1 & cond2, in1, in2, out);`

Hierarchical Verilog Example

- Build up more complex modules using simpler modules
- Example: 4-bit wide mux from four 1-bit muxes
 - Again, just "drawing" boxes and wires

```
module mux2to1_4(Sel, A, B, O);  
    input [3:0] A;  
    input [3:0] B;  
    input Sel;  
    output [3:0] O;  
  
    mux2to1 mux0 (Sel, A[0], B[0], O[0]);  
    mux2to1 mux1 (Sel, A[1], B[1], O[1]);  
    mux2to1 mux2 (Sel, A[2], B[2], O[2]);  
    mux2to1 mux3 (Sel, A[3], B[3], O[3]);  
endmodule
```

Connections by Name

- Can (should?) specify module connections by name

- Helps keep the bugs away

- Example

```
mux2to1 mux0 (.S(Sel), .A(A[0]), .B(B[0]), .O(O[0]));
```

- Also, then order doesn't matter

```
mux2to1 mux1 (.A(A[1]), .B(B[1]), .O(O[1]), .S(Sel));
```

Arrays of Modules

- Verilog also supports arrays of module instances

- Well, at least some Verilog tools

- Support for this feature varies (may not work well in Xilinx)

```
module mux2to1_4(Sel, A, B, O);
```

```
    input [3:0] A;
```

```
    input [3:0] B;
```

```
    input Sel;
```

```
    output [3:0] O;
```

```
    mux2to1 mux0[3:0] (Sel, A, B, O);
```

```
endmodule
```

Parameters

- Allow per-instantiation module parameters

- Use "parameter" statement

- modname #(10, 20, 30) instname (in1, out1);

- Example:

```
module mux2to1_N(Sel, A, B, O);
```

```
    parameter N = 1
```

```
    input [N-1:0] A;
```

```
    input [N-1:0] B;
```

```
    input Sel;
```

```
    output [N-1:0] O;
```

```
    mux2to1 mux0[N-1:0] (Sel, A, B, O);
```

```
endmodule
```

```
...
```

```
mux2to1_N #(4) mux1 (S, in1, in2, out)
```

Last Multiplexer Example

- Using conditional operator

```
module mux2to1_N(Sel, A, B, Out);
```

```
    parameter N = 1
```

```
    input [N-1:0] A;
```

```
    input [N-1:0] B;
```

```
    input Sel;
```

```
    output [N-1:0] Out;
```

```
    assign Out = Sel ? B : A
```

```
endmodule
```

```
mux2to1_N #(4) mux1 (S, in1, in2, out)
```

Comments

- C/Java style comments
 - // comment until end of line
 - /* comment between markers */
- Note: all variable names are case sensitive

Verilog Pre-Processor

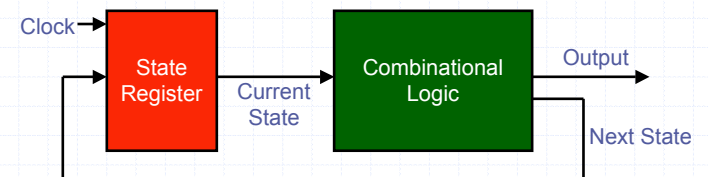
- Like the C pre-processor
 - But uses ` (back-tick) instead of #
 - Constants: ``define`
 - No parameterized macros
 - Use ` before expanding constant macro
- Conditional compilation: ``ifdef`, ``endif`
- File inclusion: ``include`
- Parameter vs `define
 - Parameter only for "per instance" constants
 - `define for "global" constants

Recall: Two Types of Digital Circuits

- **Combinational Logic**
 - Logic without state variables
 - Examples: adders, multiplexers, decoders, encoders
 - No clock involved
- **Sequential Logic**
 - Logic with state variables
 - State variables: latches, flip-flops, registers, memories
 - Clocked
 - State machines, multi-cycle arithmetic, processors
- **Sequential Logic in Verilog**
 - Special idioms using behavioral constructs that synthesize into latches, memories

Designing Sequential Logic

- CSE372 design rule: separate comb. logic from sequential state elements
 - Not enforced by Verilog, but a very good idea
 - Possible exceptions: counters, shift registers
- We'll give you a flip-flop module (see next slide)
 - Edge-triggered, not a transparent latch
 - Parameterized to create a n -bit register
- Example use: state machine



Sequential Logic in Verilog

- How do we specify state-holding constructs in Verilog?

```
module dff (out, in, wen, rst, clk);
```

```
    output out;
    input in;
    input wen, rst, clk;
```

wen = write enable
rst = reset
clk = clock

```
    reg out;
    always @(posedge clk)
    begin
        if (rst)
            out = 0;
        else if (wen)
            out = in;
    end
```

```
endmodule
```

CSE 372 (Martin): Synthesizable Verilog

Larger memories done similarly
(we will also give these to you)

33

Sequential Logic in Verilog

- How do we specify state-holding constructs in Verilog?

```
module register (out, in, wen, rst, clk);
```

```
    parameter n = 1;
    output [n-1:0] out;
    input [n-1:0] in;
    input wen, rst, clk;
```

wen = write enable
rst = reset
clk = clock

```
    reg [n-1:0] out;
    always @(posedge clk)
    begin
        if (rst)
            out = 0;
        else if (wen)
            out = in;
    end
```

```
end
```

```
endmodule
```

CSE 372 (Martin): Synthesizable Verilog

34

Clocks Signals

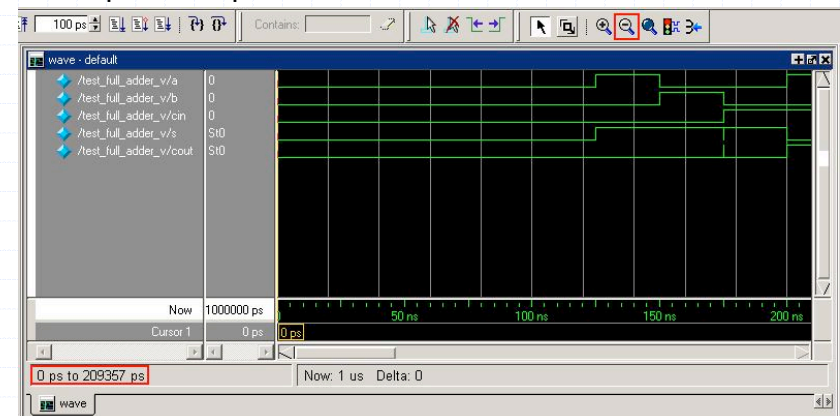
- Clocks & reset signals are **not** normal signals
- Travel on dedicated "clock" wires
 - Reach all parts of the chip
 - Special "low-skew" routing
- Ramifications:
 - Never do logic operations on the clocks
 - If you want to add a "write enable" to a flip-flop:
 - Use a mux to route the old value back into it
 - (or use the flip-flop with write enable we give you!)
 - Do not just "and" the write-enable signal with the clock!
- Messing with the clock can cause a errors
 - Often can only be found using timing simulation

CSE 372 (Martin): Synthesizable Verilog

35

Simulation

- Used to test and debug our designs
- Graphical output via waveforms



CSE 372 (Martin): Synthesizable Verilog

36

Levels of Simulation

- Functional (or Behavioral) Simulation
 - Simulates Verilog abstractly
 - No timing information, can't detect "timing bugs"
- Post-synthesis Timing Simulation
 - Simulating devices generated via synthesis
 - Gates, transistors, FPGA logical units (LUTs or lookup tables)
 - No interconnect delay
 - Not all internal signals may still exist
 - Synthesis might have optimized or changed the design
 - Slower
- Layout Timing Simulation
 - After synthesis, the tool "places and routes" the logic blocks
 - Includes all sources of delay
 - Even slower

Testbenches

- How does one test code?
- In C/Java?
 - Write test code in C/Java to test C/Java
 - "Test harness", "unit testing"
- For Verilog/VHDL?
 - Write test code in Verilog to test Verilog
 - Verilog has advanced "behavioral" commands to facilitate this:
 - Delay for n units of time
 - Full high-level constructs: if, while, sequential assignment, ints
 - Input/output: file I/O, output to display, etc.
- Example as part of Lab 1

Common Errors

- Tools are from a less gentle time
 - More like C, less like Java
 - Assume that you mean what you say
- Common errors:
 - Not assigning a wire a value
 - Assigning a wire a value more than once
 - Implicit wire declarations (default to type "wire" 1-bit wide)
 - Disable by adding the following to the file:
 - ``default_nettype none`
 - Does not work with ModelSim
- Avoid names such as:
 - clock, clk, power, pwr, ground, gnd, vdd, vcc, init, reset, rst
 - Some of these are "special" and will silently cause errors

Additional Verilog Resources

- Elements of Logic Design Style by Shing Kong, 2001
 - Dos, do-nots, tips
 - <http://www.cis.upenn.edu/~milom/elements-of-logic-design-style/>
- Verilog HDL Synthesis: A Practical Primer
 - By J. Bhasker, 1998
 - To the point (<200 pages)
- Advanced Digital Design with the Verilog HDL
 - By Michael D. Ciletti, 2003
 - Verilog plus lots of digital logic design (~1000 pages)
- Verilog tutorial on CD from "Computer Org. and Design"