CIS 372
Computer Organization and Design Lab

Prof. Amir Roth

Unit 1: (Synthesizable) Verilog HDL

Assignments

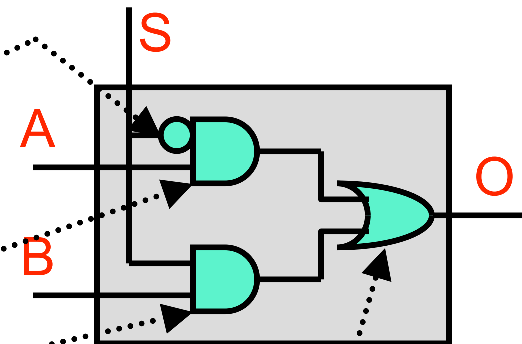
- Lab 0: LC4 programmable timer device
  - http://www.seas.upenn.edu/~cse372/lab0-s09/
  - Due Fri. 2/6 (in 9 days)
  - Demo & writeup in K-lab by 6:30pm
  - ISE tutorial + assignment should take ~ 3 hours
- Lab 1: single-cycle LC4 datapath
  - Up next week
  - Due Fri. 2/27 (3 weeks after lab0)
  - Demo (maelstrom) & writeup in K-lab by 6:30pm
  - Should take 10-20 hours

Hardware Description Languages (HDLs)

- Textual representation of a hardware design
  - \+ Easier to edit and revise than schematics
  - – Still need to *think* in terms of schematics (pictures)

```
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
```



HDLs are not SDLs

- Similar in some (intentional) ways …
  - Syntax
    - Named entities, constants, scoping, etc.
  - Tool chain: synthesis tool analogous to compiler
    - Multiple levels of representation
    - "Optimization"
    - Multiple targets (portability)
  - "Software engineering"
    - Modular structure and parameterization
    - Libraries and code repositories

- … but different in many others
  - One of the most difficult conceptual leaps of this course

Hardware is not Software

- Just two different beasts (or two parts of the same beast)
  - Things that make sense in hardware, don't in software, vice versa
  - One of the main themes of 371/372
- **Software is sequential**
  - Hardware is inherently parallel, at multiple levels
  - Have to work to get hardware to *not* do things in parallel
- Software atoms are purely functional ("digital")
  - Hardware atoms have quantitative ("analog") properties too
  - Including correctness properties!
- Software mostly about quality ("functionality")
  - Hardware mostly about quantity: performance, area, power, etc.
- One reason that HDLs are not SDLs

HDL & SDL: Specification and Synthesis

- One use of HDL: specifying a hardware design
  - Analogous usage model for SDL
- Specification can be translated automatically to hardware…
  - Called **synthesis**
  - Analogous to compilation of SDL
- … or automatically checked for equivalence with manually optimized hardware design
  - Can conceive of a software analog, but not a common usage model
  - Useful for hardware because quantitative optimization so important
  - Humans (can see big picture) can sometimes do a better job

HDL: Simulation

- Another use of HDL: simulating & testing a hardware design
  - No software analog for this, not even interpretation
  - Many free tools (e.g., Icarus) do simulation only, but no synthesis
- Another reason that HDLs are not SDLs
- Both have synthesis features
- **HDLs have simulation features too**
  - High level data types: integers, FP-numbers, timestamps
  - HLL control structures: for-loops, conditionals
  - Routines for I/O: error messages, file operations
  - Obviously, these cannot be synthesized into circuits
- Also another reason for HDL/SDL confusion
  - HDLs have "SDL" features for simulation

HDL: Behavioral Constructs

- HDLs have **low-level structural** constructs
  - Specify hardware structures directly
  - Gates (`and`, `not`) and wires, hierarchy via modules
- Also have **multi-level behavioral** constructs
  - Specify operations, not hardware to perform them
  - Low-to-medium-level: `&`, `~`, `+`, `*`
  - High-level: `if-then-else`, `for`
  - Some of these are synthesizable
    - For `for` loops, tools try to guess what you want
    - Often guess wrong, or "not quite right"
    - – Higher-level → more difficult to know what it will synthesize to!
  - Software parallel is declarative languages like SQL
- HDLs are like Java and SQL in one language
  - Where you don't really know where the boundary is!

HDL: Multi-Level Structural Constructs

- HDLs can specify structure at both gate and transistor level
  - Software analog would be Java and assembly in one language
- So HDLs are really assembly + Java + SQL in one!
  - Good times

HDL History

- 1970s
  - First HDLs
- Late 1970s: VHDL
  - VHDL introduced: inspired by Ada language
  - VHDL: VHSIC (Very High Speed Integrated Circuit) HDL
- 1980s
  - VHDL standardized
  - Verilog introduced: inspired by C
- 1990s
  - Verilog standardized (Verilog-1995 standard)
- 2000s
  - Continued evolution (Verilog-2001 standard)
- Both VHDL and Verilog evolving, still in use today
  - Some companies (e.g., Intel) have proprietary languages

Verilog HDL

- Verilog is a (surprisingly) big language
  - Structural constructs at both gate and transistor level
  - Facilities for specifying memories (including ROMs)
  - Precise timing specification and simulation
  - All C behavioral constructs (and some others)
  - C-style procedural variables, including arrays
  - A pre-processor
  - VPI: Verilog programming interface

372 Verilog HDL

- **We're going to learn a focused subset of Verilog**
- For synthesis
  - Structural constructs at gate-level only
  - A few behavioral constructs
  - Pre-processor
- For testing and debugging
  - C-style procedural variables
  - Some VPI

Rule I: if you haven't seen it in lecture, you can't use it!

Rule IA: when in doubt, ask … me!

Basic Verilog Syntax

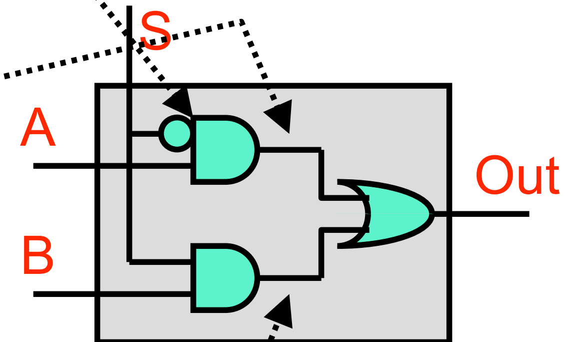
- Have already seen basic syntax, looks like C
  - C/C++/Java style comments
  - Names are case sensitive, and can use _ (underscore)
  - Avoid: clock, clk, power, pwr, ground, gnd, vdd, vcc, init, reset, rst
    - Some of these are "special" and will silently cause errors

```
/* this is a module */
module mux2to1(S, A, B, O);
    input S, A, B;
    output O;
    wire S_, AnS_, BnS;
    // these are gates
    not (S_, S);
    and (AnS_, A, S_);
    and (BnS, B, S);
    or (O, AnS_, BnS);
endmodule
```

(Gate-Level) Structural Verilog

- Primitive "data type": **`wire`**
  - Declare

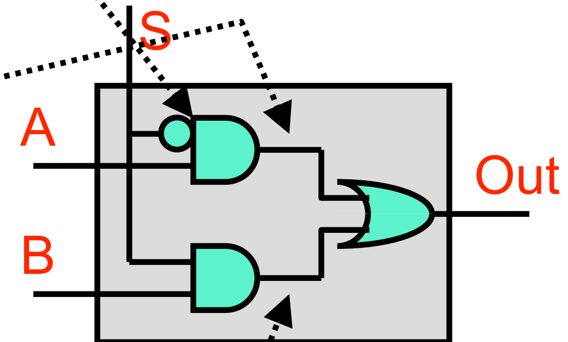
```
module mux2to1(S, A, B, Out);
  input S, A, B;
  output Out;
  wire S_, AnS_, BnS;
  not (S_, S);
  and (AnS_, A, S_);
  and (BnS, B, S);
  or (Out, AnS_, BnS);
endmodule
```



(Gate-Level) Structural Verilog

- Primitive "operators": gates
  - Specifically: **and**, **or**, **xor**, **nand**, **nor**, **xnor**, **not**, **buf**
  - Can be multi-input: e.g., **or (C, A, B, D)** (C= A+B+D)
  - "Operator" **buf** just repeats input signal (may amplify it)

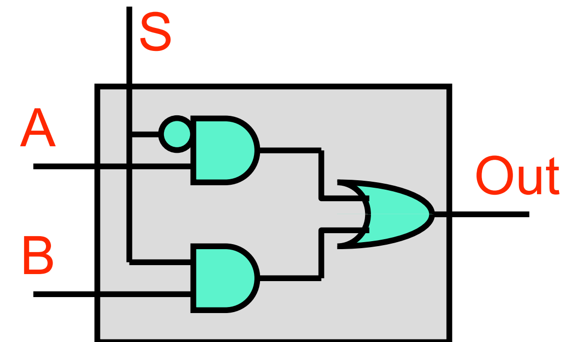
```
module mux2to1(S, A, B, Out);
  input S, A, B;
  output Out;
  wire S_, AnS_, BnS;
  not (S_, S);
  and (AnS_, A, S_);
  and (BnS, B, S);
  or (Out, AnS_, BnS);
endmodule
```



(Gate-Level) Behavioral Verilog

- Primitive "operators": boolean operators
  - Specifically: **&**, **|**, **^**, **~**
  - Can be combined into expressions
  - Can be mixed with structural Verilog

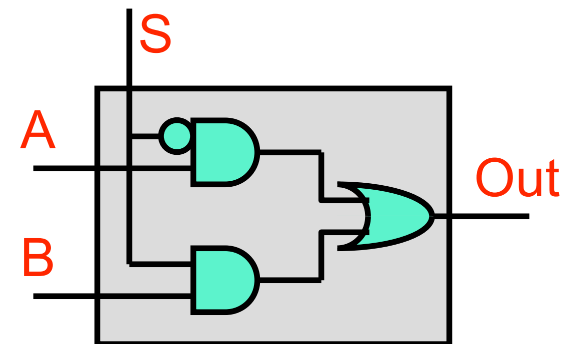
```
module mux2to1(S, A, B, Out);
  input S, A, B;
  output Out = (~S & A) | (S & B);
endmodule
```



Easiest Way to do a Mux?

- Verilog supports `?:` conditional assignment operator
  - Much more useful (and common) in Verilog than in C/Java

```
module mux2to1(S, A, B, Out);
  input S, A, B;
  output Out = S ? B : A;
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 without assign

```
wire c = a | b;      // same thing
```

Wires Are Not C-like Variables!

- Order of assignment doesn't matter
  - This works fine

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

- Can't "reuse" a wire

```
assign temp = a & b;
assign temp = a | b;
```

  - Actually, you can; but don't do it

Wire Vectors

- Wire vectors: also called "arrays" or "buses"

```
wire [7:0] w1;          // 8 bits, w1[7] is MSB
wire [0:7] w2;          // 8 bits, w2[0] is MSB
wire [7:0] w1, w2;      // two 8 bit buses
```

Wire and Wire Vector Constants

```
wire [3:0] w = 4'b0101;
```

- The "4" is the number of bits
- The "b" means "binary" - "h" for hex, "o" for octal, "d" for decimal
- The "0101" are the digits (in binary in this case)

```
wire [3:0] w = 4'd5; // same thing, effectively
```

- Here is a single wire constant

```
wire w = 1'b0;
```

- A useful example of wire-vector constants: 4-to-1 mux

```
assign out = (S == 2'b00) ? A :
             (S == 2'b01) ? B :
             (S == 2'b10) ? C : D;
```

Wire Select and Concatenate

- Bit and range select

```
wire [2:0] vec1, vec2;
assign vec2[2] = vec1[0];
assign vec2[1:0] = vec1[2:1];
```

- Concatenate

```
wire [15:0] vec1, vec2;
assign vec2 = {vec1[7:0], vec1[15:8]};
```

Ultra-handy, you will use these … a lot!

Repeated Signals

- Can also repeat a signal a constant number of times

```
wire x;
wire [15:0] vec = {16{x}};  // 16 copies of x
```

- Also handy
  - What does this do?

```
wire [7:0] a;
wire [15:0] zxa = {{8{1'b0}}, a[7:0]};
```

  - What about this?

```
wire [15:0] sxa = {{8{a[7]}}, a[7:0]};
```

Behavioral Vector Operators

- Verilog also supports behavioral vector operators

- **Logical bitwise and reduction**: ~,&,|,^

```
wire [7:0] vec1, vec2;
wire [7:0] vec3 = vec1 & vec2; // bitwise AND
wire w1 = ~|vec1;              // NOR reduction
```

- **Integer arithmetic comparison**: +,–,*,/,%,==,!=,<,>

```
wire [7:0] vec4 = vec1 + vec2; // vec1 + vec2
```

  - Important: all arithmetic is unsigned, want signed? "roll your own"
  - Good: in 2C, signed/unsigned +, –, * produces same output
    - Just a matter of interpretation
  - Bad: in 2C, signed/unsigned /, % is not the same
  - Ugly: Xilinx will not synthesize /, % anyway!
    - Must rewrite maelstrom not to use DIV and MOD instructions

Why Use a High-Level Operator?

- Abstraction
  - Why write assembly, when you can write C? (not a great example)

- Take advantage of built-in high level implementation
  - Virtex-IIPro FPGAs have integer multipliers on them
  - Xilinx will use these rather than synthesizing a multiplier from gates
    - Much faster and more efficient
  - How hard is it for Xilinx to figure out you were doing a multiply?
    - If you use `*`: easy
    - If you "roll your own" using gates: nearly impossible

- Why not use high-level operators?
  - Less certain what they will synthesize to
  - Or even if it will synthesize at all: e.g., `/`, `%`

Hierarchical Design using Modules

- Interface specification

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

- Alternative: Verilog 2001 interface specification

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

- Can also have **`inout`**: bidirectional wire
  - We will not need

Hierarchical Verilog

- Build up more complex modules using simpler modules
  - E.g.: 4-bit wide mux from four 1-bit muxes
  - **Instances of non-primitive modules must be named**
    - So that you can "drill-down" into them
  - Names are locally scoped (can reuse outside the module)

```
module mux2to1_4(S, A, B, O);
    input [3:0] A, B;
    input S;
    output [3:0] O;
    mux2to1 m0 (S, A[0], B[0], O[0]);
    mux2to1 m1 (S, A[1], B[1], O[1]);
    mux2to1 m2 (S, A[2], B[2], O[2]);
    mux2to1 m3 (S, A[3], B[3], O[3]);
endmodule
```

Connections by Name

- Can (and should) specify module connections by name
  - Helps keep the bugs away
  - Also, then order doesn't matter

```
module mux2to1_4(S, A, B, O);
    input [3:0] A, B;
    input S;
    output [3:0] O;
    mux2to1 m0 (.S(S), .A(A[0]), .B(B[0]), .O(O[0]));
    mux2to1 m2 (.B(B[2]), .O(O[2]), .S(S), .A(A[2]));
    mux2to1 m3 (.O(O[3]), .B(B[3]), .A(A[3]), .S(S));
    mux2to1 m1 (.A(A[1]), .S(S), .B(B[1]), .O(O[1]));
endmodule
```

Arrays of Modules

- Verilog also supports arrays of module instances
  - Well, at least some Verilog tools do
  - Support for this feature varies (may not work well in Xilinx)

```
module mux2to1_4(S, A, B, O);
    input [3:0] A, B;
    input S;
    output [3:0] O;
    mux2to1 m[3:0] (S, A, B, O);
endmodule
```

Per-Instance Module Parameters

- Module parameters: useful for defines varying bus widths
  - Similar to C++ templates …
  - But for widths, not "types" (in HDL "width" == "type")

```
module mux2to1_N(S, A, B, O);
   parameter N = 1;
   input [N-1:0] A, B; input S; output [N-1:0] O;
   mux2to1 m[N-1:0] (S, A, B, O);
endmodule
```

  - Two ways to instantiate: implicit

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

  - And explicit

```
mux2to1_N mux1 (S, in1, in2, out)
defparam mux1.N = 4;
```

  - Multiple parameters per module allowed

Verilog Pre-Processor

- Like C pre-processor, but uses ` (back-tick) instead of #

- Constants

```
`define LETTER_A 8'h41
wire [7:0] vec = `LETTER_A;
`define SEVENTH_BIT [7]
assign vec`SEVENTH_BIT = 1'b1;
```

  - Use macros for global constants
  - Use parameters for per-instance constants

- Conditional compilation and file inclusion

```
`ifdef INCLUDE_THIS_FILE
   `include "this_file.v"
`endif
```

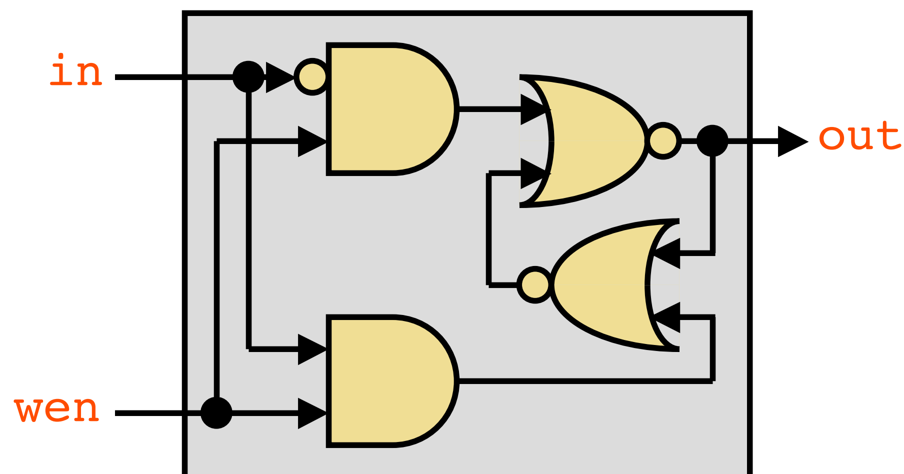
Two Types of Digital Circuits

- **Combinational**
  - No state variables
  - No clock
  - Examples: adders, multiplexers, decoders, encoders
- **Sequential logic**
  - State variables: latches, flip-flops, registers, memories
  - Clock
  - Examples: state machines, multi-cycle arithmetic, processors

Sequential Logic In Verilog

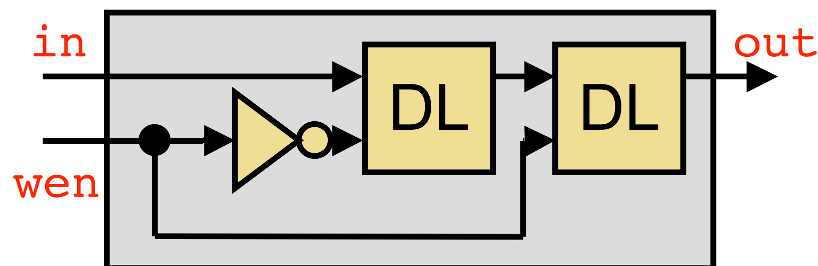
- How are state-holding variables specified in verilog?
  - First instinct: structurally
  - After all, real latches and flip-flops are made from gates…

```
module dl (out, in, wen);
   output out; input in, wen; wire out_
   assign out_ = ~(out | (wen & ~in));
   assign out = ~(out_ | (wen & in));
endmodule
```



Sequential Logic In Verilog

```
module dff (out, in, wen);
    output out; input in, wen; wire ms;
    dl master (ms, in, ~wen);
    dl slave (out, ms, wen);
endmodule
```



- This should work, right? RIGHT?
  - Logically, yes … but timing may get screwed up on FPGA
  - FPGA is not a “real” processor
- Also, FPGA has flip-flop primitives (like multiplier), use that
  - But how do you specify them? (`*` specifies multiplier primitive)
  - Behaviorally

Behavioral Register

```
module Nbit_reg (out, in, write_enable, reset, clock);
  parameter n = 1;
  output [n-1:0] out; input [n-1:0] in;
  input write_enable, reset, clock;
  reg [n-1:0] out;
  always @(posedge clock) begin
      if (reset)                   out = 0;
      else if (write_enable) out = in;
  end
endmodule
```

- **reg**: interface-less storage bit
- **always @ ()**: synthesizable behavioral sequential Verilog
  - Tricky: hard to know exactly what it will synthesize to
  - We will give this to you, don't write your own
  - "Creativity is a poor substitute for knowing what you're doing"

The Plan

- Lab 0: programmable timer device (for LC4)
  - "Goes off" after programmable number of ms (milli-seconds)
  - Resets when "gone off" and "read"
  - Can use Nbit_reg + any combinational feature we've covered
  - Can do this in 3-4 lines of Verilog (but more is fine too)
  - Due Friday 2/6, 6:30pm (worksheet + demo)
- Friday 1/30, meet in K-lab (Moore 204)
  - Boards, lockers handed out
  - I will demo lab0 and lab1
  - You can get started on lab0, TAs and I will be around to help
- Next 372 lecture, Monday 2/9
  - Will cover programmatic variables, VPI, clocking, and simulation
  - Enough to get you started on Lab 1