Overview

• Why Verilog?
• High-level description of Verilog
• Details:
  • Primitives
  • Modules and instances
  • Wires and Nets
  • Operators
  • Sequential logic
  • Miscellaneous
    • Parameters, Pre-processor, case statements, Common errors, system tasks
• Case study

How To Represent Hardware?

• If you’re going to design a computer, you need to write down the design so that:
  • You can read it again later
  • Someone else can read and understand it
  • It can be simulated and verified
  • Even software people may read it!
  • It can be synthesized into specific gates
  • It can be built and shipped and make money

Ways to represent hardware:

• Draw schematics
  • Hand-drawn
  • Machine-drawn
• Write a netlist
  • Z52BH 11234 (N123, N234, N4567);
• Write primitive Boolean equations
  • \( AAA = abc \text{ DEF} + ABC \text{ def} \)
• Use a Hardware Description Language (HDL)
  • assign overflow = c31 ^ c32;
Hardware Description Languages (HDLs)

- Textual representation of a digital logic design
  - Can represent specific gates, like a netlist, or more abstract logic
- HDLs are not "programming languages"
  - No, really. Even if they look like it, they are not.
  - For many people, a difficult conceptual leap
- Similar development chain
  - Compiler: source code → assembly code → binary machine code
  - Synthesis tool: HDL source → gate-level specification → hardware

Why use an HDL?

- Easy to write and edit
- Compact
- Don't have to follow a maze of lines
- Easy to analyze with various tools

Why not to use an HDL

- You still need to visualize the flow of logic
- A schematic can be a work of art
  - But often isn't!

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)
  - Continued evolution (Verilog-2001 standard)
  - Both VHDL and Verilog evolving, still in use today

And the answer is...

- In general, digital logic is captured in an HDL
- For government / aerospace work, it's VHDL
- For all else, it's Verilog
  - (This is, of course, a generalization...)

Verilog is not perfect!

- But then, neither is the X86 instruction set.
- And it's nowhere near that bad.
- In fact, it's pretty good...
  - If you know what to watch out for.
Starting with an example...

module fulladd {
    input A, B, Cin, 
    output sum, Cout);

assign sum = A ^ B ^ Cin;
assign Cout = (A & B)
| (A & Cin)
| (B & Cin);
endmodule

So, about Verilog...

Verilog is a (surprisingly) big language
- Lots of features for synthesis and simulation of hardware
- Can represent low-level features, e.g. individual transistors
- Can act like a programming language, with "for" loops etc.

- We're going to learn a focused subset of Verilog
  - We will use it at a level appropriate for computer design
  - Focus on synthesizable constructs
  - Focus on avoiding subtle synthesis errors
  - Initially restrict some features just because they aren't necessary
  - Rule: if you haven't seen it approved, you can't use it
  - Ask me if you have any questions

Why an HDL is not a Programming Language
- In a program, we start at the beginning (e.g. "main"), and we proceed sequentially through the code as directed
- The program represents an algorithm, a step-by-step sequence of actions to solve some problem
  for (i = 0; i<10; i=i+1) {
    if (newPattern == oldPattern[i]) match = i;
  }
- Hardware is all active at once; there is no starting point

Pitfalls of trying to "program" in Verilog
- If you program sequentially, the synthesizer may add a lot of hardware to try to do what you say
  - In last example, need a priority encoder
- If you program in parallel (multiple "always" blocks), you can get non-deterministic execution
  - Which "always" happens first?
- You create lots of state that you didn't intend
  - If (x == 1) out = 0;
  - If (y == 1) out = 1;
  - // else out retains previous state? R-S latch!
- You don't realize how much hardware you're specifying
  - x = x + 1 can be a LOT of hardware
- Slight changes may suddenly make your code "blow up"
  - A chip that previously fit suddenly is too large or slow
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
    - You need procedural code for testbench and only for testbench

- Trends: a moving target
  - Good: better synthesis tools for higher-level constructs
  - Bad: harder than ever to know what is synthesizable or not

- Important distinction: What is a "higher-level" construct and what is "procedural code"?

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

- Not all behavioral constructs are synthesizable
  - We've already talked about the pitfalls of trying to "program"
  - But even some combinational logic won't synthesize well
  - out = a % b // modulo operation – what does this synthesize to?
  - We will **not** use: + - * / % > >= < <= >> <<

Verilog Structural vs Behavioral Example

Structural

```verilog
module mux2to1(
    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
```

Behavioral

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

- Why Verilog?
- High-level description of Verilog

Details:
- Primitives
- Modules and instances
- Wires and Nets
- Operators
- Sequential logic
- Miscellaneous
  - Parameters, Pre-processor, case statements, Common errors, system tasks
- Case study

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

Verilog Structural Primitives

- Gate-level
  - One-output boolean operators: and, or, xor, nand, nor, xnor
    - E.g., \( C = A+B \)
    - \( \text{or} \ (C, A, B) \);
    - E.g., \( C = A+B+D \)
    - \( \text{or} \ (C, A, B, D) \);
  - One-input operators: not
    - E.g., \( A = \neg Z \)
    - \( \text{not} \ (A, Z) \);
    - E.g., \( A = \neg Z, B = \neg Z \)
    - \( \text{not} \ (A, B, Z) \);
    - Buf is like not but just replicates signals – we don’t need
  - Transistor-level primitives too
    - We will not use

Three Module Components

- Interface specification – new style (Verilog 2001)

```
module mux2to1(
    input S, A, B,
    output O);
```
- Can also have inout: bidirectional wire (we will not need or use)
- Declarations
  - Internal wires, i.e., wires that remain within this module
  - 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(
    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
```

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(
    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 mux1 (.A (A[1]),
                   .B (B[1]),
                   .O (O[1]),
                   .S (Sel));
    ```
  - Verilog won't complain about the order (but it is still poor practice to mix them up):

Wire and Vector Assignment

- Wire assignment: "continuous assignment"
  - Connect combinational logic block or other wire to wire input
  - Order of statements not important to Verilog, executed totally in parallel
  - But order of statements can be important to clarity of thought!
  - When right-hand-side changes, it immediately flows through to left
  - Designated by the keyword `assign`
    ```
    wire c;
    assign c = a | b;
    wire c = a | b;   // same thing
    ```
Vectors of Wires

- Wire vectors:
  ```verilog
class wire [7:0] W1; // 8 bits, w1[7] is MSB
  Also called "buses"
```

- Operations
  - Bit select: W1[3]
  - Range select: W1[3:2]
  - Concatenate:
    ```verilog
    vec = {x, y, z};
    {carry, sum} = vec[0:1];
    example, swap high and low-order bytes of 16-bit vector
    wire [15:0] w1, w2;
    assign w2 = {w1[7:0], w1[15:8]}
    ```

Operators

- Operators similar to C or Java
- On wires:
  - & (and), | (or), ~ (not), ^ (xor)
- On vectors:
  - & (and), | (or), ~ (not), ^ (xor) (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)

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

Conditional Operator

- Verilog supports the ?: conditional operator
  - Just like in C
  - But much more common in Verilog

- Examples:
  ```verilog
  assign out = S ? B : A;
  ```

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

  What do these do?

Miscellaneous

- Operators and expressions can be used with modules
  ```verilog
  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)
  Another example: assign xyz = 8'hFF;
  ```

- C/Java style comments
  ```verilog
  // comment until end of line
  /* comment between markers */
  ```

- All variable names are case sensitive
  ```verilog
  But it is a bad idea to make names that differ only in case
  ```

- Constants:
  ```verilog
  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)
  Another example: assign xyz = 8'hFF;
  ```
Overview

- Why Verilog?
- High-level description of Verilog
- Details:
  - Primitives
  - Modules and instances
  - Wires and Nets
  - Operators
  - Sequential logic
  - Miscellaneous
    - Parameters, Pre-processor, case statements, Common errors, system tasks
- Case study

Sequential Logic in Verilog

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

```verilog
module dff (input Clock, D, Reset, output reg Q);
always @(posedge Clock)
begin
  if (Reset)
    Q = 1'b0;
  else
    Q = D;
end
endmodule
```

Designing Sequential Logic

- CS/ECE 552 design rule: separate combinational logic from sequential state elements in lowest-level modules
  - Not enforced by Verilog, but a very good idea
  - Possible exceptions: counters, shift registers
- We'll give you a 1-bit flip-flop module (see previous slide)
  - Edge-triggered, not a latch
  - Use it to build n-bit register, registers with "load" inputs, etc.
- Example use: state machine

Clocks Signals

- Clocks 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
    - Do not just "and" the write-enable signal with the clock!
- Messing with the clock can cause errors
  - Often can only be found using timing simulation
# Overview

- **Why Verilog?**
- **High-level description of Verilog**

## Details:
- Primitives
- Modules and instances
- Wires and Nets
- Operators
- Sequential logic

## Miscellaneous
- Parameters, Pre-processor, case statements, Common errors, system tasks
- Case study

---

# 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
  ```

## Verilog Pre-Processor

- Like the C pre-processor
  - But uses `(back-tick)` instead of `#`
  - Constants: `define`
  - No parameterized macros
  - Use `` before expanding constant macro
  ```
  define letter_A #`h41
  wire w = `letter_A;
  ```
  - Conditional compilation: `ifdef, `endif
  - File inclusion: `include`

## 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
- Avoid names such as:
  - clock, power, pwr, ground, gnd, vdd, vcc, init, reset
  - Some of these are "special" and will silently cause errors
  - We will use "clk" and "rst", but only for their intended uses
Repeated Signals

- Previously we discussed vector concatenation
  \[
  \text{assign } \text{vec} = \{x, y, z\};
  \]
- Can also repeat a signal \(n\) times
  \[
  \text{assign } \text{vec} = \{16\times\text{x}\}; \quad // \text{16 copies of } x
  \]
- Example uses (what does this do?):
  \[
  \text{wire}[7:0] \text{ out};
  \text{wire}[3:0] \text{ A};
  \text{assign } \text{out} = \{\{4\times0\}\}, \text{A}[3:0];
  \]
- What about this?
  \[
  \text{assign } \text{out} = \{\{4\times\text{A}[3]\}\}, \text{A}[3:0];
  \]

Non-binary Hardware Values

- A hardware signal can have four values
  \[
  0, 1, x, z
  \]
  - \(x\): don’t know, don’t care
  - \(z\): high-impedance (no current flowing)
- Two meanings of “\(x\)”
  - Simulator indicating an unknown state
  - Or: You telling 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 nothing (z)
  - Many tri-states drive the same wire, all but one must be “z”
  - Example: multiplexer
- Why Verilog allows multiple assignments to same wire.

Case Statements

- Useful to make big muxes
- Very useful for “next-state” logic
- But they are easy to abuse
- If you don’t set a value, it retains its previous state
- Which is a latch!
- We will allow case statements, but with some severe restrictions:
  - Every value is set in every case
  - Every possible combination of select inputs must be covered
  - Each case lives in its own “always” block, sensitive to changes in all of its input signals
  - This is our only use of “always” blocks

\[
\text{case } (\langle e x p r \rangle )
  \langle \text{match-constant1} \rangle : <\text{stmt}>
  \langle \text{match-constant2} \rangle : <\text{stmt}>
  \langle \text{match-constant3} \rangle , \langle \text{match-constant4} \rangle : <\text{stmt}>
  \text{default} : <\text{stmt}>
\text{endcase}
\]
Case Statement Example

```verilog
always @*
case ({goBack, currentState, inputA, inputB})
  6'b1_???_?_?  : begin out = 0; newState = 3'b000; err=0; end
  6'b0_000_0_? : begin out = 0; newState = 3'b000; err=0; end
  6'b0_000_1_? : begin out = 0; newState = 3'b001; err=0; end
  6'b0_001_1_? : begin out = 0; newState = 3'b001; err=0; end
  6'b0_001_0_0 : begin out = 0; newState = 3'b001; err=0; end
  6'b0_001_0_1 : begin out = 0; newState = 3'b001; err=0; end
  6'b0_010_?_0 : begin out = 0; newState = 3'b010; err=0; end
  6'b0_010_?_1 : begin out = 0; newState = 3'b011; err=0; end
  6'b0_011_?_1 : begin out = 0; newState = 3'b011; err=0; end
  6'b0_011_?_0 : begin out = 0; newState = 3'b100; err=0; end
  6'b0_100_?_? : begin out = 1; newState = 3'b000; err=0; end
  6'b0_101_?_? : begin out = 0; newState = 3'b000; err=1; end
  6'b0_110_?_? : begin out = 0; newState = 3'b000; err=1; end
  6'b0_111_?_? : begin out = 0; newState = 3'b000; err=1; end
default:              begin out = 0; newState = 3'b000; err=1; end
decase
```

What happens if it’s wrong?

Here are our rules:

- A case statement should always have a default
- Hitting this default is an error
- Every module has an "err" output
- Can be used for other checks, like illegal inputs
- OR together all "err" signals -- bring "err" all the way to top
- Our clock/reset module will print a message if err == 1

System tasks

- Start with $
- For output:
  - `$display(<fmtstring>,<signal>*);`
  - `$fdisplay(<fhandle>,<fmtstring>,<signal>*);`
  - Signal printf/fprintf
  - `$monitor(<fmtstring>,<signal>*);`
  - Non-procedural printf, prints out when a signal changes
  - `$dumpvars(<,signal>*);`
  - Similar to monitor
  - VCD format for waveform viewing (gtkwave)
  - Output is in dumpfile.vcd

More System Tasks

- `$time`
  - Simulator’s internal clock (64-bit unsigned)
  - Can be used as both integer and auto-formatted string
- `$finish`
  - Terminate simulation
- `$stop`
  - Pause simulation and debug
- `$readmemh(<fname>,<mem>,<start>,<end>);`
- `$writememh(<fname>,<mem>,<start>,<end>);`
  - Load contents of ASCII file to memory array (and vice versa)
  - Parameters `<start>,<end>` are optional
  - Useful for loading initial images, dumping final images
Overview

- Why Verilog?
- High-level description of Verilog
- Details:
  - Primitives
  - Modules and instances
  - Wires and Nets
  - Operators
  - Sequential logic
  - Miscellaneous
    - Parameters, Pre-processor, case statements, Common errors, system tasks
- Case study: Sequence Detector

Sequential Circuits

- A sequence detector is sequential logic
- Design Rule: separate combinational logic from sequential state elements in lowest-level modules
- We will give you a 1-bit flip-flop module to hold state and a clock/reset generator
  - See the course web site

Finite State Machine

- Let's revisit the HW1 solution
- Target Sequence: 10010110 (96)

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>in = 0</td>
<td>in = 0</td>
<td>0</td>
</tr>
<tr>
<td>S0</td>
<td>S1</td>
<td>0</td>
</tr>
<tr>
<td>S1</td>
<td>S2</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>S3</td>
<td>0</td>
</tr>
<tr>
<td>S3</td>
<td>S4</td>
<td>0</td>
</tr>
<tr>
<td>S4</td>
<td>S5</td>
<td>0</td>
</tr>
<tr>
<td>S5</td>
<td>S6</td>
<td>0</td>
</tr>
<tr>
<td>S6</td>
<td>S7</td>
<td>0</td>
</tr>
<tr>
<td>S7</td>
<td>S8</td>
<td>0</td>
</tr>
<tr>
<td>S8</td>
<td>S9</td>
<td>1</td>
</tr>
</tbody>
</table>

How do we represent this??

State Representation

- We'll encode the states as binary numbers
- Need three bits to represent

```verilog
module seq_logic (
    input [2:0] NextState,
    input clk,
    input rst,
    output [2:0] CurState );

dff bit0 ( .Clock( clk ),
                .D( NextState[0] ),
                .Reset( rst ),
                .Q( CurState[0] ));
dff bit1 ( .Clock( clk ),
                .D( NextState[1] ),
                .Reset( rst ),
                .Q( CurState[1] ));
dff bit2 ( .Clock( clk ),
                .Reset( rst ),
                .Q( CurState[2] ));
endmodule
```
Combinational Logic

- We'll do two implementations
- The first is based on the state equations:

\[
\begin{align*}
Y &= \overline{y_2} \cdot y_0 \cdot \overline{y_1} + y_1 \cdot \overline{y_0} \cdot \overline{y_1} + y_2 \\
\overline{Y} &= \overline{y_2} \cdot \overline{y_1} \cdot \overline{y_0} + \overline{y_1} \cdot \overline{y_0} \cdot \overline{y_2} \\
\overline{\overline{Y}} &= y_2 \cdot y_1 \cdot y_0
\end{align*}
\]

- For the second, we'll use a simple case statement

Combinational Logic - Implementation #1

```verilog
module comb_logic (
    input [2:0] y, //current state
    input in,
    input clk, rst,
    output [2:0] Y); // next state

endmodule
```

Combinational Logic - Implementation #2

```verilog
module comb_logic(
    input [2:0] y, //current state
    input in,
    input clk, rst,
    output [2:0] Y); // next state
output out);

reg [2:0] Y; // signals assigned in an always block must be registers
always @ *
case (y, in)
    4'b000_0: begin Y = 3'b000; out = 1'b0; end
    4'b000_1: begin Y = 3'b001; out = 1'b0; end
    4'b001_0: begin Y = 3'b010; out = 1'b0; end
    4'b001_1: begin Y = 3'b001; out = 1'b0; end
    4'b010_0: begin Y = 3'b011; out = 1'b0; end
    4'b010_1: begin Y = 3'b011; out = 1'b0; end
    4'b011_0: begin Y = 3'b100; out = 1'b0; end
    4'b011_1: begin Y = 3'b100; out = 1'b0; end
    4'b100_0: begin Y = 3'b100; out = 1'b0; end
    4'b100_1: begin Y = 3'b100; out = 1'b0; end
    4'b101_0: begin Y = 3'b101; out = 1'b0; end
    4'b101_1: begin Y = 3'b101; out = 1'b0; end
    4'b110_0: begin Y = 3'b110; out = 1'b0; end
    4'b110_1: begin Y = 3'b110; out = 1'b0; end
    4'b111_0: begin Y = 3'b110; out = 1'b0; end
    4'b111_1: begin Y = 3'b110; out = 1'b0; end
    default: begin Y = 3'b000; out = 1'b0; end
endcase
endmodule
```

Top-Level Design

- Top-level design contains ONLY modules

```verilog
module seq_detect(
    input in,
    input clk, rst,
    output out);

wire [2:0] CurState;
wire [2:0] NextState;

seq_logic (NextState, clk, rst, CurState); // always block
comb_logic (CurState, in, clk, rst, NextState, out); // always block
endmodule
```

```verilog
module seq_logic(
    input NextState, clk, rst, CurState);

comb_logic (CurState, in, clk, rst, NextState, out);
endmodule
```

```verilog
module comb_logic(
    input [2:0] y, //current state
    input in,
    input clk, rst,
    output [2:0] Y); // next state

endmodule
```
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"

Backup Slides:
Constructs we won’t use

Traditional Module Header

```verilog
module mux2to1_4(A, B, Sel, 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
```

Behavioral Statements

- Like in C, but use `begin-end` instead of `{}` to group

```verilog
if (expr) <stmt> else if <stmt>
for <stmt>:<expr>:<stmt> <stmt>
```

- Careful: No `++` operator in Verilog
Behavior Invocation: Always

always @( <sensitivity> or sensitivity >*)
begin
  <stmt>*
end

- Defines reaction of module to changes in input
- sensitivity list: signals or signal edges that trigger change
- Keyword or: disjunction of multiple sensitivity elements
- Multiple always sections are allowed
  - Careful: don’t know order in which signals arrive
  - Best to use one

Signal and Signal Edge Sensitivity

- Signal sensitivity: evaluate block on any signal change
  always @(CLK)
- Edge sensitivity: evaluate block on particular signal change
  always @(posedge CLK)

- Quiz: what’s the difference?
  always @(D or CLK) if (CLK) Q <= D;
  always @(posedge CLK) Q <= D;

Auxiliary Variables

- C style variables that are used procedurally
  - Understood to be “program helpers”, not pieces of hardware
  - integer i; // signed 32-bit (not int)
  - time t; // unsigned 64-bit
  - real r; // double precision FP
  - memory (i.e., C) like array syntax
  - integer iarray[63:0]; // array of 64 integers
  - E.g.,
    integer i;
    for (i = 0; i < N; i = i + 1)
      memory[i] <= 0;
  - E.g.,
    time sim_num_insn; // retired instructions

Behavior Modules: Tasks

module memory ()
  reg [7:0] memory [1023:0];
  integer i;
  task clear;
  begin
    for (i = 0; i < 1024; i = i + 1)
      memory[i] <= 8'b0;
  end
endtask
endmodule

- Tasks: module “methods”
  - Can be invoked from always and initial blocks
An Example Test Module

```verilog
#include "mux.v"
module main;
  reg [3:0] A, B;
  wire [3:0] O;
  reg S;
  mux2to1_4 mux (S, A, B, O);
  initial
    begin
      $monitor ($time, "S=%b, A=%d, B=%d, O=%d", S, A, B, O);
      $dumpvars(1, S, A, B, O);
      #5 A=4'b1010; B=4'b0010; S=1'b0;
      #5 S=1'b1;
      #5 $finish;
    end
endmodule
```

A Test Module With Clock

```verilog
#include "fsm.v"
module main;
  reg clk, in;
  wire out;
  fsm fsm1 (clk, in, out);
  always #5 clk <= ~clk;
  initial
    begin
      clk = 0;
      $monitor ($time, "CLK=%b", clk, fsm1.state);
      $dumpvars(1, clk, fsm1.state);
      #100 $finish;
    end
endmodule
```