EECS 150 - Components and Design Techniques for Digital Systems

Lec 04 – Hardware Description Languages / Verilog

9/6/2007

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

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

Review

- Advancing technology changes the trade-offs and design techniques
  - 2x transistors per chip every 18 months
- ASIC, Programmable Logic, Microprocessor
- Programmable logic invests chip real-estate to reduce design time & time to market
  - Canonical Forms, Logic Minimization, PLAs, →
- FPGA:
  - programmable interconnect,
  - configurable logic blocks
    - » LUT + storage
  - Block RAM
  - IO Blocks

Outline

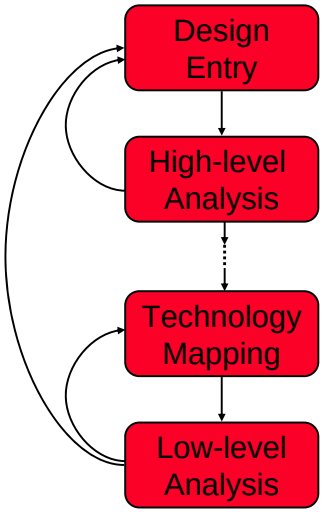
- Netlists
- Design flow
- What is a HDL?
- Verilog
- Announcements
- Structural models
- Behavioral models
- Elements of the language
- Lots of examples

Remember: to design is to represent

- How do we represent digital designs?
- Components
  - Logic symbol, truth table
  - Storage symbol, timing diagram
- Connections
  - Schematics

Human readable or machine readable???

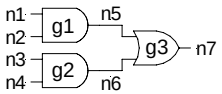
Design Flow



Netlist

- A key data structure (or representation) in the design process is the "netlist":
  - Network List
- A netlist lists components and connects them with nodes:

ex:



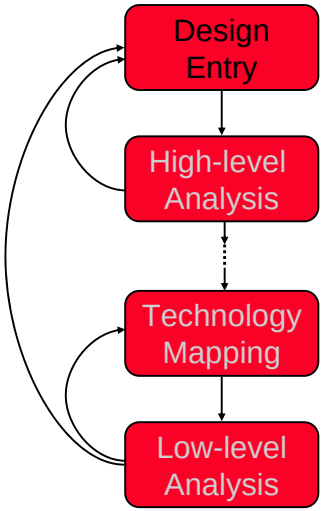

```
g1 "and" n1 n2 n5
g2 "and" n3 n4 n6
g3 "or" n5 n6 n7
```

Alternative format:

```
n1 g1.in1
n2 g1.in2
n3 g2.in1
n4 g2.in2
n5 g1.out g3.in1
n6 g2.out g3.in2
n7 g3.out
g1 "and"
g2 "and"
g3 "or"
```

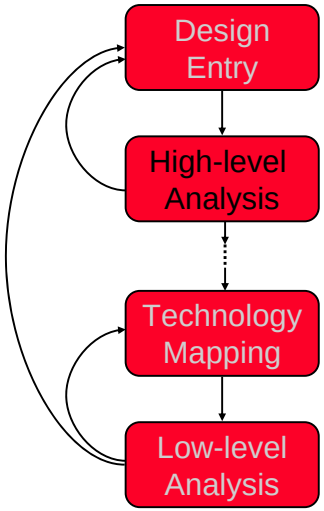
- Netlist is needed for simulation and implementation.
- Could be at the transistor level, gate level, ...
- Could be hierarchical or flat.
- How do we generate a netlist?

Design Flow



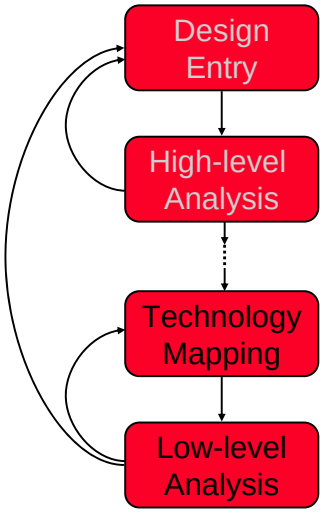
- Circuit is described and represented:
  - Graphically (Schematics)
  - Textually (HDL)
- Result of circuit specification (and compilation) is a netlist of:
  - generic primitives - logic gates, flip-flops, or
  - technology specific primitives - LUTs/CLBs, transistors, discrete gates, or
  - higher level library elements - adders, ALUs, register files, decoders, etc.

Design Flow



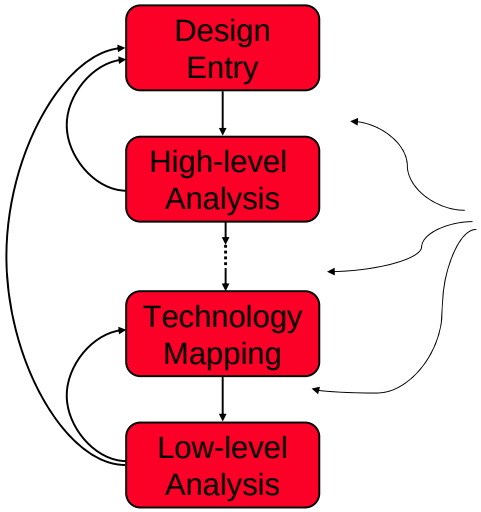
- High-level Analysis is used to verify:
  - correct function
  - rough:
    - » timing
    - » power
    - » cost
- Common tools used are:
  - simulator - check functional correctness, and
  - static timing analyzer
    - » estimates circuit delays based on timing model and delay parameters for library elements (or primitives).

Design Flow



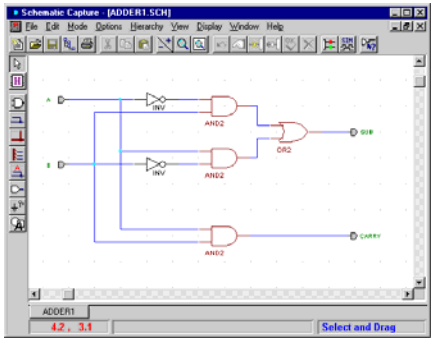
- **Technology Mapping:**
  - Converts netlist to implementation technology dependent details
    - » Expands library elements,
    - » performs:
      - partitioning,
      - placement,
      - routing
- **Low-level Analysis**
  - Simulation and Analysis Tools perform low-level checks with:
    - » accurate timing models,
    - » wire delay
  - For FPGAs this step could also use the actual device.

Design Flow



Design Entry

- Schematic entry/editing used to be the standard method in industry
- Used in EECS150 until recently
- ☺ Schematics are intuitive. They match our use of gate-level or block diagrams.
- ☺ Somewhat physical. They imply a physical implementation.
- ☹ Require a special tool (editor).
- ☹ Unless hierarchy is carefully designed, schematics can be confusing and difficult to follow.



- Hardware Description Languages (HDLs) are the new standard
  - except for PC board design, where schematics are still used.

HDLs

- Basic Idea:
  - Language constructs describe circuits with two basic forms:
  - *Structural descriptions* similar to hierarchical netlist.
  - *Behavioral descriptions* use higher-level constructs (similar to conventional programming).
- Originally designed to help in abstraction and simulation.
  - Now "logic synthesis" tools exist to automatically convert from behavioral descriptions to gate netlist.
  - Greatly improves designer productivity.
  - However, this may lead you to falsely believe that hardware design can be reduced to writing programs!

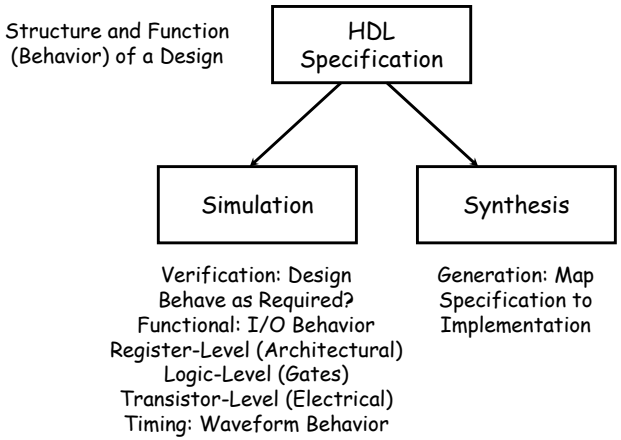
"Structural" example:

```
Decoder(output x0,x1,x2,x3;
                inputs a,b)
{
  wire abar, bbar;
  inv(bbar, b);
  inv(abar, a);
  nand(x0, abar, bbar);
  nand(x1, abar, b   );
  nand(x2, a,    bbar);
  nand(x3, a,    b   );
}
```

"Behavioral" example:

```
Decoder(output x0,x1,x2,x3;
                inputs a,b)
{
  case [a b]
    00: [x0 x1 x2 x3] = 0x0;
    01: [x0 x1 x2 x3] = 0x2;
    10: [x0 x1 x2 x3] = 0x4;
    11: [x0 x1 x2 x3] = 0x8;
  endcase;
}
```

Design Methodology



Quick History of HDLs

- ISP (circa 1977) - research project at CMU
  - Simulation, but no synthesis
- Abel (circa 1983) - developed by Data-I/O
  - Targeted to programmable logic devices
  - Not good for much more than state machines
- Verilog (circa 1985) - developed by Gateway (now Cadence)
  - Similar to Pascal and C, originally developed for simulation
  - Fairly efficient and easy to write
  - 80s Berkeley develops synthesis tools
  - IEEE standard
- VHDL (circa 1987) - DoD sponsored standard
  - Similar to Ada (emphasis on re-use and maintainability)
  - Simulation semantics visible
  - Very general but verbose
  - IEEE standard

Verilog

- Supports structural and behavioral descriptions
- Structural
  - Explicit structure of the circuit
  - How a module is composed as an interconnection of more primitive modules/components
  - E.g., each logic gate instantiated and connected to others
- Behavioral
  - Program describes input/output behavior of circuit
  - Many structural implementations could have same behavior
  - E.g., different implementations of one Boolean function

Verilog Introduction

- the `module` describes a component in the circuit
- Two ways to describe:
  - Structural Verilog
    - » List of components and how they are connected
    - » Just like schematics, but using text
      - A net list
    - » tedious to write, hard to decode
    - » Essential without integrated design tools
  - Behavioral Verilog
    - » Describe *what* a component does, not *how* it does it
    - » Synthesized into a circuit that has this behavior
    - » Result is only as good as the tools
- Build up a hierarchy of modules

Structural Model - XOR

```
module xor_gate ( out, a, b );
  input     a, b;
  output    out;
  wire      abar, bbar, t1, t2;

  inverter invA (abar, a);
  inverter invB (bbar, b);
  and_gate and1 (t1, a, bbar);
  and_gate and2 (t2, b, abar);
  or_gate  or1 (out, t1, t2);

endmodule
```

module name

port list

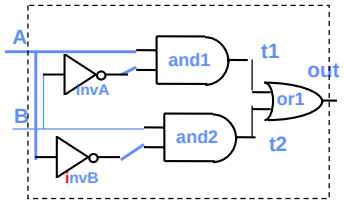
declarations

Built-in gates

statements

interconnections

Instance name



- Composition of primitive gates to form more complex module
- Note use of `wire` declaration!

By default, identifiers are wires

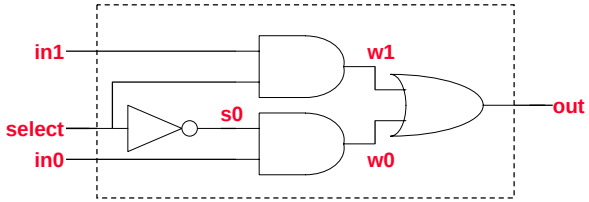
Structural Model: 2-to1 mux

```
//2-input multiplexor in gates
module mux2 (in0, in1, select, out);
   input in0,in1,select;
   output out;
   wire s0,w0,w1;

   not (s0, select);
   and (w0, s0, in0),
       (w1, select, in1);
   or  (out, w0, w1);

endmodule // mux2
```

- **Notes:**
  - comments
  - "module"
  - port list
  - declarations
  - wire type
  - primitive gates
  - Instance names?
  - List per type



Simple Behavioral Model

- **Combinational logic**
  - **Describe output as a function of inputs**
  - **Note use of `assign` keyword: *continuous* assignment**

```
module and_gate (out, in1, in2);
  input         in1, in2;
  output        out;

  assign out = in1 & in2;

endmodule
```

Output port of a *primitive* must be first in the list of ports

Restriction does not apply to *modules* in general

When is this "evaluated"?

2-to-1 mux behavioral description

```
// Behavioral model of 2-to-1
//  multiplexor.
module mux2 (in0,in1,select,out);
   input in0,in1,select;
   output out;
   //
   reg out;
   always @ (in0 or in1 or select)
     if (select) out=in1;
     else out=in0;
endmodule // mux2
```

- Notes:
  - behavioral descriptions using keyword **`always`** followed by blocking *procedural* assignments
  - Target output of procedural assignments must of of type `reg` **(not a real register)**
  - Unlike `wire` types where the target output of an assignment may be continuously updated, a `reg` type retains it value until a new value is assigned (the assigning statement is executed).
  - Optional initial statement

Sensitivity list

Behavioral 4-to1 mux

```
//Does not assume that we have
// defined a 2-input mux.

//4-input mux behavioral description
module mux4 (in0, in1, in2, in3, select, out);
   input in0,in1,in2,in3;
   input [1:0] select;
   output      out;
   reg         out;

  always @ (in0 in1 in2 in3 select)
      case (select)
            2'b00: out=in0;
            2'b01: out=in1;
            2'b10: out=in2;
            2'b11: out=in3;
      endcase
endmodule // mux4
```

- Notes:
  - No instantiation
  - Case construct equivalent to nested if constructs.
  - *Definition:* A structural description is one where the function of the module is defined by the instantiation and interconnection of sub-modules.
  - A behavioral description uses higher level language constructs and operators.
  - Verilog allows modules to mix both behavioral constructs and sub-module instantiation.

Mixed Structural/Behavioral Model

- Example 4-bit ripple adder

```
module full_addr (S, Cout, A, B, Cin );
  input     A, B, Cin;
  output    S, Cout;

  assign {Cout, S} = A + B + Cin;
endmodule

module adder4 (S, Cout, A, B, Cin);
  input  [3:0] A, B;
  input        Cin;
  output [3:0] S;
  output       Cout;
  wire         C1, C2, C3;

  full_addr fa0 (S[0], C1,  A[0], B[0], Cin);
  full_addr fa1 (S[1], C2,  A[1], B[1], C1);
  full_addr fa2 (S[2], C3,  A[2], B[2], C2);
  full_addr fa3 (S[3], Cout, A[3], B[3], C3);
endmodule
```

Behavior

Structural

Order of ports?

Announcements

Office hours will be posted on schedule.php

Homework 1 due tomorrow (2 pm outside 125)

Homework 2 out today

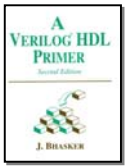
Feedback on labs, Lab lectures

Reading:

- these notes
- verilog code you see in lab

Verilog Help

- The lecture notes only cover the basics of Verilog and mostly the conceptual issues.
  - Lab Lectures have more detail focused on lab material
- Textbook has examples.
- Bhasker book is a good tutorial.
- http://www.doe.carleton.ca/~shams/97350/PetervrlK.pdf pretty good
- The complete language specification from the IEEE is available on the class website under "Refs/Links"
- http://toolbox.xilinx.com/docsan/xilinx4/data/docs/xst/verilog2.html



Verilog Data Types and Values

- **Bits - value on a wire**
  - `0, 1`
  - `X` **- don't care/don't know**
  - `Z` **- undriven, tri-state**
- **Vectors of bits**
  - `A[3:0]` - vector of 4 bits: `A[3], A[2], A[1], A[0]`
  - **Treated as an *unsigned* integer value**
    - » **e.g. ,** `A < 0` **??**
  - **Concatenating bits/vectors into a vector**
    - » **e.g., sign extend**
    - » `B[7:0] = {A[3], A[3], A[3], A[3], A[3:0]};`
    - » `B[7:0] = {3{A[3]}, A[3:0]};`
  - **Style: Use** `a[7:0] = b[7:0] + c;`
    Not: `a = b + c;` **// need to look at declaration**

Verilog Numbers

- `14` **- ordinary decimal number**
- `-14` **- 2's complement representation**
- `12'b0000_0100_0110` **- binary number with 12 bits (_ is ignored)**
- `12'h046` **- hexadecimal number with 12 bits**
- **Verilog values are *unsigned***
  - **e.g.,** `C[4:0] = A[3:0] + B[3:0];`
  - **if A = 0110 (6) and B = 1010(-6)**
    C = 10000 not 00000
    i.e., B is zero-padded, not sign-extended

Verilog Operators

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
! ~ & | ~& ~| ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	Logical Bit-wise Reduction Reduction Reduction Reduction Reduction Reduction
+ -	unary (sign) plus unary (sign) minus	Arithmetic Arithmetic
{ }	concatenation	Concatenation
{{ }}	replication	Replication
* / %	multiply divide modulus	Arithmetic Arithmetic Arithmetic
+ -	binary plus binary minus	Arithmetic Arithmetic
<< >>	shift left shift right	Shift Shift

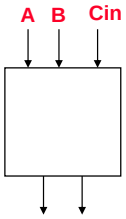
> >= < <=	greater than greater than or equal to less than less than or equal to	Relational Relational Relational Relational
== !=	logical equality logical inequality	Equality Equality
=== !==	case equality case inequality	Equality Equality
&	bit-wise AND	Bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	Bit-wise Bit-wise
|	bit-wise OR	Bit-wise
&&	logical AND	Logical
||	logical OR	Logical
?:	conditional	Conditional

Verilog Variables

- **wire**
  - **Variable used simply to connect components together**
- **reg**
  - **Variable that saves a value as part of a behavioral description**
  - **Usually corresponds to a wire in the circuit**
  - **Is *NOT* necessarily a register in the circuit**
- **usage:**
  - **Don't confuse reg assignments with the combinational continuous** `assign` **statement! (more soon)**
  - **Reg should only be used with a** `always` **blocks (sequential logic, to be presented …)**

Verilog Module

- Corresponds to a circuit component
  - "Parameter list" is the list of external connections, aka "ports"
  - Ports are declared "input", "output" or "inout"
    - » inout ports used on tri-state buses
  - Port declarations imply that the variables are wires



module name

ports

```
module full_addr (A, B, Cin, S, Cout);
  input     A, B, Cin;
  output    S, Cout;

  assign  {Cout, S} = A + B + Cin;
endmodule
```

inputs/outputs

Verilog Continuous Assignment

- Assignment is continuously evaluated
- `assign` corresponds to a connection or a simple component with the described function
- Target is *NEVER* a reg variable
- Dataflow style

```
assign A = X | (Y & ~Z);
assign B[3:0] = 4'b01XX;
assign C[15:0] = 4'h00ff;
assign #3 {Cout, S[3:0]} = A[3:0] + B[3:0] + Cin;
```

use of Boolean operators (~ for bit-wise, ! for logical negation)

bits can take on four values (0, 1, X, Z)

variables can be n-bits wide (MSB:LSB)

use of arithmetic operator

multiple assignment (concatenation)

delay of performing computation, only used by simulator, not synthesis

Comparator Example

```
module Compare1 (A, B, Equal, Alarger, Blarger);
  input     A, B;
  output    Equal, Alarger, Blarger;

  assign Equal = (A & B) | (~A & ~B);
  assign Alarger = (A & ~B);
  assign Blarger = (~A & B);
endmodule
```

When evaluated?

What is synthesized?

Comparator Example

```
// Make a 4-bit comparator from 4 1-bit comparators

module Compare4(A4, B4, Equal, Alarger, Blarger);
  input [3:0] A4, B4;
  output Equal, Alarger, Blarger;
  wire e0, e1, e2, e3, Al0, Al1, Al2, Al3, B10, Bl1, Bl2, Bl3;

  Compare1 cp0(A4[0], B4[0], e0, Al0, Bl0);
  Compare1 cp1(A4[1], B4[1], e1, Al1, Bl1);
  Compare1 cp2(A4[2], B4[2], e2, Al2, Bl2);
  Compare1 cp3(A4[3], B4[3], e3, Al3, Bl3);

  assign Equal = (e0 & e1 & e2 & e3);
  assign Alarger = (Al3 | (Al2 & e3) |
                    (Al1 & e3 & e2) |
                    (Al0 & e3 & e2 & e1));
  assign Blarger = (~Alarger & ~Equal);
endmodule
```

What would be a "better" behavioral version?

Simple Behavioral Model - the `always` block

- `always` block
  - Always waiting for a change to a trigger signal
  - Then executes the body

```
module and_gate (out, in1, in2);
  input   in1, in2;
  output  out;
  reg     out;

  always @(in1 or in2) begin
     out = in1 & in2;
  end
endmodule
```

Not a real register!!
A Verilog register
Needed because of
assignment in always
block

Specifies when block is executed
I.e., triggered by which signals

`always` Block

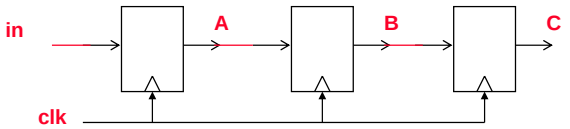
- A procedure that describes the function of a circuit
  - Can contain many statements including `if`, `for`, `while`, `case`
  - Statements in the `always` block are executed *sequentially*
    - » *"blocking" assignment*
    - » Continuous assignments <= are executed in *parallel*
      - Non-blocking
  - The entire block is executed 'at once'
    - » But the meaning is established by sequential interpretation
      - Simulation micro time vs macro time
      - synthesis
  - The *final* result describes the function of the circuit for current set of inputs
    - » intermediate assignments don't matter, only the final result
- `begin`/`end` used to group statements

What Verilog generates storage elements?

- Expressions produce combinational logic
  - Map inputs to outputs
- Storage elements carries same values forward in time

State Example

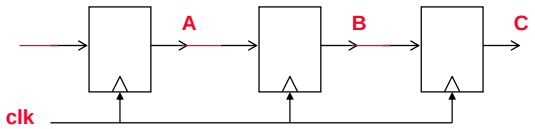
```
module shifter (in, A,B,C,clk);
  input in, clk;
  input A,B,C;
  reg A, B, C;
  always @ (posedge clk) begin
    C = B;
    B = A;
    A = in;
  end
endmodule
```



- Block interpreted sequentially, but action happens "at once"

State Example2 – Non blocking

```
module shifter (in, A,B,C,clk);
  input in, clk;
  input A,B,C;
  reg A, B, C;
  always @ (posedge clk) begin
    B <= A;
    A <= in;
    C <= B;
  end
endmodule
```

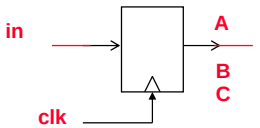


- Non-blocking: all statements interpreted in parallel
  - Everything on the RHS evaluated,
  - Then all assignments performed

State Example2 – interactive quiz

```
module shifter (in, A,B,C,clk);
  input in, clk;
  input A,B,C;
  reg A, B, C;
  always @ (posedge clk) begin
    A = in;
    B = A;
    C = B;
  end
endmodule
```

- Variable becomes a storage element if its value is preserved (carried forward in time) despite changes in variables the produce it.
- Not whether it is declared as a `wire` or a `reg`!



"Complete" Assignments

- If an `always` block executes, and a variable is *not* assigned
  - Variable keeps its old value (think implicit state!)
  - *NOT* combinational logic ⇒ latch is inserted (implied memory)
  - This is usually *not* what you want: dangerous for the novice!
- Any variable assigned in an `always` block should be assigned for any (and every!) execution of the block.

Incomplete Triggers

- Leaving out an input trigger usually results in a sequential circuit
- Example: The output of this "and" gate depends on the input history

```
module and_gate (out, in1, in2);
  input      in1, in2;
  output     out;
  reg        out;

  always @(in1) begin
     out = in1 & in2;
  end

endmodule
```

What Hardware would this generate?

Behavioral with Bit Vectors

```
//Behavioral model of 32-bitwide 2-to-1 multiplexor.
module mux32 (in0,in1,select,out);
   input [31:0] in0,in1;
   input      select;
   output [31:0] out;
   //
   reg [31:0] out;
   always @ (in0 or in1 or select)
     if (select) out=in1;
     else out=in0;
endmodule // Mux

//Behavioral model of 32-bit adder.
module add32 (S,A,B);
   input [31:0] A,B;
   output [31:0] S;
   reg [31:0] S;
   always @ (A or B)
     S = A + B;
endmodule // Add
```

- **Notes:**
  - **inputs, outputs 32-bits wide**

Hierarchy & Bit Vectors

```
//Assuming we have already
// defined a 2-input mux (either
// structurally or behaviorally,

//4-input mux built from 3 2-input muxes
module mux4 (in0, in1, in2, in3, select, out);
   input in0,in1,in2,in3;
   input [1:0] select;
   output      out;
   wire        w0,w1;

   mux2
     m0 (.select(select[0]), .in0(in0), .in1(in1), .out(w0)),
     m1 (.select(select[0]), .in0(in2), .in1(in3), .out(w1)),
     m3 (.select(select[1]), .in0(w0), .in1(w1), .out(out));
endmodule // mux4
```

Which select?

- Notes:
  - instantiation similar to primitives
  - select is 2-bits wide
  - *named* port assignment

Verilog `if`

- **Same syntax as C if statement**
- **Sequential meaning, action "at once"**

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;    // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;              // target of assignment

  always @(sel or A or B or C or D)
    if (sel == 2'b00) Y = A;
    else if (sel == 2'b01) Y = B;
    else if (sel == 2'b10) Y = C;
    else if (sel == 2'b11) Y = D;

endmodule
```


Verilog `if`

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;    // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;              // target of assignment

  always @(sel or A or B or C or D)
    if (sel[0] == 0)
      if (sel[1] == 0) Y = A;
      else             Y = B;
    else
      if (sel[1] == 0) Y = C;
      else             Y = D;
endmodule
```

Verilog case

- **Sequential execution of cases**
  - Only first case that matches is executed (no break)
  - Default case can be used

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel;      // 2-bit control signal
input A, B, C, D;
output Y;
reg Y;                // target of assignment

  always @(sel or A or B or C or D)
    case (sel)
      2'b00: Y = A;
      2'b01: Y = B;
      2'b10: Y = C;
      2'b11: Y = D;
    endcase
endmodule
```

Conditions tested in top to bottom order

Verilog case

- **Without the default case, this example would create a latch for Y!**
  - your generating hardware, not programming
- **Assigning X to a variable means synthesis is free to assign any value**

```
// Simple binary encoder (input is 1-hot)
module encode (A, Y);
input  [7:0] A;          // 8-bit input vector
output [2:0] Y;          // 3-bit encoded output
reg    [2:0] Y;          // target of assignment

  always @(A)
    case (A)
      8'b00000001: Y = 0;
      8'b00000010: Y = 1;
      8'b00000100: Y = 2;
      8'b00001000: Y = 3;
      8'b00010000: Y = 4;
      8'b00100000: Y = 5;
      8'b01000000: Y = 6;
      8'b10000000: Y = 7;
      default:     Y = 3'bX;      // Don't care when input is not 1-hot
    endcase
endmodule
```


Verilog case (cont)

- **Cases are executed sequentially**
  - The following implements a *priority* encoder

```
// Priority encoder
module encode (A, Y);
input  [7:0] A;          // 8-bit input vector
output [2:0] Y;          // 3-bit encoded output
reg    [2:0] Y;          // target of assignment

  always @(A)
    case (1'b1)
      A[0]:    Y = 0;
      A[1]:    Y = 1;
      A[2]:    Y = 2;
      A[3]:    Y = 3;
      A[4]:    Y = 4;
      A[5]:    Y = 5;
      A[6]:    Y = 6;
      A[7]:    Y = 7;
      default: Y = 3'bX;  // Don't care when input is all 0's
    endcase
endmodule
```


Parallel Case

- **A priority encoder is more expensive than a simple encoder**
  - If we know the input is 1-hot, we can tell the synthesis tools
  - "parallel-case" pragma says the order of cases does not matter

```
// simple encoder
module encode (A, Y);
input  [7:0] A;          // 8-bit input vector
output [2:0] Y;          // 3-bit encoded output
reg    [2:0] Y;          // target of assignment

  always @(A)
    case (1'b1)          // synthesis parallel-case
      A[0]:    Y = 0;
      A[1]:    Y = 1;
      A[2]:    Y = 2;
      A[3]:    Y = 3;
      A[4]:    Y = 4;
      A[5]:    Y = 5;
      A[6]:    Y = 6;
      A[7]:    Y = 7;
      default: Y = 3'bX;  // Don't care when input is all 0's
    endcase
endmodule
```

Verilog casex

- Like case, but cases can include 'X'
  - X bits not used when evaluating the cases
  - In other words, you don't care about those bits!

casex Example

```
// Priority encoder
module encode (A, valid, Y);
input  [7:0] A;              // 8-bit input vector
output [2:0] Y;              // 3-bit encoded output
output valid;                // Asserted when an input is not all 0's
reg    [2:0] Y;              // target of assignment
reg    valid;

  always @(A) begin
    valid = 1;
    casex (A)
      8'bXXXXXXX1: Y = 0;
      8'bXXXXXX10: Y = 1;
      8'bXXXXX100: Y = 2;
      8'bXXXX1000: Y = 3;
      8'bXXX10000: Y = 4;
      8'bXX100000: Y = 5;
      8'bX1000000: Y = 6;
      8'b10000000: Y = 7;
      default:  begin
         valid = 0;
         Y = 3'bX;  // Don't care when input is all 0's
      end
    endcase
  end
endmodule
```


Verilog `for`

- `for` is similar to C
- `for` statement is executed at compile time (like macro expansion)
  - Result is all that matters, not how result is calculated
  - Use in testbenches only!

```
// simple encoder
module encode (A, Y);
input  [7:0] A;           // 8-bit input vector
output [2:0] Y;           // 3-bit encoded output
reg    [2:0] Y;           // target of assignment

integer i;                // Temporary variables for program only
reg [7:0] test;

  always @(A) begin
    test = 8b'00000001;
    Y = 3'bX;
    for (i = 0; i < 8; i = i + 1) begin
       if (A == test) Y = N;
       test = test << 1;
    end
  end
endmodule
```


Another Behavioral Example

- Computing Conway's Game of Life rule
  - Cell with no neighbors or 4 neighbors dies; with 2-3 neighbors lives

```
module life (neighbors, self, out);
  input         self;
  input [7:0]   neighbors;
  output        out;
  reg           out;
  integer       count;
  integer       i;

  always @(neighbors or self) begin
    count = 0;
    for (i = 0; i<8; i = i+1) count = count + neighbors[i];
    out = 0;
    out = out | (count == 3);
    out = out | ((self == 1) & (count == 2));
  end
endmodule
```

integers are temporary compiler variables

always block is executed instantaneously,
if there are no delays only the final result is used

Verilog while/repeat/forever

- `while` (expression) statement
  - Execute statement while expression is true
- `repeat` (expression) statement
  - Execute statement a fixed number of times
- `forever` statement
  - Execute statement forever

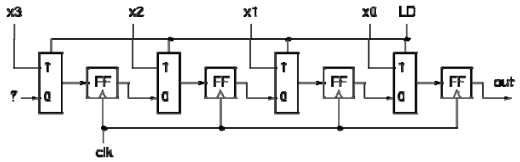
full-case and parallel-case

- `// synthesis parallel_case`
  - Tells compiler that ordering of cases is not important
  - That is, cases do not overlap
    - » e. g., state machine - can't be in multiple states
  - Gives cheaper implementation
- `// synthesis full_case`
  - Tells compiler that cases left out can be treated as don't cares
  - Avoids incomplete specification and resulting latches

Sequential Logic

```
//Parallel to Serial converter
module ParToSer(LD, X, out, CLK);
  input [3:0] X;
  input LD, CLK;
  output out;
  reg out;
  reg [3:0] Q;
  assign out = Q[0];
  always @ (posedge CLK)
       if (LD) Q=X;
       else Q = Q>>1;
endmodule // mux2

module FF (CLK,Q,D);
   input D, CLK;
   output Q; reg Q;
   always @ (posedge CLK) Q=D;
endmodule // FF
```



- Notes:
  - "always @ (posedge CLK)" forces Q register to be rewritten every simulation cycle.
  - ">>" operator does right shift (shifts in a zero on the left).
  - Shifts on non-reg variables can be done with concatenation:

```
wire [3:0] A, B;
assign B = {1'b0, A[3:1]}
```


Testbench

Top-level modules written specifically to test sub-modules.

Generally no ports.

```
module testmux;
  reg a, b, s;
  wire f;
  reg expected;

  mux2 myMux (.select(s), .in0(a), .in1(b), .out(f));

  initial
       begin
            s=0; a=0; b=1; expected=0;
          #10 a=1; b=0; expected=1;
          #10 s=1; a=0; b=1; expected=1;
       end
   initial
       $monitor(
           "select=%b in0=%b in1=%b out=%b, expected out=%b time=%d",
            s, a, b, f, expected, $time);
endmodule // testmux
```

- Notes:
  - initial block similar to always except only executes once (at beginning of simulation)
  - #n's needed to advance time
  - $monitor - prints output
  - A variety of other "system functions", similar to `monitor` exist for displaying output and controlling the simulation.

Final thoughts

- **Verilog looks like C, but it describes hardware**
  - **Multiple physical elements, Parallel activities**
  - **Temporal relationships**
  - **Basis for simulation and synthesis**
  - **figure out the circuit you want, then figure out how to express it in Verilog**
- **Understand the elements of the language**
  - **Modules, ports, wires, reg, primitive, continuous assignment, blocking statements, sensitivity lists, hierarchy**
  - **Best done through experience**
- **Behavioral constructs hide a lot of the circuit details but you as the designer must still manage the structure, data-communication, parallelism, and timing of your design.**