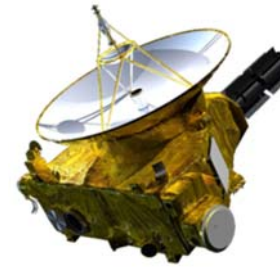# Intro to Verilog

- Circuits in the real world
- Verilog
  -- structural: modules, instances
  -- dataflow: continuous assignment
  -- sequential behavior: always blocks
  -- other useful features

Handouts
- lecture slides

Reminder: Lab 1 Checkoff Thu (A-M), Fr (N-Z)

New Horizon Probe

- Transmitter power 12 watts
- Transit time to earth 4.5 hours from Pluto
- Received signal strength ~ 10^{-19} watts!
- Reception possible because of *interleaving, forward error correction* and super low noise amplifiers.

https://upload.wikimedia.org/wikipedia/commons/thumb/4/4f/New_Horizons_Transparent.png/257px-New_Horizons_Transparent.png
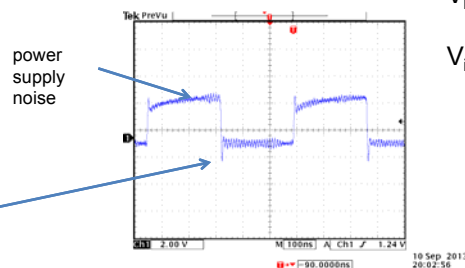

Wires Theory vs Reality

Integrated circuit
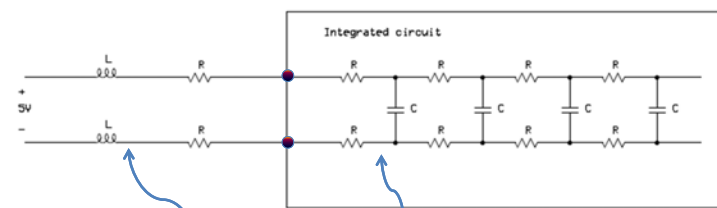
30-50mv voltage drop in chip

$V_{ih} > 2.0$

$V_{il} < 0.8$

Wires have inductance and resistance

$L \frac{di}{dt}$ noise during transitions

Voltage drop across wires

LC ringing after transitions

power supply noise

Bypass (Decoupling) Capacitors

Electrolytic Capacitor 10uf

Bypass capacitor 0.1uf typical

- Provides additional filtering from main power supply
- Used as local energy source – provides peak current during transitions
- Provided decoupling of noise spikes during transitions
- Placed as close to the IC as possible.
- Use small capacitors for high frequency response.
- Use large capacitors to localize bulk energy storage

Through hole PCB (ancient) shown for clarity.

Nexys4 Power System
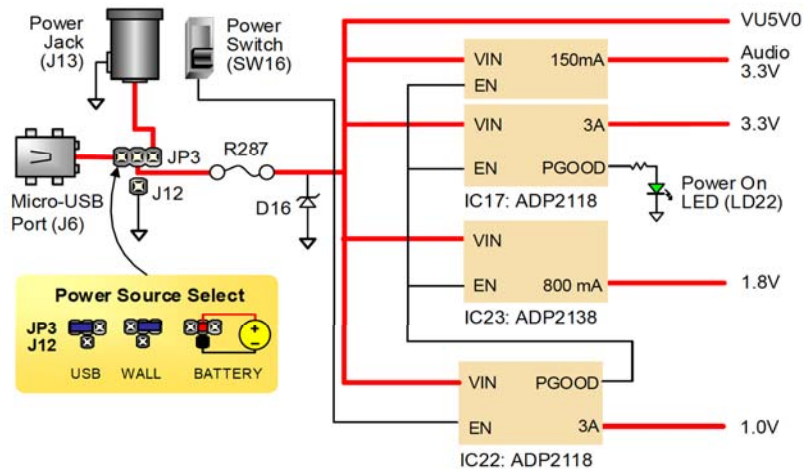
ADP: Analog Devices

Input Output Design

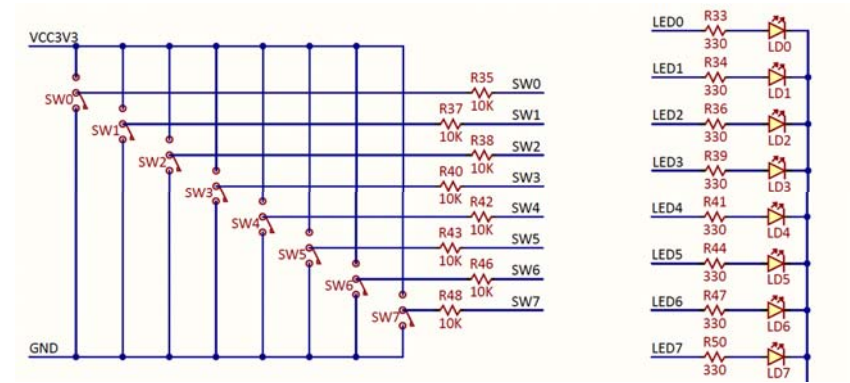Floating input is undefined

Limit output current for LED

Nexys I/O PMOD Limitations
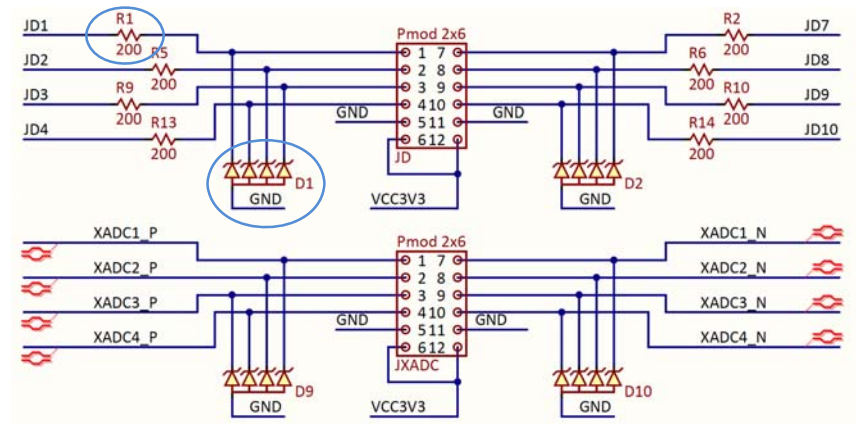
- Max current: 1 amp/PMOD, constrained by
  - USB source (~0.5-1amp)
  - External power supply
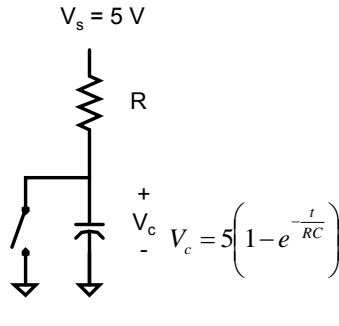 - 3.3V regulator (3 amp)

- Max frequency: <50Mhz

PMOD Port Protection

R: Short circuit protection   D: Over voltage protection

RC Equation

$V_s = 5\ V$

$V_s = 5\ V$

R

$+$
V_c
$-$

$V_c = 5\left(1 - e^{-\frac{t}{RC}}\right)$

Is RC in units of time?

Switch is closed t<0

Switch opens t>0

$V_s = V_R + V_C$

$V_s = i_R R + V_c \quad i_R = C\frac{dV_c}{dt}$

$V_s = RC\frac{dV_c}{dt} + V_c$

$V_c = V_s\left(1 - e^{-\frac{t}{RC}}\right)$

$$e^x \;=\; 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

$$=\; \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Diodes
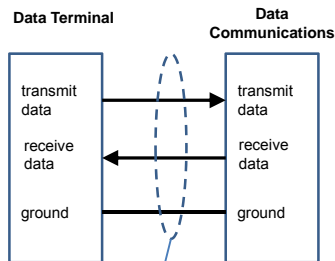
- Reversed diodes have capacitances ~ 200pf
- Reversed biased diodes are used as varactors aka varicap diode, tuning diode, variable capacitor diode ..
- Depletion layer (spacing) increases with reversed voltage; capacitance decreases

e^{-1} = 0.37 $\qquad V_c = 5\left(1 - e^{-\frac{t}{RC}}\right)$

- R=200Ω, C=200pf   RC=20ns   Max F << 50Mhz

RS-232 - Serial Data Transfer

- Standard developed in 1970
- Clear definition of terminal and data communications (modem)
- Data is sent over a single wire one bit at a time at a fixed data rate.
- Data rates can range from 300 bits per second (bps) to megabits/second.
- First modem was 300 bps.

Data Terminal

transmit data

receive data

ground

Data Communications

transmit data

receive data

ground

Straight through cable

Is RX input or output?

RX is input

Data Terminal

transmit data

receive data

ground

Data Terminal

transmit data

receive data

ground
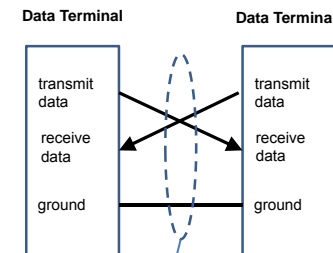
Crossover "null modem" cable

Synthesis – Really?

led[4]: Logical NAND of sw[1] and sw[2]

assign led[4] = !(sw[1] & sw[2]);

SystemVerilog *logic* values

Since we're describing hardware, we'll need to represent the values that can appear on wires. SystemVerilog uses a 4-valued logic:

When using a tri-state bus, we'll need to represent the values that can appear on bus and need to use Verilog with a 4-valued logic:

Value	Meaning
0	Logic zero, "low"
1	Logic one, "high"
Z or ?	High impedance (tri-state buses)
X	Unknown value (simulation)

"X" is used by simulators when a wire hasn't been initialized to a known value or when the predicted value is an illegitimate logic value (e.g., due to contention on a tri-state bus).

Numeric Constants

Constant values can be specified with a specific width and radix:

```
123          // default: decimal radix, unspecified width
'd123        // 'd = decimal radix
'h7B         // 'h = hex radix
'o173        // 'o = octal radix
'b111_1011   // 'b = binary radix, "_" are ignored
'hxx         // can include X, Z or ? in non-decimal constants
16'd5        // 16-bit constant 'b0000_0000_0000_0101
11'h1X?      // 11-bit constant 'b001_XXXX_ZZZZ
```

By default constants are unsigned and will be extended with 0's on left if need be (if high-order bit is X or Z, the extended bits will be X or Z too). You can specify a signed constant as follows:

```
8'shFF       // 8-bit twos-complement representation of -1
```

To be absolutely clear in your intent it's usually best to explicitly specify the width and radix.

Basic building block: modules

In Verilog we design modules, one of which will be identified as our top-level module. Modules usually have named, directional ports (specified as input, output or inout) which are used to communicate with the module.

Don't forget this ";"

```
// 2-to-1 multiplexer with dual-polarity outputs
module mux2(input a,b,sel, output z,zbar);
   wire selbar,z1,z2;    // wires internal to the module
   // order doesn't matter - all statements are
   // executed concurrently!
   not i1(selbar,sel);   // inverter, name is "i1"
   and a1(z1,a,selbar);  // port order is (out,in1,in2,…)
   and a2(z2,b,sel);
   or  o1(z,z1,z2);
   not i2(zbar,z);
endmodule
```
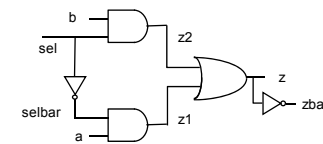

In this example the module's behavior is specified using Verilog's built-in Boolean modules: not, buf, and, nand, or, nor, xor, xnor. Just say no! We want to specify behavior, not implementation!

Continuous assignments

If we want to specify a behavior equivalent to combinational logic, use Verilog's operators and continuous assignment statements:

```verilog
// 2-to-1 multiplexer with dual-polarity outputs
module mux2(input a,b,sel, output z,zbar);
    // again order doesn't matter (concurrent execution!)
    // syntax is "assign LHS = RHS" where LHS is a wire/bus
    // and RHS is an expression
    assign z = sel ? b : a;
    assign zbar = ~z;
endmodule
```

Conceptually `assign`'s are evaluated continuously, so whenever a value used in the RHS changes, the RHS is re-evaluated and the value of the wire/bus specified on the LHS is updated.

This type of execution model is called "dataflow" since evaluations are triggered by data values flowing through the network of wires and operators.

Boolean operators

- Bitwise operators perform bit-oriented operations on vectors
  - ~(4'b0101) = {~0,~1,~0,~1} = 4'b1010
  - 4'b0101 & 4'b0011 = {0&0, 1&0, 0&1, 1&1} = 4'b0001
- Reduction operators act on each bit of a single input vector
  - &(4'b0101) = 0 & 1 & 0 & 1 = 1'b0
- Logical operators return one-bit (true/false) results
  - !(4'b0101) = 1'b0

Bitwise	
~a	NOT
a & b	AND
a | b	OR
a ^ b	XOR
a ~^ b a ^~ b	XNOR

Reduction	
&a	AND
~&a	NAND
|a	OR
~|a	NOR
^a	XOR
~^a ^~a	XNOR

Logical (checks for z and x)

!a	NOT
a && b	AND
a || b	OR
a == b a != b	[in]equality returns x when x or z in bits. Else returns 0 or 1
a === b a !== b	case [in]equality returns 0 or 1 based on bit by bit comparison

Note distinction between ~a and !a when operating on multi-bit values

---

Boolean operators

- ^ is NOT exponentiation (**)
- Logical operator with z and x
  - 4'bz0x1 === 4'bz0x1 = 1   4'bz0x1 === 4'bz001 = 0
- Bitwise operator with z and x
  - 4'b0001 & 4'b1001 = 0001   4'b1001 & 4'bx001 = x001

Bitwise	
~a	NOT
a & b	AND
a | b	OR
a ^ b	XOR
a ~^ b a ^~ b	XNOR

Reduction	
&a	AND
~&a	NAND
|a	OR
~|a	NOR
^a	XOR
~^a ^~a	XNOR

Logical

!a	NOT
a && b	AND
a || b	OR
a == b a != b	[in]equality returns x when x or z in bits. Else returns 0 or 1
a === b a !== b	case [in]equality returns 0 or 1 based on bit by bit comparison

Note distinction between ~a and !a when operating on multi-bit values

---

Integer Arithmetic

- Verilog's built-in arithmetic makes a 32-bit adder easy:

```verilog
module add32
    (input[31:0] a, b,
     output[31:0] sum);

    assign sum = a + b;
endmodule
```

- A 32-bit adder with carry-in and carry-out:

```verilog
module add32_carry
    (input[31:0] a,b,
     input cin,
     output[31:0] sum,
     output cout);

    assign {cout, sum} = a + b + cin;
endmodule
```

concatenation

Other operators

Conditional

a ? b : c	If a then b else c

Relational

a > b	greater than
a >= b	greater than or equal
a < b	Less than
a <= b	Less than or equal

Arithmetic

-a	negate
a + b	add
a - b	subtract
a * b	multiply
a / b	divide
a % b	modulus
a ** b	exponentiate
a << b	logical left shift
a >> b	logical right shift
a <<< b	arithmetic left shift
a >>> b	arithmetic right shift

Hierarchy: module instances

Our descriptions are often hierarchical, where a module's behavior is specified by a circuit of module instances:

```verilog
// 4-to-1 multiplexer
module mux4(input d0,d1,d2,d3, input [1:0] sel, output z);
  wire z1,z2;
  // instances must have unique names within current module.
  // connections are made using .portname(expression) syntax.
  // once again order doesn't matter…
  mux2 m1(.sel(sel[0]),.a(d0),.b(d1),.z(z1));  // not using zbar
  mux2 m2(.sel(sel[0]),.a(d2),.b(d3),.z(z2));
  mux2 m3(.sel(sel[1]),.a(z1),.b(z2),.z(z));
  // could also write "mux2 m3(z1,z2,sel[1],z,)" NOT A GOOD IDEA!
endmodule
```

Connections to module's ports are made using a syntax that specifies both the port name and the wire(s) that connects to it, so ordering of the ports doesn't have to be remembered ("explicit").

This type of hierarchical behavioral model is called "structural" since we're building up a structure of instances connected by wires. We often mix dataflow and structural modeling when describing a module's behavior.

Parameterized modules

```verilog
// 2-to-1 multiplexer, W-bit data
module mux2 #(parameter W=1)  // data width, default 1 bit
          (input [W-1:0] a,b,
           input sel,
           output [W-1:0] z);
  assign z = sel ? b : a;
  assign zbar = ~z;
endmodule

// 4-to-1 multiplexer, W-bit data
module mux4 #(parameter W=1)  // data width, default 1 bit
          (input [W-1:0] d0,d1,d2,d3,
           input [1:0] sel,
           output [W-1:0] z);
  wire [W-1:0] z1,z2;

  mux2 #(.W(W)) m1(.sel(sel[0]),.a(d0),.b(d1),.z(z1));
  mux2 #(.W(W)) m2(.sel(sel[0]),.a(d2),.b(d3),.z(z2));
  mux2 #(.W(W)) m3(.sel(sel[1]),.a(z1),.b(z2),.z(z));
endmodule
```

could be an expression evaluable at compile time; if parameter not specified, default value is used

---

Sequential behaviors

There are times when we'd like to use sequential semantics and more powerful control structures – these are available inside sequential always blocks:

```verilog
// 4-to-1 multiplexer
module mux4(input a,b,c,d, input [1:0] sel, output reg z,zbar);
  always @(*) begin
    if (sel == 2'b00) z = a;
    else if (sel == 2'b01) z = b;
    else if (sel == 2'b10) z = c;
    else if (sel == 2'b11) z = d;
    else z = 1'bx;    // when sel is X or Z
    // statement order matters inside always blocks
    // so the following assignment happens *after* the
    // if statement has been evaluated
    zbar = ~z;
  end
endmodule
```

always @(*) blocks are evaluated whenever any value used inside changes. Equivalently we could have written

always @(a, b, c, d, sel) begin … end   // careful, prone to error!

Historical Usage: `reg` vs `wire`

Verilog (not SystemVerilog in this case) uses `wire` declarations when naming nets (ports are declared as wires by default).

However nets appearing on the LHS of assignment statements inside of `always` blocks must be declared as type `reg`.

> I don't know why Verilog has this rule! I think it's because traditionally *always* blocks were used for sequential logic (the topic of next lecture) which led to the synthesis of hardware registers instead of simply wires. Always blocks typically include flip-flops (storage), hence the concept of *always_ff* in SystemVerilog. So this seemingly unnecessary rule really supports historical usage – the declaration would help the reader distinguish registered values from combinational values.

SystemVerilog logic keyword replaces reg and wire.

Case statements

Chains of if-then-else statements aren't the best way to indicate the intent to provide an alternative action for every possible control value. Instead use `case`:

```
// 4-to-1 multiplexer
module mux4(input a,b,c,d, input [1:0] sel, output reg z,zbar);
  always @(*) begin
    case (sel)
      2'b00: z = a;
      2'b01: z = b;
      2'b10: z = c;
      2'b11: z = d;
      default: z = 1'bx;   // in case sel is X or Z
    endcase
    zbar = ~z;
  end
endmodule
```

`case` looks for an exact bit-by-bit match of the value of the case expression (e.g., sel) against each case item, working through the items in the specified order. `casex`/`casez` statements treat X/Z values in the selectors as don't cares when doing the matching that determines which clause will be executed.

Unintentional creation of state

Suppose there are multiple execution paths inside an `always` block, i.e., it contains `if` or `case` statements, and that on some paths a net is assigned and on others it isn't.

```
// 3-to-1 multiplexer ????
module mux3(input a,b,c, input [1:0] sel, output reg z);
  always @(*) begin
    case (sel)
      2'b00: z = a;
      2'b01: z = b;
      2'b10: z = c;
      // if sel is 2'b11, no assignment to z!!??
    endcase
  end
endmodule
```
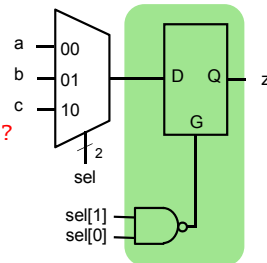

So sometimes z changes and sometimes it doesn't (and hence keeps its old value). That means the synthesized hardware has to have a way of remembering the state of z (i.e., it's old value) since it's no longer just a combinational function of sel, a, b, and c. Not what was intended here. More on this in next lecture.

Keeping logic combinational

To avoid the unintentional creation of state, ensure that each variable that's assigned in an `always` block always gets assigned a new value at least once on every possible execution path.

```
    // 3-to-1 multiplexer
    module mux3(input a,b,c, input [1:0] sel, output reg z);
      always @ (*) begin
        z = 1'bx;      // a second assignment may happen below
        case (sel)
          2'b00: z = a;
          2'b01: z = b;
          2'b10: z = c;
          default: z = 1'bx;
        endcase
      end
    endmodule
```

Use one or the other

It's good practice when writing combinational `always` blocks to provide a `default:` clause for each `case` statement and an `else` clause for each `if` statement.

Other useful Verilog features

- Additional control structures: `for`, `while`, `repeat`, `forever`
- Procedure-like constructs: functions, tasks
- One-time-only initialization: `initial` blocks
- Compile-time computations: `generate`, `genvar`
- System tasks to help write simulation test jigs
  - Stop the simulation: `$finish(…)`
  - Print out text, values: `$display(…)`
  - Initialize memory from a file: `$readmemh(…)`, `$readmemb(…)`
  - Capture simulation values: `$dumpfile(…)`, `$dumpvars(…)`
  - Explicit time delays (simulation only!!!!) : `#nnn`
- Compiler directives
  - Macro definitions: `` `define ``
  - Conditional compilation: `` `ifdef ``, …
  - Control simulation time units: `` `timescale ``
  - No implicit net declarations: `` `default_nettype none ``

For Loops, Repeat Loops in Simulation

```
integer i;    // index must be declared as integer
integer irepeat;

// this will just wait 10ns, repeated 32x.
// simulation only! Cannot implement #10 in hardware!
    irepeat =0;
    repeat(32) begin
    #10;
    irepeat = irepeat + 1;
    end


// this will wait #10ns before incrementing the for loop
    for (i=0; i<16; i=i+1) begin
     #10;     // wait #10 before increment.
       // @(posedge clk);
       // add to index on posedge
    end

// other loops:  forever, while
```
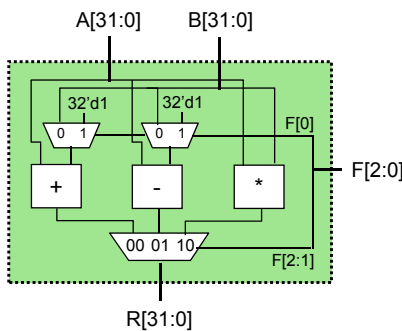

Defining Processor ALU in 5 mins

- Modularity is essential to the success of large designs
- High-level primitives enable direct synthesis of behavioral descriptions (functions such as additions, subtractions, shifts (<< and >>), etc.

Example: A 32-bit ALU

Function Table

F2	F1	F0	Function
0	0	0	A + B
0	0	1	A + 1
0	1	0	A - B
0	1	1	A - 1
1	0	X	A * B

Module Definitions

2-to-1 MUX

```
module mux32two
      (input [31:0] i0,i1,
       input sel,
       output [31:0] out);

   assign out = sel ? i1 : i0;
endmodule
```

32-bit Adder

```
module add32
      (input [31:0] i0,i1,
       output [31:0] sum);

   assign sum = i0 + i1;
endmodule
```

32-bit Subtracter

```
module sub32
      (input [31:0] i0,i1,
       output [31:0] diff);

   assign diff = i0 - i1;
endmodule
```

3-to-1 MUX

```
module mux32three
      (input [31:0] i0,i1,i2,
       input [1:0] sel,
       output reg [31:0] out);

always @ (i0 or i1 or i2 or sel)
begin
  case (sel)
    2'b00: out = i0;
    2'b01: out = i1;
    2'b10: out = i2;
    default: out = 32'bx;
  endcase
end
endmodule
```

16-bit Multiplier

```
module mul16
      (input [15:0] i0,i1,
       output [31:0] prod);

// this is a magnitude multiplier
// signed arithmetic later
assign prod = i0 * i1;

endmodule
```

Top-Level ALU Declaration

- Given submodules:

```
module mux32two(i0,i1,sel,out);
module mux32three(i0,i1,i2,sel,out);
module add32(i0,i1,sum);
module sub32(i0,i1,diff);
module mul16(i0,i1,prod);
```

- Declaration of the ALU Module:

```
module alu
   (input [31:0] a, b,
    input [2:0] f,
    output [31:0] r);
```

A[31:0]   B[31:0]

alu

32'd1

0   1   F[0]

+   -   *   F[2:0]

00 01 10   F[2:1]

R[31:0]

```
wire [31:0] submux_out;
wire [31:0] add_out, sub_out, mul_out;
```

intermediate output nodes●

```
mux32two    sub_mux(b, 32'd1, f[0], submux_out);
add32       our_adder(a, addmux_out, add_out);
sub32       our_subtracter(a, submux_out, sub_out);
mul16       our_multiplier(a[15:0], b[15:0], mul_out);
mux32three  output_mux(add_out, sub_out, mul_out, f[2:1], r);

endmodule
```

module names	(unique) instance names	corresponding wires/regs in module alu

---
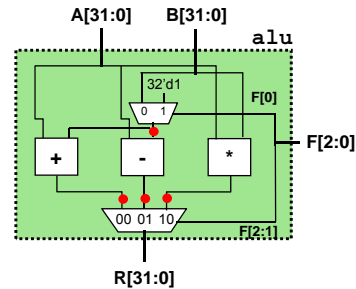
Use Explicit Port Declarations

```
mux32two  adder_mux(b, 32'd1, f[0], addmux_out);
```

Order of the ports and bit width matters!

```
mux32two  adder_mux(,i0(b), .i1(32'd1),
             .sel(f[0]), .out(addmux_out));
```

Identify *input and output* in port names

```
xvga xvga1(.vclock_in(clock_65mhz),
          .hcount_out(hcount),.vcount_out(vcount),
          .hsync_out(hsync),.vsync_out(vsync),
          .blank_out(blank));




module xvga(input vclock_in,
          output reg [10:0] hcount_out,
          output reg [9:0] vcount_out,
          output reg vsync_out, hsync_out,
          output reg blank_out);
```

Checkoff Reminder

- May checkoff at any time prior to checkoff date.
- On checkoff date, checkoff will staff's be main priority
- Two checkoff dates: last name A-M (Thu), N-Z (Fri)
- Thu checkoff starts at 5pm, Fri 1pm
- Schedule time on google doc