VERILOG 2:

LANGUAGE BASICS

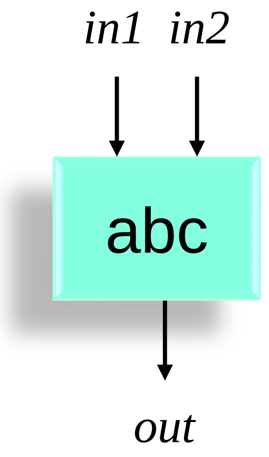
Verilog `module`

- Modules are basic building blocks. These are two example module definitions which you should use:

```
// Safer traditional method
module abc (in1, in2, out);
   input  in1;
   input  in2;
   output out;
<body of module>
endmodule
```

or

```
// Shorter, info not repeated
module abc (
   input  in1,
   input  in2,
   output out        // no comma on last signal
   );
<body of module>
endmodule
```



Others exist, but use only one of these two forms in this class

Verilog Comments and Values

- Comments
  - Single line
    ```
    assign b = c;      // A comment
    ```
  - Multiple lines
    ```
    /* This module filters a series of images at a
       rate of "f" frames per second      */
    ```
- Values of a single wire or register (not buses)
  - **`0`** and **`1`** // Binary zero and one
  - **`x`** `or` **`X`** // Unknown value due to things such as uninitialized state or // two drivers driving the same net
  - **`z`** `or` **`Z`** // High impedance, e.g., a node not driven by any circuit. // This is identical to the "z" state of a tri-state output driver.
  - others // Don't worry about others

Verilog Constants

- Can be specified in a number of formats; use only these four in this class:
  - binary
  - hexadecimal
  - octal
  - decimal
- Syntax: [size.in.bits]'[first.letter.of.base.of.representation][value]
- Underscore characters ("_") are ignored and can greatly help readability
- Make sure to specify enough digits to cover the full range of the constant. Although Quartus will probably not complain, other CAD tools may do something you are not expecting especially with more complex number formats.
- Examples:

	Value in binary	Comment
• 1'b0	0	
• 1'b1	1	
• 4'b0101	0101	
• 5'h0B	01011	// two hex digits for 5 bits, range of [0, +31]
• 16'h3F09	0011111100001001	// four hex digits for 16 bits
• 12'b0000_1010_0101	000010100101	// underscores are ignored
• 8'd003	00000011	// three base 10 digits for 8 bits // which has range of [0, +255]

Constants With *parameter* and *`define*

- There are two main methods to simplify constants by using readable text to represent a number
  - **parameter**
    - Local to a module
    - Usage:
      ```
      parameter   HALT = 4'b0101;
      …
      if (inst == HALT) begin
      ```
    - Definitely use this for state names in state machines in this class
  - **`define** macro
    - Global text macro substitution using a compiler directive
    - Usage:
      ```
      `define   HALT  4'b0101
      …
      if (inst == `HALT) begin     // requires "back tick" "grave accent"
      ```
    - Best when helpful to put all definitions in a global file; probably do not use in this class

Verilog Operators

- Operators: bit-wise

negation	`~`
AND	`&`
OR	`|`
XOR	`^`
Shift *a* left by *b* bits	`a << b`
Shift *a* right by *b* bits	`a >> b`

- Operators: logical (e.g., test for if-then-else)

negation	`!`
AND	`&&`
OR	`||`

- Basic arithmetic

addition	`+`
subtraction	`-`
multiplication	`*`
division	`/`
modulus	`%`

Verilog Operators

- Equality, inequalities, and relational operators—all return a 1-bit true or false result

– equal	==
– not equal	!=
– less than	<
– greater than	>
– less than or equal	<=
• Not to be confused with the non-blocking assignment which is used with flip-flops	
– greater than or equal	>=

Verilog Operators

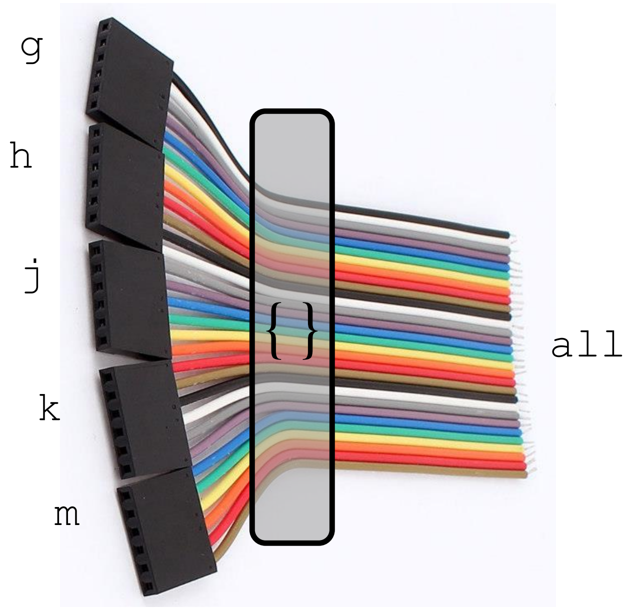
- Concatenation `x = {a,b,c}`
  - Each input may be a **`wire`** or a **`reg`**
  - The output may be a **`wire`** or a **`reg`**
  - Example: if g, h, j, l, m are all 6 bits wide, then

    ```
    all = {g,h,j,k,m}
    ```

    is 30 bits wide
  - Example: to replicate the sign bit of a 4-bit value *a* two times and assign it to *b*:

    ```
    reg [5:0] b;
    b = {a[3], a[3], a};
    ```

    If *a* were `1010`, then *b* would be `111010`



* 3 Ways to Specify Hardware *

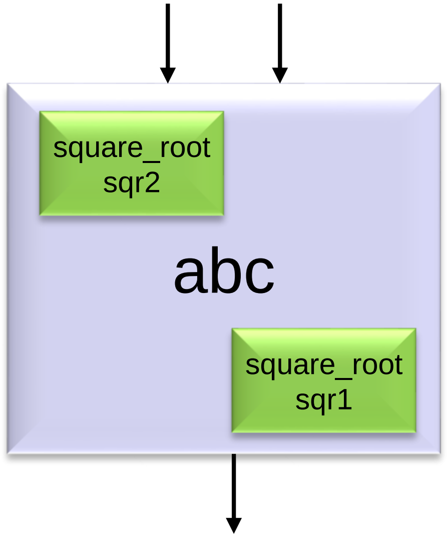
- There are three primary means to specify hardware circuits:
  1) Instantiate another module
  2) *wires, assign* statements
  3) *registers, always* blocks
- Example instantiating modules inside a main module

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

   assign...

   always...

   always...

   square_root sqr1 (clk, reset, in1, out1);

   square_root sqr2 (clk, reset, in2, out2);

endmodule
```

module name

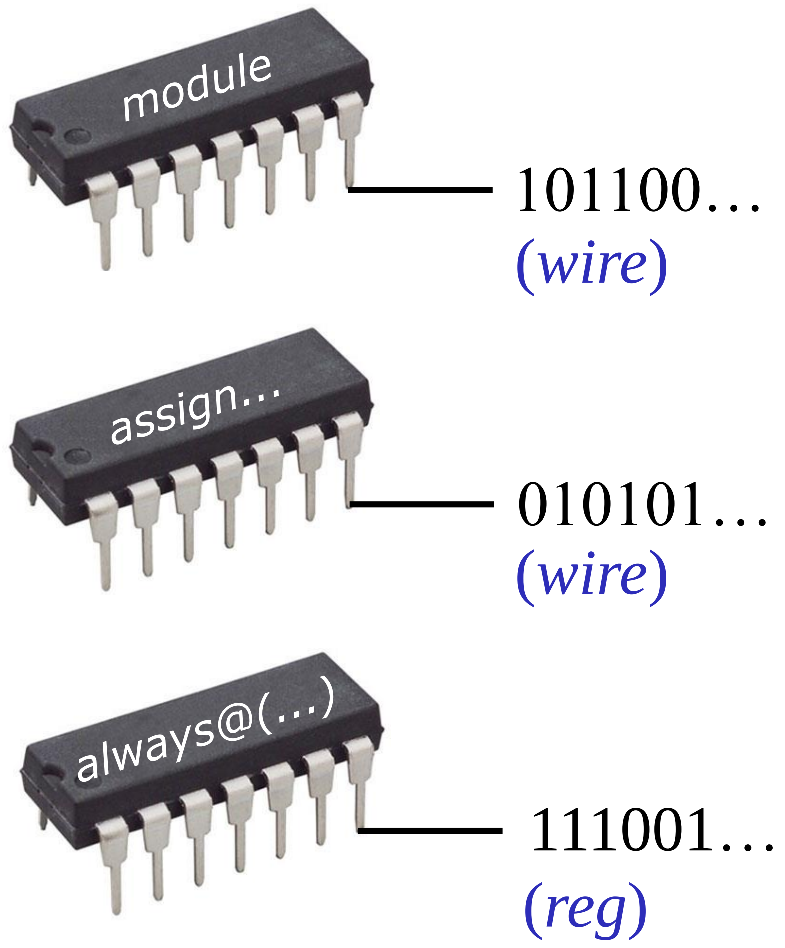
instance names

module name



Concurrency

- All circuits operate independently and concurrently
  - Different from most programming paradigms
- This is natural if we remember "hardware verilog" describes real circuit hardware: transistors and wires



Declaring and Referencing Signals

- Single-bit wire and reg signals
  - `wire reset;`
  - `reg start;`
- Multiple-bit signals
  - By convention, write **[**(*MSB-1*)**:0]**
- Multiple-bit wire and reg signals
  - `wire [7:0] phase;          // 8-bit signal`
  - `reg [31:0] elevation;      // 32-bit signal`
- To reference part of a multi-bit signal
  - `phase[0]                   // LSB of phase`
  - `elevation[7:0]             // lowest byte`

Verilog Instantiation Syntax

- Ports of an instantiated module can be connected to signals referenced in the module's declaration assuming they are in the same order but this is dangerous so don't do it. Instead write out both the port name and the connected signal as shown below.

```
module abc (in1, in2, out);
   input  in1;
   input  in2;
   output out;
   ...
endmodule
```

-
  ```
  // Don't use this method! It works but typos can be difficult to catch
  abc instance1 (phase3, angle, magnitude3);  // phase3 connected to in1, etc.
  ```

-
  ```
  // This is good. Ports are in the same order as in the module declaration
  abc instance2 (
     .in1    (phase1),
     .in2    (angle),
     .out    (magnitude1) );        // no comma on last port
  ```

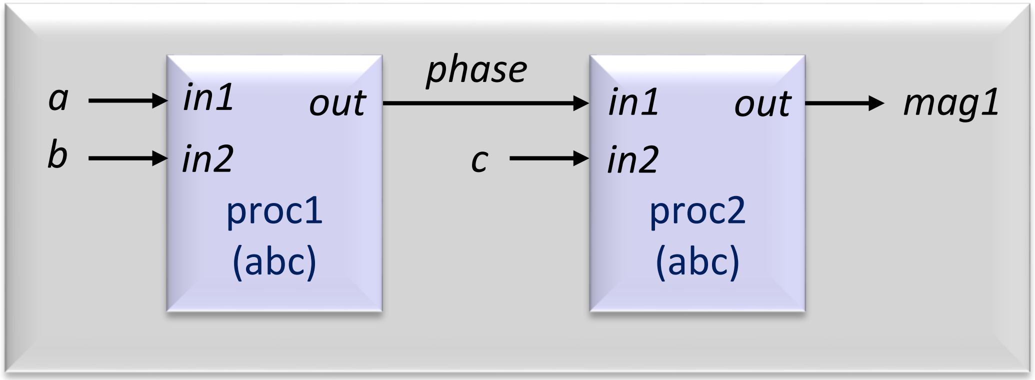
-
  ```
  // This is good. Ports are not in the same order as in the module declaration
  abc instance3 (
     .in2    (angle),               // in2 comes before in1 here but everything
     .in1    (phase2),              //   still works ok
     .out    (magnitude2) );
  ```

Verilog Instantiation Example

- In this example, two copies of the module “abc” are instantiated in the higher-level module. As described in a later slide, only wires can connect to the outputs of modules.

```
wire phase;   // must be a wire
wire mag1;    // must be a wire
// a, b, c may be wires, regs,
// or inputs of the module

abc proc1 (
   .in1   (a),
   .in2   (b),
   .out   (phase) );

abc proc2 (
   .in1   (phase),
   .in2   (c),
   .out   (mag1) );
```

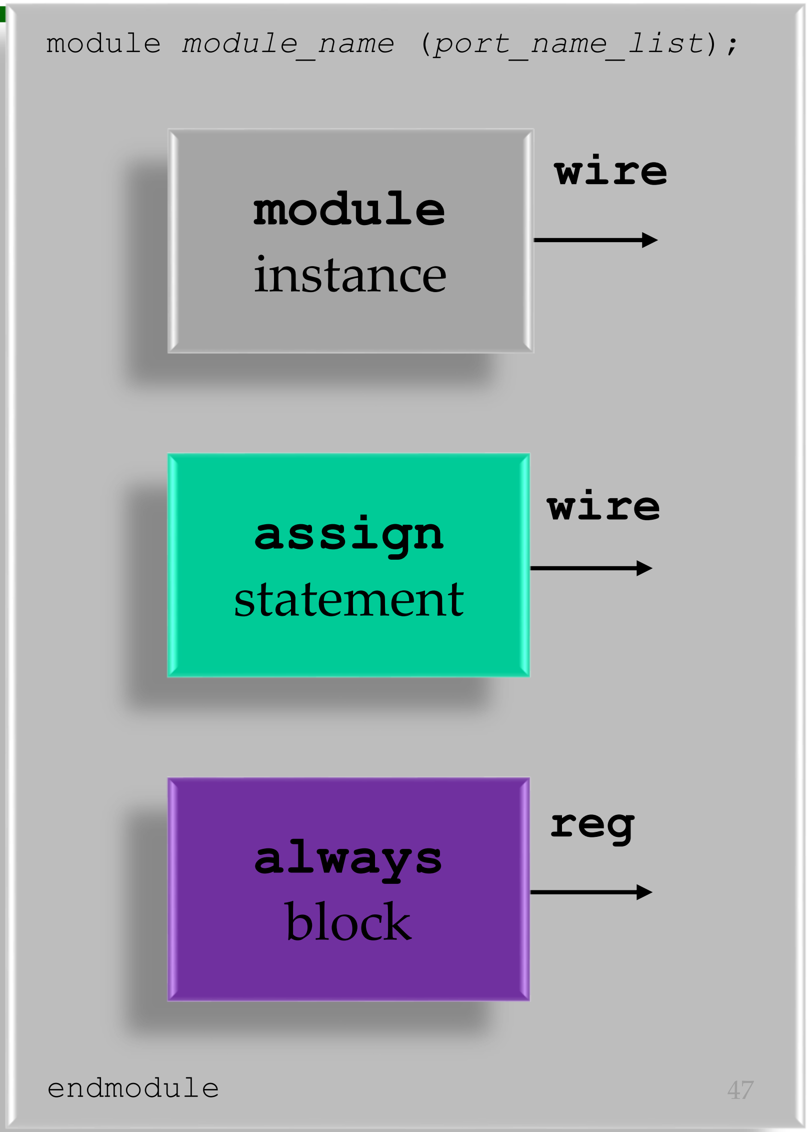
```
module abc (in1, in2, out);
    input   in1;
    input   in2;
    output  out;
    ...
endmodule
```



Describing Hardware

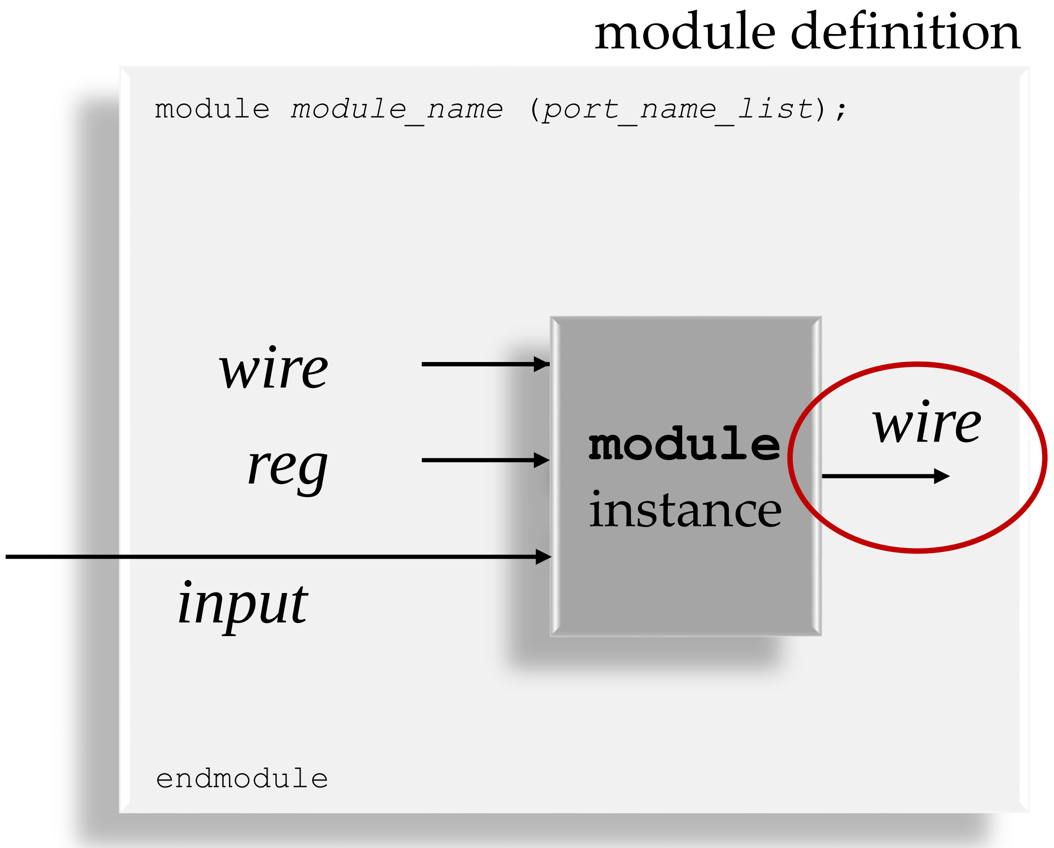
- As previously stated, there are three main ways to describe hardware circuits which produce a "signal", "electrical node", "word", (whatever you like to call it) inside a module definition:
  - Instantiate a **`module`** which has **`wire`**s connected to its outputs
  - The **`assign`** command which defines a **`wire`**
  - The **`always`** command which defines a **`reg`**
- All of these must be declared at the module definition level—not inside each other (e.g., a module instance can not be declared inside an always block)

module definition



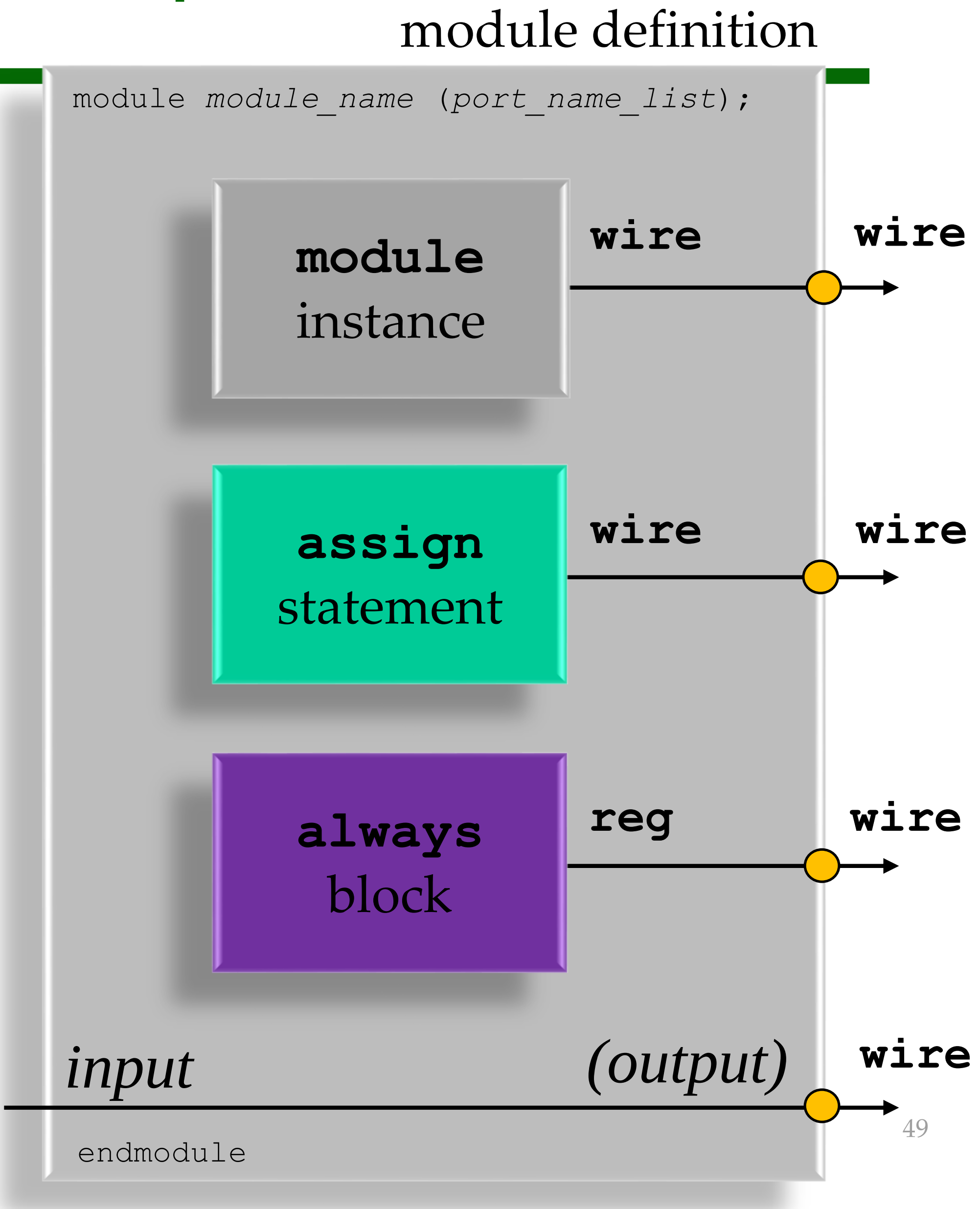
Module Inputs and Outputs

- There are three main possible inputs to a module instance:
  - A **`wire`**
  - A **`reg`**
  - An **`input`** into the module (behaves just like a wire)
- The output of a module instance is always a **`wire`**, at least for this class
  - This is perhaps the most tricky case



Module Outputs

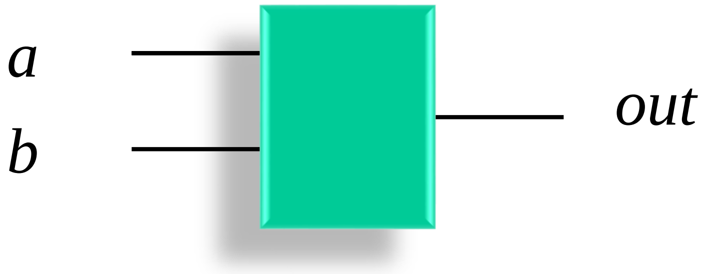
- All of these signal types may be used as outputs in a module definition:
  - **`wire`**
  - **`reg`**
  - Another possibility which is typically uncommon is for an **`input`** to pass directly to a module **`output`** port

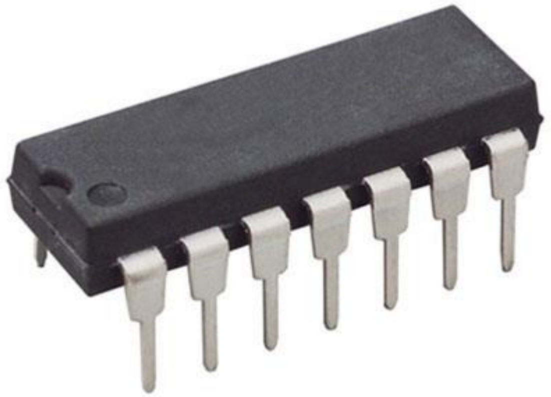


2) *wire, assign*

- Picture "always active" hardwired logic
- For now, declare all wires

```
wire out;
// a and b can be wires or regs or module inputs
```

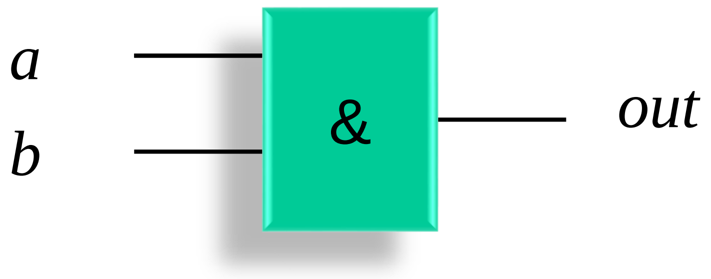


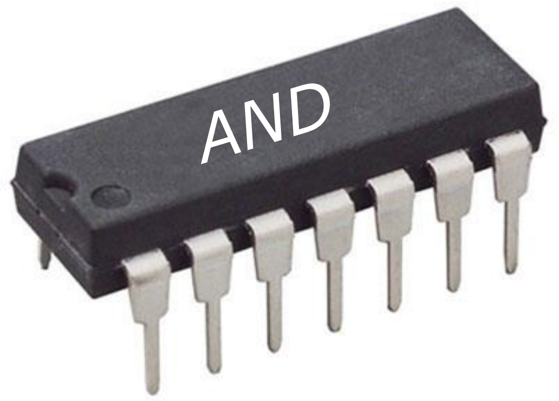


2) *wire, assign*

- Example:

```
wire out;
assign out = a & b;
```

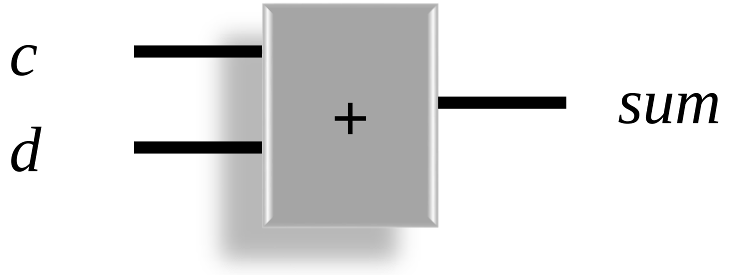


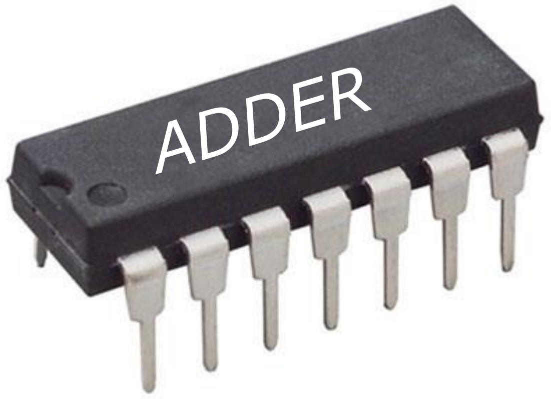


2) *wire, assign*

- Example: multibit operands

```
wire [3:0] c, d;        // c and d are both 4 bits
wire [4:0] sum;         // sum is 5 bits so no overflow
assign sum = {c[3],c} + {d[3],d};   // sign extend inputs
                                   // for 2’s complement
```





3) *reg, always*

- Picture a much more general way of assigning "wires" or "signals" or "buses"
- "if/then/else" and "case" statements are permitted
- You could, but don't use "for loops" in hardware blocks (use in testing blocks is ok)
- Sequential execution
  - *statements* execute in order to specify a *circuit*
- Syntax:

  ```
  always @(sensitivity list) begin
     statements
  end
  ```
- Operation:
  statements are executed when any signal in *sensitivity list* changes

3) *reg, always*

- Including all inputs in the sensitivity list can be tedious and prone to errors especially as the number of statements in the always block grows

  ```
  always @(sensitivity list) begin
     statements
  end
  ```

- Verilog 1364-2001 allows the use of the

  ```
  always @(*)
  ```

  or

  ```
  always @*
  ```

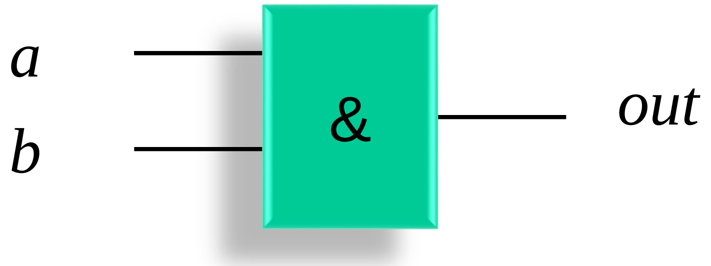
  construct which tells the simulator to include all inputs in the sensitivity list automatically. This can be very handy but is not supported by all modern CAD tools.

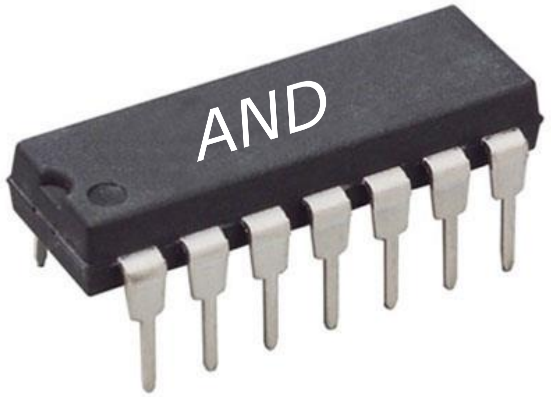
- Ok to use for this class
  - If you discover any issues, email the instructor and your TA

3) *reg, always*

- Example: there is **no** difference whatsoever in this AND gate from the AND gate built using *assign*

```
reg out;
always @(a or b) begin
   out = a & b;
end
```





If-Then-Else Statement

- The general syntax is as follows:

```
if  (condition)
       statement
else
       statement
```

- Or, taking advantage of the fact that a begin–end block acts as a single statement:

```
if  (condition)  begin
       statement;
       statement;
       ...
end
else begin
       statement;
       statement;
       ...
end
```

If-Then-Else Statement

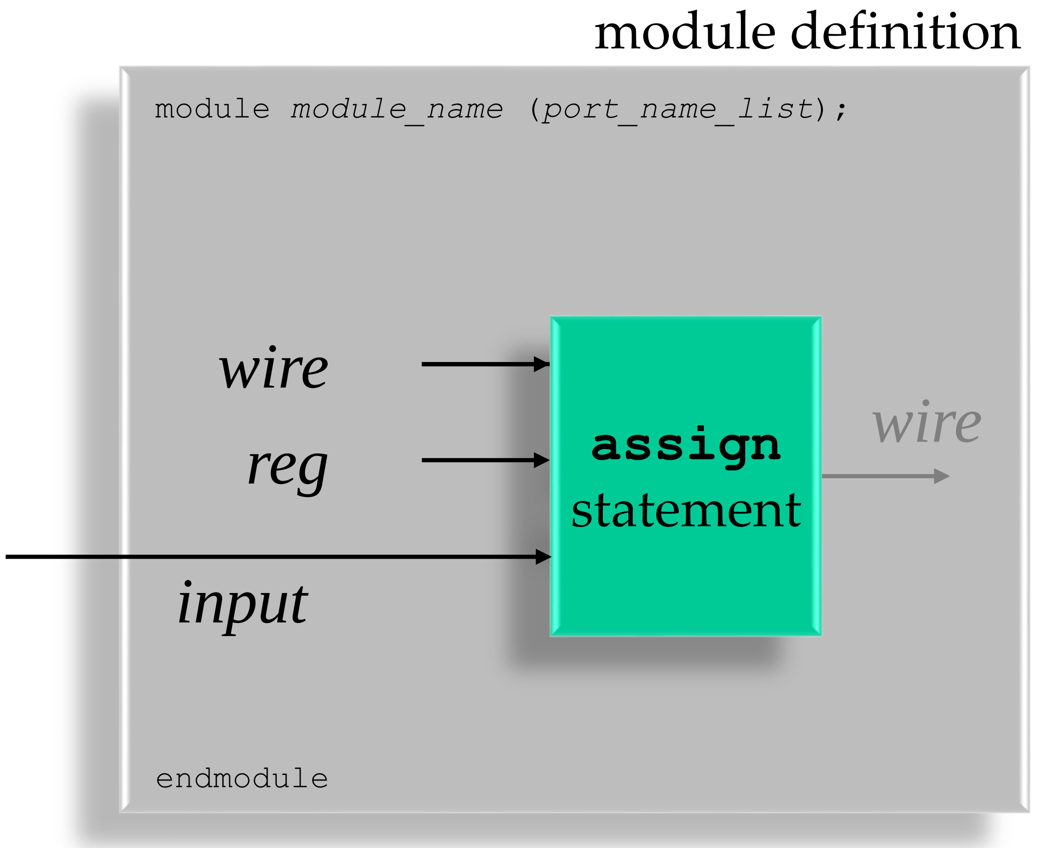
- Nesting an if block within another yields “else-if” blocks:

```
if (condition1) begin
    statement;
    statement;
    . . .
end
else if (condition2) begin
    statement;
    statement;
    . . .
end
else begin
    statement;
    statement;
    . . .
end
```

assign statement inputs

- In the same way, there are three main possible “inputs” to an assign statement:
  - A **wire**
  - A **reg**
  - An **input** into the module
- Example:

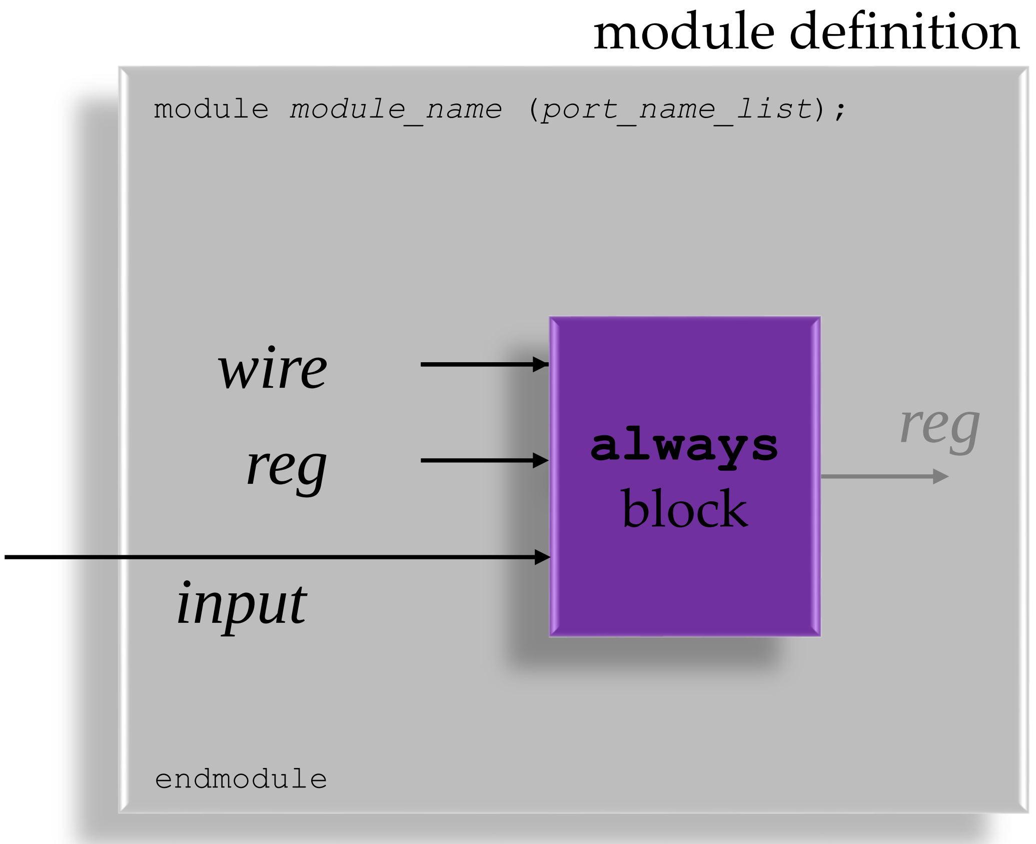
```
input a;
wire b;
reg c;

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



always block inputs

- In the same way, there are three main possible "inputs" to an always block:
  - A **wire**
  - A **reg**
  - An **input** into the module (technically still a wire)
- Example:

```
input a;
wire b;
reg c;

reg x;
always @(*) begin
  x = a & b | c;
end
```



Special Block Style: `initial`

- This block executes only once at the beginning of the simulation. It is the normal way to write testbench code.

```
initial begin
   ...
end
```

  - Example: circuit that generates a *reset* signal at the beginning of a simulation
- For our usage, initial blocks are used in only two cases
  1) Test bench code
  2) Hardware code only to specify the contents of a ROM memory (for EEC 180 FPGAs)

Special Block Style: `always begin`

- This block executes repeatedly; it begins another execution cycle as soon as it finishes. Therefore it must contain some delay. This is a good construct for a clock oscillator.

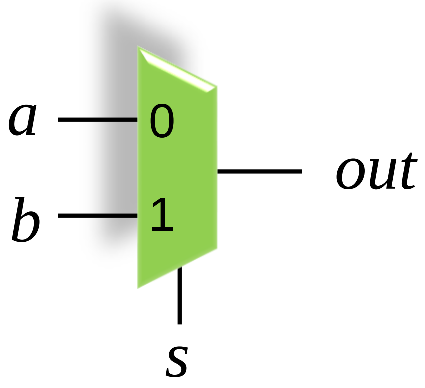
```
always begin
   ...
end
```

  - Example: *clock* signal generator
  - Can view as an **`always @(sensitivity list)`** construct where the sensitivity list is always activated immediately
- Verilog suitable for *always* blocks is also suitable for *initial* blocks

Example: 2:1 Multiplexer

- Example #1

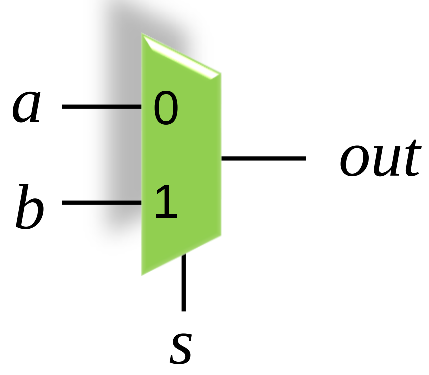
```
reg out;
always @(a or b or s) begin
  if (s == 1'b0) begin
    out = a;
    end
  else begin
    out = b;
    end
end
```



Example: 2:1 Multiplexer

- Example #1
- Normally always include *begin* and *end* statements even though they are not needed when there is only one statement in the particular block. Text struck out below could be taken out but always add it anyway in this class.

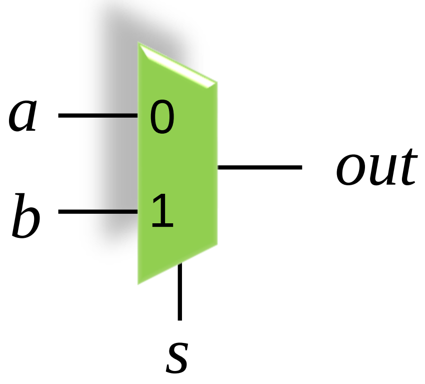
```
reg out;
always @(a or b or s) ~~begin~~
  if (s == 1'b0) ~~begin~~
    out = a;
    ~~end~~
  else ~~begin~~
    out = b;
    ~~end~~
~~end~~
```



Example: 2:1 Multiplexer

- Example #2
- May be clearer in some cases, e.g., s==1’b0 sets off auto airbag

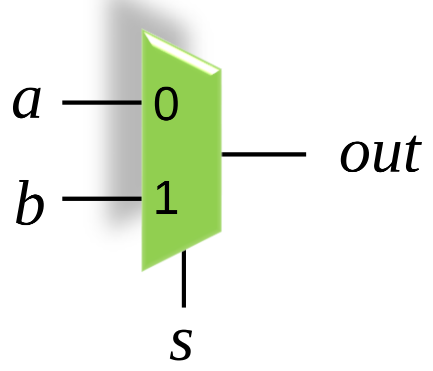
```
reg out;
always @(a or b or s) begin
   out = b;
   if (s == 1'b0) begin
      out = a;
   end
end
```



Example: 2:1 Multiplexer

- Example #3
- May be clearer in some cases

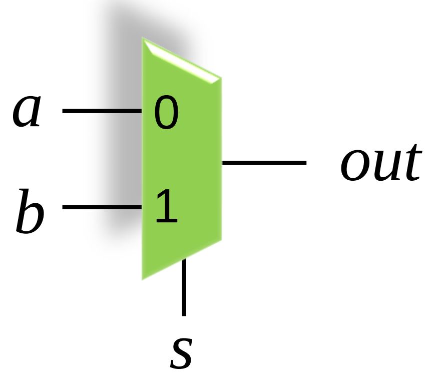
```
reg out;
always @(a or b or s) begin
  out = a;
  if (s == 1'b1) begin
    out = b;
  end
end
```



Example: 2:1 Multiplexer

- Example #4
- Simpler but less clear way of writing if/then/else called "inline if" or "conditional operator" which is also found in some programming languages

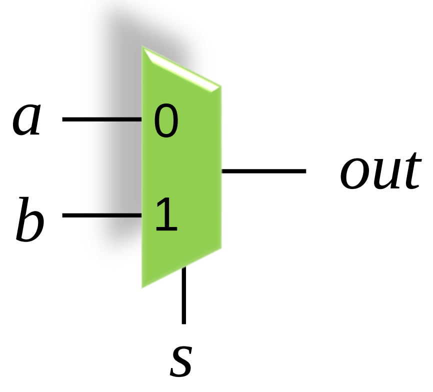
```
reg out;
always @(a or b or s) begin
  out = s ? b : a;
end
```



Example: 2:1 Multiplexer

- Example #5
- The inline conditional operator can also be used to define wires

```
wire out;
assign out = s ? b : a;
```



Case Statement

- The general syntax is as follows:
- *case_expression*
  - normally a multi-bit bus of wire or reg
- *value$_i$* targets
  - normally 0, 1, or
    a wildcard character (for casez and casex)
- *statement*
  1) An arbitrary-length block of verilog code beginning with "begin" and ending with "end"
     ```
     begin
         a = b + c;
         ....
     end
     ```
  2) A single verilog statement
- If multiple *value$_i$* targets match the *case_expression*, only the first one that matches is taken

```
case  (case_expression)
    value1:  statement
    value2:  statement
    value3:  statement
    ...
    valueN:  statement
    default:  statement
endcase
```

Case Statement: default

- The **default** case is optional
- It may be beneficial to set the output to a special value such as "x" even if you expect the default case will never be reached
  - For example:
    ```
    default: begin
       out = 4'bxxxx;
    end
    ```
  - Setting unused values to "x" makes them "don't care states" which should allow the synthesis tool to simplify logic
  - Setting unused input values to an easily-recognizable value (such as x's) could make mistakes easier to spot during debugging
  - Setting the output to "x" may cause warnings with some CAD tools

```
case  (wire or reg)
    value1 :  statement
    value2 :  statement
    value3 :  statement
    . . .
    valueN :  statement
    default:  statement
endcase
```

`casez` and `casex`

- `case`
  - Normal case statement
- `casez`
  - Allows use of wildcard “?” character for don’t cares in the target values

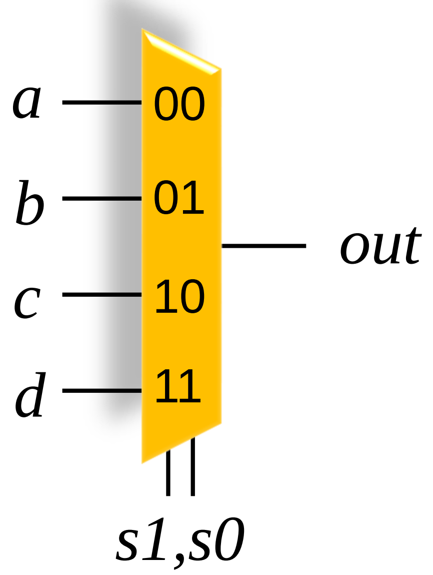
```
casez(in)
   4’b1???: out = r;
   4’b01??: out = s;
   4’b0000: out = t;
   default: out = 4’bxxxx;
endcase
```

- `casex`
  - Do not use it for this class. It can use “z” or “x” logic
  - Recommendation: probably never use it for hardware

Example: 4:1 Multiplexer

- Example: 4:1 multiplexer

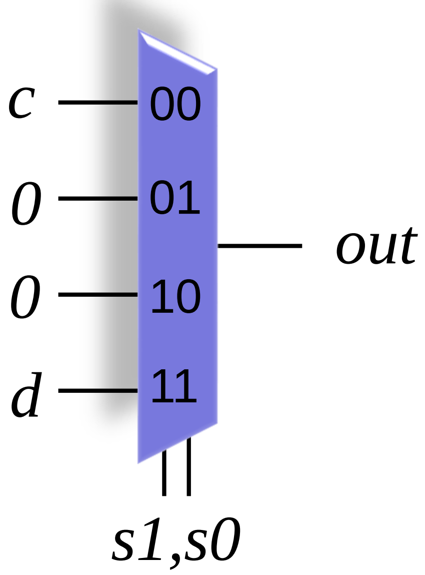
```
reg  out;           // must be a reg to be set in an always block!

always @(a or b or c or d or s1 or s0) begin
  case ({s1,s0})        // concatenate two select signals
    2'b00: begin
      out = a;
    end
    2'b01: begin
      out = b;
    end
    2'b10: begin
      out = c;
    end
    2'b11: begin
      out = d;
    end
    default: begin      // does nothing
      out = 1'bx;
    end
  endcase
end   // end of always block
```



Example: 4:1 Mux with zero on two inputs

- Example #1

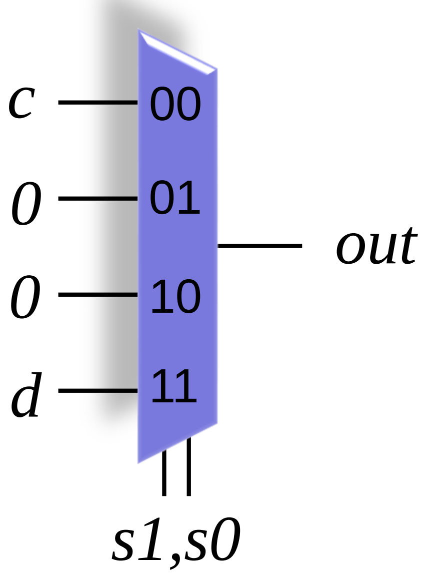
```
reg  out;          // must be a reg to be set in an always block!

always @(c or d or s1 or s0) begin
  case ({s1,s0})
    2'b00: begin
      out = c;
    end
    2'b01: begin
      out = 1'b0;
    end
    2'b10: begin
      out = 1'b0;
    end
    2'b11: begin
      out = d;
    end
    default: begin
      out = 1'b0;      // zero
    end
  endcase
end  // end of always block
```



Example: 4:1 Mux with zero on two inputs

- Example #2
- Here the case's default section *is used*

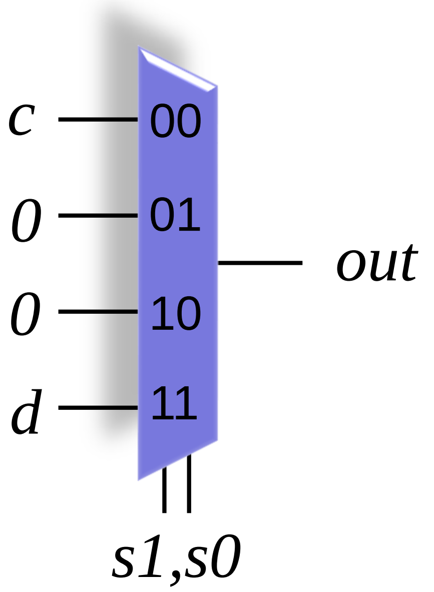
```
reg  out;          // must be a reg to be set in an always block!

always @(c or d or s1 or s0) begin
  case ({s1,s0})
    2'b00: begin
      out = c;
    end

    2'b11: begin
      out = d;
    end

    default: begin
      out = 1'b0;
    end
  endcase
end  // end of always block
```



Example: 4:1 Mux with zero on two inputs

- Example #3
- Here *out* is set to a default value before the case block

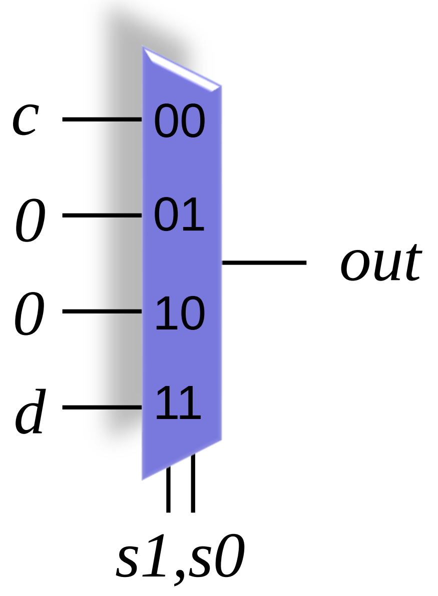
```
reg  out;          // must be a reg to be set in an always block!

always @(c or d or s1 or s0) begin
  out = 1'b0;      // set out to a "default" value

  case ({s1,s0})
    2'b00: begin
      out = c;
    end

    2'b11: begin
      out = d;
    end
  endcase
end  // end of always block
```



Example: 4:1 Mux with zero on two inputs

- Example #4
- Here *if* statements are used. Clearly there are many solutions.

```
reg  out;          // must be a reg to be set in an always block!

always @(c or d or s1 or s0) begin
  out = 1'b0;      // set "default"

  if ({s1,s0} == 2'b00) begin
    out = c;
  end

  if (s1==1'b1 && s0==1'b1) begin
    out = d;
  end
end  // end of always block
```


c
0
0
d
00
01
10
11
out
s1,s0