

Verilog II

- possible values of a net
- review
- vector
- case statement
- begin-end pair
- modeling combinational logic

Verilog data values of a net:

A net (node in EE-speak) in Verilog can have four possible values:

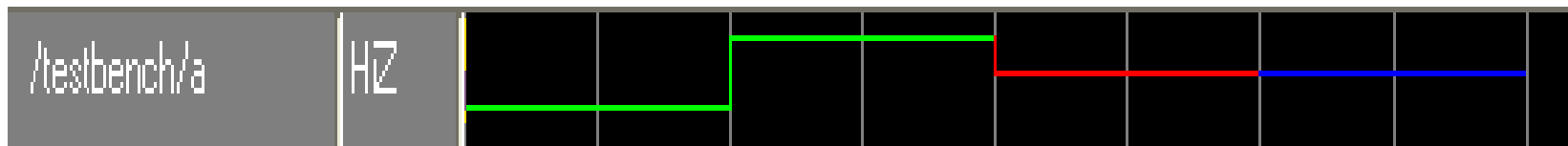
0 = logic 0 (FALSE) = low voltage

1 = logic 1 (TRUE) = high voltage

x = unknown (but must be either 0 or 1)

z = high impedance (disconnected)

A simulator output, showing 0, 1, x, and z.



recap:

A module has:

1. module name
2. ports (port list and port declaration)
3. some design inside

```
module foo (a, b, c, d);  
    input a, b;  
    output c, d;  
  
    // some design here  
  
endmodule
```

vector

- vector is a group of related signals
- declare size at input, output, wire, or reg
- default is 1 wire (1-bit)
- size goes from (n-1) down to 0 for n wires

```
module foo (a, b, c, d);  
    input a;  
    input [3:0] b, c;  
    output [7:0] d;  
  
    reg [7:0] d;  
    ...  
endmodule
```

a is 1 wire (1-bit)
b and c are vectors:
4-wire each
designated
b[3], b[2], b[1], b[0]
and
c[3], c[2], c[1], c[0]

vector

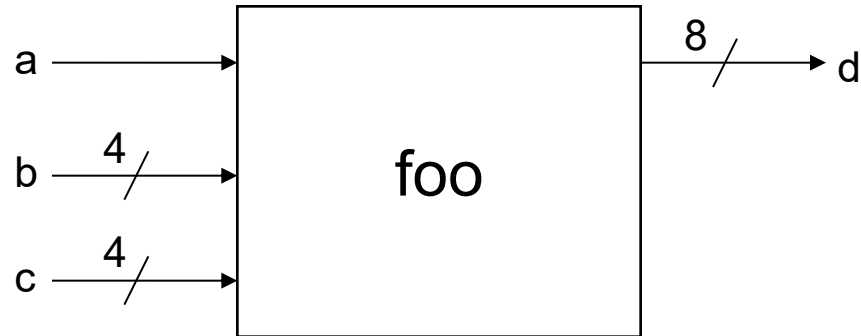
- vector is a group of related signals
- declare size at input, output, wire, or reg
- default is 1 wire (1-bit)
- size goes from (n-1) down to 0 for n wires

```
module foo (a, b, c, d);  
    input a;  
    input [3:0] b, c;  
    output [7:0] d;  
  
    reg [7:0] d;  
    ...  
endmodule
```

d is 8-bit output

size must match

vector



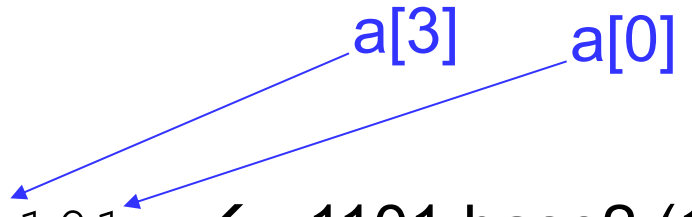
```
module foo (a, b, c, d);  
    input a;  
    input [3:0] b, c;  
    output [7:0] d;  
  
    reg [7:0] d;  
    ...  
endmodule
```

accessing data in vector

Suppose we have

```
input [3:0] a;
```

```
assign a = 4'b1101; ← 1101 base2 (13 base10)
```

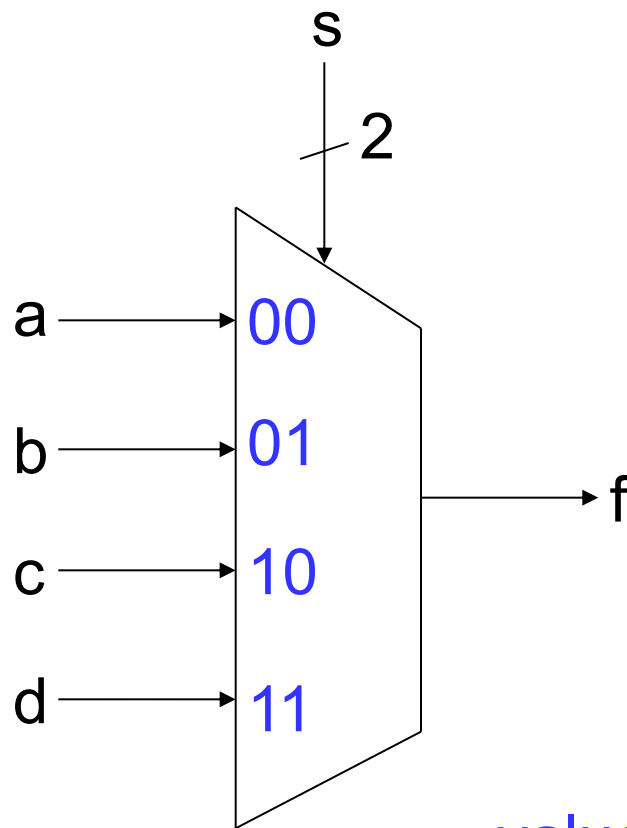


The diagram shows two blue arrows pointing from the text 'a[3]' and 'a[0]' to the first and last bits of the binary value '1101' in the assignment statement. This illustrates how specific bits within a vector are accessed using square brackets.

| we write | what it means | the value |
|----------|--------------------------------|-----------|
| a | the whole a | 1101 (13) |
| a[3:0] | the whole a | 1101 (13) |
| a[3] | rightmost bit | 1 (1) |
| a[1] | 2 nd bit from right | 0 (0) |
| a[2:0] | 3 rightmost bits | 101 (5) |
| a[3:2] | 2 leftmost bits | 11 (3) |

case statement

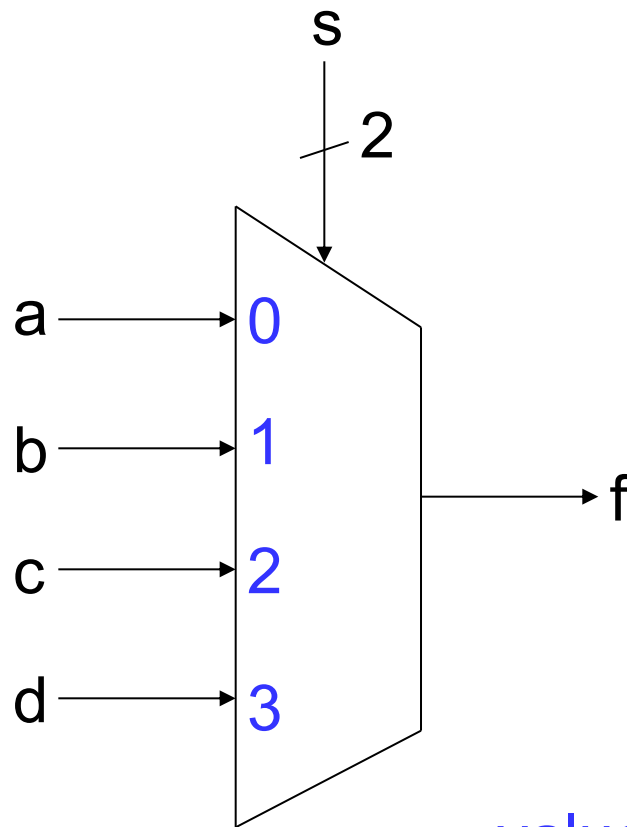
- look at one signal and select what to do based on its value



values of **s** written in binary

case statement

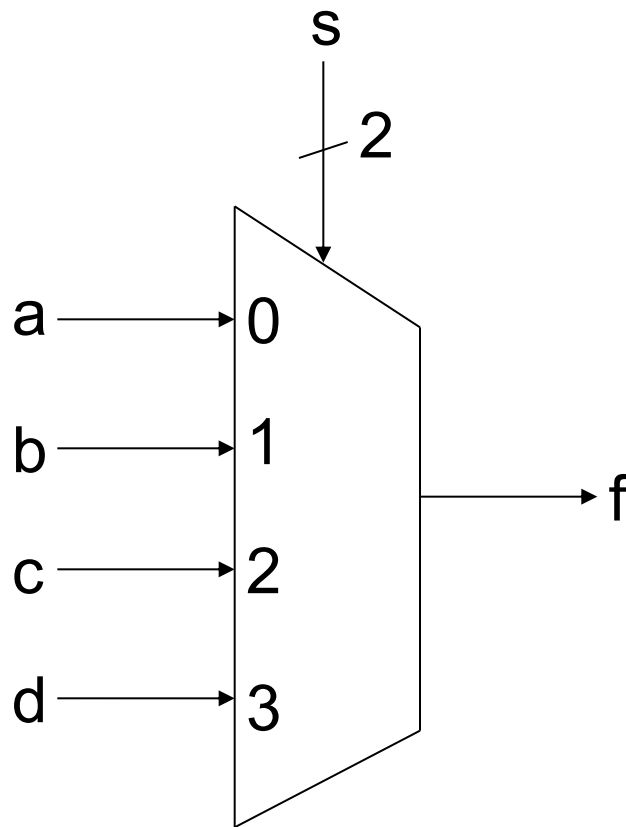
- look at one signal and select what to do based on its values



values of s written in decimal

case statement

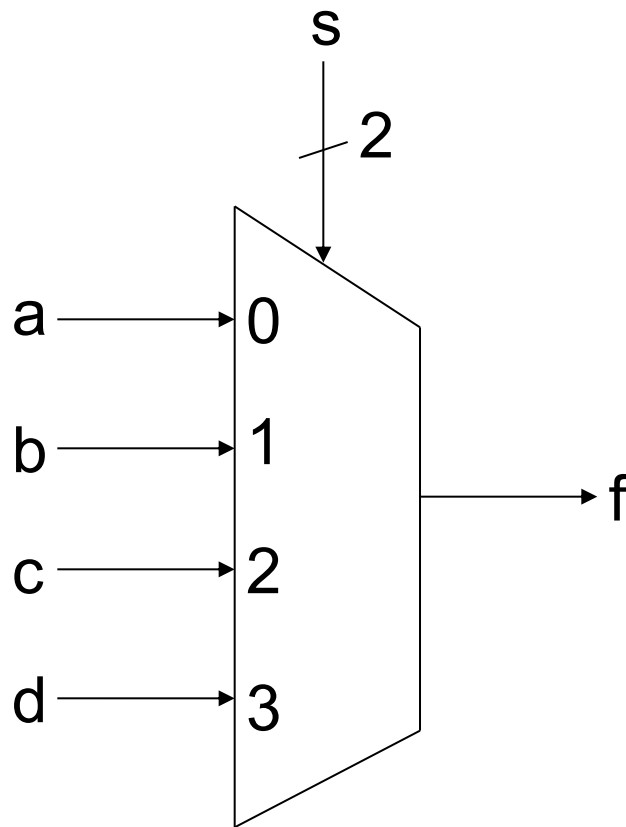
- look at one signal and select what to do based on its values



```
module foo (s, a, b, c, d, f);  
  input [1:0] s;  
  input a, b, c, d;  
  output f;  
  
  reg f;  
  
  ...  
  
endmodule
```

case statement

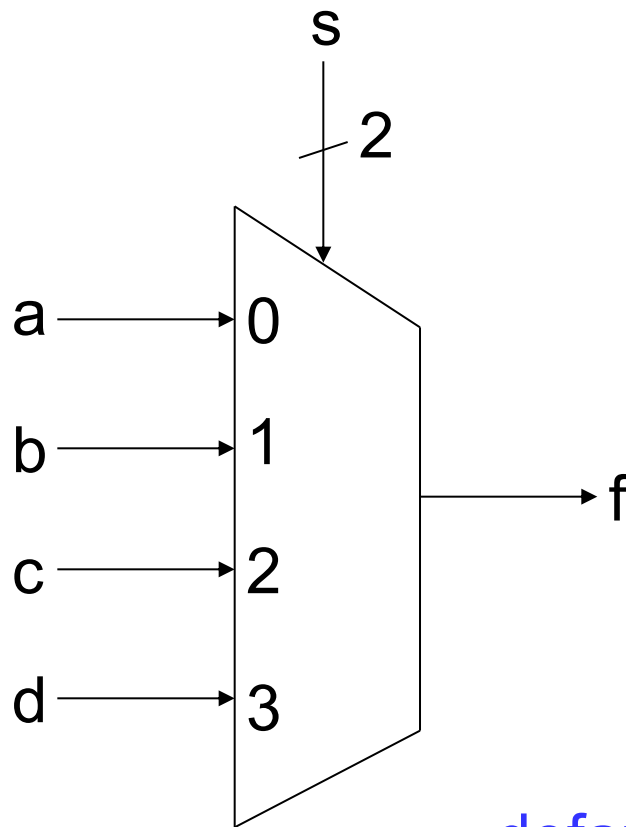
- look at one signal and select what to do based on its values



```
...  
reg f;  
  
always@(s or a or b or c or d)  
    case (s)  
        0: f = a;  
        1: f = b;  
        2: f = c;  
        3: f = d;  
    endcase  
  
...
```

case statement

- look at one signal and select what to do based on its values



```
...  
reg f;  
  
always@(s or a or b or c or d)  
    case (s)  
        0: f = a;  
        1: f = b;  
        2: f = c;  
        default: f = d;  
    endcase  
  
...
```

default means “none of the above”

begin-end pair

- group multiple statements into one statement
- similar to { ... } in C, but use *begin* and *end* instead

Example: formal Verilog syntax is: `always @ (...)`
`<one statement>`

This is ok: `always @ (a or b)`

`f = a & b;`

 1 statement

This is not ok: `always @ (a or b)`

`f = a & b;
g = a | b;`

 2 statements

begin-end pair

- group multiple statements into one statement
- similar to { ... } in C, but use *begin* and *end* instead

This is not ok:

```
always @ (a or b)
```

```
    f = a & b;
```

```
    g = a | b;
```

2 statements

This is ok:

```
always @ (a or b)
```

```
begin
```

```
    f = a & b;
```

```
    g = a | b;
```

```
end
```

1 statement

begin-end pair

- group multiple statements into one statement
- similar to { ... } in C, but use *begin* and *end* instead

This is not ok:

```
if (s == 1)
    f = 0; g = 1;
else
    f = 1; g = 0;
```

2 statements

2 statements

This is ok:

```
if (s == 1)
    begin f = 0; g = 1; end
else
    begin f = 1; g = 0; end
```

1 statement

1 statement

modeling combinational logic

2 important rules:

1. NEVER mix input and output
2. output must be assigned for every possible input

modeling combinational logic

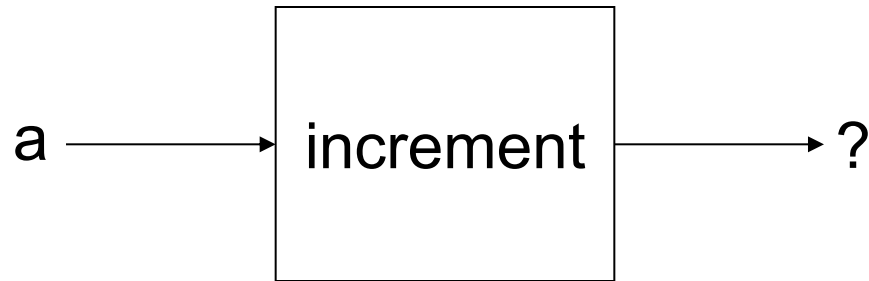
2 important rules:

1. NEVER mix input and output

```
always @ (a)
```

```
    a = a + 1;
```

← NO!!!



modeling combinational logic

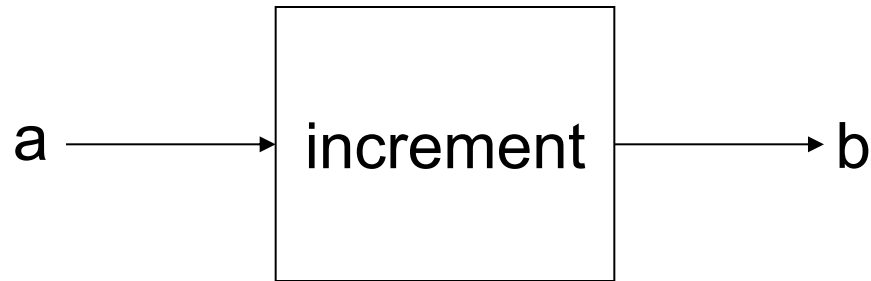
2 important rules:

1. NEVER mix input and output

```
always @ (a)
```

```
    b = a + 1;
```

OK



modeling combinational logic

2 important rules:

2. output must be assigned for every possible input

```
always @ (s or a)  
    if (s == 1)  
        f = a;
```

← NO!!!



modeling combinational logic

2 important rules:

2. output must be assigned for every possible input

```
always @ (s or a)
    if (s == 1)
        f = a;
```

← if s is 0, then what is f?



modeling combinational logic

2 important rules:

2. output must be assigned for every possible input

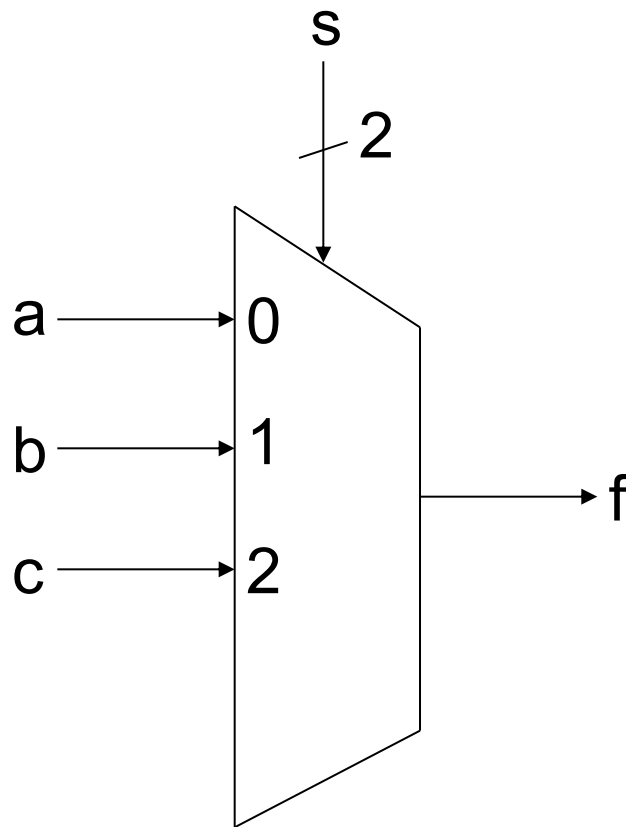
```
always @ (s or a)
    if (s == 1)
        f = a;
    else
        f = ~a;
```

← ok. whatever s or a are,
f will get some value.



modeling combinational logic

2. output must be assigned for every possible input

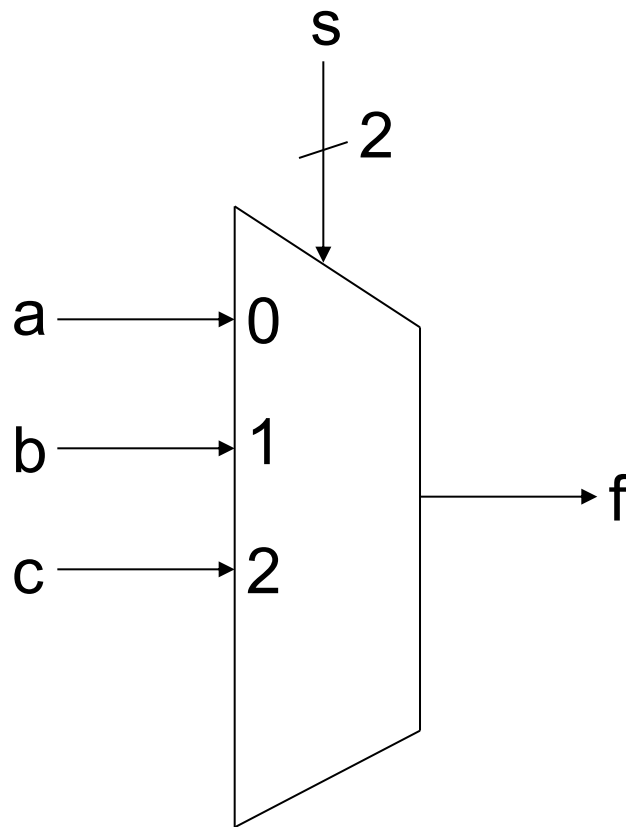


```
...  
reg f;  
  
always@(s or a or b or)  
    case (s)  
        0: f = a;  
        1: f = b;  
        2: f = c;  
    endcase  
...
```

NO. s is 2-bit, so it has 4 possible values (0, 1, 2, 3). f is not assigned when s is 3.

modeling combinational logic

2. output must be assigned for every possible input



```
...  
reg f;  
  
always@(s or a or b or)  
    case (s)  
        0: f = a;  
        1: f = b;  
        default: f = c;  
    endcase  
...
```

ok. if s is 2 or 3, f is c

What if I break the rules?

- simulation may run ok but you cannot get correct hardware after synthesis
- which means your design is useless.