Verilog Wiring Tips

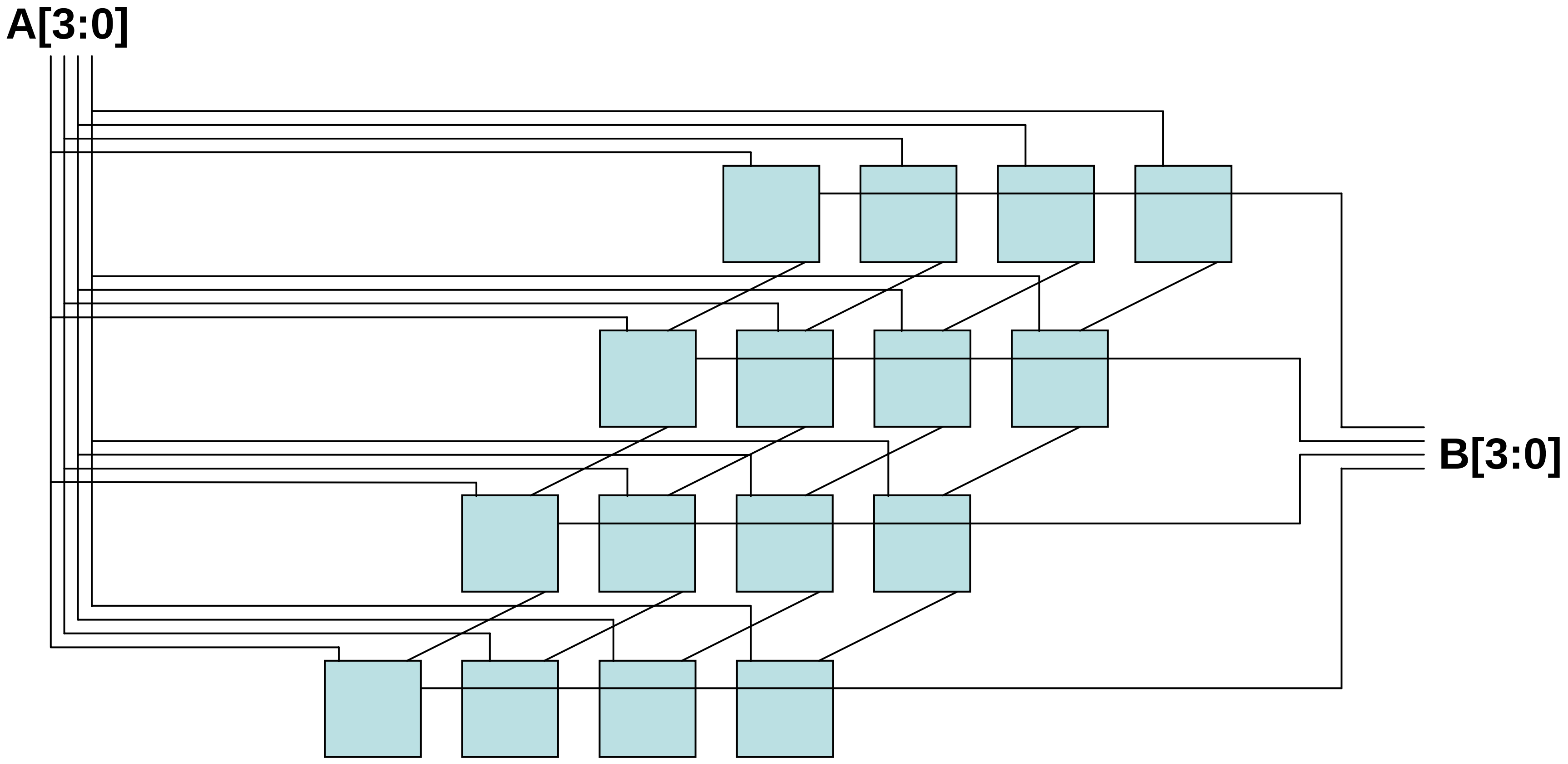
EE371

Omid Azizi

The Problem: A Wiring Mess

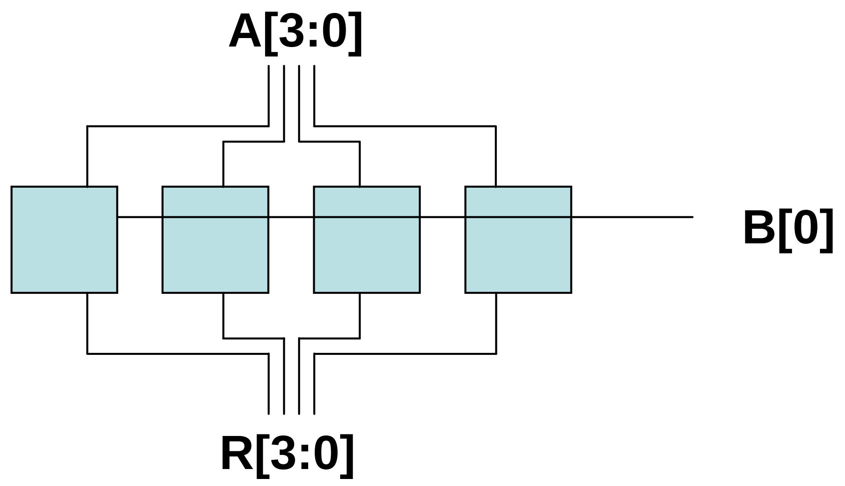
Goal: We want to wire up the following structure

- Its a made-up example, but similar to a multiplier array
- Whoa! A lot of work (even for 4 bit by 4 bit)



Wiring a Row

Lets break this down
- take only one row



METHOD 1 - MANUAL

```
module row(A, B, R)
  input  [3:0] A;
  input        B;
  output [3:0] R;

  block b1 (A[3], B, R[3]);
  block b2 (A[2], B, R[2]);
  block b3 (A[1], B, R[1]);
  block b4 (A[0], B, R[0]);
endmodule
```

METHOD 2 - VERILOG

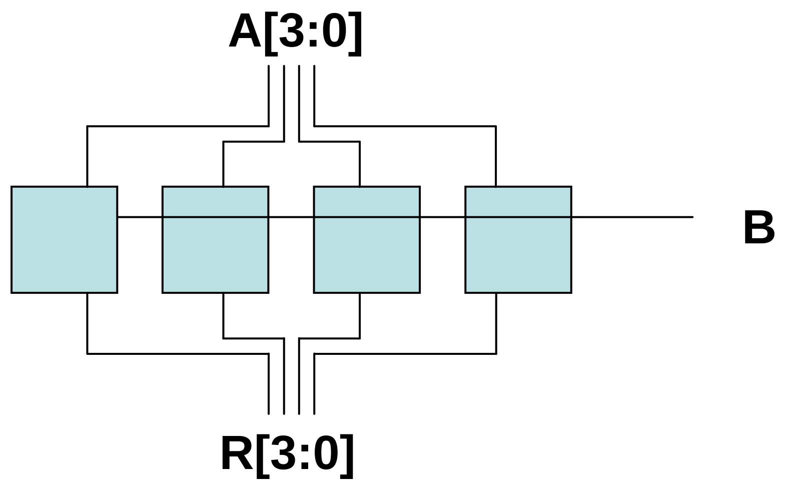
```
module row(A, B, R)
  input  [3:0] A;
  input        B;
  output [3:0] R;

  block b [3:0] (A, B, R);
endmodule
```

Wiring a Row

What happened?

We instantiated all 4 blocks at once

But how did the connections (wiring) work?



Instantiate 4 blocks

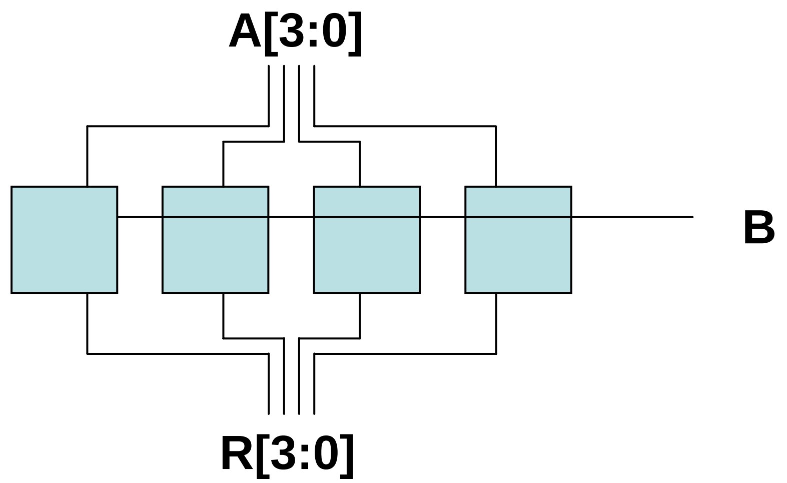
METHOD 2 - VERILOG

```
module row(A, B, R)
  input  [3:0] A;
  input        B;
  output [3:0] R;

  block b [3:0] (A, B, R);
endmodule
```

Automatic Connections

CASE 1: input port expects N-bit wire and you provide N-bit wire

Then, same signal goes to all blocks



METHOD 2 - VERILOG

```
module row(A, B, R)
  input  [3:0] A;
  input        B;
  output [3:0] R;

  block b [3:0] (A, B, R);
endmodule
```

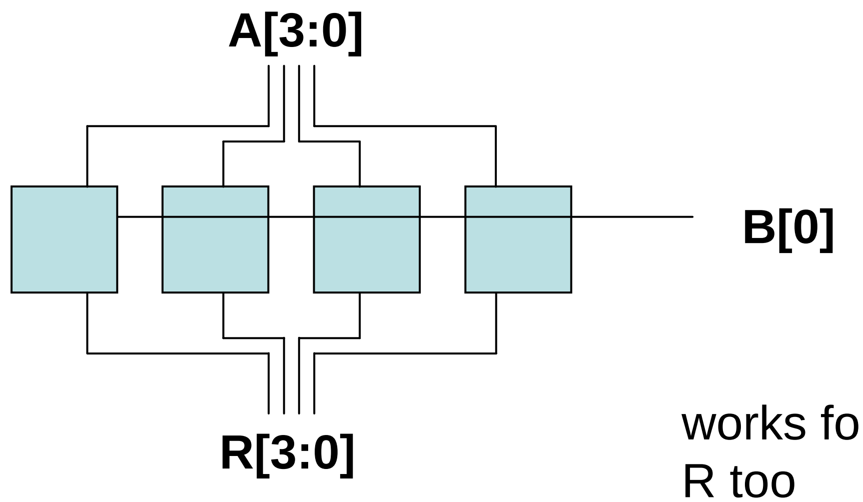
signal B is 1-bit
block expects 1-bit input

So, B is connected to all blocks
(as in diagram)

Automatic Connections

CASE 2: input port expects N-bit wire and you provide (M*N)-bit wire (M is number of blocks)

- Then, block 0 gets first N signals, block 1 gets next N signals, etc...



works for R too

METHOD 2 - VERILOG

```
module row(A, B, R)
  input  [3:0] A;
  input        B;
  output [3:0] R;

  block b [3:0] (A, B, R);
endmodule
```

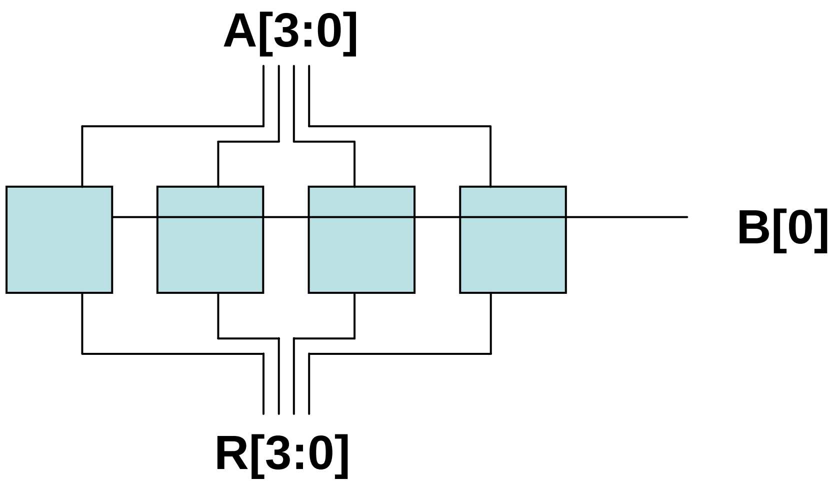
signal A is 4-bit
block expects 1-bit input

Bits are stripped off from A as connections are made.
So block 0 gets A[0], block 1 gets A[1], ...

Automatic Connections

CASE 3: neither case 1 nor 2 apply

Then, Error!



```
METHOD 2 - VERILOG
module row(A, B, R)
  input  [6:0] A;
  input        B;
  output [3:0] R;

  block b [3:0] (A, B, R);
endmodule
```

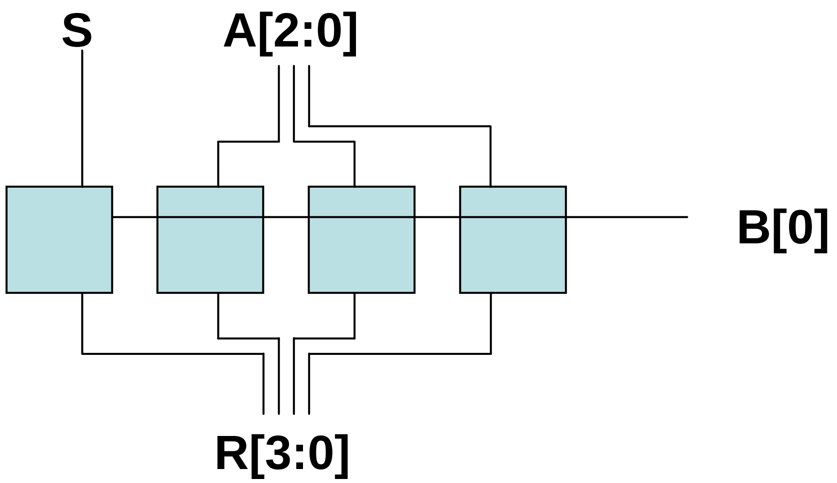
assume signal A was 6-bit
block still expects 1-bit input

Then Verilog wouldn’t know how to distribute A to the blocks...error!

Automatic Connections

What if you want to send a signal that is not a bus?

Then, make it a bus!



```
module row(A, S, B, R)
  input  [2:0] A;
  input        S;
  input        B;
  output [3:0] R;

  wire [3:0] A2 = {S,A};

  block b [3:0] (A2, B, R);
endmodule
```

Make a 4-bit wire so connections are made automatically

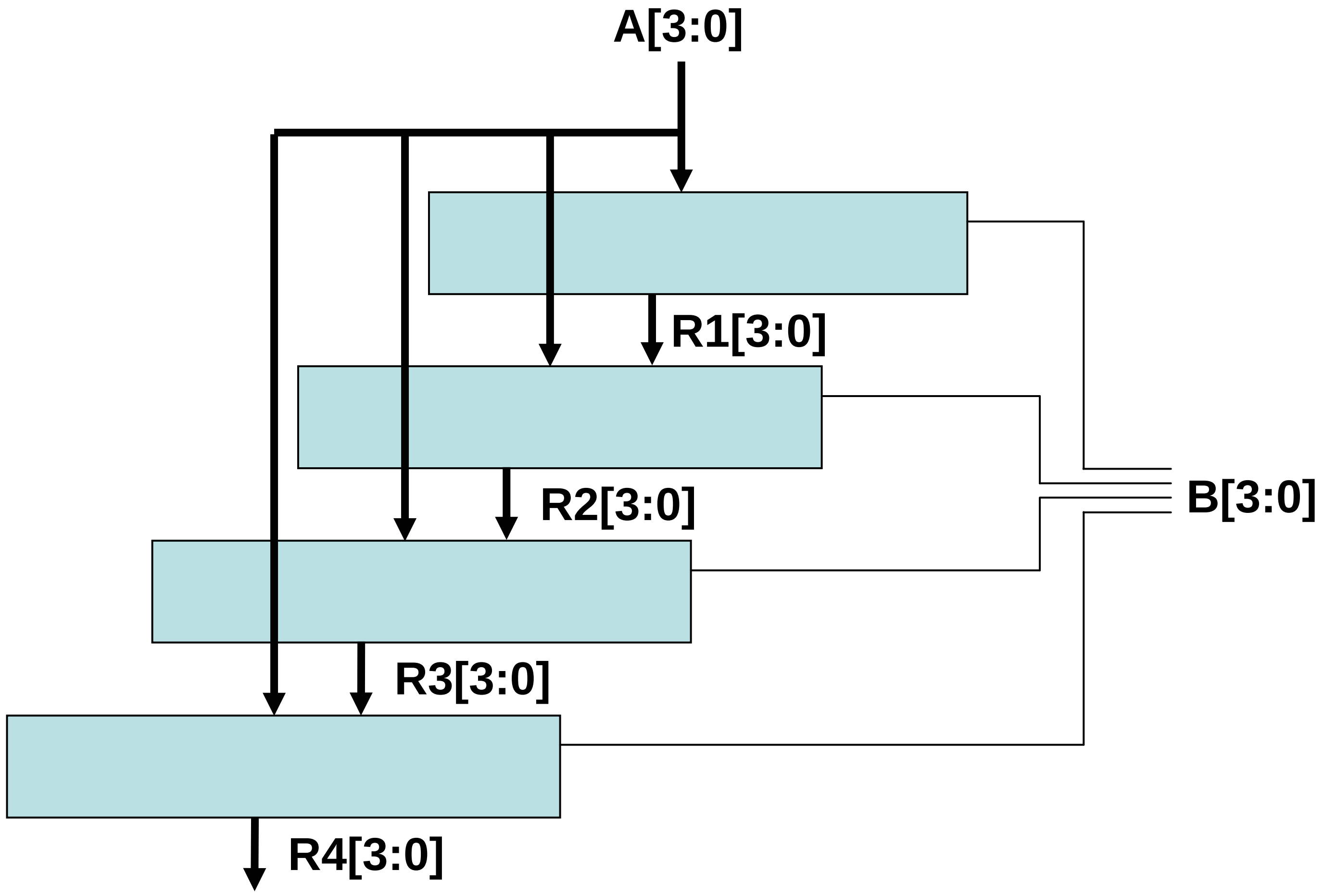
Automatic Connections

What you have learned up to now is very simple, but can save you a lot of time

- You have to make a 32x32 multiplier...
- a lot of instantiations and a lot wires

Taking the example further

We know how to make a row, now we have to make an array



A First Try

```
module row(A, B, Rin, Rout)
  input  [3:0] A;
  input        B;
  input  [3:0] Rin;
  output [3:0] Rout;

  block b [3:0] (A, B, Rin, Rout);
endmodule
```

essentially same row as before, except now we also take the result of the previous row (Rin).

```
module array(A, B, R)
  input  [3:0] A;
  input  [3:0] B;
  output [3:0] R;

  row array_row [3:0] (A, B, Rin, Rout);
  assign R = Rout;
endmodule
```

Is it this easy now?
What is wrong?

A First Try

```
module row(A, B, Rin, Rout)
  input  [3:0] A;
  input        B;
  input  [3:0] Rin;
  output [3:0] Rout;

  block b [3:0] (A, B, Rin, Rout);
endmodule


module array(A, B, R)
  input  [3:0] A;
  input  [3:0] B;
  output [3:0] R;

  row array_row [3:0] (A, B, Rin, Rout);
  assign R = Rout;
endmodule
```

A and B are hooked up correctly. All of A goes to each row (case 1). B gets distributed (case 2).

A First Try

```
module row(A, B, Rin, Rout)
  input  [3:0] A;
  input        B;
  input  [3:0] Rin;
  output [3:0] Rout;

  block b [3:0] (A, B, Rin, Rout);
endmodule


module array(A, B, R)
  input  [3:0] A;
  input  [3:0] B;
  output [3:0] R;

  row array_row [3:0] (A, B, Rin, Rout);
  assign R = Rout;
endmodule
```

The R connections don’t make any sense at all.

We want connections between array_rows.

This is something new. Its not obvious how to do this. Could just give up and instantiate all rows manually, or...

A Trick

```
module array(A, B, R)
  input  [3:0] A;
  input  [3:0] B;
  output [3:0] R;

  wire [15:0] Rin;
  wire [15:0] Rout;

 // Make Rin of one row
 // come from Rout of previous row
 // This is done with a subtle shift

  assign Rin[3:0] = 4’b0000;
  assign Rin[15:4] = Rout[11:0];

  row array_row [3:0] (A, B, Rin, Rout);

  assign R = Rout[15:12];
endmodule
```

Rin of first row is just zeros.

Make a relationship between Rin and Rout. Rin of one row is Rout of previous row.

Output is Rout of last row

A Trick

```
module array(A, B, R)
  input  [3:0] A;
  input  [3:0] B;
  output [3:0] R;

  wire [15:0] Rin;
  wire [15:0] Rout;

  // Make Rin of one row
  // come from Rout of previous row
  // This is done with a subtle shift

  assign Rin[3:0] = 4’b0000;
  assign Rin[15:4] = Rout[11:0];

  row array_row [3:0] (A, B, Rin, Rout);

  assign R = Rout[15:12];
endmodule
```

This works. And code is short.

What if we wanted to make a 64x64-bit array?
- code is just as short!

Some Final Words

- The Verilog 2001 standard has generate statements (like VHDL):

```
generate
  genvar i;
  for (i=0; i <= 3; i=i+1) begin : u
    row array_row (A, B, R[i*4+3:i*4], R[(i+1)*4+3:(i+1)*4]);
  end
endgenerate
```

- Think of it as a preprocessor that does automatic instantiations.
- Support for generate statements may be limited since its a newer standard
- But with generate support, everything you just learned almost becomes unnecessary!