

Introduction to Verilog (Combinational Logic)

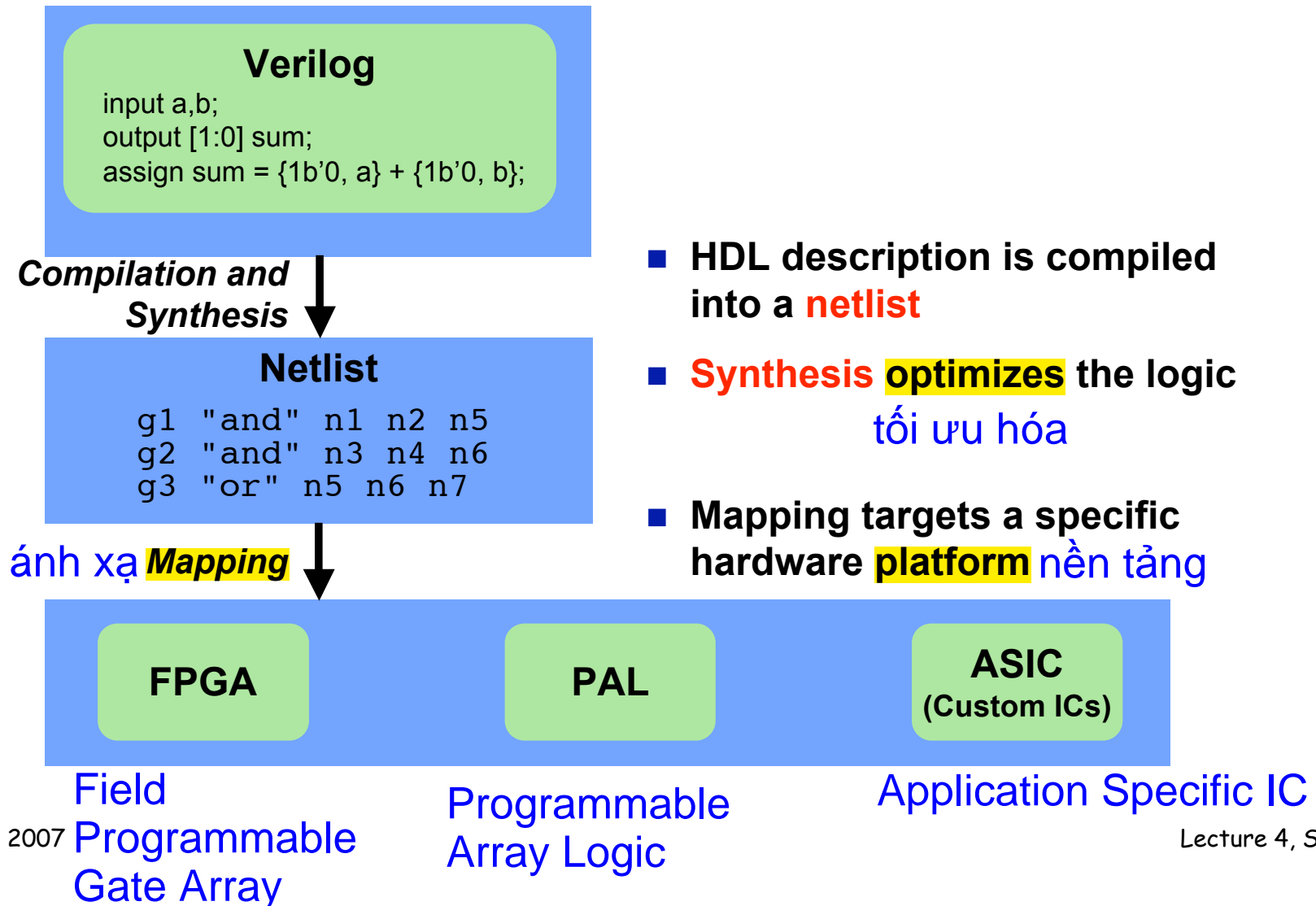
Acknowledgements : Anantha Chandrakasan, Rex Min

Verilog References:

- **Samir Palnitkar, *Verilog HDL*, Pearson Education (2nd edition).**
- **Donald Thomas, Philip Moorby, *The Verilog Hardware Description Language*, Fifth Edition, Kluwer Academic Publishers.**
- **J. Bhasker, *Verilog HDL Synthesis (A Practical Primer)*, Star Galaxy Publishing**

Synthesis and HDLs

- Hardware description language (HDL) is a convenient, device-independent representation of digital logic



Synthesis and Mapping for FPGAs

tập lệnh

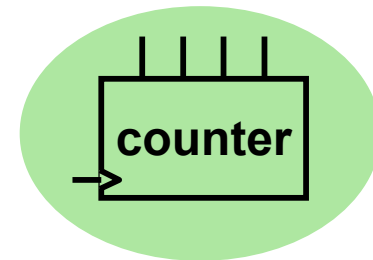
- Infer **macros**: choose the FPGA macros that efficiently **implement** various parts of the HDL code

thực thi

```
...  
always @ (posedge clk)  
begin  
    count <= count + 1;  
end  
...
```

HDL Code

“This section of code looks like a counter. My FPGA has some of those...”

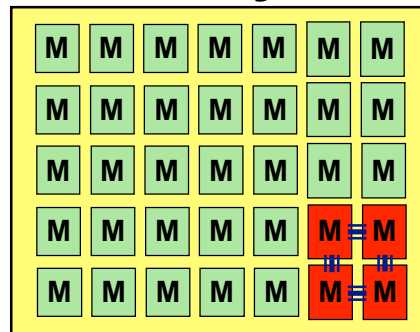


Inferred Macro

hàm ý

- Place-and-route: with area and/or speed in mind, choose the needed macros by location and **route** the interconnect

định tuyến



“This design only uses 10% of the FPGA. Let’s use the macros in one corner to minimize the distance between blocks.”

Verilog: The Module

- Verilog designs consist of interconnected **modules**.
- A module can be an element or collection of lower level design blocks.
- A simple module with combinational logic might look like this:

mạch đa hợp

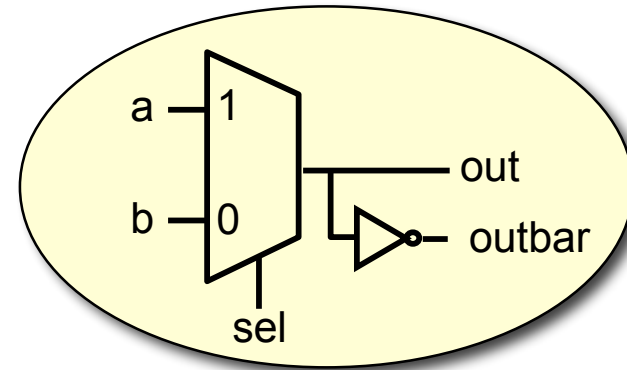
```
module mux_2_to_1(a, b, out,  
                  outbar, sel);
```

```
// This is 2:1 multiplexor
```

```
input a, b, sel;  
output out, outbar;
```

```
assign out = sel ? a : b;  
assign outbar = ~out;
```

```
endmodule
```



$$\text{Out} = \text{sel} \cdot a + \overline{\text{sel}} \cdot b$$

2-to-1 multiplexer with inverted output

Declare and name a module; list its ports. Don't forget that **semicolon**. `";"`

Comment starts with `//`
Verilog skips from `//` to end of the line

Specify each port as input, output, or inout

cách xử lí

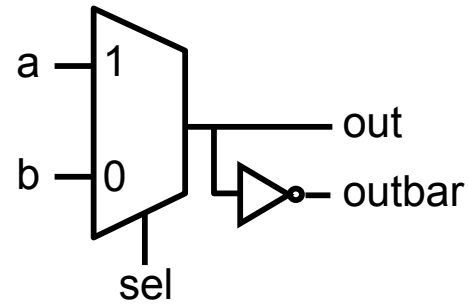
Express the module's **behavior**.
Each statement executes in parallel;
order does not matter.

Conclude the module code.

kết luận

Continuous (Dataflow) Assignment

```
module mux_2_to_1(a, b, out,  
                  outbar, sel);  
  
    input a, b, sel;  
    output out, outbar;  
  
    assign out = sel ? a : b;  
    assign outbar = ~out;  
  
endmodule
```



- Continuous assignments use the assign keyword
- A simple and natural way to represent combinational logic
- Conceptually, the right-hand expression is continuously evaluated as a function of **arbitrarily**-changing inputs...just like **dataflow** *luồng dữ liệu* *tùy ý*
- The target of a continuous assignment is a net driven by combinational logic
- Left side of the assignment must be a **scalar** *vô hướng* or vector net or a **concatenation** *ghép nối* of scalar and vector nets. It can't be a scalar or vector register (*discussed later*). Right side can be register or nets
- Dataflow operators are fairly low-level:
 - Conditional operator: (conditional_expression) ? (value-if-true) : (value-if-false);
 - Boolean logic: ~, &, |, ^
 - Arithmetic: +, -, *
- Nested conditional operator (4:1 mux)
 - assign out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0);

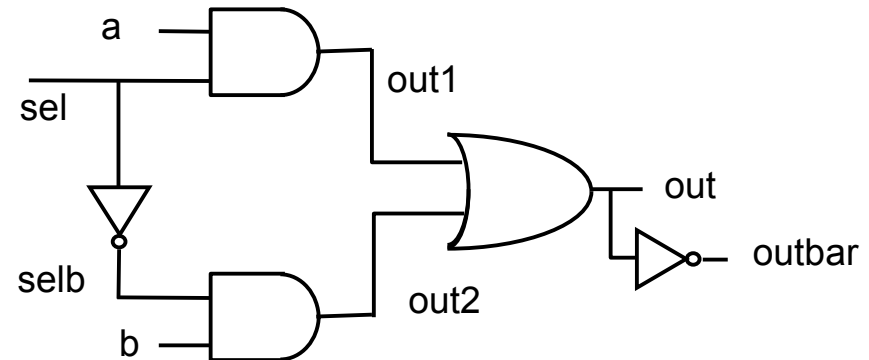
Gate Level Description

```

module muxgate (a, b, out, outbar, sel);
  input a, b, sel;
  output out, outbar;
  wire out1, out2, selb;

  and a1 (out1, a, sel);
  not i1 (selb, sel);
  and a2 (out2, b, selb);
  or o1 (out, out1, out2);
  assign outbar = ~out;
endmodule

```



- Verilog supports basic logic gates as **primitives**
 - and, nand, or, nor, xor, xnor, not, **buf**
 - can be extended to multiple inputs: e.g., nand nand3in (out, in1, in2,in3);
 - bufif1 and bufif0 are tri-state buffers
- Net represents connections between hardware elements. Nets are declared with the keyword **wire**.

| buf | | |
|-----|--|---|
| A | | Y |
| 0 | | 0 |
| 1 | | 1 |

Procedural Assignment with always

- Procedural assignment allows an alternative, often higher-level, behavioral description of combinational logic
- Two structured procedural statements: `initial` and `always`
- Supports richer, C-like control structures such as `if`, `for`, `while`, `case`

```
module mux_2_to_1(a, b, out,
                  outbar, sel);
    input a, b, sel;
    output out, outbar;
```

Exactly the same as before.

```
    reg out, outbar;
```

Anything assigned in an `always` block must *also* be declared as type `reg` (next slide)

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

Conceptually, the `always` block runs *once* whenever a signal in the **sensitivity list** changes value

```
    begin
```

```
        if (sel) out = a;
        else out = b;

        outbar = ~out;
```

Statements within the `always` block are executed **sequentially**. Order matters! tuần tự

```
    end
```

Surround multiple statements in a single `always` block with `begin/end`.

```
endmodule
```

Verilog Registers

- In digital design, registers represent memory elements (we will study these in the next few lectures)
- Digital registers need a clock to operate and update their state on certain phase or edge
- Registers in Verilog should not be confused with hardware registers
- In Verilog, the term register (`reg`) simply means a variable that can hold a value
- Verilog registers don't need a clock and don't need to be driven like a net. Values of registers can be changed anytime in a simulation by assigning a new value to the register

Mix-and-Match Assignments

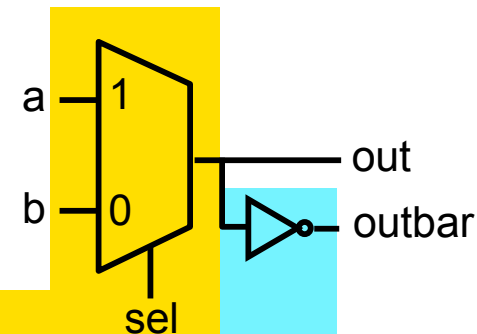
- Procedural and continuous assignments can (and often do) co-exist within a module
- Procedural assignments update the value of `reg`. The value will remain unchanged till another procedural assignment updates the variable. This is the main difference with continuous assignments in which the right hand expression is constantly placed on the left-side

```
module mux_2_to_1(a, b, out,  
                  outbar, sel);  
  input a, b, sel;  
  output out, outbar;  
  reg out;
```

```
  always @ (a or b or sel)  
  begin  
    if (sel) out = a;  
    else out = b;  
  end
```

```
  assign outbar = ~out;
```

```
endmodule
```



*procedural
description*

*continuous
description*

The case Statement

có thể thay thế cho nhau

- case and if may be used **interchangeably** to implement conditional execution within always blocks
- case is easier to read than a long string of if...else statements

```
module mux_2_to_1(a, b, out,  
                  outbar, sel);  
    input a, b, sel;  
    output out, outbar;  
    reg out;  
  
    always @ (a or b or sel)  
    begin  
        if (sel) out = a;  
        else out = b;  
    end  
  
    assign outbar = ~out;  
  
endmodule
```

*Note: Number specification notation: <size>'<base><number>
(4'b1010 if a 4-bit binary value, 16'h6cda is a 16 bit hex number, and
8'd40 is an 8-bit decimal value)*

```
module mux_2_to_1(a, b, out,  
                  outbar, sel);  
    input a, b, sel;  
    output out, outbar;  
    reg out;  
  
    always @ (a or b or sel)  
    begin  
        case (sel)  
            1'b1: out = a;  
            1'b0: out = b;  
        endcase  
    end  
  
    assign outbar = ~out;  
  
endmodule
```

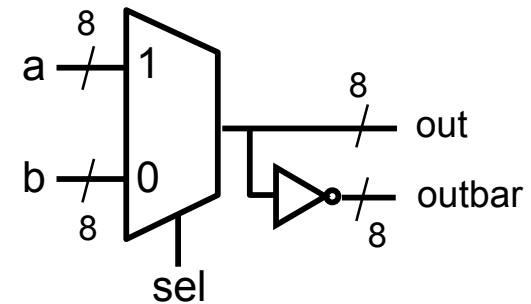
đầy đủ

Note: cases should be **exhaustive** otherwise you'll get unexpected behavior! More about this in a few slides...

The Power of Verilog: n -bit Signals

- Multi-bit signals and buses are easy in Verilog.
- 2-to-1 multiplexer with 8-bit **operands**: toán hạng

```
module mux_2_to_1(a, b, out,  
                  outbar, sel);  
    input[7:0] a, b;  
    input sel;  
    output[7:0] out, outbar;  
    reg[7:0] out;  
    always @ (a or b or sel)  
    begin  
        if (sel) out = a;  
        else out = b;  
    end  
    assign outbar = ~out;  
endmodule
```



Concatenate signals using the **{ }** operator

```
assign {b[7:0],b[15:8]} = {a[15:8],a[7:0]};  
effects a byte swap
```

hoán đổi

The Power of Verilog: Integer Arithmetic

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

```
module add32(a, b, sum);  
    input[31:0] a,b;  
    output[31:0] sum;  
    assign sum = a + b;  
endmodule
```

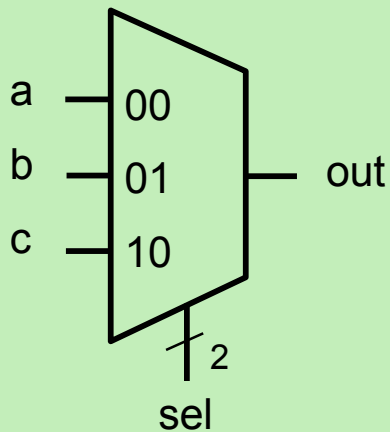
- A 32-bit adder with carry-in and carry-out:

```
module add32_carry(a, b, cin, sum, cout);  
    input[31:0] a,b;  
    input cin;  
    output[31:0] sum;  
    output cout;  
    assign {cout, sum} = a + b + cin;  
endmodule
```

Dangers of Verilog: Incomplete Specification

đặc tả

Goal:



3-to-1 MUX

('11' input is a don't-care)

Proposed Verilog Code:

```
module maybe_mux_3to1(a, b, c,
                      sel, out);

    input [1:0] sel;
    input a,b,c;
    output out;
    reg out;

    always @(a or b or c or sel)
    begin
        case (sel)
            2'b00: out = a;
            2'b01: out = b;
            2'b10: out = c;
        endcase
    end
endmodule
```

Is this a 3-to-1 multiplexer? NO

Incomplete Specification Infers Latches

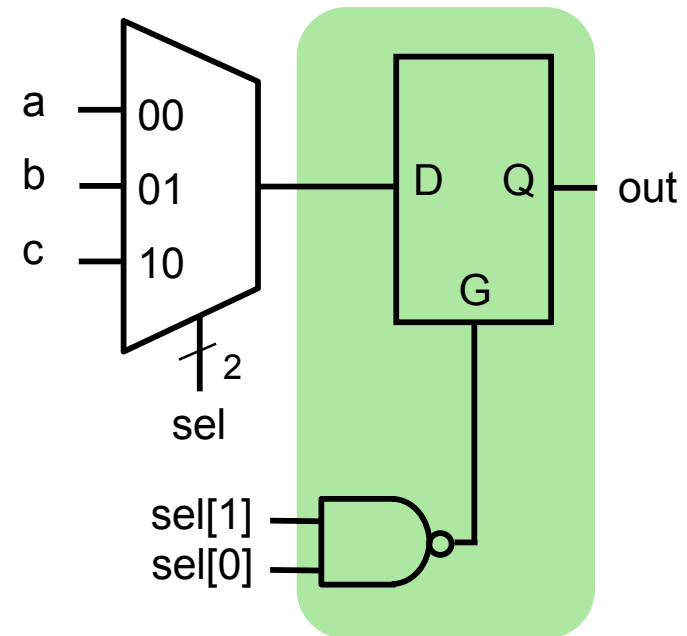
chốt

```
module maybe_mux_3to1(a, b, c,  
                      sel, out);  
  
  input [1:0] sel;  
  input a,b,c;  
  output out;  
  reg out;  
  
  always @(a or b or c or sel)  
  begin  
    case (sel)  
      2'b00: out = a;  
      2'b01: out = b;  
      2'b10: out = c;  
    endcase  
  end  
endmodule
```

if out is not assigned
during any pass through
the always block, then **the
previous value must be
retained!**

giữ lại

Synthesized Result:



- Latch memory “latches” old data when G=0 (we will discuss latches later)
- In practice, we almost *never* intend this

Avoiding Incomplete Specification

- Precede all conditionals with a default assignment for all signals assigned within them...

```
always @(a or b or c or sel)
begin
    out = 1'bx;
    case (sel)
        2'b00: out = a;
        2'b01: out = b;
        2'b10: out = c;
    endcase
end
endmodule
```

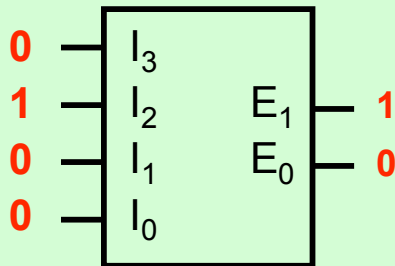
```
always @(a or b or c or sel)
begin
    case (sel)
        2'b00: out = a;
        2'b01: out = b;
        2'b10: out = c;
        default: out = 1'bx;
    endcase
end
endmodule
```

- ...or, fully specify all ^{nhánh} **branches** of conditionals and assign all signals from all branches
 - For each if, include else
 - For each case, include default

Dangers of Verilog: Priority Logic

Goal:

mạch mã hóa 4-to-2 Binary Encoder



| I_3 | I_2 | I_1 | I_0 | E_1 | E_0 |
|------------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| all others | | | | X | X |

Proposed Verilog Code:

```
module binary_encoder(i, e);
    input [3:0] i;
    output [1:0] e;
    reg e;

    always @(i)
    begin
        if (i[0]) e = 2'b00;
        else if (i[1]) e = 2'b01;
        else if (i[2]) e = 2'b10;
        else if (i[3]) e = 2'b11;
        else e = 2'bxx;
    end
endmodule
```

What is the resulting circuit?

Priority Logic

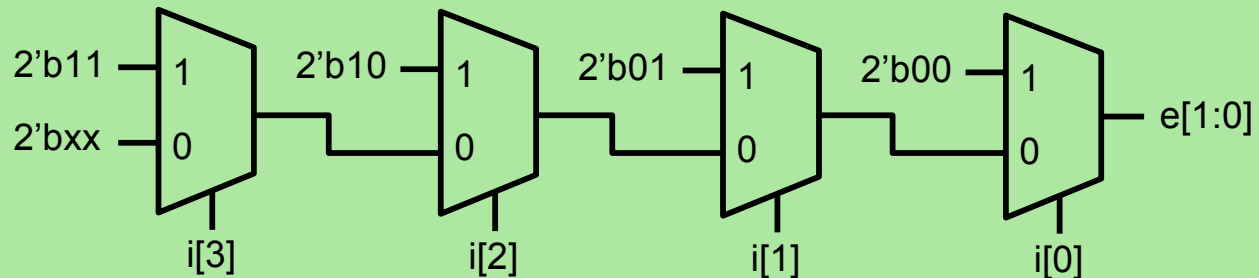
Intent: if more than one input is 1, the result is a don't-care.

| i_3 | i_2 | i_1 | i_0 | E_1 | E_0 |
|------------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| all others | | | | X | X |

Code: if $i[0]$ is 1, the result is 00 regardless of the other inputs.
 $i[0]$ takes the highest priority.

```
if (i[0]) e = 2'b00;
else if (i[1]) e = 2'b01;
else if (i[2]) e = 2'b10;
else if (i[3]) e = 2'b11;
else e = 2'bxx;
end
```

Inferred Result:



diễn giải

- if-else and case statements are **interpreted** very **literally!** Beware of **unintended** **priority logic**.

theo từng chữ

không mong muốn

Avoiding (Unintended) Priority Logic

- Make sure that if-else and case statements are *parallel*

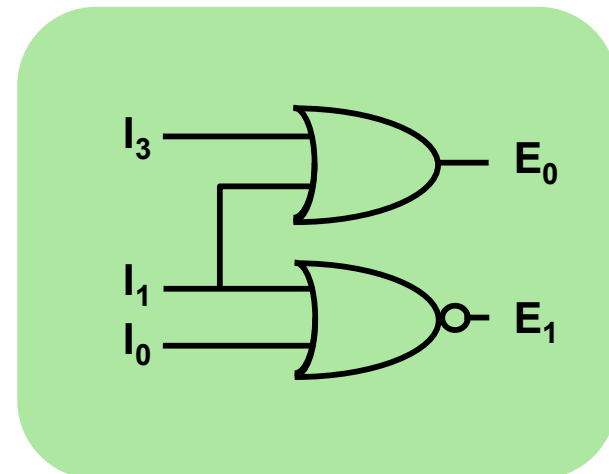
loại trừ lẫn
nhau

- If **mutually exclusive conditions** are chosen for each branch...
- ...then synthesis tool can generate a simpler circuit that evaluates the branches in parallel

Parallel Code:

```
module binary_encoder(i, e);  
  input [3:0] i;  
  output [1:0] e;  
  reg e;  
  
  always @(i)  
  begin  
    if (i == 4'b0001) e = 2'b00;  
    else if (i == 4'b0010) e = 2'b01;  
    else if (i == 4'b0100) e = 2'b10;  
    else if (i == 4'b1000) e = 2'b11;  
    else e = 2'bxx;  
  end  
endmodule
```

Minimized Result:

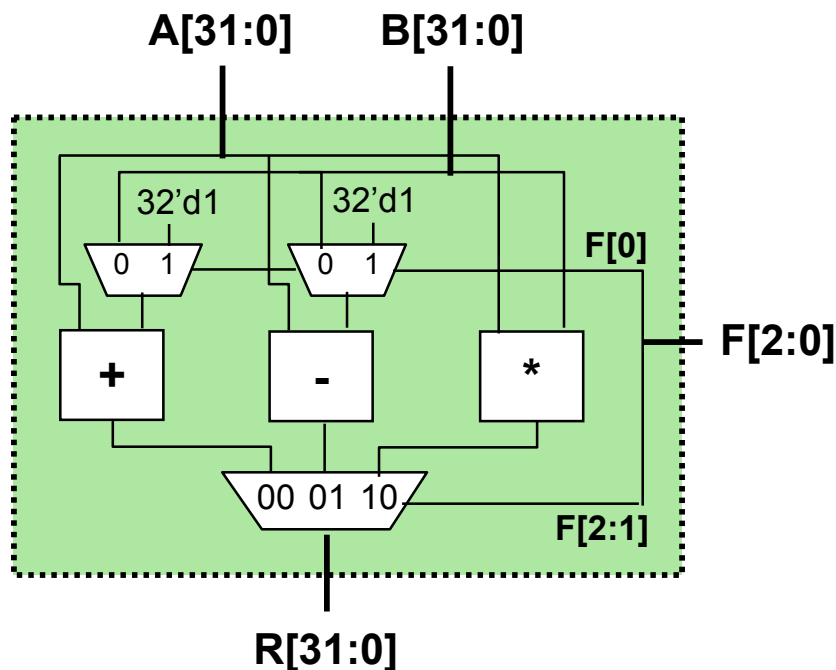


Interconnecting Modules

- Modularity is essential to the success of large designs
- A Verilog module may contain **submodules** that are “wired together”
module con
- High-level primitives enable direct synthesis of behavioral descriptions (functions such as additions, subtractions, shifts (<< and >>), etc.

Arithmetic Logical Unit

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

assign out = sel ? i1 : i0;

endmodule
```

3-to-1 MUX

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

32-bit Adder

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

assign sum = i0 + i1;

endmodule
```

32-bit Subtractor

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

assign diff = i0 - i1;

endmodule
```

16-bit Multiplier

```
module mul16(i0,i1,prod);
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(a, b, f, r);
  input [31:0] a, b;
  input [2:0] f;
  output [31:0] r;

```

```

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

```

```

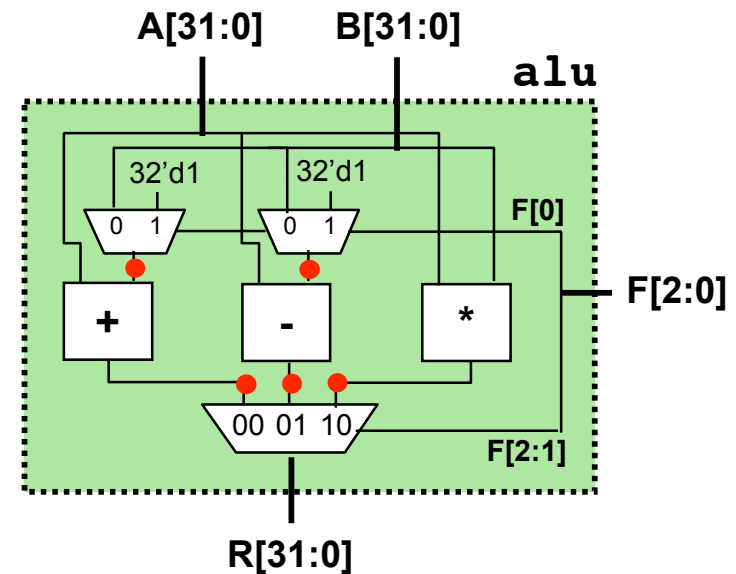
  mux32two      adder_mux(b, 32'd1, f[0], addmux_out);
  mux32two      sub_mux(b, 32'd1, f[0], submux_out);
  add32         our_adder(a, addmux_out, add_out);
  sub32         our_subtractor(a, submux_out, sub_out);
  mul16         our_multiplier(a[15:0], b[15:0], mul_out);
  mux32three    output_mux(add_out,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

```

module
names

(unique)
instance
names

corresponding
wires/regs in
module alu



intermediate output nodes

More on Module Interconnection

rõ ràng

tùy ý

- **Explicit** port naming allows port mappings in **arbitrary order**: better scaling for large, **evolving** designs

suy ra

Given Submodule Declaration:

```
module mux32three(i0,i1,i2,sel,out);
```

Module Instantiation with Ordered Ports:

```
mux32three output_mux(add_out, sub_out, mul_out, f[2:1], r);
```

Module Instantiation with Named Ports:

```
mux32three output_mux(.sel(f[2:1]), .out(r), .i0(add_out),  
                      .i1(sub_out), .i2(mul_out));
```

submodule's
port name

corresponding
wire/reg in
outer module

- Built-in Verilog gate primitives may be instantiated as well

- Instantiations may **omit instance** name and must be ordered:

```
buf(out1,out2,...,outN, in); and(in1,in2,...inN,out);
```

bỏ qua phiên bản

Useful Boolean Operators

- **Bitwise operators** perform bit-sliced operations on vectors
 - $\sim(4'b0101) = \{\sim 0, \sim 1, \sim 0, \sim 1\} = 4'b1010$
 - $4'b0101 \& 4'b0011 = 4'b0001$
- **Logical operators** return one-bit (true/false) results
 - $!(4'b0101) = \sim 1 = 1'b0$
- **Reduction operators** act on each bit of a single input vector
 - $\&(4'b0101) = 0 \& 1 \& 0 \& 1 = 1'b0$
- **Comparison operators** perform a Boolean test on two arguments

Bitwise

| | |
|-------------------|------|
| $\sim a$ | NOT |
| $a \& b$ | AND |
| $a b$ | OR |
| $a \wedge b$ | XOR |
| $a \sim \wedge b$ | XNOR |

Logical

| | |
|------------|-----|
| $!a$ | NOT |
| $a \&\& b$ | AND |
| $a b$ | OR |

Reduction

| | |
|-----------|------|
| $\&a$ | AND |
| $\sim \&$ | NAND |
| $ $ | OR |
| $\sim $ | NOR |
| \wedge | XOR |

Comparison

| | |
|--|--|
| $a < b$ $a > b$ $a \leq b$ $a \geq b$ | Relational |
| $a == b$ $a != b$ | [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 |

quan hệ

Note distinction between $\sim a$ and $!a$

Summary

- **Multiple levels of description: behavior, dataflow, logic and switch (not used in 6.111)**
- **Gate level is typically not used as it requires working out the interconnects**
- **Continuous assignment using `assign` allows specifying dataflow structures**
- **Procedural Assignment using `always` allows efficient behavioral description. Must carefully specify the sensitivity list**
- **Incomplete specification of `case` or `if` statements can result in non-combinational logic**
- **Verilog registers (`reg`) is not to be confused with a hardware memory element**
- **Modular design approach to manage complexity**