# Tham khảo Verilog HDL



## Nội dung chính

1. Mô hình hành vi
2. Mô hình hành vi dựa trên phương trình boole
3. Mô hình hành vi vòng
4. Mô hình hành vi cho các khối cơ bản





## Mô hình hành vi

- Khái quát mô hình hành vi
- Kiểu dữ liệu cho mô hình hành vi
- Các phép toán cho mô hình hành vi





## Mô hình cấu trúc và mô hình hành vi trong HDLs

- *Cấu trúc (Structural)* chỉ ra cấu trúc phần cứng thật sự của mạch
  - Mức trừu tượng thấp
    - Các cổng cơ bản (ví dụ and, or, not)
    - Cấu trúc phân cấp thông qua các module
  - Tương tự lập trình hợp ngữ
- *Hành vi (Behavioral)* chỉ ra hoạt động của mạch trên các bit
  - Mức trừu tượng cao hơn
    - Biểu diễn bằng các biểu thức (ví dụ out = (a & b) | c)
- Không phải tất cả các đặc tả hành vi đều tổng hợp được
  - Không nên sử dụng: / %

## Mô hình hành vi – đặc điểm

- Thiết kế các vi mạch lớn
- Mô tả chức năng (what) và cách xây dựng (how) phần cứng
- Không quan tâm đến trễ truyền lan (được quan tâm trong giai đoạn tổng hợp)
- Các bước thiết kế
  - Nhanh chóng đưa ra nguyên mẫu (prototype)
  - Kiểm tra chức năng
  - Dùng công cụ tổng hợp tối ưu và ánh xạ công nghệ





## Kiểu dữ liệu cho mô hình hành vi

- Biến trong Verilog biểu diễn một tín hiệu dạng nhị phân của mạch
- Tất cả các biến trong Verilog được định nghĩa kiểu trước khi sử dụng
  - net
  - register
- Net hoạt động như dây nối vật lý
  - ***wire***
- Register hoạt động giống như biến trong các ngôn ngữ lập trình cấp cao
  - ***reg***
  - ***integer***
- Kích thước mặc định của kiểu dữ liệu ***reg*** và ***wire*** là 1 bit





## Các toán tử trong Verilog

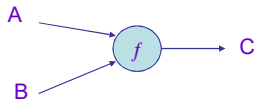
| Toán tử | Tên | Nhóm |
|---|---|---|
| [ ] | Chọn | |
| () | Ngoặc | |
| ! | Phủ định (đảo) | Logical |
| ~ | Phủ định (not) | Bit-wise |
| & | Thu giảm AND | Thu giảm |
| \| | Thu giảm OR | |
| ~& | Thu giảm NAND | |
| ~\| | Thu giảm NOR | |
| ^ | Thu giảm XOR | |
| ~^ or ^~ | Thu giảm XNOR | |
| + | Dấu dương (một ngôi) | Số học |
| – | Dấu âm (một ngôi) | |
| {} | Nối | Nối |
| {{}} | Nhân bản | Nhân bản |
| * | nhân | Số học |
| / | chia | |
| % | Chia lấy dư | |
| + | Cộng (hai ngôi) | |
| – | Trừ (hai ngôi) | |

| Toán tử | Tên | Nhóm |
|---|---|---|
| << | Dịch trái | Dịch |
| >> | Dịch phải | |
| > | Lớn hơn | Quan hệ |
| >= | Lớn hơn hay bằng | |
| < | Nhỏ hơn | |
| <= | Nhỏ hơn hay bằng | |
| == | Bằng (logic) | So sánh bằng |
| != | Không bằng (logic) | |
| === | Bằng (case) | |
| !== | Không bằng (case) | |
| & | bit-wise AND | Bit-wise |
| ^ | bit-wise XOR | |
| ^~ or ~^ | bit-wise XNOR | |
| \| | bit-wise OR | |
| && | logical AND | Logic |
| \|\| | logical OR | |
| ?: | Điều kiện | Điều kiện |





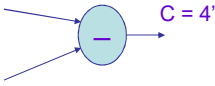
## Toán tử số học (+, -, *, /, %)




- Bất kỳ bit nào trong toán hạng là x hoặc z thì kết quả là x
- Kích thước kết quả
  - Phép nhân thì kích thước kết quả bằng tổng kích thước 2 toán hạng
  - Các phép toán khác bằng chiều dài lớn nhất của toán hạng
- Biểu diễn dấu: MSB

A = 4'b0010 (2)

B = 4'b0101 (5)

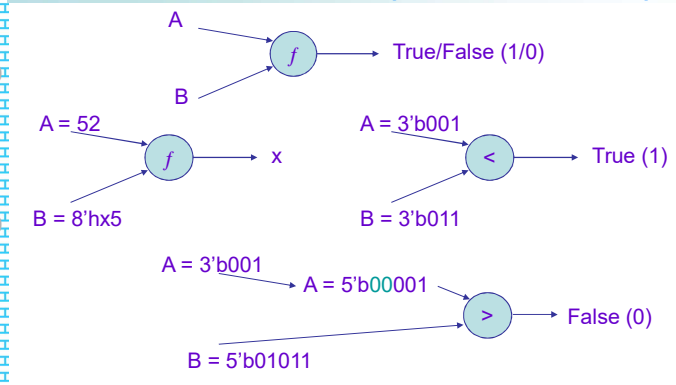



C = 4'b1101 (13)

```
integer intA, intB;
intA = -12;
intB = -'d12;
```

```
intA/3 =         -4 (11111111111111111111111111111100)
intB/3 = 1431655761 (01010101010101010101010101010001)
```

## Toán tử quan hệ (<, >, <=, >=)




- Return **1** if true, return **0** if false.
- "x" is returned if x or z in any operand.



## Toán tử bằng (==, ===, !=, !===)

- Bằng luận lý (== và !=)
  - Giá trị x và z tương tự như toán tử quan hệ
  - Kết quả có thể là x
- Case (=== và !==)
  - So sánh từng bit
  - x === x, z === z, x !== z
  - Kết quả luôn xác định (0 hoặc 1)
- Nếu kích thước 2 toán hạng không bằng nhau thì các bit 0 sẽ được thêm vào những bit trọng số cao của toán hạng có kích thước nhỏ

```
Data = 4'b11x0;
Addr = 4'b11x0;

Data == Addr  //x
Data === Addr //1
```



## Toán tử luận lý (||, &&, !)

- Toán hạng là vector khác 0 được xem như 1
- Nếu bất kỳ bit nào của toán hạng có giá trị x hay z thì toán hạng được xem như x

```
ABus = 4'b0110;
BBus = 4'b0100;

ABus || BBus // 1
ABus && BBus // 1
!ABus        // giống như !BBus
             // 0
```



## Toán tử Bit-wise (&, |, ~, ^, ^~)

| & (and) | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x | x |
| x | 0 | x | x | x |
| z | 0 | x | x | x |

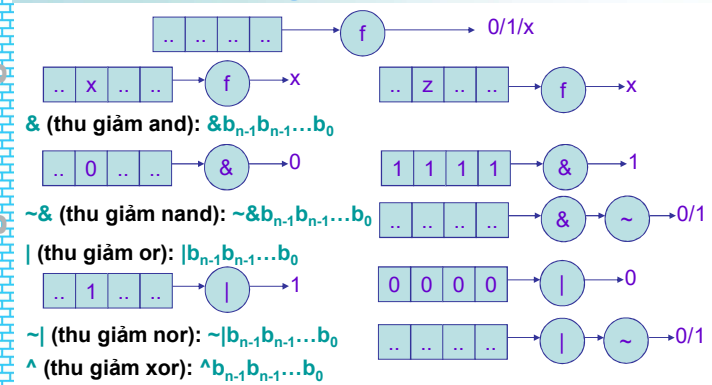
| \| (or) | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 1 | x | x |
| 1 | 1 | 1 | 1 | 1 |
| x | x | 1 | x | x |
| z | x | 1 | x | x |

| ^ (xor) | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 1 | x | x |
| 1 | 1 | 0 | x | x |
| x | x | x | x | x |
| z | x | x | x | x |

| ^~ (xnor) | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 1 | 0 | x | x |
| 1 | 0 | 1 | x | x |
| x | x | x | x | x |
| z | x | x | x | x |

| ~ (not) | 0 | 1 | x | z |
|---|---|---|---|---|
| | 1 | 0 | x | x |

# Toán tử thu giảm




**& (thu giảm and):** $\&b_{n-1}b_{n-1}...b_0$

**~& (thu giảm nand):** $\sim\&b_{n-1}b_{n-1}...b_0$

**| (thu giảm or):** $|b_{n-1}b_{n-1}...b_0$

**~| (thu giảm nor):** $\sim|b_{n-1}b_{n-1}...b_0$

**^ (thu giảm xor):** $\hat{}b_{n-1}b_{n-1}...b_0$

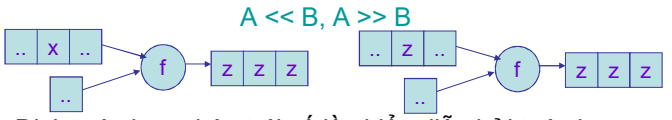
If count($b_i$ = 1) mod 2 == 0,

kết quả 0 Ngược lại kết quả 1

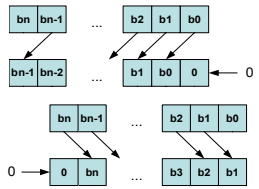
**~^/^~ (thu giảm xnor):** $\sim\hat{}b_{n-1}b_{n-1}...b_0$



# Toán tử dịch (<<, >>)




- Dịch toán hạng bên trái số lần biểu diễn bởi toán hạng bên phải
  - Dịch trái
  - Dịch phải



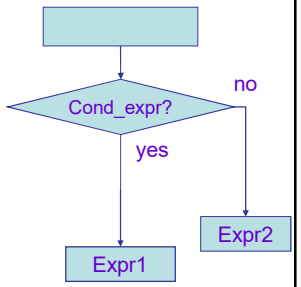

```
reg [0:7] Qreg;

Qreg = 4'b0111;
Qreg >> 2 // is 8'b0000_0001
wire [0:3] DecoderOut = 4'd1 << Address[0:1];
```



# Toán tử điều kiện

```
Cond_expr ? Expr1 : Expr2
```

- Nếu Cond_expr có chứa bất kỳ bit nào là x hoặc z thì kết quả là phép toán bit-wise trên Expr1 và Expr2 như sau
  - 0 ♣ 0 => 0
  - 1 ♣ 1 => 1
  - Trường hợp khác là x
- Toán tử điều kiện có thể được lồng nhau vô tận




```
wire[15:0]bus_a = drive_a ? data : 16'bz;
/* drive_a = 1 thì data được gán vào bus_a
   drive_a = 0 thì bus_a ở tổng trở cao
   drive_a = x thì bus_a là x */
```

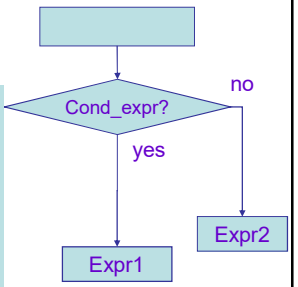


# Toán tử điều kiện

```
Cond_expr ? Expr1 : Expr2
```

- Toán tử điều kiện có thể được lồng nhau vô tận

```
Y = ( A == B ) ? C : D;

wire [1:0] select;
wire [15:0] D1, D2, D3, D4;
wire [15:0] bus = (select == 2'b00) ? D1 :
                  (select == 2'b01) ? D2 :
                  (select == 2'b10) ? D3 :
                  (select == 2'b11) ? D4 : 16'bx
```



## Toán tử kết nối/nhân bản

- Kết nối {expr1, expr2,… , exprN}
  - Những hằng số không biết kích thước không thể thực hiện kết nối

```
wire [7:0] Dbus;
wire [11:0] Abus;

assign Dbus[7:4] = {Dbus[0], Dbus[1], Dbus[2], Dbus[3]};

assign Dbus = {Dbus[3:0], Dbus[7:4]};

{Dbus, 5} // not allowed
```

- Nhân bản {rep_number {expr1, expr2,… , exprN}}

```
Abus = {3{4'b1011}};          // 12'b1011_1011_1011

{3{1'b1}} // 111
{3{Ack}}  // {Ack, Ack, Ack}
```



## Operator Precedence (Độ ưu tiên toán tử

| Operator precedence | Operator symbol |
|---|---|
| Highest | - ! ~ (unary) |
| | * / % |
| | + - (binary) |
| | << >> |
| | < <= > >= |
| | == != === !== |
| | & ~& |
| | ^ ^~ ~^ |
| | \| ~\| |
| | && |
| | \|\| |
| Lowest | ? : |

Dùng dấu ngoặc đơn ( ) để tăng độ ưu tiên



## Nội dung chính

1. Mô hình hành vi
2. **Mô hình hành vi dựa trên phương trình boole**
3. Mô hình hành vi vòng
4. Mô hình hành vi cho các khối cơ bản



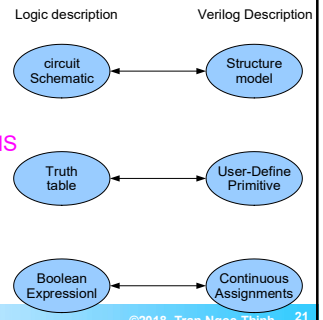
## Mô hình hành vi dựa trên phương trình boole

- Phép gán liên tục
- Thời gian trễ truyền lan và phép gán liên tục
- Latch và mạch tích cực mức trong Verilog

# Phép gán liên tục

- Phép gán liên tục
  - assign variable = boolean_expression
  - wire wire_name = boolean_expression
  - Boolean_expression được tính lại khi bất kỳ tín hiệu nào trong nó thay đổi
- Một module có thể có nhiều phát biểu gán liên tục được thực thi đồng thời

```
input x_in1, x_in2;
output y_out;
        assign y_out = ~(x_in & x_in2);
LHS                          RHS
input x_in1, x_in2;
output y_out;
wire y_out = ~(x_in & x_in2);
```








# Ví dụ - Full Adder

```
module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;
        assign s = x ^ y ^ Cin;
        assign Cout =  x& y | x& Cin | y& Cin;
endmodule
```

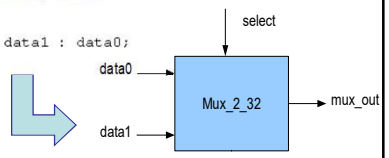
```
module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;
        assign   {Cout, s} = x + y + Cin;
endmodule
```

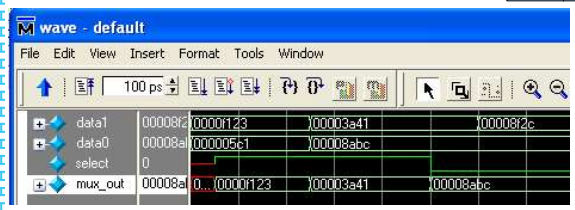




# Ví dụ - Mux_2_32

```
D:/Teaching/Verilog/Verilog/mux_2_32.v
ln #
1   module Mux_2_32 (mux_out, data1, data0, select);
2       parameter world_size = 32;
3       output [world_size - 1:0] mux_out;
4       input [world_size - 1:0] data1, data0;
5       input select;
6
7       assign mux_out = select ? data1 : data0;
8   endmodule
```








# Trễ truyền lan và gán liên tục

- Kết hợp trễ truyền lan với phép gán liên tục để mô phỏng đúng tính năng và đặc tính thời gian như ở mức cổng
- Công cụ tổng hợp tự động tạo ra mạch thật sự dựa trên phép gán

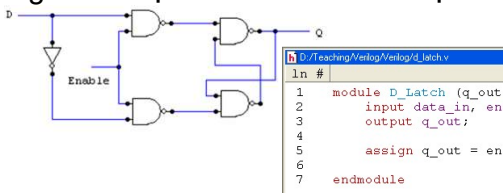
```
input x_in1, x_in2, x_in3;
output y_out;
        wire #1 y_1 = ~(x_in & x_in2);
        wire #1 y_out = ~(y_1|x_in3);
```

# Latch và mạch tích cực

- Tập hợp các phát biểu gán liên tục có thể hồi tiếp (feedback) tín hiệu

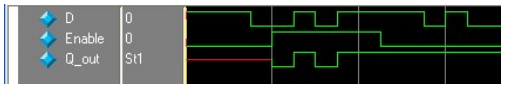
```
assign q = set ~& qbar;
assign qbar = rst ~& q;
```

- Công cụ tổng hợp chỉ hỗ trợ hồi tiếp bằng phép gán liên tục với toán tử điều kiện




D:/Teaching/Verilog/Verilog/d_latch.v

```
ln #
1   module D_Latch (q_out, data_in, enable);
2       input data_in, enable;
3       output q_out;
4
5       assign q_out = enable ? data_in : q_out;
6
7   endmodule
```



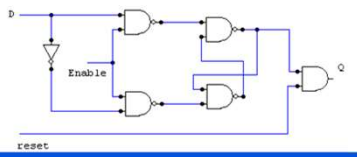


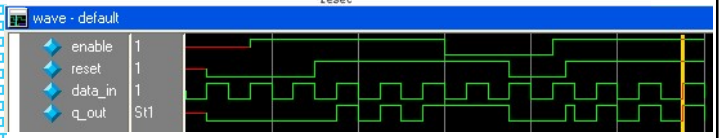



# Ví dụ - Latch với tín hiệu reset

D:/Teaching/Verilog/Verilog/D_latch_reset.v

```
ln #
1   module D_Latch_reset (q_out, data_in, enable, reset);
2       input data_in, enable, reset;
3       output q_out;
4
5       assign q_out = !reset ? 0 : enable ? data_in : q_out;
6
7   endmodule
```




⇨ Mạch phức tạp khó hiện thực bằng hàm boolean





# Nội dung chính

1. Mô hình hành vi
2. Mô hình hành vi dựa trên phương trình boole
3. **Mô hình hành vi vòng**
4. Mô hình hành vi cho các khối cơ bản





# Mô hình hành vi vòng

- Khái quát mô hình hành vi vòng
- Mô hình hành vi vòng cho mạch phát hiện cạnh
- So sánh các loại mô hình hành vi
  - Mô hình phép gán liên tục
  - Mô hình dòng dữ liệu
  - Mô hình dựa trên thuật toán

# Mô hình hành vi vòng – khái quát

- Mô hình hành vi gán liên tục không hiện thực được mạch kích cạnh (edge-sensitive)
- Thực thi những phát biểu thủ tục (procedure)
- Mô hình hành vi vòng
  - Các phát biểu được thực thi tuần tự và quay lại khi thực thi xong phát biểu cuối cùng
  - Thực thi không điều kiện dưới sự điều khiển của một biểu thức

```
always @ (<event_expression>)
begin
   procedural statements
end
```

- Sử dụng trong cả kích cạnh (đồng bộ) và kích mức



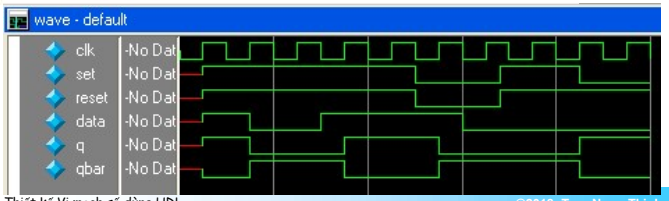
# Ví dụ - D Flip-Flop

D:/Teaching/Verilog/Verilog/df_behav.v

```
ln #
 1  module df_behav (q, qbar, data, set, reset, clk);
 2      input data, set, reset, clk;
 3      output q, qbar;
 4      reg q;
 5
 6      assign qbar = ~q;
 7      always @ (posedge clk)
 8      begin
 9          if (reset == 0) q <= 0;
10          else if (set == 0) q <= 1;
11          else q <= data;
12      end
13  endmodule
```

Non-blocking/concurrent assignment

Cạnh lên xung clock






# Phát hiện cạnh

- Các phát biểu thủ tục trong hành vi vòng sẽ được thực thi khi sự kiện điều khiển xảy ra
- posedge và negedge dùng phát hiện cạnh lên hay xuống của tín hiệu

D:/Teaching/Verilog/asynch_df_behav.v

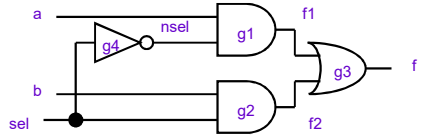
```
ln #
 1  module asynch_df_behave (q, q_bar, data, set, clk, reset);
 2      input data, set, clk, reset;
 3      output q, q_bar;
 4      reg q;
 5
 6      assign q_bar = ~q;
 7
 8      always @(negedge set or negedge reset or posedge clk)
 9      begin
10          if (reset == 0) q <= 0;
11          else if (set == 0) q <= 1;
12          else q <= data;
13      end
14  endmodule
```

D:/Teaching/Verilog/tr_latch.v

```
ln #
 1  module tr_latch (q_out, enable, data);
 2      output q_out;
 3      input enable, data;
 4      reg q_out;
 5
 6      always @(enable or data)
 7      begin
 8          if (enable) q_out = data;
 9      end
10  endmodule
```



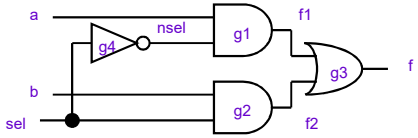
# Multiplexer 2-1




- Có bao nhiêu cách viết mux?

# Multiplexer Built From Primitives

```
module mux(f, a, b, sel);
output f;
input a, b, sel;

and     g1(f1, a, nsel),
        g2(f2, b, sel);
or      g3(f, f1, f2);
not     g4(nsel, sel);

endmodule
```



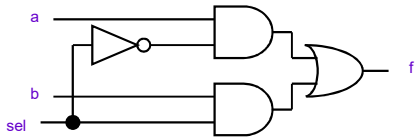



# Mux with Continuous Assignment

```
module mux(f, a, b, sel);
output f;
input a, b, sel;

assign f = sel ? a : b;

endmodule
```

LHS is always set to the value on the RHS

Any change on the right causes reevaluation



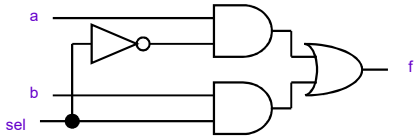



# Multiplexer Built With Always

```
module mux(f, a, b, sel);
output f;
input a, b, sel;
reg f;

always @(a or b or sel)
  if (sel) f = a;
  else f = b;

endmodule
```

Modules may contain one or more *always* blocks

Sensitivity list contains signals whose change triggers the execution of the block





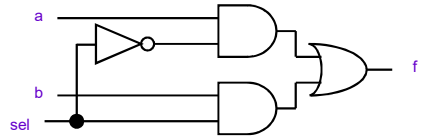

# Mux with User-Defined Primitive

```
primitive mux(f, a, b, sel);
output f;
input a, b, sel;

table
  1?0 : 1;
  0?0 : 0;
  ?11 : 1;
  ?01 : 0;
  11? : 1;
  00? : 0;
endtable
endprimitive
```

Behavior defined using a truth table that includes “don’t cares”

This is a less pessimistic than others: when a & b match, sel is ignored

(others produce X)



## So sánh giữa các loại mô hình hành vi

- Mô hình gán liên tục (Continuous-Assignment models)
- Dataflow/RTL models
- Mô hình dựa trên thuật toán (Algorithm-Based models)



## Mô hình gán liên tục

- Mô tả những hành vi nhạy mức (level-sensitive)
- Những biểu thức gán liên tục được thực hiện đồng thời

D:/Teaching/Verilog/compare_assign.v

```
ln #
1   module compare_2 (A_lt_B, A_gt_B, A_eq_B, A, B);
2       input [1:0] A, B;
3       output A_lt_B, A_gt_B, A_eq_B;
4
5       assign A_lt_B = (A<B);
6       assign A_gt_B = (A>B);
7       assign A_eq_B = (A==B);
8   endmodule
```

D:/Teaching/Verilog/compare_assign.v

```
ln #
1   module compare_32 (A_lt_B, A_gt_B, A_eq_B, A, B);
2       parameter world_size = 32;
3       input [world_size-1:0] A, B;
4       output A_lt_B, A_gt_B, A_eq_B;
5
6       assign A_lt_B = (A<B);
7       assign A_gt_B = (A>B);
8       assign A_eq_B = (A==B);
9   endmodule
```

D:/Teaching/Verilog/compare_boolean.v

```
ln #
1   module compare_2 (A_lt_B, A_gt_B, A_eq_B, A, B);
2       input [1:0] A, B;
3       output A_lt_B, A_gt_B, A_eq_B;
4       assign A_lt_B = (~A[1])&B[1]|(~A[1])&(~A[0])&B[0]|(~A[0])&B[1]&B[0];
5       assign A_gt_B = A[1]&(~B[1])|A[0]&(~B[1])&(~B[0])|A[1]&A[0]&(~B[0]);
6       assign A_eq_B = (~A[1])&(~A[0])&(~B[1])&(~B[0])|
7         (~A[1])&A[0]&(~B[1])&B[0]|A[1]&A[0]&B[1]&B[0]|A[1]&(~A[0])&B[1]&(~B[0]);
8   endmodule
```



## Mô hình Dataflow/RTL (1)

- Mô hình dòng dữ liệu của mạch tổ hợp mô tả những hoạt động đồng thời trên các tín hiệu
- Trong máy trạng thái đồng bộ các tính toán được thực hiện khi có cạnh tích cực của xung clock và được lưu trữ vào thanh ghi ở chu kỳ tiếp theo
- Mô hình dòng dữ liệu cho máy trạng thái đồng bộ được gọi là mô hình RTL (register transfer level)
- Mô hình RTL chỉ ra kiến trúc các thanh ghi đường dữ liệu và các hoạt động của máy
- Mô hình hành vi của mạch tổ hợp có thể được mô tả bằng tập hợp các phát biểu gán liên tục hoặc bằng một hành vi vòng



## Assignments (overview)

- Continuous assignments – fixed connection
  - **assign** f1 = a && b;
  - **assign** f2 = ~ f1;
- Blocking assignments – evaluate in order
  - **=** in **always** block
    - **begin**
      Q1 = D; // new Q1 will be used in evaluating all subsequent statements in this block
      Q2 = Q1; // new Q1 goes to Q2, so Q2 is equal to D.
    - **end**
- Non-blocking assignments – evaluate in parallel
  - **<=** in **always** block
    - **begin**
      Q1<= D;
      Q2<= Q1; // old Q1 goes to Q2
    - **end**
  - The order of statements doesn't matter

## Blocking and Nonblocking Assignments

- Fundamental problem:
  - In a synchronous system, all flip-flops sample simultaneously
  - In Verilog, always @(posedge clk) blocks run in some undefined sequence

Blocking: assignments are evaluated in some order, but we do not know in what

```
reg d1, d2, d3, d4;

always @(posedge clk) d2 = d1;
always @(posedge clk) d3 = d2;
always @(posedge clk) d4 = d3;
```

Nonblocking: RHS evaluated when assignment runs

```
reg d1, d2, d3, d4;

always @(posedge clk) d2 <= d1;
always @(posedge clk) d3 <= d2;
always @(posedge clk) d4 <= d3;
```

LHS updated only after all events for the current instant have run



## Blocking and Nonblocking Assignments

- A sequence of nonblocking assignments don't communicate

```
a = 1;
b = a;
c = b;
```

Blocking assignment:
a = b = c = 1

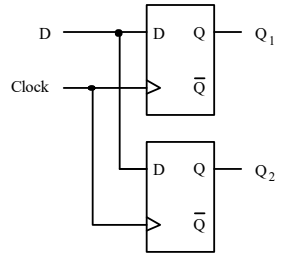
```
a <= 1;
b <= a;
c <= b;
```

Nonblocking assignment:
a = 1
b = old value of a
c = old value of b



## Blocking assign

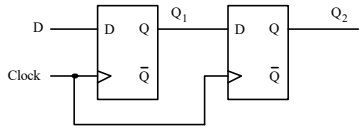
```
module example7_3 (D, Clock, Q1, Q2);
    input D, Clock;
    output Q1, Q2;
    reg Q1, Q2;

    always @(posedge Clock)
    begin
        Q1 = D;
        Q2 = Q1;
    end

endmodule
```




## Two cascaded flip-flops

Non-blocking assignments

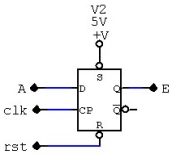
```
module example7_4 (D, Clock, Q1, Q2);
    input D, Clock;
    output Q1, Q2;
    reg Q1, Q2;

    always @(posedge Clock)
    begin
        Q1 <= D;
        Q2 <= Q1;
    end

endmodule
```



# Mô hình Dataflow/RTL (2)

```
1   module shiftreg_5 (E, A, clk, rst);
2       output E;
3       input A, clk, rst;
4       reg B, C, D, E;
5
6       always @(posedge clk or posedge rst) begin
7           if (rst) begin
8               B = 0; C = 0; D = 0; E = 0;
9           end
10          else begin
11              E = D;
12              D = C;
13              C = B;
14              B = A;
15          end
16      end
17  endmodule
```

time




```
1   module shiftreg_5 (E, A, clk, rst);
2       output E;
3       input A, clk, rst;
4       reg B, C, D, E;
5
6       always @(posedge clk or posedge rst) begin
7           if (rst) begin
8               B = 0; C = 0; D = 0; E = 0;
9           end
10          else begin
11              B = A;
12              C = B;
13              D = C;
14              E = D;
15          end
16      end
17  endmodule
```

Blocked assignment



# Non-blocking vs. Blocking

- Non-blocking
  - Toán tử <=
  - Các phát biểu thực thi đồng thời (song song)
  - Thứ tự các phát biểu không ảnh hưởng đến kết quả cuối cùng
  - Khi thực hiện hành vi vòng bộ mô phỏng tính giá trị biểu thức bên vế phải trước khi gán cho vế trái
- Blocking
  - Toán tử =
  - Các phát biểu thực thi tuần tự
  - Thứ tự các phát biểu có thể ảnh hưởng đến kết quả cuối cùng
  - Khi thực hiện hành vi vòng bộ mô phỏng chỉ tính giá trị biểu thức bên phải ngay sau khi phát biểu trước đó hoàn tất



# Mô phỏng trong mô hình hành vi

- Sự kiện xảy ra ở những tín hiệu nhập hay trong vế phải kích hoạt bộ mô phỏng tính toán giá trị ngõ xuất
- Khi hành vi vòng được kích hoạt, bộ mô phỏng thực thi những phát biểu cho đến
  - Toán tử điều khiển trễ (#)
  - Toán tử điều khiển sự kiện (@)
  - Cấu trúc đợi (wait)
  - Phát biểu cuối cùng của hành vi vòng
- Nếu có nhiều hành vi vòng được kích hoạt cùng lúc thì thứ tự thực thi không xác định được



# Nội dung chính

1. Mô hình hành vi
2. Mô hình hành vi dựa trên phương trình boole
3. Mô hình hành vi vòng
4. **Mô hình hành vi cho các khối cơ bản**

## Mô hình hành vi cho các khối cơ bản

- Phân kênh (multiplexer)
- Mã hóa (encoder)
- Giải mã (decoder)
- Thanh ghi dịch hồi tiếp tuyến tính

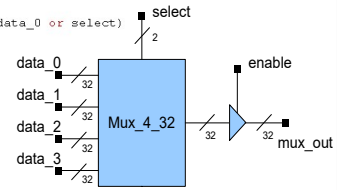


## Bộ phân kênh (1)

- Cấu trúc case tương tự như switch trong C, tìm các giá trị từ trên xuống, và thực hiện giá trị trùng đầu tiên.
- Giá trị default nên có trong tất cả các cấu trúc case

D:/Teaching/Verilog/mux_4_32_case.v

```
ln #
1   module Mux_4_32_Case (mux_out, data_3, data_2, data_1,
2                         data_0, select, enable);
3       output [31:0] mux_out;
4       input [31:0] data_3, data_2, data_1, data_0;
5       input [1:0] select;
6       input enable;
7
8       reg [31:0] mux_buff;
9
10      assign mux_out = enable?mux_buff:32'bz;
11
12      always @(data_3 or data_2 or data_1 or data_0 or select)
13      case (select)
14          2'b00: mux_buff = data_0;
15          2'b01: mux_buff = data_1;
16          2'b10: mux_buff = data_2;
17          2'b11: mux_buff = data_3;
18          default: mux_buff = 32'bx;
19      endcase
20  endmodule
```






## Bộ phân kênh (2)

D:/Teaching/Verilog/Mux_4_42_if.v

```
ln #
1   module Mux_4_32_if (mux_out, data_3, data_2, data_1,
2                       data_0, select, enable);
3       output [31:0] mux_out;
4       input [31:0] data_3, data_2, data_1, data_0;
5       input [1:0] select;
6       input enable;
7
8       reg [31:0] mux_buff;
9
10      assign mux_out = enable?mux_buff:32'bz;
11
12      always @(data_3 or data_2 or data_1 or data_0 or select)
13      if (select == 2'b00) mux_buff = data_0;
14      else if (select == 2'b01) mux_buff = data_1;
15           else if (select == 2'b10) mux_buff = data_2;
16                else if (select == 2'b11) mux_buff = data_3;
17                     else mux_buff = 32'bx;
18  endmodule
```

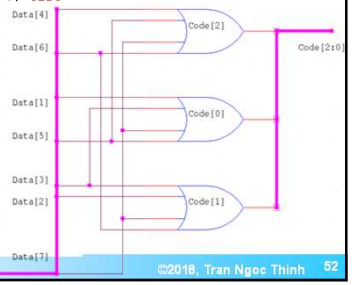
D:/Teaching/Verilog/Mux_4_32_CA.v

```
ln #
1   module Mux_4_32_CA (mux_out, data_3, data_2, data_1,
2                       data_0, select, enable);
3       output [31:0] mux_out;
4       input [31:0] data_3, data_2, data_1, data_0;
5       input [1:0] select;
6       input enable;
7       wire mux_buff;
8
9       assign mux_out = enable?mux_buff:32'bz;
10      assign mux_buff = (select == 2'b00)? data_0:
11                        (select == 2'b01)? data_1:
12                        (select == 2'b10)? data_2:
13                        (select == 2'b11)? data_3: 32'bx;
14  endmodule
```



## Bộ mã hóa – encoder

```
ln #
1   module encoder (Code, Data);
2       output [2:0] Code;
3       input [7:0 Data;
4       reg [2:0] Code;
5
6       always @ (Data)
7       begin
8           if (Data == 8'b00000001) Code = 0; else
9           if (Data == 8'b00000010) Code = 1; else
10          if (Data == 8'b00000100) Code = 2; else
11          if (Data == 8'b00001000) Code = 3; else
12          if (Data == 8'b00010000) Code = 4; else
13          if (Data == 8'b00100000) Code = 5; else
14          if (Data == 8'b01000000) Code = 6; else
15          if (Data == 8'b10000000) Code = 7; else
16          Code = 3'bx;
17      end
18
19      /*alternative description
20      always @ (Data)
21      case (Data)
22          8'b00000001: Code = 0
23          8'b00000010: Code = 1
24          8'b00000100: Code = 2
25          8'b00001000: Code = 3
26          8'b00010000: Code = 4
27          8'b00100000: Code = 5
28          8'b01000000: Code = 6
29          8'b10000000: Code = 7
30          default: Code = 3'bx;
31      endcase */
32  endmodule
```



# Mã hóa ưu tiên – priority encoder

- Cấu trúc casex cho phép tín hiệu x trong phép so sánh

D:/Teaching/Verilog/priority.v

```
ln #
 1   module priority (Code, Data);
 2       output [2:0] Code;
 3       input [7:0] Data;
 4       reg [2:0] Code;
 5
 6       always @(Data)
 7       casex (Data)
 8           8'b1xxxxxxx: Code = 7;
 9           8'b01xxxxxx: Code = 6;
10           8'b001xxxxx: Code = 5;
11           8'b0001xxxx: Code = 4;
12           8'b00001xxx: Code = 3;
13           8'b000001xx: Code = 2;
14           8'b0000001x: Code = 1;
15           8'b00000001: Code = 0;
16           default : Code = 3'bx;
17      endcase
18  endmodule
```

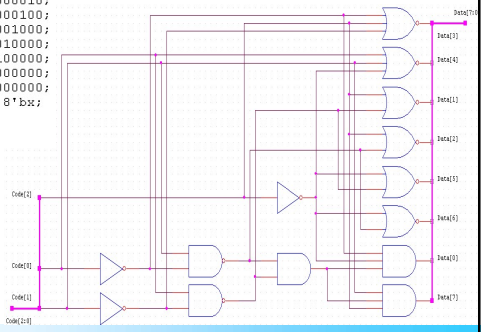




# Bộ giải mã - decoder

D:/Teaching/Verilog/decoder.v

```
ln #
 1   module decoder (Data, Code);
 2       output [7:0] Data;
 3       input [2:0] Code;
 4       reg [7:0] Data;
 5
 6       always @(Data)
 7       case (Code)
 8           0: Data = 8'b00000001;
 9           1: Data = 8'b00000010;
10           2: Data = 8'b00000100;
11           3: Data = 8'b00001000;
12           4: Data = 8'b00010000;
13           5: Data = 8'b00100000;
14           6: Data = 8'b01000000;
15           7: Data = 8'b10000000;
16           default: Data = 8'bx;
17      endcase
18  endmodule
```

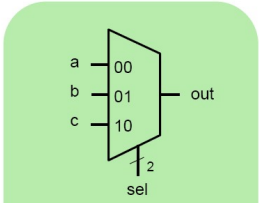






# Dangers of Verilog: Incomplete Specification

**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?***

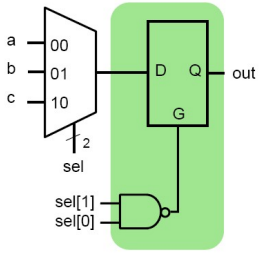


# Incomplete Specification Infers Latches

```
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!**

**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
```

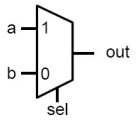
- …or, fully specify all branches of conditionals and assign all signals from all branches
  - For each `if`, include `else`
  - For each `case`, include `default`

## The Sequential always Block

- Edge-triggered circuits are described using a sequential `always` block

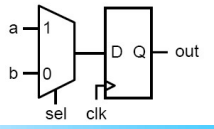
**Combinational**

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




**Sequential**

```
module sequential(a, b, sel,
                  clk, out);
  input a, b;
  input sel, clk;
  output out;
  reg out;
  always @ (posedge clk)
  begin
    if (sel) out <= a;
    else out <= b;
  end
endmodule
```



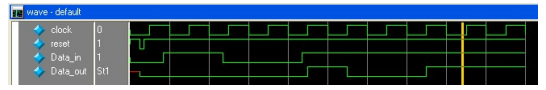

## Mô hình hóa máy số bằng các giải thuật lặp

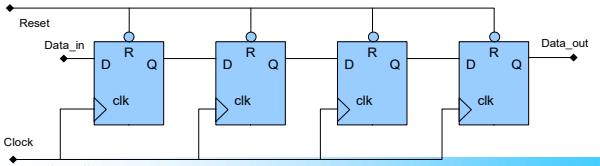
- Cấu trúc lặp for (có thể tổng hợp)
  - for *(initial_statement; control_expression; index_statement)*
    - *statement_for_expression;*
- Cấu trúc lặp repeat
  - repeat *(num_of_loop)*
    - *statement_repeat_expression;*
- Cấu trúc lặp while
  - while *(condition_expression)*
    - *statement_while_expression;*
- Cấu trúc lặp forever
  - forever *statement_forever_expression;*



## Thanh ghi dịch - shift register

```
1   module Shift_reg4 (Data_out, Data_in, clock, reset);
2       output Data_out;
3       input Data_in, clock, reset;
4       reg [3:0] Data_reg;
5
6       assign Data_out = Data_reg[0];
7       always @(negedge reset or posedge clock)
8       begin
9           if (reset == 1'b0) Data_reg <= 4'b0;
10          else Data_reg <= {Data_in,Data_reg[3:1]};
11      end
12  endmodule
```

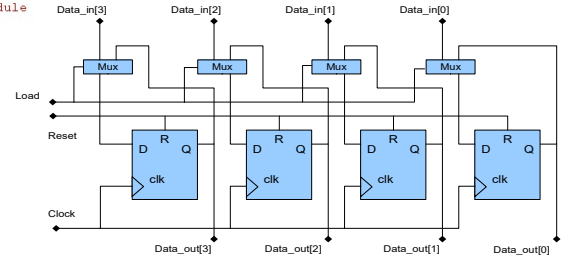


# Thanh ghi chuyển dữ liệu song song – parallel load

D:/Teaching/Verilog/par_load_reg4.v

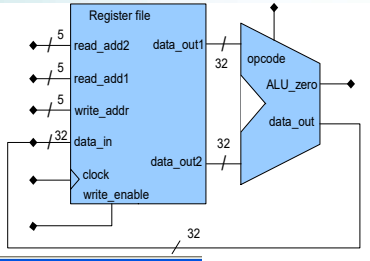
```
ln #
1   module par_load_reg4 (data_out, data_in, load, clock, reset);
2       output [3:0] data_out;
3       input [3:0] data_in;
4       input clock, load, reset;
5
6       reg[3:0] data_out;
7
8       always @(posedge clock or posedge reset)
9       begin
10          if (reset == 1'b0) data_out <= 4'b0;
11          else if (load == 1'b1) data_out <= data_in;
12      end
13  endmodule
```






# Tập thanh ghi

- Tập thanh ghi bao gồm một số lượng nhỏ các thanh ghi hỗ trợ đọc ghi thường được hiện thực bằng D Flip-Flop




D:/Teaching/Verilog/reg_file.v

```
ln #
1   module Register_File (data_out1, data_out2, data_in, read_add1,
2                         read_add2, write_add, write_enable, clock);
3       output [31:0] data_out1, data_out2;
4       input [31:0] data_in;
5       input [4:0] read_add1, read_add2, write_add;
6       input write_enable, clock;
7       reg [31:0] reg_file [31:0]; // 32bit x 32 word
8
9       assign data_out1 = reg_file[read_add1];
10      assign data_out2 = reg_file[read_add2];
11
12      always @(posedge clock)
13      begin
14          if (write_enable) reg_file[write_add] <= data_in;
15      end
16  endmodule
```