CHƯƠNG 3: RT- LEVEL COMBINATION CIRCUIT

1. **Giới thiệu**

Các mạch cấp cổng (gate-level) đã được bàn luận trong chương 1 sử dụng các toán tử bitwise đơn giản để mô tả thiết kế mức cổng, chúng bao gồm các ô logic đơn giản. Trong chương này, chúng ta hãy xem mô tả HDL của các mạch bao gồm các thành phần có kích thước trung bình như: bộ cộng, bộ so sánh, bộ ghép kênh. Vì những thành phần này là các khối xây dựng cơ bản sử dụng trong phương pháp thanh ghi dịch. Nó cũng thỉnh thoảng được tham chiếu đến như thiết kế RT-level. Chúng ta bàn luận về nhưng toán tử Verilog phức tạp hơn, khối always và cấu trúc định tuyến. Sau đó, mô tả về thiết kế mạch tổ hợp RT-level thông qua một chuỗi ví dụ.

1. **Các toán tử**

Verilog bao gồm khoảng 20 toán tử. Thêm vào đó các toán tử bitwise đã được thảo luận trong chương 1: các toán tử số học, dịch, và các phép toán quan hệ. Những toán tử này tương ứng với các thành phần có kích thước trung bình, chẳng hạn như bộ cộng và bộ so sánh. Chúng ta xem sét các toán tử trong phần này cũng như bao gồm các cấu trúc Verilog liên quan đến việc tổng hợp khác.

|  |  |  |  |
| --- | --- | --- | --- |
| Kiểu | Ký hiệu | Mô tả | Số toán hạng |
| Toán học | + | Cộng | 2 |
| - | Trừ | 2 |
| \* | Nhân | 2 |
| / | Chia | 2 |
| % | Lấy dư | 2 |
| \*\* | Lũy thừa | 2 |
| Dịch | >> | Dịch phải | 2 |
| << | Dịch trái | 2 |
| >>> |  | 2 |
| <<< |  | 2 |
| Quan hệ | > | Lớn hơn | 2 |
| < | Nhỏ hơn | 2 |
| >= | Lớn hơn hoặc bằng | 2 |
| <= | Nhỏ hơn hoặc bằng | 2 |
| Bình đẳng | == | Bằng nhau | 2 |
| != | Khác nhau | 2 |
| === | Trường hợp tương đương | 2 |
| !== | Trường hợp không tương đương | 2 |
| Bitwise | ~ | Đảo bit | 1 |
| & | Cộng bit | 2 |
| | | Hoặc bit | 2 |
| ^ | Xor bit | 2 |
| Giảm |  |  |  |
|  |  |  |
|  |  |  |
| Logic | ! | Phủ định | 1 |
| && | Và | 2 |
| || | Hoặc | 2 |
| Kết hợp | {} | Kết hợp |  |
| { {} } | Nhân rộng |  |
| Điều kiện | ? : | Điều kiện | 3 |

* 1. **Toán tử số học**

Có 6 toán tử số học: +, -, \*, /, %, \*\*. Chúng đại diện cho cộng, trừ, nhân, chia, chia lấy dư, và lũy thừa. Các toán tử + và - cũng có thể được sử dụng như các toán tử một ngôi, như trong -a. Trong quá trình tổng hợp, các toán tử + và - suy ra bộ cộng và bộ trừ và chúng được tổng hợp bởi các ô logic của FPGA. Phép nhân là một hoạt động phức tạp và tổng hợp của toán tử nhân \* phụ thuộc vào phần mềm tổng hợp và công nghệ của thiết bị đích. Xilinx Spartan-3, họ FPGA Xilinx có chứa các khối nhân tổ hợp được chế tạo sẵn. Phần mềm Xilinx XST chỉ ra có thể suy ra các khối này trong quá trình tổng hợp và do đó toán tử nhân có thể được sử dụng trong mã HDL. Thiết bị XCS200 của bo mạch S3 bao gồm mười hai khối nhân 18x18. Mặc dù việc tổng hợp toán tử nhân được hỗ trợ nhưng chúng ta vẫn cần nhận thức được giới hạn về số lượng và chiều rộng đầu vào của các khối này và sử dụng chúng một cách cẩn thận.

Các toán tử /, % và \*\* thường không thể được tổng hợp tự động.

* 1. **Toán tử dịch**

Có bốn toán tử shift: >>, <<, >>> và <<<. Hai cái đầu tiên đại diện cho sự dịch chuyển logic sang phải và trái, hai cái cuối cùng đại diện cho sự dịch chuyển số học sang phải và trái.

Các số 0 được chuyển sang cho một phép toán dịch chuyển logic (>> và <<). Các bit dấu như MSB thì được chuyển vào cho hoạt động >>> và các bit 0 được chuyển vào cho hoạt động <<<.

Lưu ý rằng không có sự khác biệt giữa các phép toán << và <<<. Cái sau được bao gồm để đầy đủ. Một số ví dụ về chuyển dịch được thể hiện trong Bảng 3.3. Nếu cả hai toán hạng của toán tử dịch đều là các tín hiệu, như a << b, toán tử suy luận về bộ dịch chuyển thùng, là một mạch khá phức tạp. Mặt khác, nếu số lượng dịch chuyển là cố định, như a << 2, hoạt động này không có logic và chỉ liên quan đến việc định tuyến các tín hiệu đầu vào.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| a | a >> 2 | a >>> 2 | a << 2 | a <<< 2 |
| 0100\_1111 | 0001\_0011 | 0001\_0011 |  |  |
| 1100\_1111 |  |  |  |  |

Kiểu hoạt động này cũng có thể được mô tả bằng cách sử dụng toán tử nối được thảo luận trong Phần 3.2.5.

* 1. **Toán tử quan hệ và bình đẳng**

Có bốn toán tử quan hệ: >, <, <= và >=. Các toán tử này so sánh hai toán hạng và trả về một kết quả Boolean (có thể là sai hoặc đúng, được biểu diễn bằng giá trị vô hướng l-bit I). Có bốn toán tử bình đẳng: ==, ! =, === và !==. Như với các toán tử quan hệ, chúng trả về false (I-bit 0) hoặc true (I-bit 1). Các toán tử === và !==, được gọi là toán tử tương đương và không tương đương bằng cách xem xét các kết quả khớp của các bit x và z trong các toán hạng. Chúng không thể được tổng hợp.

Các toán tử quan hệ và toán tử == và != suy ra bộ so sánh trong quá trình tổng hợp.

* 1. **Toán tử bitwise, giảm và logic**

Các toán tử bitwise, giảm và logic gần giống nhau và thực hiện các phép toán AND, OR, XOR và NOT. Các toán tử này được thực hiện bởi các ô logic cơ bản.

Toán tử bitwise: Có 4 toán tử bitwise cơ bản: & (and), | (or), ^ (xor) và ~ (not). 3 toán tử đầu tiên yêu cầu 2 toán hạng. Đảo và xor có thể kêt hợp với nhau để tạo ra xnor (~^ hoặc ^~).

Ví dụ: a, b, c với 4 bit tín hiệu.

Wire [3:0] a, b, c;

assigne c = a | b;

Giống như:

assigned c[3] = a[3] | b [3];

assigned c[2] = a[2] | b [2];

assigned c[1] = a[1] | b [1];

assigned c[0] = a[0] | b [0];

* Toán tử giảm: trước toán tử &, | và ^ có thể có chỉ một toán hạng. Toán hạng đơn thường có kiểu dữ liệu mảng. Phép tính được chọn sẽ được thực hiện trên tất cả các phần tử của mảng và trả về 1 bit kết quả. Ví dụ 4 bit tín hiệu a và 1 bit tín hiệu y.

Wire [3:0] a;

Wire y;

Assigned y = | a;

Tương tự:

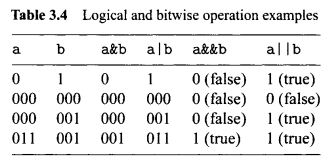
Assigned y = a[3] | a[2] | a[1] | a[0];

* Toán tử logic

Có 3 toán tử logic: && (logic and), || (logic or) và ! (logic phủ định). Toán tử logic khác toán tử bitwise. Nếu chúng ta giả sử rằng không có x và z được sử dụng thì các toán hạng của một toán tử logic sẽ bị gián đoạn vì false (khi tất cả các bit là 0) hoặc true (khi ít nhất một bit là 1) và luôn trả về 1 bit kết quả. Như tên cho thấy, toán tử logic nên được sử dụng như kết nối logic của các biểu thức Boolean.

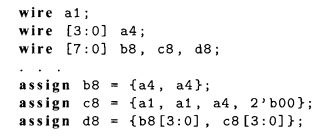


Một số ví dụ trong bảng 3.4. Toán tử bitwise cũng được thêm vào để minh họa sự khác nhau giữa 2 kiểu toán hạng này. Vì Verilog sử dụng 0 và 1 to biểu thị cho giá trị false và true, toán tử bitwise và logic có thể được sử dụng thay thế cho nhau trong một số trường hợp. Tuy nhiên, bạn nên sử dụng toán tử logic cho biểu thức Boolean và sử dụng toán tử bitwise để thao tác tín hiệu.

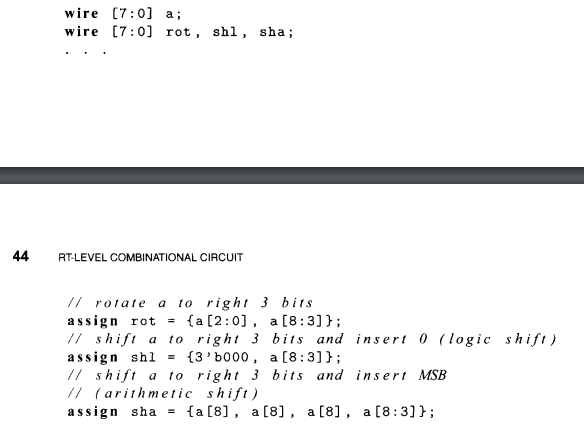


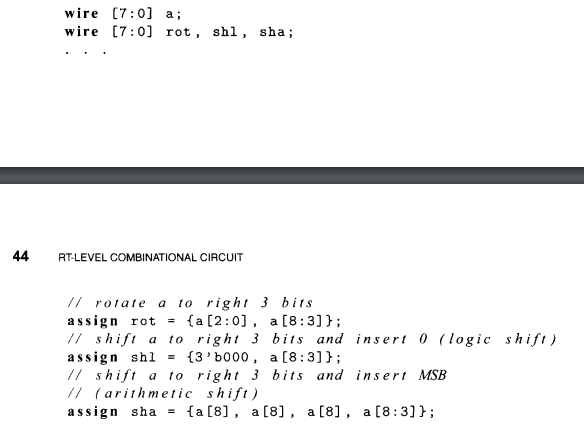
* 1. **Toán tử nối và sao chép**

Toán tử nối { } kết hợp các đoạn của các phần tử và các mảng nhỏ để tạo thành một mảng lớn. Ví dụ sau minh họa việc sử dụng nó:



Việc thực hiện toán tử nối liên quan đến việc kết nối lại các tín hiệu đầu vào và đầu ra và chỉ yêu cầu "nối dây". Một ứng dụng của toán tử nối là dịch chuyển và xoay tín hiệu theo một số lượng cố định, như thể hiện trong ví dụ:





Toán tử nối, N { }, sao chép chuỗi kèm theo. Hằng số sao chép (N), chỉ định số lần lặp lại. Ví dụ: {4 {2 'b01}} trả về 8' b01010101.

Hoạt động thay đổi số học trước đó có thể được đơn giản hóa:

assign sha = C3{a[81), a[8:3]};

* 1. **Toán tử điều kiện**

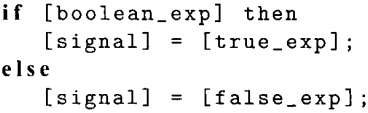
Toán tử điều kiện “? :” có ba toán hạng và định dạng chung của nó là:



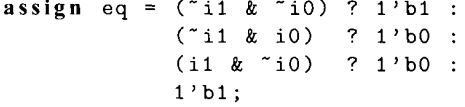
[Boolean-exp] là một biểu thức Boolean trả về true (1'b1) hoặc false (1'b0). [Signal] nhận [true-exp] nếu đúng và [false-exp] nếu sai. Ví dụ, đoạn mạch sau đây thu được tối đa a và b:



Toán tử có thể được coi là một câu lệnh if-then-else được đơn giản hóa:



Mặc dù đơn giản, các toán tử điều kiện có thể được xếp tầng hoặc lồng vào nhau để chỉ định lựa chọn mong muốn. Ví dụ, mạch eq1 được mô tả trong Bảng 1.1 có thể được viết lại bằng cách sử dụng các toán tử có điều kiện:



Chúng ta có thể mở rộng mạch với cực đại để trả về cực đại của a, b và c:



Trong khi được tổng hợp, một toán tử điều kiện suy ra một mạch ghép kênh 2 to 1. Chi tiết được thảo luận trong phần 3.6.

* 1. **Ưu tiên toán tử**

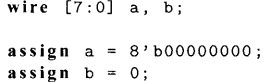
Ưu tiên toán tử chỉ định thứ tự đánh giá. Mức độ ưu tiên được thể hiện trong Table 3.2. Khi một biểu thức được đánh giá, toán tử có mức độ ưu tiên cao hơn sẽ được đánh giá đầu tiên. Ví dụ, trong biểu thức a + b >> 1, a + b được ước lượng trước và sau đó >> 1 được ước lượng. Chúng ta có thể sử dụng dấu ngoặc đơn để thay đổi mức độ ưu tiên, như trong a + (b >> 1). Một thực tiễn tốt là sử dụng dấu ngoặc đơn để làm cho biểu thức rõ ràng hơn, như trong (a + b) >> 1, ngay cả khi chúng không được yêu cầu.

* 1. **Điều chỉnh độ dài bit biểu thức**

Như các tín hiệu trong phần cứng thực, net và biến trong chương trình Verilog thường có số lượng bit khác nhau (tức là độ dài hoặc độ rộng bit). Trong một câu lệnh Verilog, độ dài bit của các toán hạng có thể khác nhau và việc điều chỉnh được xác định bởi một tập hợp các quy tắc ngầm định:

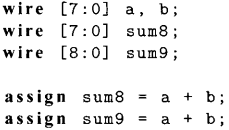
* Xác định độ dài bit tối đa của các toán hạng trong ngữ cảnh, bao gồm biểu thức bên phải và tín hiệu bên trái.
* Mở rộng độ dài bit của toán hạng ở phía bên phải đến mức tối đa và đánh giá biểu thức.
* Gán kết quả cho tín hiệu bên trái. Cắt ngắn các MSB nếu độ dài bit của tín hiệu nhỏ hơn.

Đầu tiên chúng ta hãy xem xét một ví dụ đơn giản:



Câu lệnh đầu tiên chỉ định giá trị 8 bit "00000000" cho a. Câu lệnh thứ hai gán số nguyên 0 cho b. Nhớ lại rằng số nguyên trong Verilog là 32 bit và do đó 0 được biểu thị là "00000000000000000000000000000000". Vì b rộng 8 bit nên nó bị cắt bớt thành "00000000" trong quá trình gán. Mặc dù cả hai câu lệnh đều gán một mẫu bằng không cho các tín hiệu, nhưng chúng ta cần biết cách thu được các giá trị.

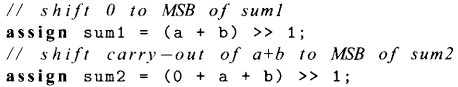
Chúng ta hãy xem xét một ví dụ khác:



Trong phép gán đầu tiên, tất cả các toán hạng có độ rộng 8 bit và phép cộng 8 bit được thực hiện. Phần thực hiện của phần bổ sung bị loại bỏ. Trong phép gán thứ hai, tín hiệu a và b được mở rộng thành 9 bit, độ dài bit của tín hiệu sum9 và phép cộng 9-bit được thực hiện. Bit tổng [9] nhận được kết quả là bit thực hiện. Chúng ta cũng có thể sử dụng toán tử nối nếu muốn có tín hiệu thực hiện rõ ràng:

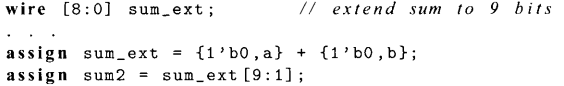


Mặc dù quy tắc chuyển đổi cơ bản rất đơn giản và trực quan, nhưng sự phức tạp có thể gây ra lỗi. Ví dụ: cho a, b, sum1 và sum2 là các tín hiệu 8 bit. Các câu lệnh sau đây cho một kết quả khác:



Trong phép gán đầu tiên, tất cả các toán hạng có độ rộng 8 bit và phép cộng 8 bit được thực hiện. Bit mang bị loại bỏ. Khi hoạt động thay đổi được thực hiện, 0 được chuyển vào MSB. Trong phép gán thứ hai, 0 là một số nguyên và do đó có chiều rộng 32 bit. A và b được mở rộng đến 32 bit để cộng và tổng được thay đổi. Kết quả sau đó được cắt ngắn thành 8 bit khi được gán cho sum2 và sum2[7] nhận được bit thực hiện ban đầu. Việc chuyển đổi trở nên liên quan hơn khi kiểu dữ liệu đã ký được sử dụng (được thảo luận trong phần 7.3).

Một giải pháp thay thế an toàn nhưng hơi rườm rà là điều chỉnh độ dài bit của các toán hạng theo cách thủ công. Ví dụ, một giải pháp thay thế có thể được sử dụng để lấy sum2 là:



Mã dài hơn nhưng mang tính mô tả cao hơn và ít bị lỗi hơn.

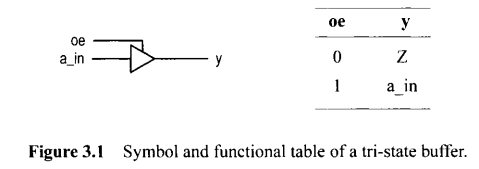
Tóm lại, chúng ta phải biết về cơ chế điều chỉnh độ dài bit tự động của Verilog. Sự không khớp về độ dài bit ngoài ý muốn có thể dẫn đến các lỗi tinh vi, khó tìm. Ngoại trừ những điều chỉnh nhỏ, chẳng hạn như gán một mẫu tất cả 0 với một số nguyên 0, chúng ta nên hoặc điều chỉnh độ dài bit theo cách thủ công hoặc ghi lại kỹ lưỡng điều chỉnh tự động mong muốn.

* 1. **Tổng hợp các giá trị z và x**

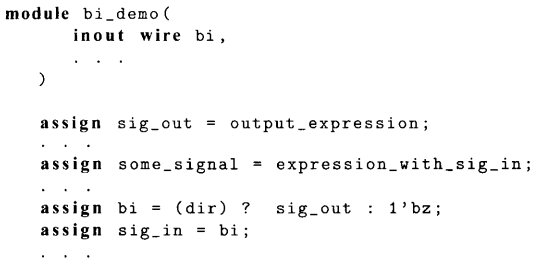
Ngoài logic thông thường 0 và logic 1, net và biến có thể chứa các giá trị z và x. Mặc dù chúng không phải là toán tử. Chúng ta sẽ thảo luận khía cạnh tổng hợp của hai giá trị này trong tiểu mục này.

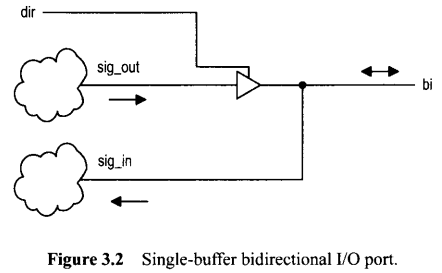
**Tổng hợp z:** giá trị z ngụ ý trở kháng cao hoặc mạch hở. Nó không phải là một giá trị logic thông thường và chỉ có thể được tổng hợp bởi một bufer ba trạng thái. Biểu tượng và bảng chức năng của bộ đệm ba trạng thái được thể hiện trong Hình 3.1. Hoạt động của bộ đệm được điều khiển bởi một tín hiệu cho phép, “oe” (đối với "cho phép đầu ra"). Khi nó là 1, đầu vào được chuyển sang đầu ra. Mặt khác, khi nó bằng 0, đầu ra y dường như là một mạch hở. Mã của bộ đệm ba trạng thái là.





Ứng dụng phổ biến nhất cho bộ đệm ba trạng thái là triển khai cổng hai chiều để sử dụng tốt hơn chân 110 vật lý. Một ví dụ đơn giản được thể hiện trong Hình 3.2. Tín hiệu dir điều khiển hướng dòng tín hiệu của chân bi. Khi nó bằng 0, bộ đệm ba trạng thái ở trạng thái trở kháng cao và tín hiệu ra bị chặn. Chân được sử dụng như một cổng đầu vào và tín hiệu đầu vào được chuyển đến tín hiệu sig\_in. Khi tín hiệu dir là 1, chân được sử dụng làm cổng ra và tín hiệu sig\_out được chuyển đến mạch bên ngoài. Mã HDL có thể được lấy theo sơ đồ:

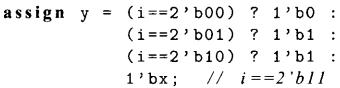


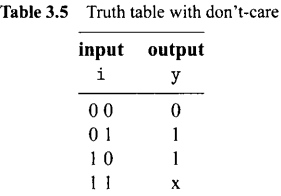


Lưu ý rằng chế độ của cổng bi phải được khai báo là không hoạt động cho hoạt động hai chiều.

Đối với thiết bị Xilinx Spartan-3, bộ đệm ba trạng thái chỉ tồn tại trong khối 110 (IOB) của chân vật lý. Do đó, bộ đệm ba trạng thái chỉ có thể được sử dụng cho các cổng I / O được ánh xạ tới các chân vật lý Xilinx của thiết bị FPGA.

**Tổng hợp x:** Trong một số mạch tổ hợp, một số mẫu đầu vào nhất định có thể không bao giờ xảy ra và do đó giá trị đầu ra không liên quan. Chúng tôi thường chỉ định giá trị không quan tâm cho đầu ra. Trong quá trình tổng hợp, giá trị không quan tâm sẽ được gán một giá trị (0 hoặc 1) có thể giúp quá trình tối ưu hóa. Hãy xem xét bảng sự thật được hiển thị trong Table 3.5. Chúng ta giả định rằng i sẽ không bao giờ là 11 và do đó đầu ra tương ứng được chỉ định là không quan tâm. Trong tổng hợp, chúng ta có thể sử dụng x cho giá trị không quan tâm. Một mã có thể có cho bảng trước là





Mặc dù cách tiếp cận này giúp giảm thiểu mạch, nhưng nó tạo ra sự khác biệt giữa mô phỏng và tổng hợp. Trong mô phỏng, x là một giá trị duy nhất thay vì 0 hoặc 1. Nếu đầu vào là 11 trong mô phỏng, đầu ra sẽ trở thành x và không phù hợp với kết quả tổng hợp (có thể là 0 hoặc 1). Tuy nhiên, vì mẫu 11 không bao giờ xảy ra trong đặc điểm kỹ thuật ban đầu, sự xuất hiện của giá trị x có thể được sử dụng để báo hiệu các lỗi tiềm ẩn trong testbench.

1. **Bock ALWAYS cho một mạch tổ hợp**

Để tạo điều kiện thuận lợi cho việc lập mô hình hệ thống, Verilog chứa một số câu lệnh thủ tục được thực thi trong tuần tự. Vì hoạt động của chúng khác với mô hình mạch đồng thời, các câu lệnh được đóng gói bên trong một khối always hoặc initial. Khối initial được thực hiện một khi mô phỏng bắt đầu. Nó có thể được sử dụng trong mô phỏng, như trong ví dụ testbench trong Listing 1.7. Chỉ khối always có thể được tổng hợp và nó được bàn luận trong phần này. Vì câu lệnh thủ tục trừu tượng, loại mã này đôi khi được gọi là mô tả hành vi.

Một khối always có thể được coi như một hộp đen có hành vi được mô tả bằng các câu lệnh thủ tục bên trong nó. Các câu lệnh thủ tục bao gồm nhiều loại cấu trúc phong phú nhưng nhiều cấu trúc trong số chúng không có đối tác phần cứng rõ ràng. Một khối always được mã hóa kém thường xuyên dẫn đến việc triển khai phức tạp không cần thiết hoặc không thể tổng hợp được.

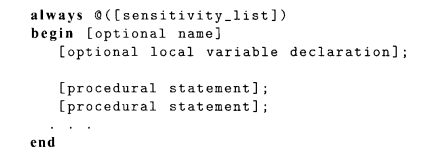
Trọng tâm của phần này là tổng hợp các mạch tổ hợp và chúng ta giới hạn cuộc thảo luận ở ba dạng:

* Blocking procedural assignment
* If statement
* Case statement

Hai cấu trúc sau có thể được coi là cấu trúc suy ra cấu trúc định tuyến.

* 1. Cú pháp và hành vi cơ bản

Cú pháp đơn giản của một khối always với một sensitivity list:



Thuật ngữ [sensitivity list] là danh sách các tín hiệu và sự kiện mà khối always phản hồi. Đối với mạch tổ hợp, tất cả các tín hiệu đầu vào nên được đưa vào danh sách này. Phần thân bao gồm bất kỳ số lượng tuyên bố thủ tục nào. Các dấu phân cách begin và end có thể bị bỏ qua nếu chỉ có một câu lệnh thủ tục trong phần nội dung. Thuật ngữ @ ([sensitivity list]) thực sự là một cấu trúc điều khiển thời gian. Nó thường là cấu trúc điều khiển thời gian duy nhất trong một khối always có thể tổng hợp.

Một khối always có thể được xem như một phần mạch phức tạp. Nó có thể bị đình chỉ hoặc hoạt động. Khi bất khì tín hiệu của sensitivity list thay đổi hoặc một sự kiện được đưa ra, phần được kích hoạt và thực hiện các câu lệnh thủ tục nội bộ.

Vì không có cấu trúc điều khiển thời gian nào khác, quá trình thực thi tiếp tục đến cuối và phần bị tạm dừng. Do đó, một khối always thực sự "lặp lại mãi mãi" và việc bắt đầu mỗi vòng lặp được điều khiển bởi sensitivity list.

* 1. **Gán thủ tục**

Phép gán thủ tục chỉ có thể được sử dụng trong một khối always hoặc khối initial.

Có 2 kiểu gán: blocking assignment và nonblocking assignment.



Trong blocking assignment, biểu thức được đánh giá và sau đó được gán cho biến ngay lập tức, trước khi thực hiện câu lệnh tiếp theo (việc gán do đó "chặn" việc thực thi các câu lệnh khác). Hành vi của nó giống như gán biến bình thường trong ngôn ngữ C. Trong một nonblocking assignment, biểu thức đánh giá được gán tại cuối của khối always (vì thế không block thực thi các tuyên bố khác).

Assignment blocking và nonblocking thường hiểu lầm với người mới dùng Verilog và không hiểu hành vi của chúng sẽ dẫn đến những hành vi không mong muốn hoặc một cuộc chạy đua.

Quy tắc cơ bản:

Assignment blocking: sử dụng cho mạch tổ hợp.

Nonblocking sử dụng cho mạch tuần tự.

Phần này sẽ được giải thích chi tiết trong phần 7.1 Vì chúng ta tập chung vào mạch tổ hợp trong chương này nên chỉ blocking được sử dụng.

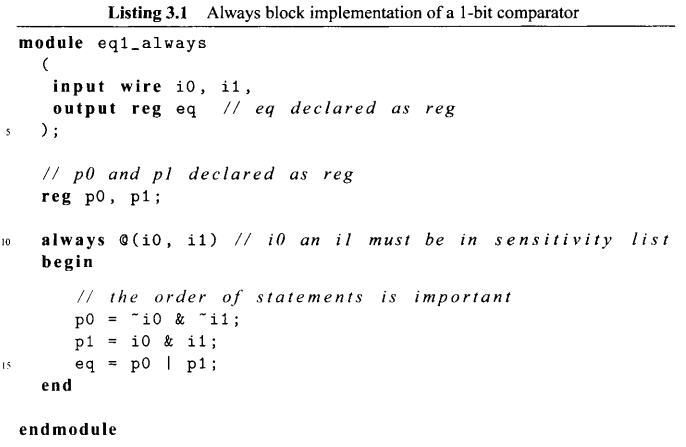
* 1. **Kiểu dữ liệu biến**

Trong các thủ tục gán, một biểu thức có thể chỉ được gán tới một đầu ra với một kiểu dữ liệu của biến là reg, integer, real, time hoặc realtime. Kiểu dữ liệu reg như kiểu dữ liệu wire, nhưng được sử dụng với thủ tục đầu ra. Kiểu dữ kiệu integer đại diện cho kích thước cố định một số nguyên có dấu 32-bit. Vì kích thước của nó là cố định, chúng ta luôn luôn không sử dụng nó trong khi tổng hợp. Các kiểu dữ liệu khác cho mô hình hóa và mô phỏng và không thể được tổng hợp.

* 1. **Ví dụ đơn giản**

Chúng tôi sử dụng hai ví dụ đơn giản để minh họa việc sử dụng và hành vi của khối always và thủ tục blocking assignment.

Bộ so sánh 1-bit: Chúng ta có thể viết lại mạch so sánh 1-bit trước đó trong List 1.1 sử dụng một khối always. Mã được hiển thị trong Listing 3.1



Khi tín hiệu eq, p0 và p1 được gán trong khối always, chúng được khai báo như kiểu dữ liệu reg. Sensitivity list bao gồm i0, i1, thứ được phân cách bởi dấu “,”. Khi một trong số chúng thay đổi, khối always được kích hoạt. Ba khối blocking assignment được thực thi tuần tự giống như các câu lệnh trong trương trình C. Thứ tự của các câu lệnh là quan trọng và p0 và p1 phải được gán giá trị trước khi được sử dụng.

Trong Verilog-1995, keywork “or” được sử dụng để thay thế “,” trong sensitivity list. Ví dụ:



Được biết lại như:



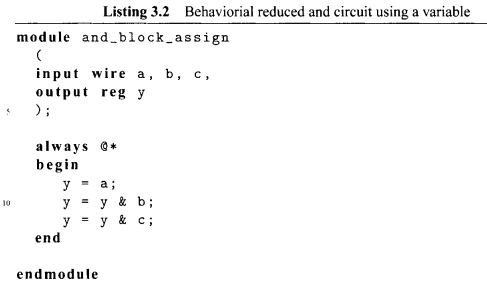
Chúng ta chỉ sử đụng “,” trong cuốn sách này.

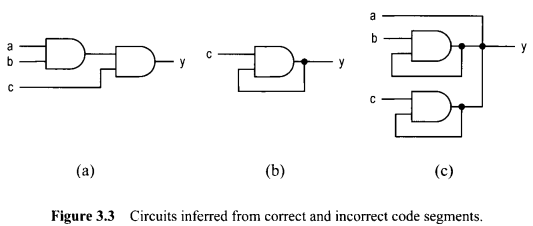
Một mạch tổ hợp phải bảo gồm tất cả các tín hiệu đầu vào trong sensitivity list để mô hình chính xác hành vi mong muốn. Bỏ qua 1 tín hiệu có thể dẫn đến sự khác biệt giữa tổng hợp và mô phỏng. Trong Verilog-2001, chúng ta có thể sử dụng ký hiệu để ngầm định bao gồm tất cả các tín hiệu đầu vào. Trong cuốn sách này, chúng tôi sử dụng cấu trúc này cho mạch tổ hợp.



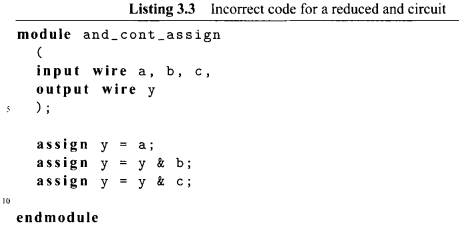
Ba đầu vào và mạch: giống như mã trong List 1.1 và 3.1 hơi sai lệch. Hành vi của các phép gán liên tục và các câu lệnh thủ tục khá khác nhau.

Xem sét mã trong List 3.2. Nó là mạch thực hiện toán tử AND thông qua a, b và c (a&b&c).





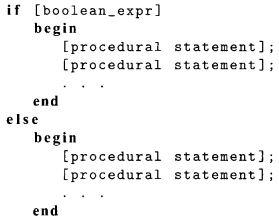
Mạch suy ra như hình 3.3 (a). Nếu chúng ta sử dụng các phép gán liên tục theo cách tương tự, như được hiển thị trong List 3.3, thì mô tả không chính xác.



Trong mã này, mỗi phép gán liên tục suy ra một phần mạch. Ba sự xuất hiện của y ở phía bên trái ngụ ý rằng ba đầu ra được gắn với nhau. Sơ đồ mạch tương ứng được thể hiện trong Hình 3.3 (c) và rõ ràng nó không phải là mạch mong muốn.

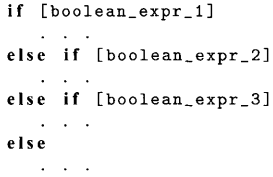
1. **IF Statement**
   1. **Cú pháp**

Cú pháp cơ bản của một tuyên bố IF.



Thuật ngữ [boolean-expr] là một biểu thức Boolean và được đánh giá đầu tiên. Nếu nó đúng, các câu lệnh trong nhanh sau được thực thi. Ngược lại các câu lệnh trong nhánh else được thực thi. Nhanh else là lựa chọn và có thể được bỏ qua. Begin và end có thể được bỏ qua khi chỉ có một một câu lệnh thủ tục trong 1 nhánh.

Nhiều câu lệnh if có thể được xếp tầng để đánh giá nhiều điều kiện Boolean và thiết lập mức độ ưu tiên, như trong:

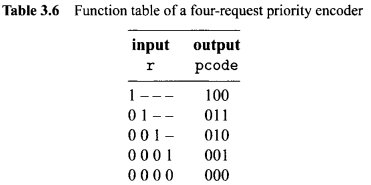


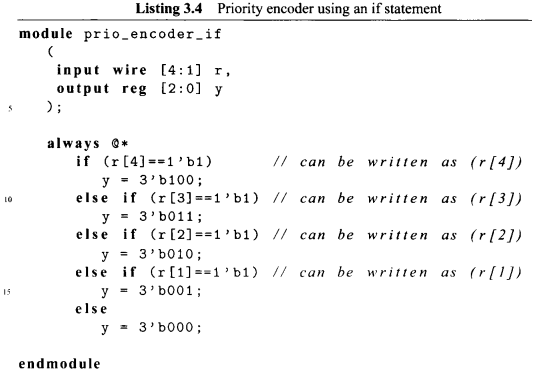
Khi được tổng hợp, các câu lệnh if suy ra các mạng định tuyến ưu tiên. Chủ đề này được thảo luận trong phần 3.6.

* 1. **Ví dụ**

Chúng ta sử dụng 2 ví dụ đơn giản để chứng minh việc sử dụng câu lệnh if.

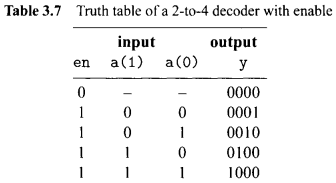
Ví dụ đầu tiền là bộ mã hóa ưu tiên “*priority encoder*”. Bộ mã hóa ưu tiên có 4 yêu cầu r[4], r[3], r[2] và r[1]. Chúng được nhóm lại như một tín hiệu đầu vào 4-bit r và r[4] có mức ưu tiên cao nhất. Đầu ra là mã nhị phân của yêu cầu thứ tự cao nhất. Bảng chức năng được hiển thị trong Table 3.6 và mã HDL được hiển thị trong List 3.4.

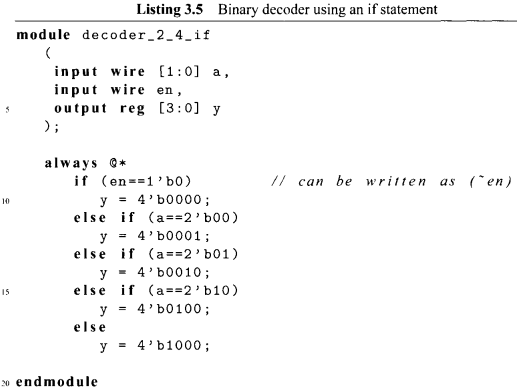




Đầu tiên, mã kiểm tra yêu cầu r[4] và gán 100 cho pcode nếu nó được khẳng định. Nó tiếp tục kiểm tra yêu cầu r[3] nếu r[4] không được khẳng định và lặp lại quá trình cho đến khi tất cả các yêu cầu được kiểm tra. Lưu ý rằng biểu thức Boolean (r[4] == l'b1) là true khi r [4] là 1. Vì giá trị true cũng được biểu thị bằng 1'b1 trong Verilog, biểu thức có thể được viết dưới dạng (r [4]).

Ví dụ thứ hai là bộ giải mã nhị phân. Bộ giải mã nhị phân n-to-2n xác nhận một bit của đầu ra 2n bit theo các kết hợp đầu vào. Bảng chức năng của bộ giải mã 2-thành-4 được thể hiện trong Table 3.7. Mạch cũng có tín hiệu điều khiển “en”, bật chức năng giải mã khi được xác nhận. Mã HDL được hiển thị trong List 3.5.

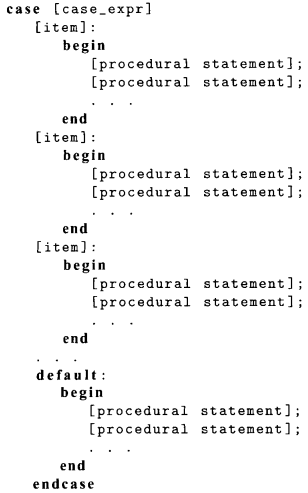




Đầu tiên, mã kiểm tra xem “en” có được xác nhận không. Nếu điều kiện là sai (tức là en là 1), nó sẽ kiểm tra bốn kết hợp nhị phân theo thứ tự. Lưu ý rằng biểu thức Boolean (en == l'b0) cũng có thể được viết là (~en).

1. **CASE Statement**
   1. **Cú pháp**

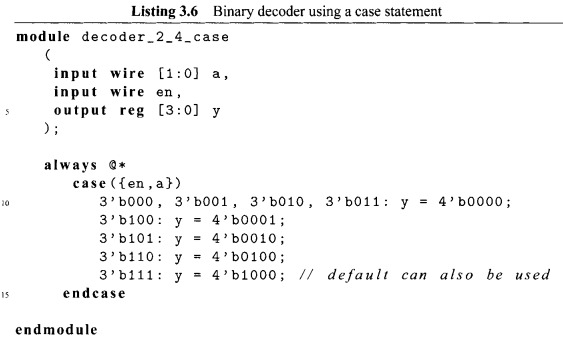
Cú pháp cơ bản của case:



Câu lệnh case là một câu lệnh quyết định nhiều chiều so sánh biểu thức [case-expr] với một số biểu thức [item]. Việc thực thi nhảy đến nhánh có [item] khớp với giá trị hiện tại của [case-expr]. Nếu có nhiều biểu thức [item] được so khớp, việc thực thi sẽ chuyển đến nhánh của kết quả khớp đầu tiên. Mục cuối cùng có thể là một từ khóa “default” tùy chọn. Nó bao gồm tất cả các giá trị không xác định của biểu thức [case-expr]. begin và end có thể bị bỏ qua nếu chỉ có một câu lệnh thủ tục trong một nhánh.

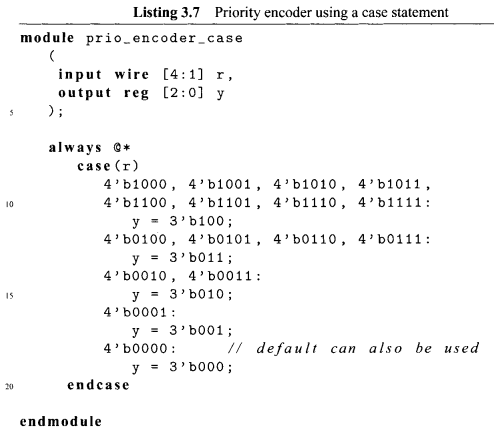
* 1. **Ví dụ**

Chúng ta sử dụng các ví dụ về bộ mã hóa và bộ giải mã ưu tiên giống trên để chứng minh việc sử dụng câu lệnh *case*. Bảng chức năng của bộ giải mã 2 to 4 được thể hiện trong Table 3.7. Mã HDL sử dụng câu lệnh *case* được hiển thị trong List 3.6.



Chúng ta có thể nhóm nhiều giá trị vào một biểu thức, như trong dòng 10, nếu các câu lệnh giống hệt nhau được sử dụng cho các giá trị này. Lưu ý rằng tất cả các giá trị có thể có của biểu thức {en, a} đều được bao phủ bởi các biểu thức item.

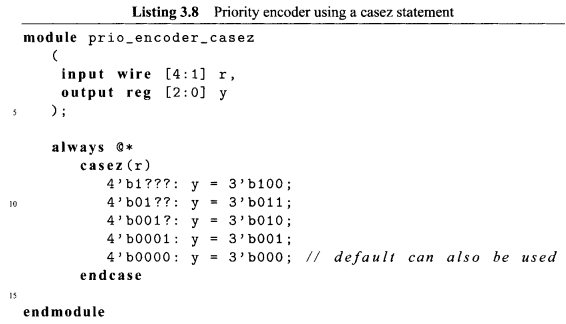
Bảng chức năng của bộ mã hóa ưu tiên được trình bày trong Table 3.6. Mã HDL được hiển thị trong List 3.7.



* 1. **Câu lệnh casez và casex**

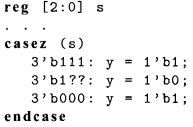
Có hai biến thể ngoài câu lệnh case thông thường. Trong câu lệnh casez, giá trị z và ký tự “?”trong biểu thức được coi là không quan tâm (tức là bit tương ứng không cần phải khớp). Trong một câu lệnh casex, các giá trị z và x và kí tự dấu ? trong biểu thức được coi là không quan tâm. Vì các giá trị z và x có thể xuất hiện trong mô phỏng, dấu ? được sử dụng nhiều.

Ví dụ: bộ mã hóa ưu tiên trước có thể được mã hóa bằng câu lệnh casez, như được hiển thị trong List 3.8.



* 1. **CASE đầy đủ và CASE song song**

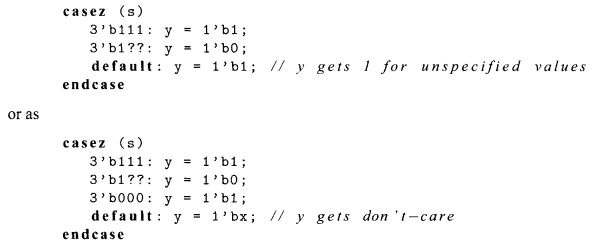
Trong Verilog, các biểu thức không cần phải bao gồm tất cả các giá trị của biểu thức [case-expr] và một số giá trị có thể được so khớp nhiều lần. Hãy xem xét câu lệnh casez sau:



Trong câu lệnh này, giá trị 3'b111 được so khớp hai lần trong các biểu thức (một lần trong 3'b111 và một lần trong 3'b1??). Kể từ lần đầu tiên có hiệu lực, y được 1’b1 nếu s là 3’b111. Nếu s là 3'b001, 3'b010 hoặc 3'b011, không có kết quả phù hợp nào và y sẽ giữ nguyên giá trị trước đó của nó.

Khi tất cả các giá trị nhị phân có thể có của biểu thức [case-expr] được bao phủ bởi các biểu thức item, thì câu lệnh được gọi là câu lệnh case đầy đủ. Đối với mạch tổ hợp, chúng ta phải sử dụng một câu lệnh case đầy đủ vì mỗi tổ hợp đầu vào phải có một giá trị đầu ra.

Chúng ta có thể thêm mục “default” để bao gồm tất cả các giá trị chưa khớp. Ví dụ: câu lệnh trước đó có thể được sửa đổi thành.



Khi các giá trị trong biểu thức mục loại trừ lẫn nhau (tức là một giá trị chỉ xuất hiện trong một biểu thức mục), thì câu lệnh được gọi là câu lệnh case song song. Ví dụ: câu lệnh casez trước đó không phải là một câu lệnh case song song vì giá trị 3'b111 xuất hiện hai lần. Các câu lệnh case của List 3.6 và 3.7 là các câu lệnh trường hợp song song.

Khi được tổng hợp, một câu lệnh case song song thường suy ra một mạng định tuyến ghép kênh và một câu lệnh case không song song thường cho một mạng định tuyến ưu tiên. Chủ đề này sẽ được thảo luận trong phần tiếp theo.

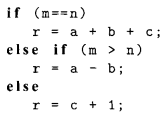
Nhiều gói phần mềm tổng hợp có chỉ thị case đầy đủ và chỉ thị case song song. Khi chúng được sử dụng, tất cả các câu lệnh trường hợp được coi là câu lệnh trường hợp đầy đủ và câu lệnh trường hợp song song và được tổng hợp tương ứng. Verilog-2001 có các thuộc tính tương tự cho mục đích này. Việc sử dụng các chỉ thị này về cơ bản ghi đè ngữ nghĩa gốc của mã Verilog và tạo ra sự khác biệt giữa mô phỏng và tổng hợp. Trong cuốn sách này, chúng tôi diễn đạt các điều kiện này bằng mã thay vì áp dụng các chỉ thị hoặc thuộc tính này.

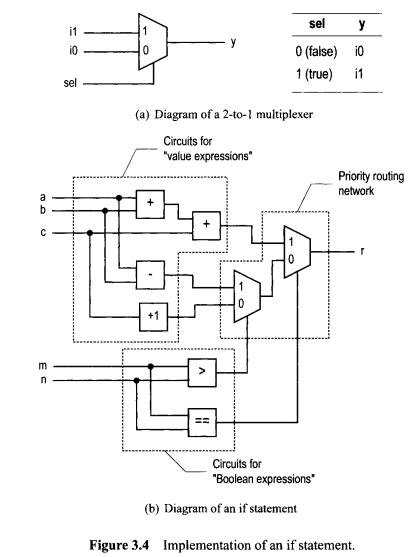
1. **Cấu trúc định tuyến của các cấu trúc điều khiển điều kiện**

Chúng tôi kiểm tra một số cấu trúc ngôn ngữ điều khiển có điều kiện, bao gồm toán tử “? :” và các câu lệnh if và case. Trong ngôn ngữ C, các cấu trúc này được thực thi tuần tự. Không có điều khiển tuần tự trong mạch tổ hợp. Các cấu trúc này được thực hiện bởi các mạng định tuyến. Tất cả các biểu thức được đánh giá đồng thời và mạng định tuyến định tuyến kết quả mong muốn đến đầu ra. Có hai loại cấu trúc định tuyến: mạng định tuyến ưu tiên và mạng ghép kênh được suy ra bởi câu lệnh kiểu if-else và câu lệnh case song song tương ứng.

* 1. **Mạng định tuyến ưu tiên**

Mạng định tuyến ưu tiên được thực hiện bởi một chuỗi các bộ ghép kênh 21. Sơ đồ và bảng sự thật của bộ ghép kênh 2-1 được thể hiện trong Hình 3.4 (a). Câu lệnh if-else ngụ ý một mạng định tuyến ưu tiên. Xem xét theo câu lệnh sau:

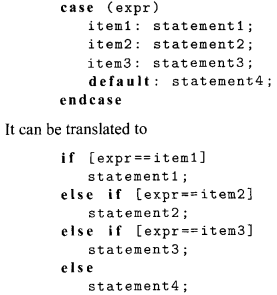




Sơ đồ khái niệm của câu lệnh được thể hiện trong Hình 3.4 (b). Hai bộ ghép kênh 2-1 tạo thành mạng định tuyến ưu tiên và các thành phần khác triển khai các biểu thức Boolean và số học khác nhau. Nếu điều kiện Boolean đầu tiên (m==n) là đúng, kết quả của a+b+c được định tuyến tới r. Ngược lại, dữ liệu được kết nối đến cổng 0 được nối tới r. Điều kiện Boolean tiếp theo (m> n) xác định xem kết quả của a-b hoặc c+l có được chuyển đến đầu ra hay không.

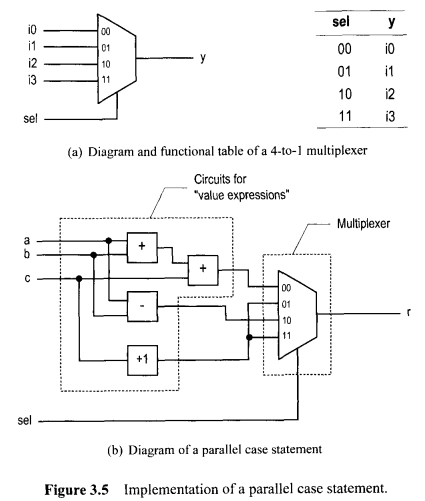
Lưu ý rằng tất cả các biểu thức Boolean và biểu thức số học được đánh giá đồng thời. Các đầu ra từ mạch Boolean đặt các tín hiệu lựa chọn của bộ ghép kênh để định tuyến giá trị mong muốn đến r. Số lượng các giai đoạn xếp tầng tăng tỷ lệ thuận với số lượng các mệnh đề if-else. Một số lượng lớn các mệnh đề if-else sẽ dẫn đến một chuỗi xếp tầng dài và tạo ra độ trễ truyền lớn.

Toán tử điều kiện (? :) giống như một câu lệnh if-else đơn giản hóa và suy ra các mạng định tuyến ưu tiên tương tự. Một câu lệnh case không song song đặt một tùy chọn trên mục phù hợp đầu tiên và do đó cũng suy ra các mạng định tuyến ưu tiên tương tự. Ví dụ, hãy xem xét câu lệnh trường hợp sau:

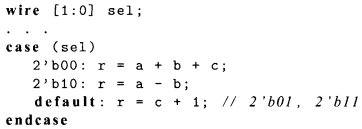


* 1. **Mạng ghép kênh**

Một mạng ghép kênh được thực hiện bởi một bộ ghép kênh n-to-1. Cổng đầu vào mong muốn được chỉ định bởi tín hiệu lựa chọn và đầu vào tương ứng được chuyển đến đầu ra. Sơ đồ và bảng chức năng của bộ ghép kênh 22-1 được thể hiện trong Hình 3.5 (a)



Trong một câu lệnh case song song, chúng ta có thể ánh xạ từng giá trị của biểu thức case tới một cổng đầu vào của bộ ghép kênh và kết nối kết quả được đánh giá tương ứng với cổng đó. Biểu thức case được kết nối với tín hiệu lựa chọn. Việc xây dựng tốt nhất có thể được giải thích bằng một ví dụ. Hãy xem xét các trạng thái trường hợp sau:



Sơ đồ khái niệm của câu lệnh này được thể hiện trong Hình 3.5 (b). Biến Sel có thể giả định bốn giá trị có thể có: 00, 01, 10 và 11. Nó ngụ ý một bộ ghép kênh 22-1 với sel là tín hiệu lựa chọn. Kết quả được đánh giá của a + b + c được chuyển đến r khi sel là 00, kết quả của a-b được chuyển đến r khi sel là 10 và kết quả của c + l được chuyển đến r khi sel là 01 hoặc 11.

Một lần nữa, lưu ý rằng tất cả các biểu thức giá trị được đánh giá đồng thời. Biến sel được sử dụng làm tín hiệu lựa chọn để định tuyến giá trị mong muốn đến đầu ra. Chiều rộng (số cổng đầu vào) của bộ ghép kênh tăng lên về mặt hình học với số lượng bit của sel.

Nói chung, mạng định tuyến ưu tiên hiệu quả hơn khi ưu tiên được đưa ra trong các điều kiện nhất định, chẳng hạn như đối với bộ mã hóa ưu tiên và mạng ghép kênh hiệu quả hơn đối với bảng sự thật hoặc mô tả dựa trên bảng hàm, chẳng hạn như đối với bộ giải mã nhị phân.

1. **Hướng dẫn viết mã chung cho một khối always**

Verilog dành cho cả mô hình hóa và tổng hợp. Trong khi viết mã để tổng hợp, chúng ta cần lưu ý về cách các cấu trúc ngôn ngữ khác nhau được ánh xạ tới phần cứng. Điều này đặc biệt đúng đối với khối always vì các biến và câu lệnh thủ tục có thể được sử dụng trong khối. Chúng ta nên nhớ rằng mục đích của mã là để suy luận phần cứng chứ không phải mô tả một thuật toán tuần tự trong C. Việc không làm như vậy thường xuyên dẫn đến các mã không thể tổng hợp, triển khai phức tạp không cần thiết hoặc sự khác biệt giữa mô phỏng và tổng hợp. Trong phần này, chúng ta xem xét một số lỗi phổ biến và đề xuất một bộ các nguyên tắc mã hóa.

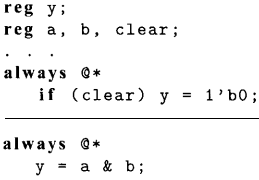
* 1. **Các lỗi thường gặp trong mã hóa mạch tổ hợp**

Sau đây là các lỗi thường gặp trong mã mạch tổ hợp:

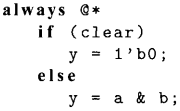
* Biến được chỉ định trong nhiều khối luôn luôn.
* Danh sách “sensitivity list” không đầy đủ.
* Phân nhánh không đầy đủ và chỉ định đầu ra không đầy đủ.

Những lỗi này được thảo luận dưới đây.

**Biến được gán trong nhiều khối always:** Trong Verilog, các biến có thể được chỉ định (xuất hiện ở phía bên trái) trong nhiều khối luôn luôn. Ví dụ, biến y được chia sẻ bởi hai khối luôn là đoạn mã sau:



Mặc dù mã chính xác về mặt cú pháp và có thể được mô phỏng, nhưng nó không thể được tổng hợp. Nhớ lại rằng mỗi khối always có thể được hiểu là một phần mạch. Đoạn mã trên chỉ ra rằng y là đầu ra của cả hai phần mạch và có thể được cập nhật theo từng phần. Không có mạch vật lý nào thể hiện loại hành vi này và do đó mã không thể được tổng hợp. Chúng ta phải nhóm các câu lệnh gán trong một khối luôn luôn duy nhất, như trong



**Danh sách “sensitivity list” không đầy đủ:** Đối với mạch tổ hợp, đầu ra là một chức năng của đầu vào và do đó bất kỳ thay đổi nào trong tín hiệu đầu vào sẽ kích hoạt mạch. Điều này ngụ ý rằng tất cả các tín hiệu đầu vào nên được đưa vào danh sách “sensitivity list”. Ví dụ, một cổng hai đầu vào và cổng có thể được viết là:



Nếu chúng ta quên thêm b, mã sẽ trở thành:



Mặc dù mã vẫn đúng về mặt cú pháp, nhưng hành vi của nó rất khác. Khi một thay đổi, khối luôn được kích hoạt và y nhận giá trị của a & b. Khi b thay đổi, khối always vẫn bị treo vì nó không nhạy cảm với b và y vẫn giữ giá trị trước đó của nó. Không có mạch vật lý nào thể hiện loại hành vi này. Hầu hết các phần mềm tổng hợp sẽ đưa ra một thông báo cảnh báo và thay vào đó suy ra cổng và. Tuy nhiên, phần mềm mô phỏng vẫn mô phỏng hành vi dự kiến và do đó tạo ra sự khác biệt giữa mô phỏng và tổng hợp.

Trong Verilog-2001, một ký hiệu đặc biệt “@ \*” được giới thiệu để bao gồm ngầm định tất cả các tín hiệu đầu vào có liên quan và do đó loại bỏ vấn đề này. Một thực tiễn tốt là sử dụng ký hiệu này để mô tả mạch tổ hợp.

**Nhánh chưa hoàn chỉnh và chỉ định đầu ra không đầy đủ:**

1. **Tham số và hằng số**
2. **Ư**
3. **Ư**
4. **Các thử nghiệm được đề suất**

Do hạn chế về mạch phần cứng nên các thử nghiệm trong phần này được thực hiện trên phần mềm mô phỏng modelSim. Toàn bộ mã nguồn được lưu giữ tại [github](https://github.com/namduongdinh2364/FPGA.git).

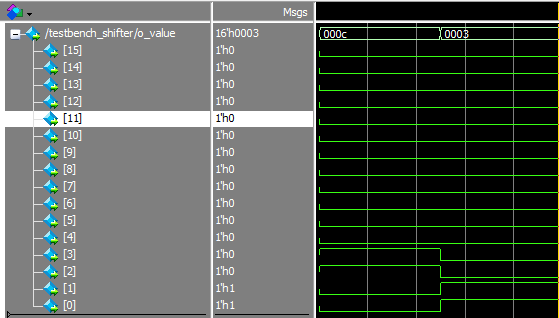
* 1. **Multifunction barrel shifter**

Thiết kế một mạch dịch 8-bit trái hoặc phải dựa vào tín hiệu lr từ đầu vào.

Các thành phần bao gồm:

* Mạch phần mạch dịch trái và dịch phải.
* Một mạch mux để lựa chọn đầu ra là phép dịch trái hay dịch phải.
* Testbench để kiểm tra chế độ vận hành của mạch

Kết quả đầu ra với đầu vào là: 0x06 sẽ được dịch trái và phải 1-bit.



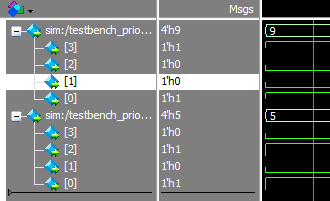
* 1. **Dual-priority encoder**

Các thành phần bao gồm:

* Mạch lấy ra highest encoder 12-4 và một mạch decoder 4-12.
* Một top-level kết hợp bộ encoder và decoder trên để lấy ra 2 mức ưu tiên cao nhất và cao thứ 2.
* Testbench để kiểm tra chế độ vận hành của mạch

Để có thể lấy ra được 2 mức ưu tiên, đầu tiên chúng ta sử dụng bộ encoder để lấy mức ưu tiên cao nhất, sau đó dùng bộ decoder để chuyển mức ưu tiên trên về 12-bit rồi tiếp theo tiến hành loại bỏ mức ưu tiên cao nhất khỏi đầu vào. Tiếp theo sau khi loại bỏ mức ưu tiên cao nhất từ đầu vào, chúng ta sẽ tiến hành cho kết quả này đi qua bộ encoder để lấy ra mức ưu tiên thứ 2.

Kết quả đầu ra với đầu vào là: 12'b000100010001 sẽ thu được mức ưu tiên cao nhất là 9 và thứ 2 là 5.



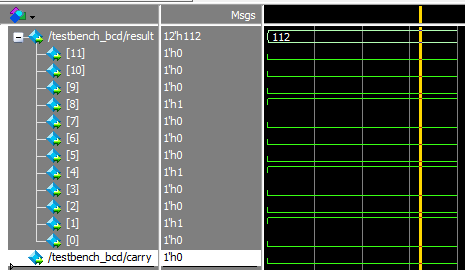
* 1. **BCD incrementor**

Bộ tăng BCD cho 3 số sẽ được tạo thành tử 3 bộ tăng BCD 1 số:

* Bộ tăng BCD 1 số.
* Top-level được tạo thành từ 3 bộ BCD 1 số.
* Testbench để kiểm tra chế độ vận hành của mạch.

Với việc tăng lên 1 đơn vị, chúng ra sẽ sử dụng 3 bộ BCD 1 số, khi BCD hàng đơn vị tăng đến giá trị giới hạn là 9 thì sẽ tiến hành tạo ra một bit nhớ làm đầu vào của BCD hàng đơn chục. Sau đó BCD hàng chục sẽ tăng giá trị. Khi đến giới hạn thì tiếp đó là BCD hàng trăm.

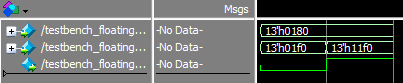
Kết quả thiết kệ bộ BCD tăng với đầu vào là: 12’d111 sẽ được đầu ra là 111 + 1=112.



* 1. **Floating-point greater-than circuit**

Thiết kế mạch so sách 2 số floating, nếu số đầu tiên lớn hơn thì sẽ tạo ra một tín hiệu xác nhận. Một số floating theo định dạng  với 1 bit đấu là bit thứ 13 và 4 bit mũ là 11 đến 8, cuối cùng là 8 bit độ lớn.

Kết quả so sánh giữa 1 và 1.875, 1 và -1.875.



* 1. **Floating-point and signed integer conversion circuit**

Thế kế mạch chuyển đổi từ số nguyên sang số float và từ số float sang số nguyên. Định dạng của số float như ví dụ ở phần 3.11.4.

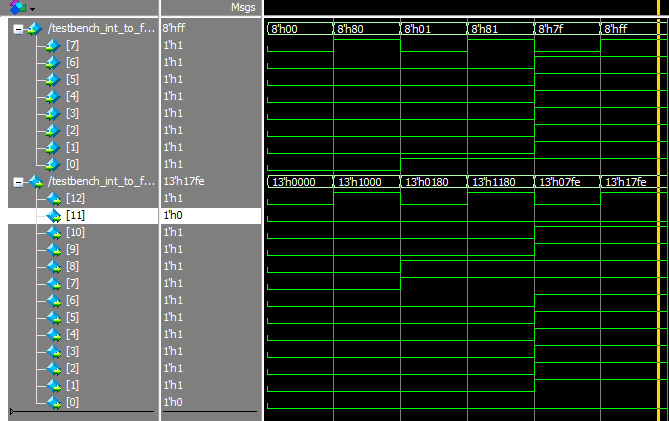


Figure 1 kết quả của việc chuyển một số nguyên sang dấu phẩy động

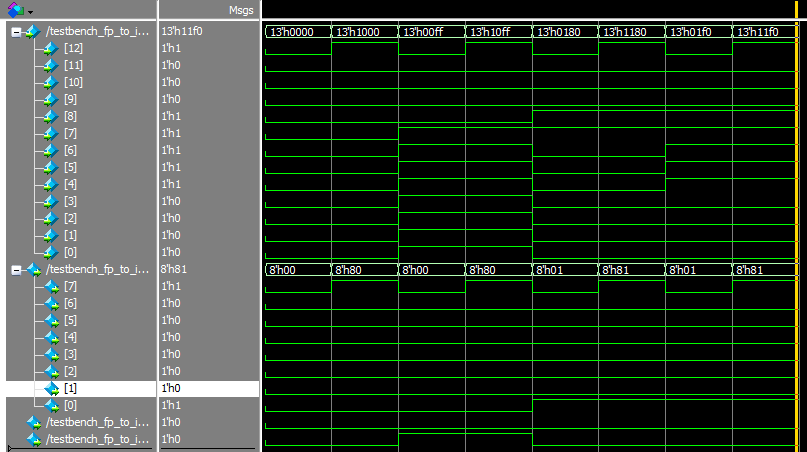


Figure 2 kết quả chuyển một số dấu phẩy động sang số nguyên

* 1. **Enhanced floating-point adder**

Với định dạng số float như trên, kết hợp với mô tả cộng 2 số dấu phẩy động như phần 3.9.4. Với yêu cầu đề bài, sử dụng thêm 3 tham số là guard, round, sticky để quyết định xem có nên làm tròn số float theo quy tắc “round to the nearest even” khi bình thường hóa số float bé hơn.

Kết quả của việc cộng 2 số phảy động lần lượt theo thứ tự: 0 + 1, 0 + (-1), 0.999 + 1, -1.875 + 1.

