ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

----------oOo----------

Giáo trình

NGÔN NGỮ MÔ TẢ PHẦN CỨNG VERILOG

Biên soạn: TS. Vũ Đức Lung

ThS. Lâm Đức Khải

Ks. Phan Đình Duy

2012

Lời nói đầu

Ngày nay, khi mạch thiết kế với hàng triệu cổng logic được tích hợp trong một con Chip thì việc thiết kế mạch và đi dây kết nối bằng tay trở nên bất khả thi, chính từ lí do đó một khái niệm ngôn ngữ có mức độ trừu tượng cao dùng để mô tả thiết kế phần cứng được ra đời, đó chính là Verilog. Cùng với sự ra đời của ngôn ngữ mô tả phần cứng Verilog là hàng loạt các công cụ EDA (Electronic Design Automation) và CAD (Computer Aided Design) đã giúp cho những kĩ sư thiết kế phần cứng tạo nên những con Chip có độ tích hợp rất cao, tốc độ siêu việt và chức năng đa dạng.

Giáo trình Ngôn ngữ mô tả phần cứng Verilog nhằm giúp sinh viên trang bị kiến thức về thiết kế vi mạch. Giáo trình tập trung vào mảng thiết kế các mạch số với mạch tổ hợp và mạch tuần tự. Giáo trình cũng giới thiệu về các bước cần thực hiện trong quá trình thiết kế vi mạch từ việc mô tả thiết kế, kiểm tra, phân tích cho đến tổng hợp phần cứng của thiết kế.

Giáo trình Ngôn ngữ mô tả phần cứng Verilog dùng cho sinh viên chuyên ngành Kĩ thuật máy tính và sinh viên các khối điện tử. Để tiếp nhận kiến thức dễ dàng, sinh viên cần trang bị trước kiến thức về thiết kế số và hệ thống số.

Giáo trình này được biên dịch và tổng hợp từ kinh nghiệm nghiên cứu giảng dạy của tác giả và ba nguồn tài liệu chính: IEEE Standard for Verilog Hardware Description Language, 2006; Verilog Digital System Design, Second Edition, McGraw-Hill; The Complete Verilog Book, Vivek Sagdeo, Sun Micro System, Inc.

Nhằm cung cấp một luồng kiến thức mạch lạc, giáo trình được chia ra làm 9 chương:

Chương 1: Dẫn nhập thiết kế hệ thống số với Verilog. Chương này sẽ giới thiệu lịch sử phát triển của ngôn ngữ mô tả phần cứng Verilog, bên

cạnh đó một qui trình thiết kế vi mạch sử dụng ngôn ngữ mô tả phần cứng Verilog cũng được trình bày cụ thể ở đây.

Chương 2: Trình bày các từ khóa được sử dụng trong môi trường mô tả thiết kế bởi Verilog.

Chương 3: Trình bày các loại dữ liệu được sử dụng trong thiết kế mạch bởi Verilog, gồm hai loại dữ liệu chính đó là loại dữ liệu net và loại dữ liệu biến.

Chương 4: Trình bày các toán tử cũng như các dạng biểu thức được hỗ trợ bởi Verilog.

Chương 5: Giới thiệu cấu trúc của một thiết kế, phương thức sử dụng thiết kế con.

Chương 6: Trình bày phương pháp thiết kế sử dụng mô hình cấu trúc, trong phương thức này, module thiết kế được xây dựng bằng cách gọi các module thiết kế nhỏ hơn và kết nối chúng lại.

Chương 7: Trình bày phương thức thiết kế sử dụng mô hình RTL bởi phép gán nối tiếp và mô hình hành vi sử dụng ngôn ngữ có tính trừu tượng cao tương tự như ngôn ngữ lập trình. Phần thiết kế máy trạng thái sử dụng mô hình hành vi cũng được nêu ra trong chương này.

Chương 8: Trình bày phương pháp thiết kế và sử dụng tác vụ và hàm.

Chương 9: Giới thiệu các phương pháp kiểm tra chức năng của thiết kế.

Do thời gian cũng như khối lượng trình bày giáo trình không cho phép tác giả đi sâu hơn về mọi khía cạnh của thiết kế vi mạch như phân tích định thời, tổng hợp phần cứng. Để có được những kiến thức này độc giả có thể tham khảo trong các tài liệu tham khảo mà giáo trình này đã cung cấp.

Mặc dù nhóm tác giả đã cố gắng biên soạn kỹ lưỡng tuy nhiên cũng khó tránh khỏi những thiếu sót. Nhóm tác giả mong nhận được những đóng góp mang tính xây dựng từ quí độc giả nhằm chỉnh sửa giáo trình hoàn thiện hơn.

Chương 1. Dẫn nhập thiết kế hệ thống số với Verilog

Khi kích thước và độ phức tạp của hệ thống thiết kế ngày càng tăng, nhiều công cụ hỗ trợ thiết kế trên máy tính (CAD) được sử dụng vào quá trình thiết kế phần cứng. Thời kì đầu, những công cụ mô phỏng và tạo ra phần cứng đã đưa ra phương pháp thiết kế, kiểm tra, phân tích, tổng hợp và tự động tạo ra phần cứng một cách phức tạp. Sự phát triển không ngừng của những công cụ thiết kế một cách tự động là do sự phát triển của những ngôn ngữ mô tả phần cứng (HDLs) và những phương pháp thiết kế dựa trên những ngôn ngữ này. Dựa trên những ngôn ngữ mô tả phần cứng (HDLs), những công cụ CAD trong thiết kế hệ thống số được phát triển và được những kĩ sư thiết kế phần cứng sử dụng rộng rãi. Hiện tại, người ta vẫn đang tiếp tục nghiên cứu để tìm ra những ngôn ngữ mô tả phần cứng tốt hơn. Một trong những ngôn ngữ mô tả phần cứng được sử dụng rộng rãi nhất đó là ngôn ngữ Verilog HDL. Do được chấp nhận rộng rãi trong ngành công nghiệp thiết kế số, Verilog đã trở thành một kiến thức được đòi hỏi phải biết đối với những kĩ sư cũng như sinh viên làm việc và học tập trong lĩnh vực phần cứng máy tính.

Chương này sẽ trình bày những công cụ và môi trường làm việc có sẵn tương thích với ngôn ngữ Verilog mà một kĩ sư thiết kế có thể sử dụng trong qui trình thiết kế tự động của mình để giúp đẩy nhanh tiến độ thiết kế. Đầu tiên sẽ trình bày từng bước về thiết kế phân cấp, thiết kế mức cao từ việc mô tả thiết kế bằng ngôn ngữ Verilog đến việc tạo ra phần cứng của thiết kế đó. Những qui trình và những từ khóa chuyên môn cũng sẽ được minh họa ở phần này. Kế tiếp sẽ thảo luận những công cụ CAD hiện có tương thích với Verilog và chức năng của nó trong môi trường thiết kế tự động. Phần cuối cùng của chương này sẽ nói về một số đặc tính của

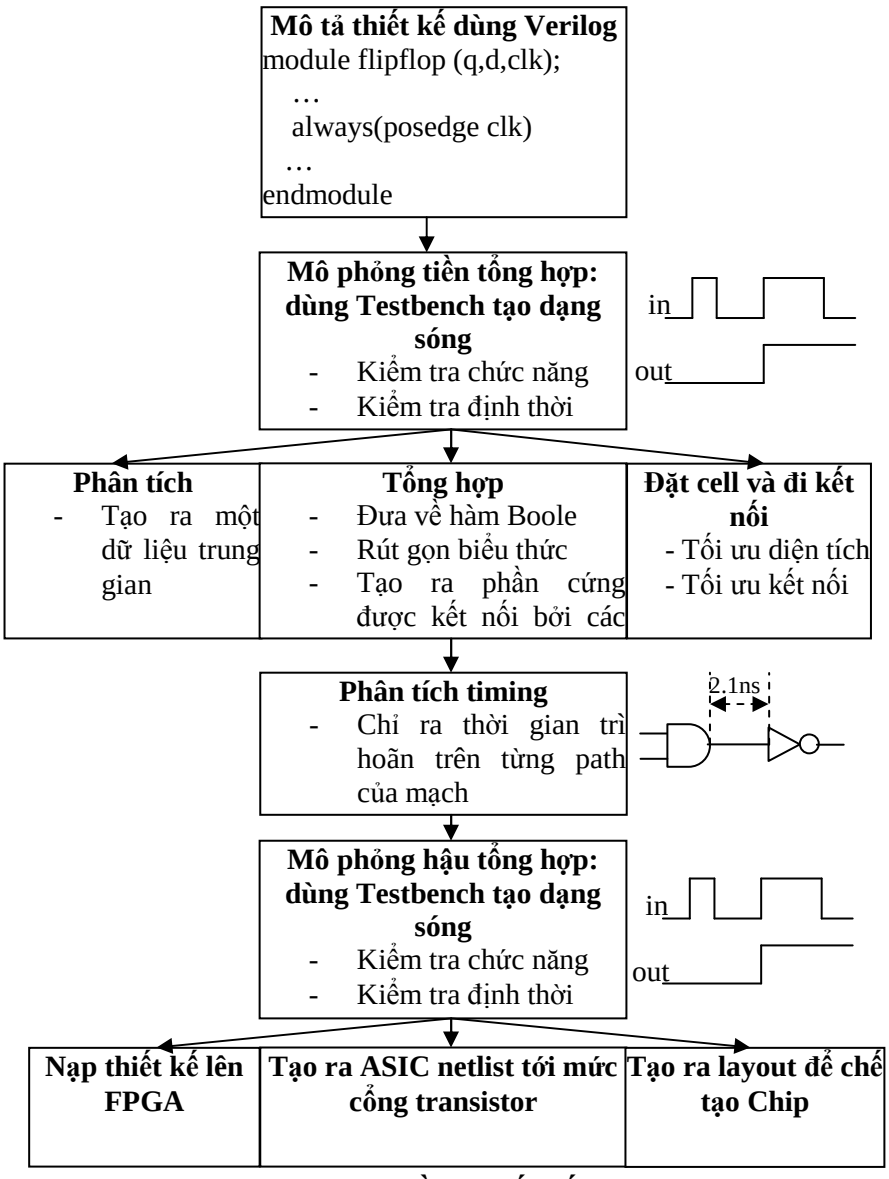
Verilog khiến nó trở thành một ngôn ngữ được nhiều kĩ sư thiết kế phần cứng lựa chọn.

1.1 Qui trình thiết kế số

Trong thiết kế một hệ thống số sử dụng môi trường thiết kế tự động, qui trình thiết kế bắt đầu bằng việc mô tả thiết kế tại nhiều mức độ trừu tượng khác nhau và kết thúc bằng việc tạo ra danh sách các linh kiện cũng như các đường kết nối giữa các linh kiện với nhau (netlist) cho một mạch tích hợp với ứng dụng cụ thể (ASIC), mạch in (layout) cho một mạch tích hợp theo yêu cầu khách hàng (custom IC), hoặc một chương trình cho một thiết bị logic có khả năng lập trình được (PLD). Hình 1.1 mô tả từng bước trong qui trình thiết kế này.

Bước đầu của thiết kế, một thiết kế sẽ được mô tả bởi sự hỗn hợp giữa mô tả ở mức độ hành vi (behavioural) Verilog, sử dụng những gói (module) thiết kế Verilog đã được thiết kế sẵn, và việc gán hệ thống các bus và wire để liên kết các gói thiết kế này thành một hệ thống hoàn chỉnh. Kĩ sư thiết kế cũng phải có trách nhiệm tạo ra dữ liệu để kiểm tra (testbench) xem thiết kế đúng chức năng hay chưa cũng như dùng để kiểm tra thiết kế sau khi tổng hợp. Việc kiểm tra thiết kế có thể thực hiện được bằng việc mô phỏng, chèn những kĩ thuật kiểm tra, kiểm tra thông thường hoặc kết hợp cả ba phương pháp trên. Sau bước kiểm tra đánh giá thiết kế (bước này được gọi là kiểm tra tiền tổng hợp (presynthesis verification)), thiết kế sẽ được tiếp tục bằng việc tổng hợp để tạo ra phần cứng thực sự cho hệ thống thiết kế cuối cùng (ASIC, custom IC or FPLD,…). Nếu hệ thống thiết kế là ASIC, thiết kế sẽ sẽ được sản xuất bởi nhà sản xuất khác; nếu là custom IC, thiết kế sẽ được sản xuất trực tiếp; nếu là FPLD, thiết kế sẽ được nạp lên thiết bị lập trình được. Sau bước tổng hợp và trước khi

phần cứng thực sự được tạo ra, một quá trình mô phỏng khác (hậu tổng hợp (postsynthesis)) phải được thực hiện. Việc mô phỏng này, ta có thể sử dụng testbench tương tự testbench đã sử dụng trong mô phỏng tiền tổng hợp (presynthesis). Bằng phương pháp này, mô hình thiết kế ở mức độ hành vi và mô hình phần cứng của thiết kế được kiểm tra với cùng dữ liệu ngõ vào. Sự khác nhau giữa mô phỏng tiền tổng hợp và hậu tổng hợp đó là mức độ chi tiết có thể đạt được từ mỗi loại mô phỏng.



Hình 1.1 Luồng thiết kế ASIC

Những phần tiếp theo sẽ mô tả tỉ mỉ về mỗi khối trong Hình 1.1

1.1.1 Dẫn nhập thiết kế

Bước đầu tiên trong thiết kế hệ thống số là bước dẫn nhập thiết kế. Trong bước này, thiết kế được mô tả bằng Verilog theo phong cách phân cấp từ cao xuống thấp (top-down). Một thiết kế hoàn chỉnh có thể bao gồm những linh kiện ở mức cổng hoặc mức transistor, những khối (module) phần cứng có chức năng phức tạp hơn được mô tả ở mức độ hành vi, hoặc những linh kiện được liệt kê bởi cấu trúc bus.

Do những thiết kế Verilog ở mức cao thường được mô tả ở mức độ mà tại đó nó mô tả hệ thống những thanh ghi và sự truyền dữ liệu giữa những thanh ghi này thông qua hệ thống bus, việc mô tả hệ thống thiết kế ở mức độ này được xem như là mức độ truyền dữ liệu giữa các thanh ghi (RTL). Một thiết kế hoàn chỉnh được mô tả như vậy sẽ tạo ra được phần cứng tương ứng thực sự rõ ràng. Những cấu trúc thiết kế Verilog ở mức độ RTL sử dụng những phát biểu qui trình (producedural statements), phép gán liên tục (continuous assignments), và những phát biểu gọi sử dụng khối (module) đã xây dựng sẵn.

Những phát biểu qui trình Verilog (procedural statements) được dùng để mô tả mức độ hành vi ở mức cao. Một hệ thống hoặc một linh kiện được mô tả ở mức độ hành vi thì tương tự với việc mô tả trong ngôn ngữ phần mềm. Ví dụ, chúng ta có thể mô tả một linh kiện bằng việc kiểm tra điều kiện ngõ vào của nó, bật cờ hiệu, chờ cho đến khi có sự kiện nào đó xảy ra, quan sát những tín hiệu bắt tay và tạo ra ngõ ra. Mô tả hệ thống một cách qui trình như vậy, cấu trúc if-else, case của Verilog cũng như những ngôn ngữ phần mềm khác đều sử dụng như nhau.

Những phép gán liên tục (continuous assignment) trong Verilog là những phép gán cho việc thể hiện chức năng những khối logic, những phép

gán bus, và mô tả việc kết nối giữa hệ thống bus và các chân ngõ vào và ngõ ra. Kết hợp với những hàm Boolean và những biểu thức có điều kiện, những cấu trúc ngôn ngữ này có thể được để mô tả những linh kiện và hệ thống theo những phép gán thanh ghi và bus của chúng.

Những phát biểu gọi sử dụng khối Verilog đã được thiết kế sẵn (instantiantion statements) được dùng cho những linh kiện mức thấp trong một thiết kế ở mức độ cao hơn. Thay vì mô tả ở mức độ hành vi, chức năng, hoặc bus của một hệ thống, chúng ta có thể mô tả một hệ thống bằng Verilog bằng cách kết nối những linh kiện ở mức độ thấp hơn. Những linh kiện này có thể nhỏ như là mức cổng hay transistor, hoặc có thể lớn như là một bộ vi xử lí hoàn chỉnh.

1.1.2 Testbench trong Verilog

Một hệ thống được thiết kế dùng Verilog phải được mô phỏng và kiểm tra xem thiết kế xem đã đúng chức năng chưa trước khi tạo ra phần cứng. Trong quá trình chạy mô phỏng này, những lỗi thiết kế và sự không tương thích giữa những linh kiện dùng trong thiết kế có thể được phát hiện. Chạy mô phỏng một thiết kế đòi hỏi việc tạo ra một dữ liệu ngõ vào kiểm tra và quá trình quan sát kết quả sau khi chạy mô phỏng, dữ liệu dùng để kiểm tra này được gọi là testbench. Một testbench sử dụng cấu trúc mức cao của Verilog để tạo ra dữ liệu kiểm tra, quan sát đáp ứng ngõ ra, và cả việc bắt tay giữa những tín hiệu trong thiết kế. Bên trong testbench, hệ thống thiết kế cần chạy mô phỏng sẽ được gọi ra (instantiate) trong testbench. Dữ liệu testbench cùng với hệ thống thiết kế sẽ tạo ra một mô hình mô phỏng mà sẽ được sử dụng bởi một công cụ mô phỏng Verilog.

1.1.3 Đánh giá thiết kế

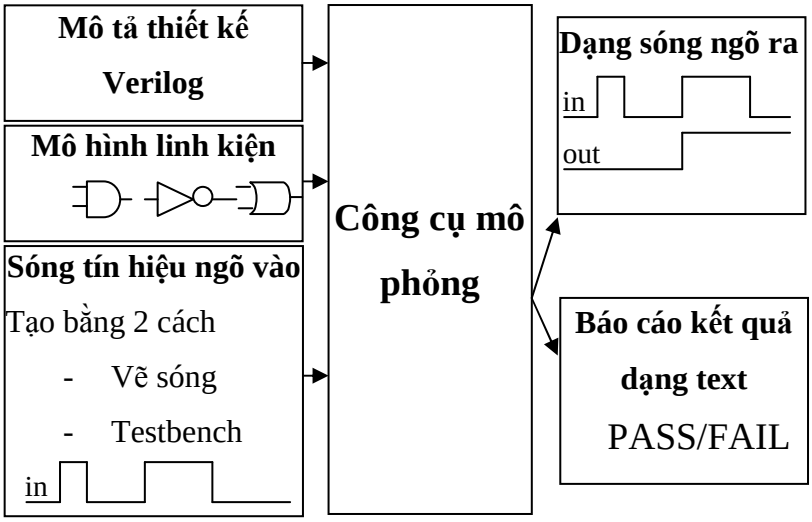
Một nhiêm vụ quan trọng trong bất kì thiết kế số nào cũng cần đó là đánh giá thiết kế. Đánh giá thiết kế là quá trình mà người thiết kế sẽ kiểm tra thiết kế của họ có sai sót nào có thể xảy ra trong suốt quá trình thiết kế hay không. Một sai sót thiết kế có thể xảy ra do sự mô tả thiết kế mơ hồ, do sai sót của người thiết kế, hoặc sử dụng không đúng những khối trong thiết kế. Đánh giá thiết kế có thể thực hiện bằng mô phỏng, bằng việc chèn những kĩ thuật kiểm tra, hoặc kiểm tra thông thường.

1.1.3.1 Mô phỏng

Chạy mô phỏng dùng trong việc đánh giá thiết kế được thực hiện trước khi thiết kế được tổng hợp. Bước chạy mô phỏng này được hiểu như mô phỏng ở mức độ hành vi, mức độ RTL hay tiền tổng hợp. Ở mức độ RTL, một thiết kế bao gồm xung thời gian clock nhưng không bao gồm trí hoãn thời gian trên cổng và dây kết nối (wire). Chạy mô phỏng ở mức độ này sẽ chính xác theo xung clock. Thời gian của việc chạy mô phỏng ở mức độ RTL là theo tín hiệu xung clock, không quan tâm đến những vấn đề như: nguy hiểm tiềm ẩn có thể khiến thiết kế bị lỗi (hazards, glitch), hiện tượng chạy đua không kiểm soát giữa những tín hiệu (race conditions), những vi phạm về thời gian setup và hold của tín hiệu ngõ vào, và những vấn đề liên quan đến định thời khác. Ưu điểm của việc mô phỏng này là tốc độ chạy mô phỏng nhanh so với chạy mô phỏng ở mức cổng hoặc mức transistor.

Chạy mô phỏng cho một thiết kế đòi hỏi dữ liệu kiểm tra. Thông thường trong môi trường mô phỏng Verilog sẽ cung cấp nhiều phương pháp khác nhau để đưa dữ liệu kiểm tra này vào thiết kế để kiểm tra. Dữ liệu kiểm tra có thể được tạo ra bằng đồ họa sử dụng những công cụ soạn

thảo dạng sóng, hoặc bằng testbench. Hình 1.2 mô tả hai cách khác nhau để định nghĩa dữ liệu kiểm tra ngõ vào của một công cụ mô phỏng. Những ngõ ra của công cụ mô phỏng là những dạng sóng ngõ ra (có thể quan sát trực quan).



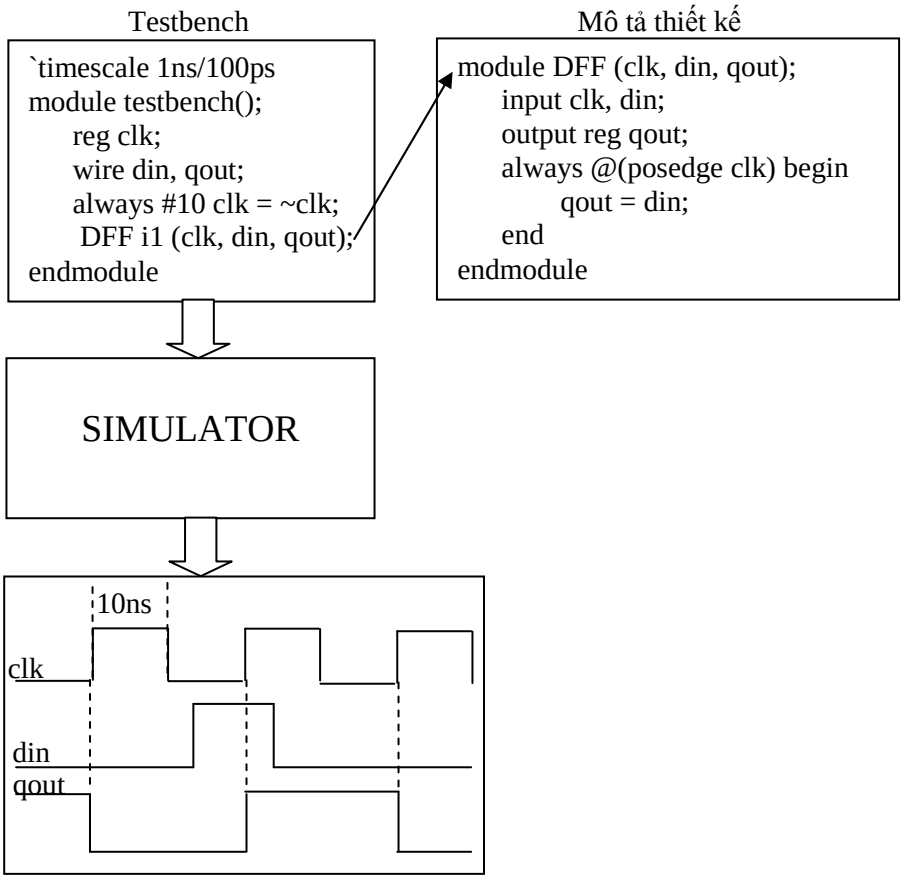
Hình 1.2 Hai cách khác nhau để định nghĩa dữ liệu kiểm tra ngõ vào

Để chạy mô phỏng với Verilog testbench, trong testbench sẽ gọi hệ thống thiết kế ra để kiểm tra, lúc này hệ thống thiết kế được xem như là một phần của testbench, testbench sẽ cung cấp dữ liệu kiểm tra đến ngõ vào của hệ thống thiết kế. Hình1.3 mô tả một đoạn code của một mạch đếm, testbench của nó, cũng như kết quả chạy mô phỏng của nó dưới dạng sóng ngõ ra. Quan sát hình ta thấy việc chạy mô phỏng sẽ đánh giá chức năng của mạch đếm. Với mỗi xung clock thì ngõ ra bộ đếm sẽ tăng lên 1. Chú ý rằng, theo biểu đồ thời gian thì ngõ ra bộ đếm thay đổi tại cạnh lên xung clock và không có thời gian trì hoãn do cổng cũng như trì hoãn trên đường truyền. Kết quả chạy mô phỏng cho thấy chức năng của mạch đếm là chính xác mà không cần quan tâm đến tần số xung clock.

Hiển nhiên, những linh kiện phần cứng thực sự sẽ có đáp ứng khác nhau. Dựa trên định thời và thời gian trì hoãn của những khối được sử

dụng, thời gian từ cạnh lên xung clock đến ngõ ra của bộ đếm sẽ có độ trì hoãn khác không. Hơn nữa, nếu tần số xung clock được cấp vào mạch thực sự quá nhanh so với tốc độ truyền tín hiệu bên trong các cổng và transistor của thiết kế thì ngõ ra của thiết kế sẽ không thể biết được.

Việc mô phỏng này không cung cấp chi tiết về các vấn đề định thời của hệ thống thiết kế được mô phỏng. Do đó, những vấn đề tiềm ẩn về định thời của phần cứng do trì hoãn trên cổng sẽ không thể phát hiện được. Đây là vấn đề điển hình của quá trỉnh mô phỏng tiền tổng hợp hoặc mô phỏng ở mức độ hành vi. Điều biết được trong Hình1.3 đó là bộ đếm của ta đếm số nhị phân. Thiết kế hoạt động nhanh chậm thế nào, hoạt đông được ở tần số nào chỉ có thể biết được bằng việc kiểm tra thiết kế sau tổng hợp.



Hình1.3 Mô tả một đoạn code của một mạch flip-flop

1.1.3.2 Kĩ thuật chèn kiểm tra (assertion)

Thay vì phải dò theo kết quả mô phỏng bằng mắt hay tạo những dữ liệu kiểm tra testbench phức tạp, kĩ thuật chèn thiết bị giám sát có thể được sử dụng để kiểm tra tuần tự những đặc tính của thiết kế trong suốt quá trình mô phỏng. Thiết bị giám sát được đặt bên trong hệ thống thiết kế được mô phỏng bởi người thiết kế. Người thiết kế sẽ quyết định xem chức năng của thiết kế đúng hay sai, những điều kiện nào thiết kế cần phải thỏa mãn. Những điều kiện này phải tuân theo những đặc tính thiết kế, và thiết bị giám sát được chèn vào hệ thống thiết kế để đảm bảo những đặc tính này không bị vi phạm. Chuỗi thiết bị giám sát này sẽ sai nếu một đặc tính nào đó được đặt vào bởi người thiết kế bị vi phạm. Nó sẽ cảnh báo người thiết kế rằng thiết kế đã không đúng chức năng như mong đợi. Thư viện OVL (Open Verification Library) cung cấp một chuỗi những thiết bị giám sát để chèn vào hệ thống thiết kế để giám sát những đặc tính thông thường của thiết kế. Người thiết kế có thể dùng những kĩ thuật giám sát của riêng mình để chèn vào thiết kế và dùng chúng kết hợp với testbench trong việc kiểm tra đánh giá thiết kế.

1.1.3.3 Kiểm tra thông thường

Kiểm tra thông thường là quá trình kiểm tra những đặc tính bất kì của thiết kế. Khi một thiết kế hoàn thành, người thiết kế sẽ xây dựng một chuỗi những đặc tính tương ứng với hành vi của thiết kế. Công cụ kiểm tra thông thường sẽ kiểm tra thiết kế để đảm bảo rằng những đặc tính được mô tả đáp ứng được tất cả những điều kiện. Nếu có một đặc tính được phát hiện là không đáp ứng đúng, đặc tính đó được xem như vi phạm. Đặc tính độ bao phủ (coverage) chỉ ra bao nhiêu phần trăm đặc tính của thiết kế đã được kiểm tra.

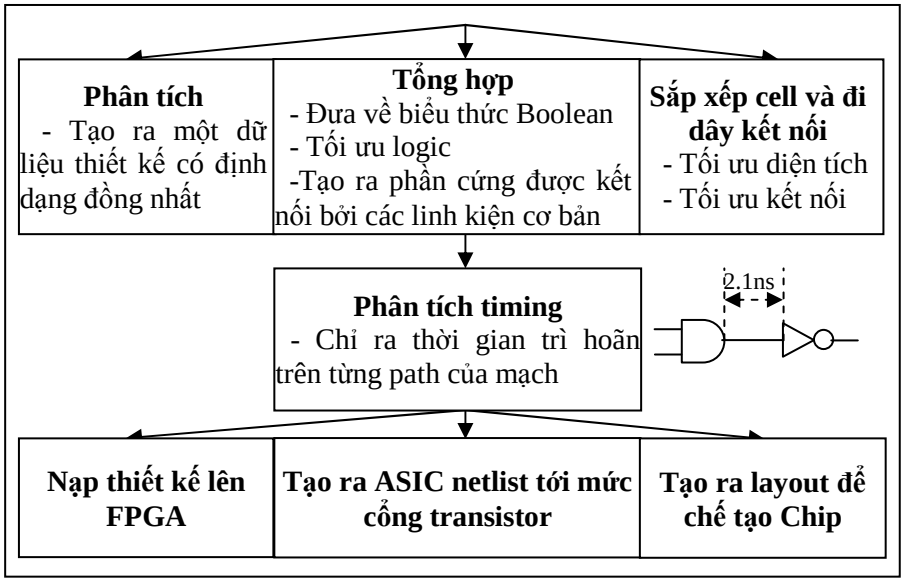
1.1.4 Biên dịch và tổng hợp thiết kế

Tổng hợp là quá trình tạo ra phần cứng tự động từ một mô tả thiết kế phần cứng tương ứng rõ ràng. Một mô tả phần cứng Verilog dùng để tổng hợp không thể bao gồm tín hiệu và mô tả định thời ở mức cổng, và những cấu trúc ngôn ngữ khác mà không dịch sang những phương trình logic tuần tự hoặc tổ hợp. Hơn thế nữa, những mô tả phần cứng Verilog dùng cho tổng hợp phải tuân theo những phong cách viết code một cách nhất định cho mạch tổ hợp cũng như mạch tuần tự. Những phong cách này và cấu trúc Verilog tương ứng của chúng được định nghĩa trong việc tổng hợp RTL.

Trong qui trình thiết kế, sau khi một thiết kế được mô tả hoàn thành và kết quả mô phỏng tiền tổng hợp của nó được kiểm tra bởi người thiết kế, nó phải được biên dịch để tiến gần hơn đến việc tạo thành phần cứng thực sự trên silicon. Bước thiết kế này đòi hỏi việc mô tả phần cứng của thiết kế phải được nhận ra. Ví dụ, chúng ta phải chỉ đến một ASIC cụ thể, hoặc một FPGA cụ thể như là thiết bị phần cứng mục đích của thiết kế. Khi thiết bị mục đích được chỉ ra, những tập tin mô tả về công nghệ (technology files) của phần cứng (ASIC, FPGA, hoặc custom IC) sẽ cung cấp chi tiết những thông tin về định thời và mô tả chức năng cho quá trình biên dịch. Quá trình biên dịch sẽ chuyển đổi những phần khác nhau của thiết kế ra một định dạng trung gian (bước phân tích), kết nối tất cả các phần lại với nhau, tạo ra mức logic tương ứng (bước tổng hợp), sắp xếp và kết nối (place and route) những linh kiện trong thiết bị phần cứng mục đích lại với nhau để thực hiên chức năng như thiết kế mong muốn và tạo ra thông tin chi tiết về định thời trong thiết kế.

Hình 1.4 mô tả quá trình biên dịch và mô tả hình ảnh kết quả ngõ ra của mỗi bước biên dịch. Như trên hình, ngõ vào của bước này là một mô tả

phần cứng bao gồm những mức độ mô tả khác nhau của Verilog, và kết quả ngõ ra của nó là một phần cứng chi tiết cho thiết bị phần cứng mục đích như FPLD hay để sản xuất chip ASIC.



Hình 1.4 Mô tả quá trình biên dịch và mô tả hình ảnh kết quả ngõ ra

1.1.4.1 Phân tích

Một thiết kế hoàn chỉnh được mô tả dùng Verilog có thể bao gồm mô tả ở nhiều mức độ khác nhau như mức độ hành vi, hệ thống bus và dây kết nối với những linh kiện Verilog khác. Trước khi một thiết kế hoàn chỉnh tạo ra phần cứng, thiết kế phải được phân tích và tạo ra một định dạng đồng nhất cho tất cả các phần trong thiết kế. Bước này cũng kiểm tra cú pháp và ngữ nghĩa của mã ngõ vào Verilog.

1.1.4.2 Tạo phần cứng

Sau khi tạo được một dữ liệu thiết kế có định dạng đồng nhất cho tất cả các linh kiện trong thiết kế, bước tổng hợp sẽ bắt đầu bằng chuyển đổi dữ liệu thiết kế trên sang những định dạng phần cứng thông thường như một chuỗi những biểu thức Boolean hay một netlist những cổng cơ bản.

1.1.4.3 Tối ưu logic

Bước kế tiếp của quá trình tổng hợp, sau khi một thiết kế được chuyển đổi sang một chuỗi những biểu thức Boolean, bước tối ưu logic được thực hiện. Bước này nhằm mục đích làm giảm những biểu thức với ngõ vào không đổi, loại bỏ những biểu thức lập lại, tối thiểu hai mức, tối thiểu nhiều mức.

Đây là quá trình tính toán rất hao tốn thời gian và công sức, một số công cụ cho phép người thiết kế quyết định mức độ tối ưu. Kết quả ngõ ra của bước này cũng dưới dạng những biểu thức Boolean, mô tả logic dưới dạng bảng, hoặc netlist gồm những cổng cơ bản.

1.1.4.4 Binding

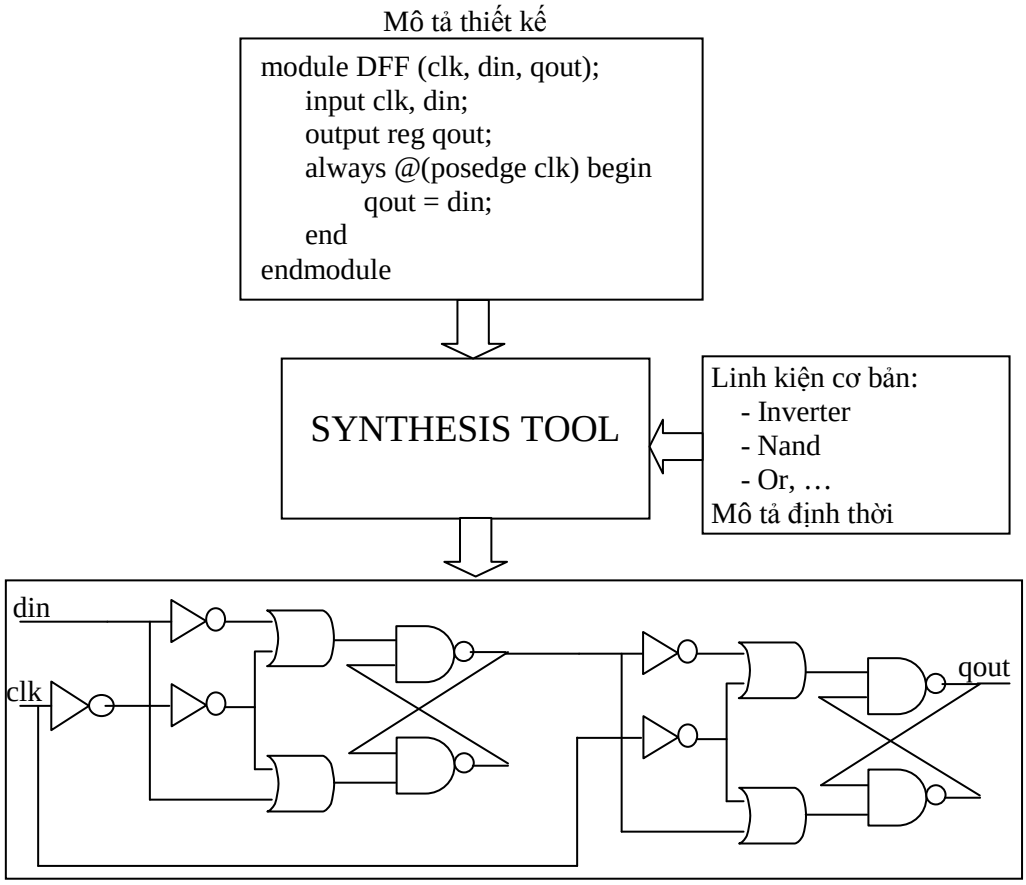
Sau bước tối ưu logic, quá trình tổng hợp sử dụng thông tin từ thiết bị phần cứng mục đích để quyết định chính xác linh kiện logic nào và thiết bị nào cần để hiện thực mạch thiết kế. Quá trình này được gọi là binding và kết quả ngõ ra của nó được chỉ định cụ thể sử dụng cho FPLD, ASIC, hay custom IC.

1.1.4.5 Sắp xếp cell và đi dây kết nối

Bước sắp xếp và đi dây kết nối sẽ quyết định việc đặt vị trí của các linh kiện trên thiết bị phần cứng mục đích. Việc kết nối các ngõ vào và ngõ ra của những linh kiện này dùng hệ thống dây liên kết và vùng chuyển mạch trên thiết bị phần cứng mục đích được quyết định bởi bước sắp xếp và đi dây liên kết này. Kết quả ngõ ra của bước này được đưa tới thiết bị phần cứng mục đích, như nạp lên FPLD, hay dùng để sản xuất ASIC.

Một ví dụ minh họa về quá trình tổng hợp được chỉ ra trên Hình 1.5. Trong hình này, mạch đếm đã được dùng chạy mô phỏng trong hình 1.3

được tổng hợp. Ngoài việc mô tả phần cứng thiết kế dùng Verilog, công cụ tổng hợp đòi hỏi những thông tin mô tả thiết bị phần cứng đích để tiến hành quá trình tổng hợp của mình. Kết quả ngõ ra của công cụ tổng hợp là danh sách các cổng và flip-flop có sẵn trong thiết bị phần cứng đích và hệ thống dây kết nối giữa chúng. Hình 1.5 cũng chỉ ra một kết quả ngõ ra mang tính trực quan mà đã được tạo ra tự động bằng công cụ tổng hợp của Altera Quartus II.



Hình 1.5 Minh họa về quá trình tổng hợp

1.1.5 Mô phỏng sau khi tổng hợp thiết kế

Sau khi quá trình tổng hợp hoàn thành, công cụ tổng hợp sẽ tạo ra một netlist hoàn chỉnh chứa những linh kiện của thiết bị phần cứng đích và các giá trị định thời của nó. Những thông tin chi tiết về các cổng được dùng để hiện thực thiết kế cũng được mô tả trong netlist này. Netlist này cũng

bao gồm những thông tin về độ trì hoãn trên đường dây và những tác động của tải lên các cổng dùng trong quá trình hậu tổng hợp. Có nhiều định dạng netlist ngõ ra có thể được tạo ra bao gồm cả định dạng Verilog. Một netlist như vậy có thể được dùng để mô phỏng, và mô phỏng này được gọi là mô phỏng hậu tổng hợp. Những vấn đề về định thời, về tần số xung clock, về hiện tượng chạy đua không kiểm soát, những nguy hiểm tiềm ẩn của thiết kế chỉ có thể kiểm tra bằng mô phỏng hậu tổng hợp thực hiện sau khi thiết kế được tổng hợp. Như trên Hình 1.1, ta có thể sử dụng dữ liệu kiểm tra mà đã dùng cho quá trình mô phỏng tiền tổng hợp để dùng cho quá trình mô phỏng hậu tổng hợp.

Do độ trì hoãn trên đường dây và các cổng, đáp ứng của thiết kế sau khi chạy mô phỏng hậu tổng hợp sẽ khác với đáp ứng của thiết kế mà người thiết kế mong muốn. Trong trường hợp này, người thiết kế phải sửa lại thiết kế và cố gắng tránh những sai sót về định thời và hiện tượng chạy đua giữa những tín hiệu mà không thể kiểm soát.

1.1.6 Phân tích thời gian

Quan sát trên Hình 1.1, bước phân tích thời gian là một phần trong quá trình biên dịch, hoặc trong một số công cụ thì bước phân tích thời gian này được thực hiện sau quá trình biên dịch. Bước này sẽ tạo ra khả năng xấu nhất về độ trì hoãn , tốc độ xung clock, độ trì hoãn từ cổng này đến cổng khác, cũng như thời gian cho việc thiết lập và giữ tín hiệu. Kết quả của bước phân tích thời gian được thể hiện dưới dạng bảng hoặc biểu đồ. Người thiết kế sử dụng những thông tin này để xác định tốc độ xung clock, hay nói cách khác là xác định tốc độ hoạt động của mạch thiết kế.

1.1.7 Tạo linh kiện phần cứng

Bước cuối cùng trong qui trình thiết kế tự động dựa trên Verilog đó là tạo ra phần cứng thực sự cho thiết kế. Bước này có thể tạo ra một netlist dùng để sản xuất ASIC, một chương trình để nạp vào FPLD, hay một mạch in cho mạch IC.

1.2 Ngôn ngữ phần cứng Verilog (Verilog HDL)

Trong phần trước, ta đã trình bày từng bước thiết kế ở mức độ RTL từ một mô tả thiết kế Verilog cho đến việc hiện thực ra một phần cứng thực sự. Qui trình thiết kế này chỉ có thể thực hiện được khi ngôn ngữ Verilog có thể hiểu được bởi người thiết kế hệ thống, người thiết kế ở mức độ RTL, người kiểm tra, công cụ mô phỏng, công cụ tổng hợp, và các máy móc liên quan. Bởi vì tầm quan trọng của nó trong qui trình thiết kế, Verilog đã trở thành một chuẩn quốc tế IEEE. Chuẩn này được sử dụng bởi người thiết kế cũng như người xây dựng công cụ thiết kế.

1.2.1 Quá trình phát triển Verilog

Verilog được ra đời vào đầu năm 1984 bởi Gateway Design Automation. Khởi đầu, ngôn ngữ đầu tiên được dùng như là một công cụ mô phỏng và kiểm tra. Sau thời gian đầu ngôn ngữ này được chấp nhận bởi ngành công nghiệp điện tử, một công cụ mô phỏng, một công cụ phân tích thời gian, và sau này vào năm 1987, công cụ tổng hợp đã được xây dựng và phát triển dựa vào ngôn ngữ này. Gateway Design Automation và những công cụ dựa trên Verilog của hãng sau này được mua bởi Cadence Design System. Từ sau đó, Cadence đóng vai trò hết sức quan trọng trong việc phát triển cũng như phổ biến ngôn ngữ mô tả phần cứng Verilog.

Vào năm 1987, VHDL trở thành một chuẩn ngôn ngữ mô tả phần cứng của IEEE. Bởi do sự hỗ trợ của Bộ quốc phòng (DoD), VHDL được sử dụng nhiều trong những dự án lớn của chính phủ Mỹ. Trong nỗ lực phổ biến Verilog, vào năm 1990, OVI (Open Verilog International) được thành lập và Verilog chiếm ưu thế trong lĩnh vực công nghiệp. Điều này đã tạo ra một sự quan tâm khá lớn từ người dùng và các nhà cung cấp EDA (Electronic Design Automation) tới Verilog.

Vào năm 1993, những nỗ lực nhằm chuẩn hóa ngôn ngữ Verilog được bắt đầu. Verilog trở thành chuẩn IEEE, IEEE Std 1364-1995, vào năm 1995. Với những công cụ mô phỏng, công cụ tổng hợp, công cụ phân tích thời gian, và những công cụ thiết kế dựa trên Verilog đã có sẵn, chuẩn Verilog IEEE này nhanh chóng được chấp nhận sâu rộng trong cộng đồng thiết kế điện tử.

Một phiên bản mới của Verilog được chấp nhận bởi IEEE vào năm 2001. Phiên bản mới này được xem như chuẩn Verilog-2001 và được dùng bởi hầu hết người sử dụng và người phát triển công cụ. Những đặc điểm mới trong phiên bản mới đó là nó cho phép bên ngoài có khả năng đọc và ghi dữ liệu, quản lí thư viện, xây dựng cấu hình thiết kế, hỗ trợ những cấu trúc có mức độ trừu tượng cao hơn, những cấu trúc mô tả sự lặp lại, cũng như thêm một số đặc tính vào phiên bản này. Quá trình cải tiến chuẩn này vẫn đang được tiếp tục với sự tài trợ của IEEE.

1.2.2 Những đặc tính của Verilog

Verilog là một ngôn ngữ mô tả phần cứng dùng để đặc tả phần cứng từ mức transistor đến mức hành vi. Ngôn ngữ này hỗ trợ những cấu trúc định thời cho việc mô phỏng định thời ở mức độ chuyển mạch và tức thời, nó cũng có khả năng mô tả phần cứng tại mức độ thuật toán trừu tượng. Một mô tả thiết kế Verilog có thể bao gồm sự trộn lẫn giữa những khối

(module) có mức độ trừu tượng khác nhau với sự khác nhau về mức độ chi tiết.

1.2.2.1 Mức độ chuyển mạch

Những đặc điểm của ngôn ngữ này khiến nó trở nên lí tưởng trong việc mô hình hóa và mô phỏng ở mức độ chuyển mạch bao gồm khả năng chuyển mạch một chiều cũng như hai chiều với những thông số về độ trì hoãn và lưu trữ điện tích. Những trì hoãn mạch điện có thể được mô hình hóa như là trì hoãn đường truyền, trì hoãn từ thấp lên cao hay từ cao xuống thấp. Đặc điểm lưu trữ điện tích ở mức độ trừu tượng trong Verilog khiến nó có khả năng mô tả những mạch điện với linh kiện động như là CMOS hay MOS.

1.2.2.2 Mức độ cổng

Những cổng cơ bản với những thông số được định nghĩa trước sẽ cung cấp một khả năng thuận tiện trong việc thể hiện netlist và mô phỏng ở mức cổng. Đối với việc mô phỏng mức cổng với mục đích chi tiết và đặc biệt, những linh kiện cổng có thể được định nghĩa ở mức độ hành vi. Verilog cũng cung cấp những công cụ cho việc định nghĩa những phần tử cơ bản với những chức năng đặc biệt. Một hệ thống số logic 4 giá trị đơn giản (0,1,x,z) được sử dụng trong Verilog để thể hiện giá trị cho tín hiệu. Tuy nhiên, để mô hình mức logic chính xác hơn, những tín hiệu Verilog gồm 16 mức giá trị về độ mạnh được thêm vào 4 giá trị đơn giản ở trên.

1.2.2.3 Độ trì hoãn giữa pin đến pin

Một tiện ích trong việc mô tả định thời cho các linh kiện tại ngõ vào và ngõ ra cũng được cung cấp trong Verilog. Tiện ích này có thể được dùng

để truy vấn lại thông tin về định thời trong mô tả tiền thiết kế ban đầu. Hơn nữa, tiện ích này cũng cho phép người viết mô hình hóa tinh chỉnh hành vi định thời của mô hình dựa trên hiện thực phần cứng.

1.2.2.4 Mô tả Bus

Những tiện ích về mô hình bus và thanh ghi cũng được cung cấp bởi Verilog. Đối với nhiều cấu trúc bus khác nhau, Verilog hỗ trợ chức năng phân giải bus và wire với hệ thống logic 4 giá trị (0,1,x,z). Với sự kết hợp giữa chức năng bus logic và chức năng phân giải, nó cho phép mô hình hóa được hầu hết các loại bus. Đối với việc mô hình hóa thanh ghi, việc mô tả xung clock mức cao và những cấu trúc điều khiển định thời có thể được sử dụng để mô tả thanh ghi với những tín hiệu xung clock và tín hiệu reset khác nhau.

1.2.2.5 Mức độ hành vi

Những khối qui trình (procedural blocks) của Verilog cho phép mô tả thuật toán của những cấu trúc phần cứng. Những cấu trúc này tương tự với ngôn ngữ lập trình phần mềm nhưng có khả năng mô tả phần cứng.

1.2.2.6 Những tiện ích hệ thống

Những tác vụ hệ thống trong Verilog cung cấp cho người thiết kế những công cụ trong việc tạo ra dữ liệu kiểm tra testbench, tập tin truy xuất đọc, ghi, xử lí dữ liệu, tạo dữ liệu, và mô hình hóa những phần cứng chuyên dụng. Những tiện ích hệ thống dùng cho bộ nhớ đọc và thiết bị logic lập trình được (PLA) cung cấp những phương pháp thuận tiện cho việc mô hình hóa những thiết bị này. Những tác vụ hiện thị và I/O có thể được sử dụng để kiểm soát tất cả những ngõ vào và ngõ ra dữ liệu của ứng

dụng và mô phỏng. Verilog cho phép việc truy xuất đọc và ghi ngẫu nhiên đến các tập tin.

1.2.2.7 PLI

Công cụ tương tác ngôn ngữ lập trình (PLI) của Verilog cung cấp một môi trường cho việc truy xuất cấu trúc dữ liệu Verilog sử dụng một thư viện chứa các hàm của ngôn ngữ C.

1.2.3 Ngôn ngữ Verilog

Ngôn ngữ Verilog HDL đáp ứng tất cả những yêu cầu cho việc thiết kế và tổng hợp những hệ thống số. Ngôn ngữ này hỗ trợ việc mô tả cấu trúc phân cấp của phần cứng từ mức độ hệ thống đến mức cổng hoặc đến cả mức công tắc chuyển mạch. Verilog cũng hỗ trợ mạnh tất cả các mức độ mô tả việc định thời và phát hiện lỗi. Việc định thời và đồng bộ mà được đòi hỏi bởi phần cứng sẽ được chú trọng một cách đặc biệt.

Trong Verilog, một linh kiện phần cứng được mô tả bởi một cấu trúc ngôn ngữ “khai báo module”. Sự mô tả một module sẽ mô tả danh sách những ngõ vào và ngõ ra của linh kiện cũng như những thanh ghi và hệ thống bus bên trong linh kiện. Bên trong một module, những phép gán đồng thời, gọi sử dụng linh kiện và những khối qui trình có thể được dùng để mô tả một linh kiện phần cứng.

Nhiều module có thể được gọi một cách phân cấp để hình thành những cấu trúc phần cứng khác nhau. Những phần tử con của việc mô tả thiết kế phân cấp có thể là những module, những linh kiện cơ bản hoặc những linh kiện do người dùng tự định nghĩa. Để mô phỏng cho thiết kế, những phần tử con trong cấu trúc phân cấp này nên được tổng hợp một cách riêng lẻ.

Hiện nay có rất nhiều công cụ và môi trường dựa trên Verilog cung cấp khả năng chạy mô phỏng, kiểm tra thiết kế và tổng hợp thiết kế. Môi trường mô phỏng cung cấp những chương trình giao diện đồ họa cho bước thiết kế trước layout (front-end) và những công cụ tạo dạng sóng và công cụ hiển thị. Những công cụ tổng hợp dựa trên nền tảng của Verilog và khi tổng hợp một thiết kế thì thiết bị phần cứng đích như FPGA hoặc ASIC cần phải được xác định trước.

1.3 Tổng kết

Phần này đã cung cấp một cái nhìn tổng quan về những cơ chế, những công cụ và những qui trình dùng trong việc mô tả một thiết kế từ bước thiết kế đến quá trình hiện thực phần cứng. Phần này cũng nói sơ lược về thông tin kiến thức mà ta sẽ đi sâu trong các phần sau. Bên cạnh đó, nó cũng cung cấp đến người đọc lịch sử phát triển của Verilog. Cùng với việc phát triển chuẩn Verilog HDL này là sự phát triển không ngừng của các công ty nghiên cứu, xây dựng và hoàn thiện các công cụ hỗ trợ đi kèm, kết quả là tạo ra những công cụ tốt hơn và những môi trường thiết kế đồng bộ hơn.

1.4 Bài tập

1. Verilog là gì ? Tại sao ta phải sử dụng ngôn ngữ mô tả phần cứng Verilog trong thiết kế Chip?
2. Tìm hiểu môi trường thiết kế trên FPGA là QuartusII của Altera và tìm hiểu môi trường mô phỏng và môi trường tổng hợp của nó. Hãy liên tưởng so sánh môi trường thiết kế này với môi trường mô phỏng và tổng hợp mà đã được trình bày trong phần này.

3. Nêu sự khác biệt giữa ngôn ngữ mô tả phần cứng nói chung (ngôn ngữ Verilog HDL nói riêng) và ngôn ngữ lập trình nói chung (ngôn ngữ C nói riêng).
4. Tìm hiểu sự khác biệt giữa hai loại ngôn ngữ mô tả phần cứng Verilog HDL và VHDL.
5. Quá trình tổng hợp (synthesis) là gì?
6. Verilog HDL có thể được sử dụng để mô tả mạch tương tự (analog) trong phần cứng không ?
7. Tìm kiếm 3 công cụ mô phỏng Verilog HDL hỗ trợ miễn phí.
8. Tìm kiếm 3 tài liệu hỗ trợ việc học và nghiên cứu Verilog HDL.
9. Tìm kiếm 3 website hỗ trợ việc học và nghiên cứu Verilog HDL.
10. Tìm kiếm các công ty thiết kế chip ở Việt Nam đang sử dụng Verilog HDL trong việc thiết kế.
11. Tìm hiểu và sử dụng thành thạo hai công cụ mô phỏng QuartusII và ModelSim.

Chương 2. Qui ước về từ khóa

2.1 Khoảng trắng

Khoảng trắng có thể chứa những kí tự đặc biệt như kí tự space, kí tự tab, kí tự xuống dòng. Những kí tự này có thể được bỏ qua trừ khi nó được sử dụng để tách biệt với những kí tự đặc biệt mang ý nghĩa khác. Tuy nhiên, những kí tự khoảng trắng và kí tự tab có thể được xem như những kí tự đặc biệt trong chuỗi (xem trong mục 2.5).

2.2 Chú thích

Ngôn ngữ Verilog HDL có hai cách để tạo chú thích. “Chú thích một dòng” có thể bắt đầu bằng hai kí tự // và kết thúc với một dòng mới. “Chú thích một khối” sẽ bắt đầu với một /* và kết thúc với */. Chú thích khối không nên quá rối rắm. Trong chú thích khối thì hai kí tự // không mang ý nghĩa gì đặc biệt cả.

2.3 Toán tử

Những toán tử như chuỗi kí tự đơn, kép hay gồm ba kí tự được dùng trong những biểu thức. Trong phần thảo luận về biểu thức ta sẽ trình bày về cách sử dụng các toán tử trong biểu thức như thế nào.

Những toán tử đơn thường xuất hiện bên trái của toán hạng của chúng. Những toán tử kép thường xuất hiện ở giữa những toán hạng của chúng. Toán tử có điều kiện thường có hai toán tử kí tự được phân biệt bởi ba toán hạng

2.4 Số học

Hằng số được mô tả như là hằng số nguyên hoặc hằng số thực.

Ví dụ 2.1

```
243      // số thập phân
1.4E9    // 1.4x10^9
-5'd18   // số thập phân -18 lưu trong 5 bit
4'b1011  // số nhị phân 1011 lưu trong 4 bit
8'hEF  // số thập lục phân EF lưu trong 8 bit
16'o56  // số bát phân 56 lưu trong 16 bit
4'bxxxx // số nhị phân tùy định
4'bzzzz   // số nhị phân có giá trị tổng trở cao
```

2.4.1 Hằng số nguyên

Hằng số nguyên có thể được mô tả theo định dạng số thập phân, thập lục phân, bát phân và nhị phân.

Có hai dạng để biểu diễn hằng số nguyên. Dạng thứ nhất là một số thập phân đơn giản, nó có thể là một chuỗi kí tự từ 0 đến 9 và có thể bắt đầu với toán tử đơn cộng hoặc trừ. Dạng thứ hai được mô tả dưới dạng hằng cơ số, nó gồm ba thành phần – một là thành phần mô tả độ rộng hằng số, một thành phần là kí tự móc đơn (') được theo sau bởi một kí tự của cơ số tương ứng ('D), và thành phần cuối cùng mô tả giá trị của số đó.

Ví dụ 2.2 Hằng số không dấu

```
374          // số thập phân
'h 473FF     // số thập lục phân
'o7439       // số bát phân
```

```
5be                    // không hợp lệ ( số thập lục phân đòi hỏi ‘h)
```

Thành phần đầu tiên, độ rộng hằng số, mô tả độ rộng số bit để chứa hằng số. Nó được mô tả như là một số thập phân không dấu khác không. Ví dụ, độ rộng của hai số hexadecimal là 8 bit bởi vì mỗi một số hexadecimal cần 4 bit để chứa.

Thành phần thứ hai, định dạng cơ số, bao gồm một kí tự có thể kí tự thường hoặc kí tự hoa để mô tả cơ số của số đó, ta có thể thêm vào hoặc không thêm vào phía trước nó kí tự s (hoặc S) để chỉ rằng nó là một số có dấu, tiếp tục phía trước nó là một kí tự móc đơn. Những cơ số được dùng có thể là d, D, h, H, o, O, b, B để mô tả cho cơ số thập phân, cơ số thập lục phân, cơ số bát phân và cơ số nhị phân một cách tương ứng.

Kí tự móc đơn và kí tự định dạng cơ số không được cách nhau bởi bất kì khoảng trắng nào.

Thành phần thứ ba, một số không dấu, bao gồm những kí tự phù hợp với cơ số đã được mô tả trong thành phần thứ hai. Thành phần số không dấu này có thể theo sau ngay thành phần cơ số hoặc có thể theo sau thành phần cơ số bởi một khoảng trắng. Những kí tự từ a đến f của số thập lục phân có thể là kí tự thường hoặc kí tự hoa.

Ví dụ 2.3 Hằng số có độ rộng bit

```
4’b1011      // số nhị phân 4 bit
5 ‘D 5       // số thập phân 5 bit
3’b10x       // số nhị phân 3 bit với bit có trọng số thấp nhất có giá trị không xác định
12’hx        // số thập lục phân 12 bit có giá trị không xác định
16’hz        // số thập lục phân 16 bit có giá trị tổng trở cao.
```

Những số thập phân đơn giản không kèm theo độ rộng bit và định dạng cơ số có thể được xem như là những số nguyên có dấu, trong khi đó những số được mô tả bởi định dạng cơ số có thể được xem như những số nguyên có dấu khi thành phần chỉ định s được kèm thêm vào hoặc nó sẽ được xem như những số nguyên không dấu khi chỉ có thành phần định dạng cơ số được sử dụng. Thành phần chỉ định số có dấu s không ảnh hưởng đến mẫu bit được mô tả mà nó chỉ ảnh hưởng trong quá trình biên dịch.

Toán tử cộng hay trừ đứng trước hằng số độ rộng là một toán tử đơn cộng hay trừ. Hai toán tử này nếu được đặt nằm giữa thành phần định dạng cơ số và số là không đúng cú pháp.

Những số âm được biểu diễn dưới dạng bù hai.

Ví dụ 2.4 Sử dụng dấu với hằng số

```
6 ‘d -7          // cú pháp không đúng
-6 ‘d 7          // số bù 2 của 7, tương đương với –(6’d 7)
4 ‘shf           // số có dấu (số bù 2) 4 bit ‘1111’, bằng -1, tương
    đương với –4’h1
-4 ‘sd15         // tương đương với –(4’sd15), hay –(-1) = 1 =
    0001
8’sd?        //tương đương 8’sbz
```

Các giá trị số đặc biệt x và z:

-Một số x dùng để biểu diễn một giá trị không xác định trong những hằng số thập lục phân, hằng số bát phân và hằng số nhị phân.

-Một số z dùng để biểu diễn một số có giá trị tổng trở cao.

Một số x có thể được thiết lập trên 4 bit để biểu diễn một số thập lục phân, trên 3 bit để biểu diễn một số bát phân, trên 1 bit để biểu diễn một số nhị phân có giá trị không xác định. Tương tự, một số z có thể được thiết lập

trên 4 bit để biểu diễn một số thập lục phân, trên 3 bit để biểu diễn một số bát phân, trên 1 bit để biểu diễn một số nhị phân có giá trị tổng trở cao.

Ví dụ 2.5 Tự động thêm vào bên trái

```
reg [11:0] m, n, p, q;
initial begin
        m = 'h x;           // tạo ra xxx
        n = 'h 4x;          // tạo ra 04x
        p = 'h z5;          // tạo ra zz5
        q = 'h 0z8;         // tạo ra 0z8
end
reg [15:0]          e, f, g;
        e = 'h4;            // tạo ra {13{1'b0}, 3'b100}
        f = 'hx             // tạo ra {16{1'hx}}
        g = 'hz;            // tạo ra {16{1'hz}}
```

Nếu độ rộng bit của số không dấu nhỏ hơn độ rộng được mô tả trong phần mô tả hằng số thì số không dấu sẽ được thêm vào bên trái nó là các số 0. Nếu bít ngoài cùng bên trái trong số không dấu là x hoặc z thì một x hoặc một z sẽ được dùng để thêm vào bên trái một cách tương ứng. Nếu độ rộng của số không dấu lớn hơn độ rộng được mô tả trong phần mô tả hằng số thì số không dấu sẽ bị cắt xén đi từ bên trái.

Số bit dùng để tạo nên một số không có độ rộng (có thể là một số thập phân đơn giản hoặc một số không mô tả độ rộng bit) nên ít nhất là 32 bit. Những hằng số không dấu, không độ rộng mà bit có trọng số cao là không xác định (x) hoặc tổng trở cao (z) thì nó sẽ được mở rộng ra đến độ rộng của biểu thức chứa hằng số.

Giá trị x và z để mô tả giá trị của một số có thể là chữ hoa hoặc chữ thường.

Khi được sử dụng để mô tả một số trong Verilog, thì kí tự dấu chấm hỏi (?) có ý nghĩa thay thế cho kí tự z. Nó cũng thiết lập 4 bit lên giá trị tổng trở cao cho số thập lục phân, 3 bit cho số bát phân và 1 bit cho số nhị phân. Dấu chấm hỏi có thể được dùng để giúp việc đọc code dễ hiểu hơn trong trường hợp giá trị tổng trở cao là một điều kiện không quan tâm (don't care). Ta sẽ thảo luận rõ hơn về vấn đề này khi trình bày về casez và casex. Kí tự dấu chấm hỏi cũng được dùng trong những bảng trạng thái do người dùng tự định nghĩa.

Trong một hằng số thập phân, số không dấu không bao gồm những kí tự x, z hoặc ? trừ trường hợp ở đó chỉ có đúng một kí tự để chỉ ra rằng mọi bit trong hằng số thập phân là x hoăc z.

Kí tự gạch dưới (_) có thể dùng ở bất kì nơi đâu trong một số, ngoại trừ kí tự đầu tiên. Kí tự gạch dưới sẽ được bỏ qua. Đặc tính này có thể được dùng để tách một số quá dài để giúp việc đọc code dễ dàng hơn.

Ví dụ 2.6 Sử dụng dấu gạch dưới trong mô tả số

```
27_195_000
16'b0011_0101_0001_1111
32 'h 12ab_f001
```

Những hằng số âm có độ rộng bit và những hằng số có dấu có độ rộng bit là những số có dấu mở rộng khi nó được gán đến một loại dữ liệu là reg bất chấp bản thân reg này có dấu hay không.

Độ dài mặc định của x và z giống như độ dài mặc định của một số nguyên.

2.4.2 Hằng số thực

Những số hằng số thực có thể được biểu diễn như được mô tả bởi chuẩn IEEE 754-1985, một chuẩn IEEE cho những số dấu chấm động có độ chính xác kép.

Những số thực có thể được mô tả bằng một trong hai cách, một là theo dạng thập phân (ví dụ, 25.13), hai là theo cách viết hàn lâm (ví dụ, 45e6, có nghĩa là 45 nhân với 10^6. Những số thực được biểu diễn với dấu chấm thập phân sẽ có ít nhất một kí số ở mỗi bên của dấu chấm thập phân.

Ví dụ 2.7

2.5

0.9

1543.34592

3.2E23 or 3.2e23

5.6e-3

0.9e-0

45E13

43E-6

354.156_972_e-19 (dấu gạch dưới được bỏ qua)

Những dạng số sau không đúng là số thực vì chúng không có ít nhất một kí số ở mỗi bên của dấu chấm thập phân.

.43

8.

7.E4

.6e-9

2.4.3 Số đảo

Số thực có thể biến đổi sang số nguyên bằng cách làm tròn số thực đến số nguyên gần nhất thay vì cắt xén số bit của nó. Biến đổi không tường minh có thể thực hiện khi một số thực được gán đến một số nguyên. Những cái đuôi nên được làm tròn khác 0. Ví dụ:

Hai số thực 48.8 và 48.5 đều trở thành 49 khi được biến đổi sang số nguyên, và số 48.3 sẽ trở thành 48.

Biến đổi số thực -5.5 sang số nguyên sẽ được -6, biến đổi số 5.5 sang số nguyên sẽ được 6.

2.5 Chuỗi

Một chuỗi là một dãy các kí tự được nằm trong hai dấu nháy kép(“”) và được ghi trên một dòng đơn. Những chuỗi được dùng như là những toán hạng trong biểu thức và trong những phép gán được xem như là những hằng số nguyên không dấu và được biểu diễn bởi một dãy kí tự 8 bit ASCII. Một kí tự ASCII biểu diễn bằng 8 bit.

2.5.1 Khai báo biến chuỗi

Biến chuỗi là biến có loại dữ liệu là reg với độ rộng bằng với số kí tự trong chuỗi nhân với 8.

Ví dụ 2.8

*Để lưu trữ một chuỗi 12 kí tự “Verilog HDL!” đòi hỏi một reg có độ rộng 8*12, hoặc 96 bit*

```
reg [8*12:1] stringvar;
initial begin
        stringvar = “Verilog HDL!”;
end
```

2.5.2 Xử lí chuỗi

Chuỗi có thể được xử lí bằng việc sử dụng các toán tử Verilog HDL. Giá trị mà được xử lí bởi toán tử là một dãy giá trị 8 bit ASCII. Các toán tử xử lý chuỗi được thể hiện chi tiết hơn trong phần 4.3.3.

2.5.3 Những kí tự đặc biệt trong chuỗi

Một số kí tự chỉ được sử dụng trong chuỗi khi đứng trước nó là một kí tự mở đầu, gọi là kí tự escape "\". Bảng bên dưới liệt kê những kí tự này và ý nghĩa của nó.

Bảng 2.1 Kí tự đặc biệt trong chuỗi

Chuỗi escape	Kí tự tạo bởi chuỗi escape
\n	Kí tự xuống dòng
\t	Kí tự tab
\\	Kí tự \
\"	Kí tự "
\ddd	Một kí tự được mô tả trong 1-3 kí số bát phân (0≤d≤7) Nếu ít hơn ba kí tự được sử dụng, kí tự theo sau không thể là một kí số bát phân. Việc thực thi có thể dẫn đến lỗi nếu kí tự được biểu diễn lớn hơn 377

2.6 Định danh, từ khóa và tên hệ thống

Định danh (indentifier) được dùng để gán cho một đối tượng (object) một tên duy nhất để nó có thể được gọi tới khi cần. Định danh có thể là một định danh đơn giản hoặc một định danh escaped. Một định danh đơn giản có thể là một dãy bất kì gồm các kí tự, kí số, dấu dollar ($), và kí tự gạch dưới (_).

Kí tự đầu tiên của một định danh không thể là một kí số hay $; nó có thể là một kí tự hoặc một dấu gạch dưới. Định danh sẽ là khác nhau giữa chữ thường và chữ hoa như trong ngôn ngữ lập trình C.

Ví dụ 2.9

kiemtra_e

net_m

fault_result

string_ab

_wire1

n$983

Ở đây có sự giới hạn về độ dài của định danh, nhưng giới hạn này ít nhất là 1024 kí tự. Nếu một định danh vượt ra khỏi giới hạn về chiều dài đã được xác định thì lỗi có thể được thông báo ra.

2.6.1 Định danh với kí tự "\"

Tên định danh escaped được bắt đầu với kí tự gạch chéo (\) và kết thúc bởi khoảng trắng (kí tự khoảng trắng, kí tự tab, kí tự xuống dòng). Chúng cung cấp cách thức để chèn thêm những kí tự ASCII có thể in được vào trong các kí tự có code từ 33 đến 126, hoặc giá trị thập lục phân từ 21 đến 7E).

Cả hai kí tự gạch chéo (\) và kí tự khoảng trắng kết thúc đều không được xem như là thành phần của tên nhận dạng. Do đó, một định danh "\abc" sẽ được xử lí giống như định danh "abc".

Ví dụ 2.10

\netc+num

\-signal

****fault-result****

\wirea/\wireb

\{m,n}

\i(k+l)*

Một từ khóa trong Verilog HDL mà đứng trước nó là một kí tự escape sẽ không được biên dịch như là một từ khóa.

2.6.2 Tác vụ hệ thống và hàm hệ thống

Dấu dollar ($) mở đầu một cấu trúc ngôn ngữ sẽ cho phép phát triển những tác vụ hệ thống và hàm hệ thống do người dùng định nghĩa. Những cấu trúc hệ thống không phải là ngôn ngữ thiết kế, mà nó muốn nói đến chức năng mô phỏng. Một tên theo sau dấu $ được biên dịch như là một tác vụ hệ thống hoặc hàm hệ thống.

Tác vụ hệ thống/hàm hệ thống có thể được định nghĩa trong ba vi trí:

- Một tập hợp chuẩn những tác vụ hệ thống và hàm hệ thống.
- Những tác vụ hệ thống và hàm hệ thống thêm vào được định nghĩa dùng cho PLI (Programming Language Interface).
- Những tác vụ hệ thống và hàm hệ thống thêm vào được định nghĩa bởi thực thi phần mềm.

Ví dụ 2.11

$time – trả về thời gian chạy mô phỏng hiện tại

$display – tương tự như hàm printf trong C

$stop – ngừng chạy mô phỏng

$finish – hoàn thành chạy mô phỏng

$monitor – giám sát chạy mô phỏng

2.7 Bài tập

1. Nêu tác dụng và sự khác biệt giữa hai hàm hệ thống $monitor và $display khi sử dụng hai hàm hệ thống này trong quá trình mô phỏng.
2. Làm sao có thể đọc và ghi một file dữ liệu trong mô tả phần cứng Verilog HDL (giả sử file chứa nội dung bộ nhớ khởi tạo).
3. Hãy tìm thêm 10 tác vụ hệ thống và nêu ý nghĩa của chúng

Chương 3. Loại dữ liệu trong Verilog

3.1 Khái quát

Verilog chỉ hỗ trợ những loại dữ liệu đã được định nghĩa trước. Những loại dữ liệu này bao gồm dữ liệu bit, mảng bit, vùng nhớ, số nguyên, số thực, sự kiện, và độ mạnh của dữ liệu. Những loại này định nghĩa trong phần lớn mô tả của Verilog. Verilog chủ yếu xử lí trên bit và byte khi mô tả mạch điện tử. Loại số thực thì hữu dụng trong việc mô tả độ trì hoãn và định thời và nó cũng rất hữu dụng trong việc mô hình hóa ở mức cao như là phân tích xác suất kết nối mạch trong hệ thống và những giải thuật xử lí tín hiệu số. Loại dữ liệu phần cứng bao gồm net và reg. Thông thường những loại này có thể được xem như là dây kết nối và thanh ghi. Dữ liệu net có thể được khai báo chi tiết hơn để tạo ra những loại dữ liệu khác như tri-stated hay non-tri-stated và phụ thuộc vào các xử lí nhiều kết nối sẽ tạo ra những phép and, or hoặc dùng giá trị trước đó. Phần tiếp theo sẽ trình bày chi tiết về những vấn đề này.

3.2 Những hệ thống giá trị

Mỗi loại dữ liệu có những mục đích cụ thể của nó trong việc mô tả. Những hệ thống giá trị định nghĩa những loại giá trị khác nhau đã được định nghĩa trong ngôn ngữ và bao gồm cả những thao tác giúp hỗ trợ những hệ thống giá trị này. Chúng cũng có những định nghĩa hằng số tương ứng. Trong Verilog có nhiều giá trị khác nhau như:

- bits and integers(32 bits), time (64 bits) – bit-vectors và integers có thể phối hợp một cách tự do. Integers được định

nghĩa có 32 bit. Giá trị time có 64 bit. Thực sự bit có hai loại sau:

- ❖ 4 giá trị trạng thái (0,1,x,z); được biết như là giá trị logic.
- ❖ 128 loại trạng thái (4 trạng thái và 64 độ mạnh (8 cho độ mạnh '0' và 8 cho độ mạnh '1')

- Loại floating point (số thực)
- Chuỗi kí tự
- Giá trị độ trì hoãn – Những giá trị này có thể là single, double, triplet hay n-tuple để chỉ độ trì hoãn cạnh lên, cạnh xuống hoặc sự chuyển đổi khác của tín hiệu.
- Giá trị chuyển trạng thái – (01) – chuyển trạng thái từ 0 sang 1. Giá trị này có thể có trong những linh kiện cơ bản do người dùng định nghĩa hoặc trong những khối mô tả (specify blocks)
- Những giá trị có điều kiện/Boole – true/false hoặc 0/1
- units (chỉ dùng cho timescale) – femtoseconds (Fs) đến seconds (s)

3.3 Khai báo loại dữ liệu

3.3.1 Giới thiệu

Những loại dữ liệu khác nhau trong Verilog được khai báo bằng phát biểu khai báo dữ liệu. Những phát biểu này xuất hiện trong những định nghĩa module trước khi sử dụng và một số trong chúng có thể được khai báo bên trong những khối tuần tự được đặt tên. Thêm vào đó, những loại giá trị có thể phân biệt với những loại của dữ liệu khác, những đặc tính phần cứng của wires so với registers cũng được phân biệt như là những khai báo net so với khai báo reg trong Verilog. Từ "driving" nghĩa là điều

khiển được dùng trong những mô tả phần cứng để mô tả cách thức một giá trị được gán đến một phần tử. Nets và regs là hai phần tử dữ liệu chính trong Verilog. Nets được điều khiển một cách nối tiếp từ những phép gán nối tiếp (continuous assignments) hoặc từ những phần tử cấu trúc như module ports, gates, transistors hoặc những phần tử cơ bản do người dùng tự định nghĩa. Regs được điều khiển một cách chặt chẽ từ những khối hành vi (behavioural blocks). Nets thông thường được thực thi như là wires trong phần cứng và regs thì có thể là wires hoặc phần tử tạm hoặc flip-flops (registers).

Những loại dữ liệu khác nhau trong Verilog được khai báo như là một trong những loại sau:

- parameter: Loại này là những biểu thức giá trị hằng số được phân tích sau quá trình biên dịch và cho phép modules được gán tham số.
- input, output, inout: Những loại dữ liệu này định nghĩa chiều và độ rộng của một port.
- net: Đây là loại dữ liệu dùng cho việc kết nối hoặc wire trong phần cứng với sự phân tích khác nhau.
- reg: Đây là loại dự liệu trừu tượng giống như là một thanh ghi (register) và được điều khiển theo hành vi.
- time: Đây là loại dữ liệu lưu trữ khoảng thời gian như độ trì hoãn và thời gian mô phỏng.
- integer: Đây là loại dữ liệu số nguyên.
- real: Đây là loại dữ liệu floating point hay số thực
- event: Đây là dữ liệu để chỉ ra rằng một cờ hiệu được bật tích cực.

Những loại dữ liệu này tất cả có thể được khai báo ở mức độ module. Những mô tả khác trong Verilog với những khả năng tạo lập mục đích bao

gồm những tác vụ, những hàm và những khối begin-end được đặt tên. Nets được điều khiển không theo hành vi (non-behaviorally) nên do đó nó không thể được khai báo cho những mục đích khác. Tất cả những loại dữ liệu khác có thể được thể hiện trong những tác vụ và trong những khối begin-end.

Ví dụ 3.1

```
input a, b;
reg [15:0] c;
time tg;
```

Dòng đầu tiên trong ví dụ trên là một dòng khai báo input, dòng thứ hai là một khai báo dữ liệu reg 16 bit. Dòng cuối cùng là khai báo cho một biến được đặt tên là tg.

3.4 Khai báo loại dữ liệu net

3.4.1 Giới thiệu

Net là một trong nhiều loại dữ liệu trong ngôn ngữ mô tả Verilog dùng để mô tả dây kết nối vật lí trong mạch điện. Net sẽ kết nối những linh kiện ở mức cổng được gọi ra, những module được gọi ra và những phép gán nối tiếp. Ngôn ngữ Verilog cho phép đọc giá trị từ một net từ bên trong những mô tả hành vi, nhưng ta không thể gán một giá trị từ một net bên trong những mô tả hành vi. Một net sẽ không lưu giữ giá trị của nó. Nó phải được điều khiển bởi một trong hai cách sau.

- Bằng việc kết nối net đến ngõ ra của một cổng hay một module.
- Bằng việc gán một giá trị đến net trong một phép gán nối tiếp.

Những loại net khác nhau được định nghĩa trong Verilog được mô tả bên dưới và trong Bảng 3.1 sẽ tóm tắt sự phân giải logic của chúng. Sự phân giải logic là một qui định để giải quyết xung đột xảy ra khi có nhiều mức logic điều khiển một net.

- Wire: một net với giá trị 0,1,x và sự phân giải logic được dựa trên sự tương đương.
- Wand: một net với giá trị 0,1,x và sự phân giải logic được dựa trên nguyên tắc của phép wired and
- Wor: một net với giá trị 0,1,x và sự phân giải logic được dựa trên wired or
- Tri: một net với giá trị 0,1,x,z và sự phân giải logic được dựa trên nguyên tắc của bus tri-state
- Tri0: một net với giá trị 0,1,x,z và sự phân giải logic được dựa trên nguyên tắc của bus tri-state và một giá trị mặc định là 0 khi không được điều khiển
- Tri1: một net với giá trị 0,1,x,z và sự phân giải logic được dựa trên nguyên tắc của bus tri-state và một giá trị mặc định là 1 khi không được điều khiển
- Trior: một net với giá trị 0,1,x,z và sự phân giải logic được dựa trên nguyên tắc của tri-state cho giá trị z-non-z sử dụng hàm ‘or’ của giá trị non-z
- Triand: một net với giá trị 0,1,x,z và sự phân giải logic được dựa trên nguyên tắc của tri-state cho giá trị z-non-z sử dụng hàm ‘and’ của giá trị non-z
- Trireg: một net với giá trị 0,1,x,z và sự phân giải logic được dựa trên nguyên tắc của tri-state cùng với giá trị lưu trữ điện tích (giá trị trước được dùng để phân giải giá trị mới)
- Supply0, supply1 (gnd và vdd)

Bảng 3.1 Sự phân giải của các loại net

tri/wire	0	1	X	Z		triand/wand	0	1	X	Z
0	0	X	X	0		0	0	0	0	0
1	X	1	X	1		1	0	1	X	1
X	X	X	X	X		X	0	X	X	X
Z	0	1	X	X		Z	0	1	X	Z
trior/wor	0	1	X	Z		tri0	0	1	X	Z
0	0	1	X	0		0	0	0	0	0
1	1	1	1	1		1	X	1	X	1
X	X	1	X	X		X	X	X	X	X
Z	0	1	X	Z		Z	0	1	X	0
trireg	0	1	X	Z		tri1	0	1	X	Z
0	0	1	X	0		0	0	X	X	0
1	1	1	1	1		1	X	1	X	1
X	X	1	X	X		X	X	X	X	X
Z	0	1	X	P		Z	0	1	X	1

3.4.2 Wire và Tri

Loại dữ liệu wire là một loại đơn giản để kết nối giữa hai linh kiện. Dữ liệu wire dùng cho những net được điều khiển bởi một cổng linh kiện đơn hay trong phép gán nối tiếp (continuous assignments). Trong Ví dụ 3.2 những khai báo 2-wire được tạo ra. Khai báo đầu tiên mô tả wire đơn (scalar wire) a1. Khai báo thứ hai mô tả một mảng (vector) b2 với 3 bits. Bit trọng số cao nhất (MSB) của nó có trọng số là 2 và bit trọng số thấp nhất (ISB) có trọng số là 0.

Ví dụ 3.2

```
wire a1;
wire   [2:0] b2;
tri abc
```

Dữ liệu tri thì hoàn toàn giống với dữ liệu wire về cú pháp sử dụng và chức năng tuy nhiên nó khác với dữ liệu wire ở chỗ, dữ liệu tri được dùng cho những net được điều khiển bởi nhiều cổng linh kiện ngõ ra. Loại dữ liệu tri (tri-state) là loại dữ liệu đặc biệt của wire trong đó sự phân giải giá trị của net được điều khiển bởi nhiều linh kiện điều khiển được thực hiện bằng việc sử dụng những qui luật của bus tri-state. Tất cả các biến mà điều khiển net tri phải có giá trị Z (tổng trở cao), ngoại trừ một biến. Biến đơn này xác định giá trị của net tri.

Trong Ví dụ 3.3, ba biến điều khiển biến out. Chúng được thiết lập trong module khác để chỉ một linh kiện điều khiển tích cực trong một thời điểm.

Ví dụ 3.3

```
module tri_kiemtra (out, m, n,p);
        input [1:0] select ,m ,n, p;
        output out;
        tri out;
        assign out = m      ; tạo kết nối cho net tri
        assign out = n;
        assign  out = p;
endmodule
module mnp (m, n, p, select)
        output m, n, p;
        input [1:0] select;
```

```
        always @(select) begin
         m = 1'bz;    // thiết lập tất cả các biến có giá trị Z
        n = 1'bz;
        p = 1'bz;
        case (select)          // chỉ thiết lập một biến non-Z
        2'b00: m = 1'b1;
        2'b01: n = 1'b0;
        2'b10: p = 1'b1;
        endcase
        end
    endmodule
    module top_tri_test ( out, m, n, p, select);
        input [1:0] select;
        input m, n, p;
        output out;
        tri out;
        mnp (m, n, p, select);
        tri_test (out, m, n, p);
    endmodule
```

3.4.3 Wired net

Wired nets bao gồm những loại dữ liệu wor, wand, trior và triand. Chúng được dủng để mô hình giá trị logic của net. Những wired net trên có bảng sự thật khác nhau để phân giải những xung đột nếu xảy ra khi có nhiều cổng linh kiện cùng điều khiển một net.

3.4.3.1 Wand/triand nets

Wand/triand là loại dữ liệu đặc biệt của wire dùng hàm *and* để tìm giá trị kết quả khi nhiều linh kiện điều khiển một net, hay nói cách khác nếu có bất kì ngõ ra linh kiện điều khiển nào có giá trị 0 thì giá trị của net được điều khiển sẽ là 0. Trong Ví dụ 3.4, hai biến điều khiển biến out. Giá trị của out được xác định bằng hàm logic and giữa b1 và b2.

Ví dụ 3.4

```
module wand_test (out, b1,b2);
        input b1, b2;
        output out;
        wand out;
        assign out = b1;
        assign out = b2;
endmodule
```

Ta có thể gán một giá trị trì hoãn trong khai báo wand, và ta có thể sử dụng những từ khóa đơn (scalar) và mảng (vector) cho việc mô phỏng.

3.4.3.2 Wor/Trior

Loại dữ liệu wor /trior là loại dữ liệu đặc biệt của wire dùng hàm *or* để tìm giá trị kết quả khi nhiều linh kiện điều khiển một net, hay nói cách khác nếu có bất kì ngõ ra linh kiện điều khiển nào có giá trị 1 thì giá trị của net được điều khiển sẽ là 1. Trong Ví dụ 3.5, hai biến điều khiển biến out. Giá trị của out được xác định bằng hàm logic OR giữa a1 và a2.

Ví dụ 3.5

```
module wor_test(a1,a2);
        input a1, a2;
```

```
        ouput out;
        wor out;
        assign out = a1;
        assign out = a2;
endmodule
```

3.4.4 Trireg net

Net trireg được dùng để mô hình giá trị điện dung lưu giữ trên net của mạch điện, nó có khả năng lưu giữ giá trị điện tích. Một trireg có thể là một trong hai trạng thái sau:

Trạng thái được điều khiển (driven state): Khi có ít nhất một ngõ ra của linh kiện điều khiển net trireg có giá trị 1, 0 hoặc x thì giá trị này sẽ được truyền đến net trireg và giá trị này điều khiển giá trị của net trireg.

Trạng thái lưu giữ điện dung: Khi tất cả các ngõ ra của linh kiện điều khiển net trireg đều có giá trị tổng trở cao (z) thì net trireg sẽ lưu giữ giá trị cuối cùng mà nó ở trạng thái được điều khiển. Giá trị tổng trở cao của các ngõ ra linh kiện điều khiển sẽ không được truyền đến net trireg.

Do đó, net trireg sẽ luôn có giá trị 0 hay 1 hoặc x và không có giá trị z. Độ mạnh giá trị trên net trireg trong trạng thái lưu giữ điện dung được mô tả bởi độ rộng, đó có thể là lớn (large), vừa (medium) hay nhỏ (small) với giá trị mặc định là medium nếu nó không được mô tả. Trong trạng thái được điều khiển, độ mạnh của net trireg sẽ phụ thuộc vào độ mạnh của linh kiện điều khiển như supply, strong, pull, weak mà ta sẽ thảo luận sau. Như một mô hình Verilog như phía dưới, ta sẽ lấy được giá trị kết quả của wire trireg khi transistor điều khiển nó bị tắt.

Ví dụ 3.6

```
module kiemtra;
```

```
    reg c0, c1, i1, i2;
    tri d0, d1, d2;
    trireg d;
    and(d0, il, i2);
    nmos nl (d1, d0, c0);
    nmos n2(d, d1, c1);
    initial
    begin
    $monitor("time = %d d = %d c0=%d c1=%d d0=%d d1=%d
    i1=%d i2=%d", $time, d, c0, c1, d0, d1, i1, i2);
    #1
    i1 = 1;
    i2 = l;
    c0 = l;
    c1 = 1;
    #5
    c0 = 0;
    end
endmodule
```

Simulation result:

time = 0 d= x c0=x c1=x d0=x d1=x i1=x i2=x

time = 1 d= 1 c1=1 c1=1 d0=1 d1=1 i1=1 i2=1

time = 6 d= 1 c0=0 c1=1 d0=1 d1=0 i1=1 i2=1

3.4.5 Tri0 và tri1 nets

Net tri0 và tri1 dùng để mô hình những net với linh kiện điện trở kéo lên hoặc kéo xuống. Một net tri0 sẽ tương đương với một net với được điều

khiển liên tục bởi giá trị 0 với độ mạnh **pull**. Một net tri1 sẽ tương đương với một net với được điều khiển liên tục bởi giá trị 1 với độ mạnh **pull**.

Khi không có linh kiện điều khiển net tri0, giá trị của nó vẫn là 0 với độ mạnh **pull.** Khi không có linh kiện điều khiển net tri1, giá trị của nó vẫn là 1 với độ mạnh **pull**. Khi có nhiều linh kiện điều khiển net tri0 hoặc tri1 thì sự phân giải độ mạnh của các linh kiện điều khiển với độ mạnh **pull** của net tri0 hoặc tri1 sẽ xác định giá trị của net.

3.4.6 Supply0/supply1 nets

Loại dữ liệu supply0 và supply1 định nghĩa những net wire được mắc cố định đến mức logic 0 (nối đất, vss) và logic 1 (nguồn cung cấp, vdd). Việc sử dụng supply0 và supply1 tương tự như khai báo một wire và sau đó gán giá trị 0 hoặc 1 lên nó. Trong Ví dụ 3.7, power được nối lên nguồn cung cấp (luôn là logic 1 – có độ mạnh lớn nhất) và gnd được nối đến đất (ground) (luôn là logic 0 – có độ mạnh nhất).

Ví dụ 3.7

```
supply0 gnd;
supply1 power;
```

3.4.7 Thời gian trì hoãn trên net

Trong thực tế bất kì net nào trong mạch điện tử cũng tạo ra độ trì hoãn trên net. Trong Verilog, độ trì hoãn có thể được khai báo kết hợp trong phát biểu khai báo net. Những giá trị độ trì hoãn này là thời gian trì hoãn được tính từ khi tín hiệu tại ngõ ra của linh kiện điều khiển thay đổi cho đến khi tín hiệu trên net thực sự thay đổi. Độ trì hoãn được mô tả bởi số hoặc biểu thức theo sau biểu tượng ‘#’. Những giá trị này là hằng số,

tham số, biểu thức của chúng hay có thể là những biểu thức động dùng những biến số khác. Độ trì hoãn có thể là rise, fall, hay hold (thời gian thay đổi đến z) và mỗi loại trì hoãn này có thể có ba giá trị - minimum, typical và maximum. Sự mô tả độ trì hoãn rise, fall, và hold được phân biệt bởi dấu phẩy (,) và sự mô tả min-typ-max được phân biệt bởi dấu hai chấm (:). Độ trì hoãn rise bao gồm thời gian trì hoãn khi giá trị tín hiệu thay đổi từ 0 lên 1, 0 đến x và từ x đến 1. Độ trì hoãn fall bao gồm thời gian trì hoãn khi giá trị tín hiệu thay đổi từ 1 xuống 0, 1 đến x và từ x đến 0. Độ trì hoãn hold bao gồm thời gian trì hoãn khi giá trị tín hiệu thay đổi từ 0 lên z, 1 đến z và từ x đến z. Khái niệm độ trì hoãn này cũng được dùng cho việc định nghĩa độ trì hoãn của cổng, transistor, linh kiện cơ bản do người dùng tự định nghĩa và những mô tả hành vi.

Ví dụ 3.8

```
tri #9 t1, t2;
wire #(10,9,8) a1, a2;
wand #(10:8:6, 9:8:6) a3;
```

Trong Ví dụ 3.8, dòng đầu tiên mô tả t1, t2 có thời gian trì hoãn rise, fall, hold đều là 9 đơn vị thời gian. Dòng thứ hai, wire a1 và a2 định nghĩa ba giá trị khác nhau cho ba sự thay đổi – 10 cho rise, 9 cho fall và 8 cho hold. Dòng cuối cùng, wand a3 định nghĩa giá trị min, type cho cả ba sự thay đổi rise, fall, hold.

Ví dụ 3.9

```
wire a1, a2;
tri[7:0] t1, t2;
trireg large trg1, trg2;
triand [31:0] #(10:5) gate1;
```

Trong ví dụ trên, dòng đầu tiên với từ khóa ‘wire’ khai báo a1 và a2 là wire đơn (scalar wire hay single bit). Dòng thứ hai khai báo hai vector wire 8 bit t1 và t2 có loại dữ liệu là tri. Dòng kế tiếp khai báo net có khả năng lưu giữ điện dung trg1 và trg2 với độ lớn điện dung là large. Dòng cuối cùng khai báo một net có độ rộng 32 bit có loại dữ liệu là triand với độ trì hoãn là tối thiểu (minimum) và trung bình (typical).

3.5 Khai báo loại dữ liệu biến - reg

Khai báo reg được thực hiện cho tất cả những tín hiệu mà được điều khiển từ những mô tả hành vi. Loại dữ liệu reg lưu giữ một giá trị được cho đến khi nó được gán một giá trị mới trong một mô tả tuần tự (khối intitial hoặc always). Loại dữ liệu reg thì có mức độ trừu tượng hơn so với loại dữ liệu net nhưng nó có quan hệ mật thiết với khái niệm thanh ghi (register) với khả năng lưu giữ giá trị và có thể được xem như là một register trong phần cứng. Tuy nhiên, chúng cũng có thể được xem như là wire hoặc phần tử nhớ tạm thời mà không phải là phần tử thực trong phần cứng, điều này phụ thuộc vào việc sử dụng chúng bên trong khối mô tả hành vi.

Ví dụ 3.10

```
reg reg1, reg2;
reg [63:0] data1, data2, data3;
```

3.6 Khai báo port

3.6.1 Giới thiệu

Ta phải khai báo thật tường minh về chiều (input, output hay bidirectional) của mỗi port xuất hiện trong danh sách khai báo port. Trong

Verilog định nghĩa ba loại port khác nhau, đó là input, output và inout. Loại dữ liệu của port có thể là net hoặc reg. Loai dữ liệu reg chỉ có thể xuất hiện ở port output. Hằng số và biểu thức luôn nằm phía dưới khai báo port.

input: tất cả port input của một module được khai báo với một phát biểu input. Loại dữ liệu mặc định của input port là wire và được điều khiển bởi cú pháp của wire. Ta có thể khai báo độ rộng của một input như một mảng (vector) của những tín hiệu, giống như input b trong ví dụ dưới. Những phát biểu input có thể xuất hiện ở bất cứ vị trí nào trong mô tả thiết kế nhưng chúng phải được khai báo trước khi chúng được sử dụng.

Ví dụ 3.11

```
input m;
input [2:0] n;
```

output: tất cả port output của một module được khai báo với một phát biểu output. Nếu không có một loại dữ liệu khác như là reg, wand, wor, hoặc tri được khai báo, thì output port sẽ có loại dữ liệu mặc định là wire và nó cũng được điều khiển bởi cú pháp của wire. Một phát biểu output có thể xuất hiện ở bất cứ vị trí nào trong mô tả thiết kế, nhưng nó phải được khai báo trước khi được sử dụng. Ta có thể khai báo độ rộng của một output như một mảng (vector) của những tín hiệu. Nếu ta sử dụng loại dữ liệu reg để khai báo cho output thì reg phải có cùng độ rộng với độ rộng của mảng (vector) của tín hiệu.

Ví dụ 3.12

```
output a;
output [2:0] b;
reg [2:0] b;
```

inout: ta có thể khai báo port hai chiều (bidirectional) với phát biểu inout. Một port inout có loại dữ liệu là wire và được điều khiển bởi cú pháp của wire. Ta phải khai báo port inout trước khi nó được sử dụng.

Ví dụ 3.13

```
inout a:
inout [2:0] b;
```

Ví dụ 3.14

```
module fulladder(cout, sum, in1, in2, in3);
    input in1, in2, in3;  // khai báo 3 ngõ vào
    output cout, sum;  //khai báo 2 ngõ ra
    wire in1, in2, in3; //khai báo kiểu dữ liệu
    reg cout, sum; //khai báo kiểu dữ liệu
    …
endmodule
```

3.7 Khai báo mảng và phần tử nhớ một và hai chiều.

3.7.1 Giới thiệu

Verilog chỉ hỗ trợ khai báo mảng một và hai chiều. Những mảng một chiều được gọi là bit-vectors và nó có thể là loại dữ liệu net hoặc reg. Những mảng hai chiều được gọi là những phần tử nhớ và là loại dữ liệu reg. Ta có thể định nghĩa độ rộng cho tất cả các loại dữ liệu được trình bày trong chương này. Việc định nghĩa độ rộng cung cấp một cách để tạo ra một bit-vector. Cú pháp của một mô tả độ rộng là [msb:lsb]. Những biểu thức của msb (bit có trọng số lớn nhất) và lsb (bit có trọng số nhỏ nhất)

phải là những biểu thức có giá trị hằng khác 0. Những biểu thức có giá trị hằng chỉ có thể tạo nên bởi những hằng số, những tham số của Verilog và các toán tử. Không có giới hạn trong việc định nghĩa độ rộng tối đa của một bit-vector trong Verilog, tuy nhiên việc giới hạn này có thể sẽ phụ thuộc vào công cụ mô phỏng, tổng hợp, hoặc những công cụ khác.

3.7.2 Mảng net

Ví dụ 3.15

```
wire [63:0] bus;
```

Ở Ví dụ 3.15 mô tả việc khai báo một wire có độ rộng 64 bits.

Ví dụ 3.16

```
wire vectored [31:0] bus1;
wire scalared [31:0] bus2;
```

Ở Ví dụ 3.16, ta sử dụng hai từ khóa chỉ dẫn ‘vectored’ và ‘scalared’, chúng đều được dùng để khai báo multi-bit nets, tuy nhiên chúng khác nhau ở chỗ có cho phép mô tả từng bit hay từng phần của net hay không.

assign bus1 [1] = 1’b1; // sai cú pháp vì sử dụng việc chọn bit của một vectored net.

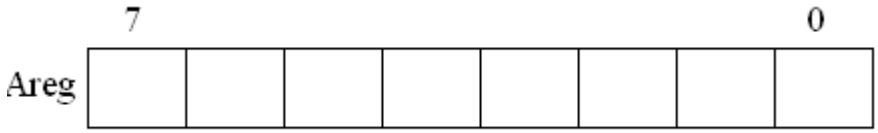
assign bus2 [1] = 1’b1; // đúng vì việc chọn bit của một scalared net là được phép.

Trình biên dịch chấp nhận cú pháp của những cấu trúc mô tả Verilog này, tuy nhiên chúng sẽ bị bỏ qua khi mạch được tổng hợp ra phần cứng.

3.7.3 Mảng thanh ghi

Ví dụ 3.17 Khai báo mảng thanh ghi

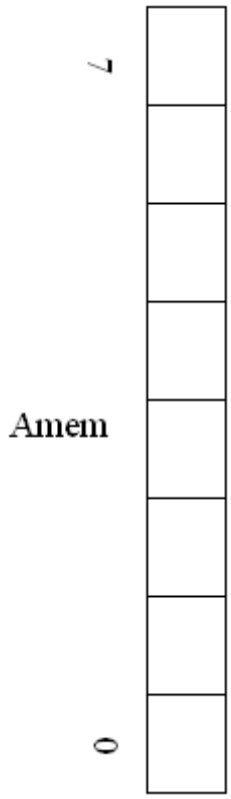
reg *[7:0] areg*



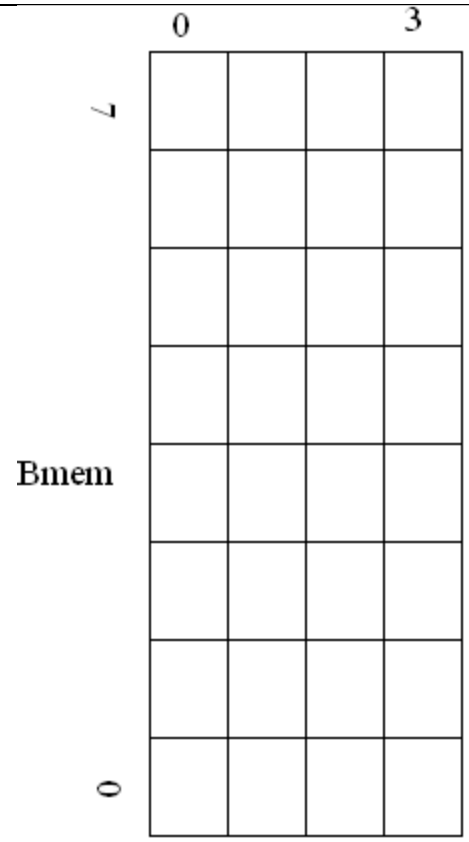
3.7.4 Mảng phần tử nhớ

Ví dụ 3.18 : Khai báo mảng phần tử nhớ

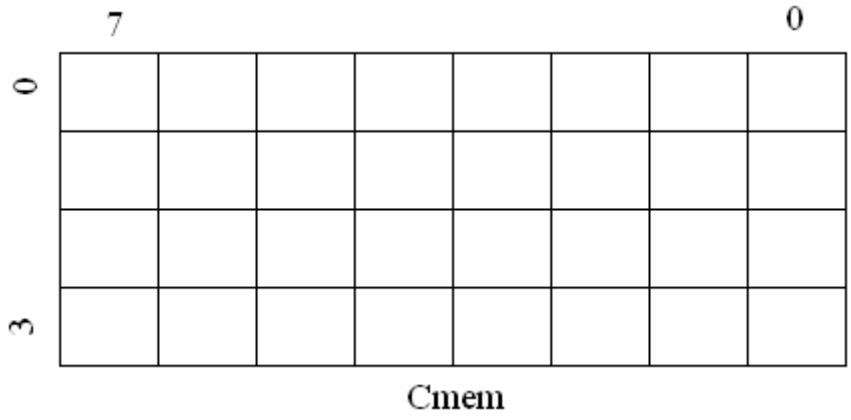
reg *amem [7:0] ;*



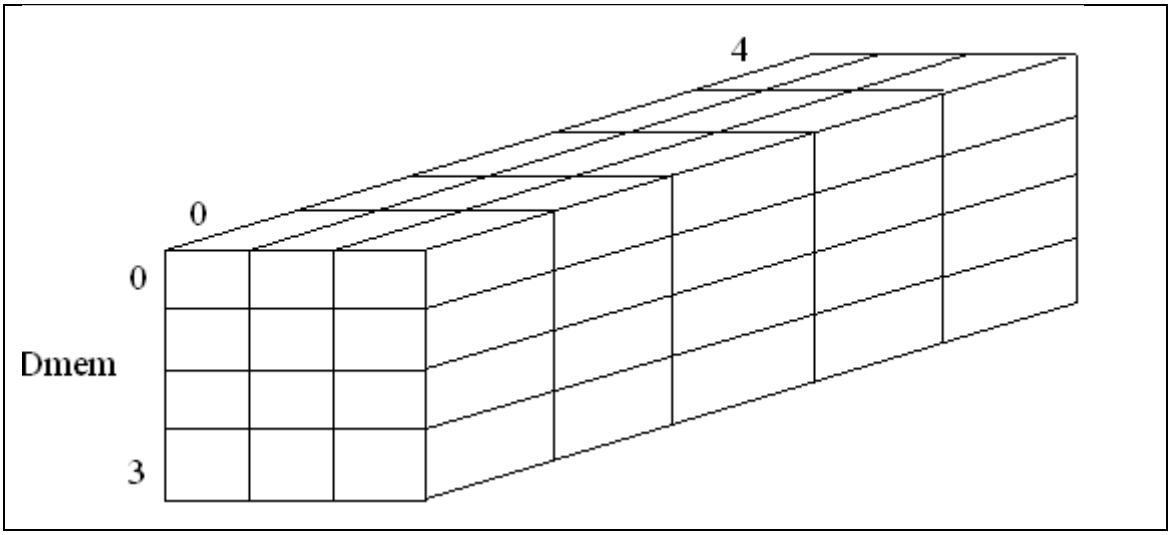
reg *bmem [7:0][0:3];*



reg *[7:0] cmem [0:3];*



reg *[2:0] dmem [0:3][0:4];*



3.8 Khai báo loại dữ liệu biến - số nguyên, thời gian, số thực, và thời gian thực

3.8.1 Giới thiệu

Thêm vào khả năng mô hình hóa cho phần cứng trong Verilog, ta có thể sử dụng thêm một số loại dữ liệu biến khác ngoài dữ liệu biến reg. Mặc dù biến dữ liệu reg có thể được dùng cho những chức năng tổng quát như đếm thời gian, lưu giữ sự thay đổi giá trị của net, biến dữ liệu integer và time thì cung cấp sự thuận lợi và dễ đọc hiểu hơn trong việc mô tả thiết kế.

3.8.2 Integer

Loại dữ liệu integer là biến có chức năng tổng quát được dùng để tính toán số lượng. Nó không được xem như là thanh ghi trong phần cứng thiết kế. Loại dữ liệu integer gồm 32 bit và nó có thể được gán và sử dụng hoàn toàn giống như loại biến dữ liệu reg. Phép gán qui trình (procedural assignment) được dùng để kích sự thay đổi giá trị của loại dữ liệu integer.

Những phép tính trên biến dữ liệu integer sẽ tạo ra những kết quả ở dưới dạng bù 2.

Ví dụ 3.19

```
integer i1, i2;
```

3.8.3 Time

Biến dữ liệu time có độ rộng 64 bit và này thường được dùng để lưu giữ giá trị output của hàm hệ thống $time hoặc để tính toán thời gian chạy mô phỏng trong những trường hợp mà ở đó việc kiểm tra định thời là bắt buộc hoặc cho những mục đích dò tìm và phát hiện lỗi của thiết kế trong quá trình mô phỏng.

Loại dữ liệu time có thể được gán và sử dụng hoàn toàn giống như loại biến dữ liệu reg. Phép gán qui trình (procedural assignment) được dùng để kích sự thay đổi giá trị của loại dữ liệu time

Ví dụ 3.20

```
time t1, t2;
```

3.8.4 Số thực (real) và thời gian thực (realtime)

Bên cạnh biến dữ liệu integer và time, Verilog còn có hỗ trợ việc sử dụng hằng số thực và biến dữ liệu thực (real). Ngoại trừ những ngoại lệ như trình bày phía dưới thì biến dữ liệu real có thể được sử dụng tương tự như integer và time.

- Không phải tất cả các phép toán trong Verilog có thể được sử dụng với những số thực.

Bảng 3.2 Toán tử với số thực

Unary+ unary-	Unary operation
+ - * / **	Arithmetic

> >= < <=	Relational
! && ||	Logical
== !=	Logical equality
?:	Conditional

Bảng 3.3 Toán tử kết nối và sao chép

{} {{}}	concatenate replicate
%	modulus
=== !==	case equality
~ & | ^ ~^ ^~	bitwise
^ ~^ ^~ & ~& | ~|	reduction
>> << >>> <<<	shift

- Biến dữ liệu không co khai báo độ rộng của biến. Việc tính toán được thực hiện dùng chuẩn định dạng IEEE floating point.
- Biến dữ liệu có giá trị mặc định là 0.

Thời gian thực (realtime) được khai báo và sử dụng tương tự như số thực (real). Chúng có thể hoán đổi cho nhau.

Ví dụ 3.21

```
real float;
realtime rtime;
```

3.9 Khai báo tham số

3.9.1 Giới thiệu

Trong Verilog HDL, loại dữ liệu tham số (parameter) không thuộc loại dữ liệu biến (variables: reg, integer, time, real, realtime) cũng như loại dữ liệu net. Dữ liệu tham số không phải là biến mà chúng là hằng số. Có hai loại tham số: tham số module (module parameter), và tham số đặc tả (specify parameter). Việc khai báo trùng tên giữa net, biến hay tham số là không được phép.

Cả hai loại tham số trên đều được phép khai báo độ rộng. Mặc định, *parameters* và *specparams* sẽ có độ rộng đủ để chứa giá trị của hằng số, ngoại trừ khi tham số đó có khai báo độ rộng.

3.9.2 Tham số module (module parameter)

Tham số module có hai loại khai báo: parameter và localparameter.

3.9.2.1 Parameter

3.9.2.1.1 Giới thiệu

Giá trị của khai báo parameter trong một module có thể được thay đổi từ bên ngoài module đó bằng phát biểu *defparam* hoặc phát biểu gọi instance của module đó. Thông thường khai báo parameter được dùng để mô tả định thời hoặc độ rộng của biến.

Ví dụ 3.22

```
parameter msb = 1; // định nghĩa tham số msb có giá trị hằng số là 1
parameter e = 43, f =789; // định nghĩa hai hằng số
parameter r = 46.7; // khai báo r là một hằng số thực
```

```
parameter byte_size = 9,
byte_mask = byte_size - 6;
parameter average_delay = (r + f) / 2;
parameter signed [3:0] mux_selector = 0;
parameter real r1 = 3.6e19;
parameter p1 = 13'h7e;
parameter [31:0] dec_const = 1'b1; // giá trị được đổi sang 32 bit
parameter newconst = 3'h4; // ngụ ý là tham số này có độ rộng [2:0]
parameter newconst = 4; // ngụ ý là tham số này có độ rộng tối thiểu
là 32 bit.
```

3.9.2.1.2 Thay đổi giá trị của tham số khai báo *parameter*

Một tham số module có thể có mô tả loại dữ liệu và mô tả độ rộng. Sự tác động của giá trị tham số mới khi nó đè lên giá trị của tham số đã được khai báo ban đầu trong module với mô tả loại dữ liệu và mô tả độ rộng sẽ tuân theo những qui luật sau:

- Một khai báo tham số mà không mô tả loại dữ liệu và độ rộng sẽ có loại dữ liệu và độ rộng mặc định của giá trị cuối cùng được gán vào tham số đó.
- Một khai báo tham số mà không mô tả loại dữ liệu mà chỉ mô tả độ rộng thì độ rộng của tham số sẽ không đổi, còn loại dữ liệu sẽ là unsigned khi giá trị mới được đè lên.
- Một khai báo tham số mà chỉ mô tả loại dữ liệu mà không mô tả độ rộng thì loại dữ liệu của tham số sẽ không đổi, còn độ rộng sẽ có giá trị đủ để chừa giá trị mới được đè lên.
- Một khai báo tham số mà mô tả cả loại dữ liệu là có dấu và mô tả cả độ rộng thì loại dữ liệu và độ rộng của tham số cũng sẽ không đổi khi giá trị mới được đè lên.

Trong Verilog có hai cách để thay đổi giá trị của tham số được khai báo bởi *parameter* : một là phát biểu defparam, với phát biểu này nó sẽ cho phép gán giá trị mới vào tham số trong module bằng cách dùng tên gọi một cách phân cấp, hai là phép gán giá trị tham số khi gọi instance của module đó, bằng cách này sẽ cho phép thay đổi giá trị tham số trong cùng một dòng với việc gọi instance của module đó.

3.9.2.1.2.1 Phát biểu *defparam*

Sử dụng phát biểu defparam, giá trị tham số có thể được thay đổi bên trong instance của module thông qua việc sử dụng tên phân cấp của tham số.

Tuy nhiên, phát biểu defparam được mô tả trong một instance hoặc một dãy các instance thì sẽ không làm thay đổi giá trị tham số trong những instance khác của cùng một module.

Biểu thức bên phải của phép gán defparam là biểu thức hằng số chỉ bao gồm số và những tham số tham chiếu đã được khai báo trước đó trong cùng module với phát biểu *defparam*.

Phát biểu defparam đặc biệt hữu dụng vì ta có thể nhóm tất cả các phép gán thay đổi giá trị các tham số của các module khác nhau chỉ trong một module.

Trong trường hợp có nhiều phát biểu defparam cho một tham số duy nhất thì giá trị tham số đó sẽ lấy giá trị của phát biểu defparam sau cùng. Nếu phát biểu defparam của một tham số được khai báo trong nhiều file khác nhau thì giá trị của tham số đó sẽ không được xác định.

Ví dụ 3.23

```
module top;
    reg clk;
```

```
        reg [0:4] in1;
        reg [0:9] in2;
        wire [0:4] o1;
        wire [0:9] o2;
        vdff m1 (o1, in1, clk);
        vdff m2 (o2, in2, clk);
endmodule
module vdff (out, in, clk);
        parameter size = 1, delay = 1;
        input [0:size-1] in;
        input clk;
        output [0:size-1] out;
        reg [0:size-1] out;
        always @(posedge clk)
        # delay out = in;
endmodule
module annotate;
        defparam
        top.m1.size = 5,
        top.m1.delay = 10,
        top.m2.size = 10,
        top.m2.delay = 20;
endmodule
```

Trong Ví dụ 3.22, module *annotate* có phát biểu defparam, giá trị từ phát biểu này sẽ đè lên những giá trị tham số *size* và *delay* trong instance *m1* và *m2* trong module top. Hai module *top* và *annotate* đều được xem như module top-level.

3.9.2.1.2.2 Phép gán giá trị tham số khi gọi instance của module

Trong Verilog có một phương pháp khác dùng để gán giá trị đến một tham số bên trong instance của một module đó là sử dụng một trong hai dạng của phép gán giá trị tham số trong instance của module. Một là phép gán theo thứ tự danh sách tham số, hai là phép gán bởi tên. Hai dạng phép gán này không thể đặt lẫn lộn với nhau mà chúng chỉ có thể là một trong hai dạng cho toàn bộ instance của module.

Việc gán giá trị tham số instance của module theo thứ tự danh sách tham số tương tự như việc gán giá trị trì hoãn cho những cổng của instance, còn việc gán giá trị tham số instance của module theo tên tham số thì tương tự như việc kết nối port của module bởi tên. Nó gán những giá trị tham số cho những instance cụ thể mà trong module của những instance này đã định nghĩa những tham số trên.

Một tham số mà đã được khai báo trong một block, một tác vụ hay một hàm chỉ có thể khai báo lại một cách trực tiếp dùng phát biểu *defparam*. Tuy nhiên, nếu giá trị tham số phụ thuộc vào một tham số thứ hai, thì việc định nghĩa lại giá trị tham số thứ hai cũng sẽ cập nhật giá trị của tham số thứ nhất.

1. Phép gán giá trị tham số theo thứ tự danh sách tham số

Thứ tự của những phép gán trong phép gán giá trị tham số theo thứ tự danh sách tham số instance của module sẽ theo thứ tự tham số lúc khai báo bên trong module. Nó không cần thiết phai gán giá trị cho tất cả các tham số có bên trong module khi dùng phương pháp này. Tuy nhiên, ta không thể nhảy qua một tham số. Do đó, để gán những giá trị cho một phần những tham số trong tất cả các tham số đã khai báo trong module thì những phép gán để thay thế giá trị của một phần những tham số đó sẽ đứng trước những khai báo của những tham số còn lại. Một phương pháp khác đó là

phải gán giá trị cho tất cả các tham số nhưng dùng giá trị mặc định (cùng có giá trị như được gán trong khai báo tham số trong định nghĩa module) cho các tham số mà không cần có giá trị mới.

Xét Ví dụ 3.24, trong ví dụ này những tham số bên trong instance của những module mod_a, mod_c, và mod_d được thay đổi trong khi gọi instance.

Ví dụ 3.24

```
module tb1;
    wire [9:0] out_a, out_d;
    wire [4:0] out_b, out_c;
    reg [9:0] in_a, in_d;
    reg [4:0] in_b, in_c;
    reg clk;
    // Tạo testbench clock và stimulus.
    // Bốn instance của module vdff với phép gán giá trị tham số theo thứ tự danh sách tham số
    // mod_a có hai giá trị tham số mới size=10 và delay=15
    // mod_b có giá trị tham số mặc định là (size=5, delay=1)
    // mod_c có một giá trị tham số mặc định là size=5 và một giá trị mới là delay=12
    // Để thay đổi giá trị của tham số delay, ta cũng cần phải mô tả giá trị mặc định của tham số size
    // mod_d có một giá trị tham số mới là size=10, và giá trị tham số delay vẫn giữ giá trị mặc định của nó.
    vdff #(10,15) mod_a (.out(out_a), .in(in_a), .clk(clk));
    vdff mod_b (.out(out_b), .in(in_b), .clk(clk));
    vdff #( 5,12) mod_c (.out(out_c), .in(in_c), .clk(clk));
```

```
    vdff #(10) mod_d (.out(out_d), .in(in_d), .clk(clk));
endmodule
module vdff (out, in, clk);
    parameter size=5, delay=1;
    output [size-1:0] out;
    input [size-1:0] in;
    input clk;
    reg [size-1:0] out;
    always @(posedge clk)
    #delay out = in;
endmodule
```

Những giá trị của tham số cục bộ không thể bị đè lên, do đó chúng không được xem như là một phần thứ tụ của danh sách cho phép gán giá trị tham số. Trong Ví dụ 3.25, addr_width sẽ được gán giá trị 12, và data_width sẽ được gán giá trị 16. Mem_size sẽ không được gán giá trị một cách tường minh do thứ tự danh sách, nhưng nó sẽ có giá trị 4096 do biểu thức khai báo của nó.

Ví dụ 3.25

```
module my_mem (addr, data);
    parameter addr_width = 16;
    localparam mem_size = 1 << addr_width;
    parameter data_width = 8;
    ...
endmodule
module top;
    ...
```

```
    my_mem #(12, 16) m(addr,data);
endmodule
```

2. Phép gán giá trị tham số bởi tên

Phép gán giá trị tham số bởi tên bao gồm tên tường minh của tham số và giá trị mới của nó. Tên của tham số sẽ là tên được mô tả trong instance của module.

Ta không cần thiết gán những giá trị đến tất cả các tham số bên trong module khi sử dụng phương pháp này. Chỉ những tham số nào mà được gán giá trị mới thì mới cần được chỉ ra.

Biểu thức tham số có thể là một lựa chọn để việc gọi instance của module có thể ghi lại việc hiện diện của một tham số mà không cần bất kì một phép gán đến nó. Những dấu đóng mở ngoặc được đòi hỏi, và trong trường hợp này tham số sẽ giữ giá trị mặc định của nó. Khi một tham số được gán một giá trị, thì một phép gán khác đến tên tham số này là không được phép.

Xét Ví dụ 3.26, trong ví dụ này cả những tham số của mod_a và chỉ một tham số của mod_c và mod_d bị thay đổi trong khi gọi instance của module.

Ví dụ 3.26

```
module tb2;
    wire [9:0] out_a, out_d;
    wire [4:0] out_b, out_c;
    reg [9:0] in_a, in_d;
    reg [4:0] in_b, in_c;
    reg clk;
    // Code tạo testbench clock & stimulus ...
```

```
        // Bốn instance của moduel vdff với giá trị tham số được gán bởi tên
        // mod_a có giá trị tham số mới là size=10 và delay=15
        // mod_b có giá trị tham số mặc định là (size=5, delay=1)
        // mod_c có một giá trị tham số mặc định là size=5 và có một giá trị
        tham số mới là delay=12
        // mod_d có một giá trị tham số mới là size=10.
        // còn tham số delay vẫn giữ giá trị mặc định
        vdff #(.size(10),.delay(15)) mod_a (.out(out_a),.in(in_a),.clk(clk));
        vdff mod_b (.out(out_b),.in(in_b),.clk(clk));
        vdff #(.delay(12)) mod_c (.out(out_c),.in(in_c),.clk(clk));
        vdff #(.delay( ),.size(10) ) mod_d (.out(out_d),.in(in_d),.clk(clk));
endmodule
module vdff (out, in, clk);
        parameter size=5, delay=1;
        output [size-1:0] out;
        input [size-1:0] in;
        input clk;
        reg [size-1:0] out;
        always @(posedge clk)
        #delay out = in;
endmodule
```

Nó thì hợp lệ khi gọi những instance của module dùng những loại định nghĩa lại tham số trong cùng module ở top-level. Xét ví dụ sau, trong ví dụ này những tham số của mod_a bị thay đổi bằng cách dùng việc định nghĩa lại tham số theo thứ tự danh sách và tham số thứ hai của mod_c được thay đổi bằng cách dùng việc định nghĩa lại tham số bằng tên trong khi gọi instance của module.

Ví dụ 3.27

```
module tb3;
    // sự pha trộn giữa instance có khai báo tham số theo thứ tự và
    instance có khai báo tham số theo tên thì hợp lệ
    vdff #(10, 15) mod_a (.out(out_a), .in(in_a), .clk(clk));
    vdff mod_b (.out(out_b), .in(in_b), .clk(clk));
    vdff #(.delay(12)) mod_c (.out(out_c), .in(in_c), .clk(clk));
endmodule
```

Nó sẽ không hợp lệ khi gọi instace của bất kì module nào dùng lẫn lộn những phép gán lại giá trị tham số bằng thứ tự danh sách tham số và tên giống như trong phép gọi instance của module mod_a ở dưới.

Ví dụ 3.28

```
    // instance mod_a  không hợp lệ do có sự pha trộn giữa các phép gán
    tham số
    vdff #(10, .delay(15)) mod_a (.out(out_a), .in(in_a), .clk(clk));
```

3.9.2.1.3 Sự phụ thuộc tham số

Một tham số (ví dụ, *memory_size*) có thể được định nghĩa với một biểu thức chứa những tham số khác (ví dụ, word_size). Tuy nhiên, việc gán đè giá trị tham số, có thể là bằng phát biểu *defparam* hoặc trong phát biểu gọi instance của module, sẽ thay thế một cách hiệu quả việc định nghĩa tham số với một biểu thức mới. Bởi vì *memory_size* phụ thuộc vào giá trị của *word_size*, bất kì có sự thay đổi nào của word_size sẽ làm thay đổi giá trị của memory_size. Ví dụ, trong khai báo tham số sau, một giá trị mới cập nhật của word_size, có thể là bởi phát biểu defparam hoặc phát biểu gọi instance của module mà trong module này đã định nghĩa những tham số

trên, thì giá trị của memory_size sẽ được tự động cập nhật. Nêu memory_size được cập nhật bởi phát biểu defparam hay một phát biểu gọi instance thì nó sẽ lấy giá trị đó mà không cần quan tâm đến giá trị của word_size.

Ví dụ 3.29

```
parameter
word_size = 32,
memory_size = word_size * 4096;
```

3.9.2.2 Tham số cục bộ (localparam)

Trong Verilog, tham số cục bộ (*localparam*) giống tương tự với tham số (*parameter*) ngoại trừ là nó không thể được gán lại giá trị bởi phát biểu *defparam* hoặc phép gán giá trị tham số khi gọi instance của module. Những tham số cục bộ (*localparam*) có thể được gán bởi những biểu thức hằng số chứa những tham số (*parameter*) mà những tham số (parameter) này có thể được gán lại giá trị bởi phát biểu *defparam* hoặc phép gán giá trị tham số khi gọi instance của module.

Việc chọn bit hay một phần của tham số cục bộ mà loại dữ liệu của nó không phải là real thì được phép.

Ví dụ 3.30

```
localparam thamso1;
localparam signed [3:0] thamso2;
localparam time t1;
localparam integer int2;
localparam var = 5*6;
```

3.9.3 Tham số đặc tả (specify parameter)

Từ khóa specparam khai báo nó là một loại đặc biệt của tham số (parameter) chỉ dùng cho mục đích cung cấp giá trị định thời (timing) và giá trị trì hoãn (delay), nhưng nó có thể xuất hiện trong bất kì biểu thức nào mà biểu thức đó không được gán đến một tham số (*parameter*) và biểu thức đó cũng không phải là phần mô tả độ rộng trong một khai báo. Những tham số đặc tả (specparams) được phép khai báo bên trong khối đặc tả (specify block) hoặc bên trong một module chính.

Một tham số đặc tả (specify parameter) khai báo bên ngoài một khối đặc tả (specify block) thì cần được khai báo trước khi nó được sử dụng. Giá trị mà được gán đến một tham số đặc tả có thể là một biểu thức hằng số bất kì. Một tham số đặc tả có thể được dùng như là phần của một biểu thức hằng số cho một khai báo tham số đặc tả kế tiếp. Không giống như một tham số module (module parameter), một tham số đặc tả không thể được gán lại giá trị từ bên trong ngôn ngữ Verilog, nhưng nó có thể được gán lại giá trị thông qua tập tin dữ liệu SDF (Standard Delay Format).

Những tham số đặc tả (specify parameter) và tham số module (module parameter) thì không thể thay thế cho nhau. Ngoài ra, tham số module (module parameter) không thể được gán bởi một biểu thức hằng số mà có chứa tham số đặc tả (specify parameter). Bảng 3.4 tóm tắt sự khác nhau giữa hai loại khai báo tham số.

Bảng 3.4 Sự khác nhau giữa hai loại khai báo tham số

Specparams (tham số đặc tả)	Parameters (tham số module)
Sử dụng từ khóa *specparam*	Sử dụng từ khóa *parameter*
Cần được khai báo bên trong một module hoặc một khối đặc tả (specify block)	Cần được khai báo bên ngoài những khối đặc tả (specify block)

Có thể được dùng bên trong một module hoặc một khối đặc tả (specify block)	Không thể được dùng bên trong những khối đặc tả (specify block).
Có thể được gán bởi tham số đặc tả (specparam) và tham số module (parameter).	Không thể được gán bởi specparams.
Sử dụng tập tin dữ liệu SDF để gán đè giá trị cho tham số đặc tả.	Dùng phát biểu defparam hoặc phép gán giá trị tham số cho instance của module để gán đè giá trị cho tham số.

Một tham số đặc tả (specify parameter) có thể được mô tả độ rộng. Độ rộng của những tham số đặc tả cần tuân theo những qui luật sau:

- Một khai báo tham số đặc tả mà không có mô tả độ rộng thì mặc định sẽ là độ rộng của giá trị cuối cùng được gán đến nó, sau khi có bất kì giá trị nào gán đè lên nó.
- Một khai báo tham số đặc tả mà có mô tả độ rộng thì độ rộng của nó sẽ theo độ rộng khai báo. Độ rộng sẽ không bị ảnh hưởng bởi bất kì giá trị nào được gán đè lên nó.

Việc chọn bit hay một phần của tham số cục bộ mà loại dữ liệu của nó không phải là real thì được phép.

Ví dụ 3.31

```
specify
specparam tRise_clk_q = 150, tFall_clk_q = 200;
specparam tRise_control = 40, tFall_control = 50;
endspecify
```

Những dòng ở giữa những từ khóa *specify* và *endspecify* là để khai báo bốn tham số đặc tả. Dòng đầu tiên khai báo hai tham số đặc tả

tRise_clk_q và tFall_clk_q với giá trị tương ứng là 150 và 200. Dòng thứ hai khai báo hai tham số đặc tả tRise_control và tFall_control với giá trị tương ứng là 40 và 50.

Ví dụ 3.32

```
module RAM16GEN (output [7:0] DOUT, input [7:0] DIN,
input [5:0] ADR,
input WE, CE);
specparam dhold = 1.0;
specparam ddly = 1.0;
parameter width = 1;
parameter regsize = dhold + 1.0; // Không hợp lệ - không thể gán tham số đặc tả (specparam) đến một tham số (parameter)
endmodule
```

3.10 Bài tập

1. Trong ngôn ngữ mô tả phần cứng Verilog HDL, có mấy loại dữ liệu cơ bản? Nêu chức năng sử dụng của mỗi loại.
2. Trong kiểu dữ liệu net có những loại khai báo dữ liệu nào? Nêu sự khác nhau giữa các loại khai báo dữ liệu net?
3. Trong kiểu dữ liệu biến có những loại khai báo dữ liệu nào? Nêu sự khác nhau giữa các loại khai báo dữ liệu biến?
4. Trong ngôn ngữ mô tả phần cứng Verilog HDL, có những loại tham số nào? Nêu sự khác biệt giữa tham số module và tham số đặc tả?
5. Khi nào ta sử dụng khai báo *defparam*?

6. Có mấy loại tham số module ? Nêu sự khác biệt giữa hai khai báo *parameter* và *localparameter* trong tham số module ?

Chương 4. Biểu thức

Biểu thức là mệnh đề mô tả các toán tử và các giá trị của các toán hạng trong Verilog HDL và cách sử dụng chúng.

Một biểu thức là một cấu trúc tổ hợp của các toán hạng với toán tử để tạo nên một kết quả là một hàm của các giá trị toán hạng và ngữ nghĩa của toán tử. Bất kỳ một toán hạng hợp lệ như là một net bit-select, bất chấp toán tử đều được xem như là một biểu thức. Bất kỳ nơi nào một giá cần một câu lệnh trong Verilog, biểu thức có thể được sử dụng.

4.1 Biểu thức giá trị hằng số

Một vài cấu trúc câu lệnh yêu cầu một biểu thức là một biểu thức giá trị hằng số. Toán hạng của biểu thức giá trị hằng số bao gồm các hằng số, chuỗi, tham biến, bit-select hằng, part-select hằng cửa tham biến, các hàm gọi hằng (10.4.5), và các hàm hệ thống gọi, nhưng chúng có thể sử dụng bất kỳ toán tử nào được định nghĩa trong Bảng 4.1.

Các hàm gọi hệ thống hằng gọi các hàm được xây dựng trong hệ thống nơi mà các đối số là các biểu thức hằng. Khi sử dụng trong biểu thức hằng số, các hàm gọi sẽ định giá trong thời gian thiết lập. Các hàm hệ thống sử dụng trong hàm gọi hệ thống hằng số gọi là các hàm thuần túy, giá trị của nó chỉ phụ thuộc vào các đối số đầu vào và không ảnh hưởng đến xung quanh. Cụ thể, các hàm hệ thống cho phép trong biểu thức hằng số được chuyển đổi từ danh sách các hàm hệ thống trong phần 17.8 và danh sách các hàm hệ thống toán học trong 17.11.

Các loại dữ liệu reg, integer, time, real và realtime là các loại dữ liệu cho biến. Mô tả liên qua tới biến áp dụng cho tất cả các loại dữ liệu này.

Một toán hạng có thể là một trong:

- Hằng số (bao gồm cả số thực) hoặc chuỗi.
- Tham biến (bao gồm cả tham biến nội và tham biến chỉ định).
- Bit-select và part-select của tham biến (không bao gồm số thực).
- Net.
- Bit-select và part-select của net.
- Biến reg, integer, hoặc time.
- Bit-select và part-select của biến reg, integer, hoặc time.
- Biến real hoặc realtime.
- Mảng các phần tử.
- Bit-select và part-select của mảng các phần tử.
- Một hàm gọi do người dùng định nghĩa hoặc hàm gọi hệ thống mà nó trả về bất kỳ giá trị nào bên trên.

4.2 Toán tử

Ký hiệu cho toán tử trong ngôn ngữ mô tả phần cứng Verilog tương tự như trong ngôn ngữ lập trình C. Bảng 4.1 là danh sách các toán tử này.

Bảng 4.1 Danh sách các toán tử

{} {{}}	Toán tử kết nối, toán tử lặp
+ -	Toán tử một ngôi
+ - * / **	Toán tử số học
%	Toán tử chia lấy phần dư
> >= < <=	Toán tử quan hệ
!	Toán tử nghịch đảo logic
&&	Toán tử logic and
||	Toán tử logic or
==	Toán tử bằng

!=	Toán tử không bằng
===	Toán tử bằng case
!==	Toán tử không bằng case
~	Toán tử phủ định bitwise
&	Toán tử and bitwise
|	Toán tử or toàn bộ bitwise
^	Toán tử or loại trì bitwise
^~ hoặc ~^	Toán tử tương đương
<<	Toán tử dịch trái logic
>>	Toán tử dịch phải logic
<<<	Toán tử dịch trái toán học
>>>	Toán tử dịch phải toán học
?:	Toán tử điều kiện

4.2.1 Toán tử với toán hạng số thực

Các toán tử trong Bảng 4.2 là hợp lệ khi áp dụng đối với toán hạng số thực. Tất cả các toán tử khác sẽ xem như là bất hợp lệ khi sử dụng với toán hạng số thực.

Kết quả khi sử dụng toán tử logic và toán tử quan hệ trên số thực là một giá trị bit đơn vô hướng .

Bảng 4.2 Danh sách các toán tử không được sử dụng đối với số thực

unary + unary -	Toán tử một ngôi
+ - * / **	Toán tử số học
>, >=, <, <=	Toán tử quan hệ
! && ||	Toán tử logic
== !=	Toán tử bằng
?:	Toán tử điều kiện

Bảng 4.3 Danh sách toán tử không được phép sử dụng đối với toán tử số thực

{} {{}}	Toán tử ghép nối, thay thế
%	Toán tử chia lấy phần dư
=== !==	Toán tử bằng
~ , &, |, ^, ^~, ~^	Toán tử bitwwise
^, ^~, ~^, &, ~&, |, ~|	Toán tử giảm
<< >> <<< >>>	Toán tử dịch

4.2.2 Toán tử ưu tiên

Thứ tự ưu tiên của các toán tử trong Verilog được mô tả trong Bảng 4.4.

Các toán tử trong cùng một dòng trong Bảng 4.4 có thứ tự ưu tiên như nhau. Các dòng được sắp xếp theo thứ tự tăng dần độ ưu tiên. Ví dụ các toán tự *, /, và % có cùng độ ưu tiên và độ ưu tiên của nó cao hơn toán tử + và -.

Tất cả các toán tử sẽ được thực hiện từ trái sang phải, ngoại trừ toán tử điều kiện nó được thực hiện từ phải sang trái. Sự kết hợp toán tử theo thứ tự đối với các toán tử có cùng độ ưu tiên. Vì vậy, trong ví dụ này B sẽ cộng với A và sau đó lấy tổng A+B trừ cho C.

A+B-C

Khi toán tử có độ ưu tiên khác nhau, thì toán tử có độ ưu tiên cao sẽ thực hiện trước. Trong ví dụ tiếp theo, B sẽ chia cho C (toán tử chia có độ ưu tiên cao hơn), và sau đó kết quả sẽ được cộng thêm A.

A+B/C

Dấu ngoặc có thể sử dụng để thay đổi độ ưu tiên của toán tử

(A+B)/C // không giống với A+B/C

Bảng 4.4 Thứ tự ưu tiên của toán tử

+ - ! ~ & ~& | ~| ^ ~^ ^~(toán tử một ngôi)	Độ ưu tiên cao nhất
**	Độ ưu tiên cao nhì
* / %	Độ ưu tiên giảm dần từ cao xuống thấp
<< >> <<< >>>	
< <= > >=	
== != === !==	
&(toán tử 2 ngôi)	
^ ^~ ~^(toán tử 2 ngôi)	
|(toán tử 2 ngôi)	
&&	
||	
?:	Độ ưu tiên thấp nhì
{} {{}}	Độ ưu tiên thấp nhất

4.2.3 Sử dụng số nguyên trong biểu thức

Số nguyên có thể sử dụng như một toán hạng trong biểu thức. Một số nguyên có thể biểu diễn như là:

- Một số nguyên không dấu, không cơ số (ví dụ 12…)
- Một số nguyên không dấu, có cơ số (ví dụ d12, sd12,…)
- Một số nguyên có dấu có cơ số (ví dụ 16’d12,16’sd12,…)

Một giá trị phủ định của một số nguyên không chỉ rõ cơ số sẽ được đánh giá khác với một số nguyên chỉ rõ cơ số. Một số nguyên không có cơ số sẽ được đánh giá như là một giá trị không dấu gồm hai phần: dấu và giá

trị. Một số nguyên không đấu có cơ số sẽ được đánh giá như là một giá trị không dấu.

Ví dụ 4.1 chỉ ra 4 cách để viết biểu thức "-12 chia 3". Chú ý rằng cả hia giá trị"-12" và "–'d12" được đánh giá là giống nhau về 2 thành phần bit, nhưng trong biểu thức –'d12 không còn được định danh như là một số phủ định có dấu.

Ví dụ 4.1

```
integer IntA;
IntA = -12 / 3;// kết quả là  -4.
IntA = -'d 12 / 3;// kết quả là 1431655761.
IntA = -'sd 12 / 3;// kết quả là -4.
IntA = -4'sd 12 / 3; // -4'sd12 là một số âm 4-bit là 1100, với -4. -(-4)
= 4.
// kết quả là 1.
```

4.2.4 Thứ tự tính toán trong biểu thức

Toán tử phải thực hiện theo các quy tắc kết hợp trong khi đánh giá một biểu thức như được miêu tả trong 4.2.2. Tuy nhiên nếu kết quả cuối cùng của biểu thức có thể được phát hiện sớm hơn, thì toàn bộ biểu thức không cần được đánh giá hết. Điều này gọi là *ngắn mạch (short-circuiting)* một đánh giá biểu thức.

Ví dụ 4.2

```
Reg regA, regB, regC, result;
Result = regA&(regB|regC)
```

Nếu giá trị của regA là 0 thì kết quả của biểu thức có thể được phát hiện là 0 mà không cần tính toán giá trị của biểu thức con regB|regC.

4.2.5 Toán tử số học

Toán tử hai ngôi được đưa ra trong Bảng 4.5

Bảng 4.5 Toán tử hai ngôi

a+b	a cộng b
a-b	a trừ b
a*b	a nhân b
a/b	a chia b
a%b	a chia b lấy dư
a**b	a lũy thừa b

Trong phép chia số nguyên cần phân tích phân số khi mẫu số là số 0. Đối với phép chia và phép chia lấy phần dư, nếu toán hạng thứ 2 là 0 thì kết quả của toàn bộ biểu thức phải là x. Trong phép chia lấy phần dư, ví dụ y%z cho ra kết quả là phần dư khi lấy y chia cho z, vì vậy khi z = 0 thì kết quả chính là y. Khi đó kết quả của phép chia lấy dư được gán bằng toán hạng đầu tiên.

Nếu một trong hai toán hạng đối với toán tử lũy thừa là số thực, thì kết quả cũng là số thực. Kết quả của toán tử lũy thừa là không xác định nếu toán hạng thứ nhất là 0 và toán hạng thứ hai không dương, hoặc nếu toán hạng thứ nhất là số âm và toán hạng thứ hai không là một số nguyên.

Nếu cả hai toán hạng của toán tử lũy thừa là số thực thì các loại kết quả được thể hiện trong phần 4.5.1và 4.6.1. Kết quả là ‘bx nếu toán hạng thứ nhất là 0 và toán hạng thứ hai là một số âm. Kết quả là 1 nếu toán hạng thứ 2 là 0.

Trong tất cả các trường hợp, toán hạng thứ hai của toán tử lũy thừa phải được xem là nửa xác thực.

Những điều này sẽ được minh họa trong Bảng 4.6.

Bảng 4.6 Toán tử lũy thừa

Toán hạng 1(op1) Toán hạng 2 (op2)	âm <-1	-1	0	1	dương > 1
dương	op1**op2	op2 là chẵn ->-1 op2 là lẻ ->1	0	1	op1**op2
0	1	1	1	1	1
âm	0	op2 là chẵn ->-1 op2 là lẻ ->1	‘bx	1	0

Toán tử số học một ngôi có quyền ưu tiên cao hơn đối với toán tử nhị phân. Toán tử một ngôi được đưa ra trong Bảng 4.7

Bảng 4.7 Toán tử số học một ngôi

+m	Toán tử một ngôi cộng m
-m	Toán tử một ngôi trừ m

Đối với toán tử số học, nếu bất kỳ toán hạng nào có giá trị bit là không xác định X và trở kháng cao Z thì kết quả chung của biểu thức phải là X.

Bảng 4.8 đưa ra một vài ví dụ về toán tử chia lấy dư và lũy thừa.

Bảng 4.8 Toán tử chia lấy dư và lũy thừa

Biểu thức	**Kết quả**	**Chú thích**
10%3	1	10 chia 3 dư 1
12%3	0	12 chia 3 không dư
-10%3	-1	Dấu của kết quả là dấu của toán hạng đầu tiên
11%-3	2	Dấu của kết quả là dấu của toán hạng đầu tiên
-4’d12%3	1	-4d’12 có giá trị là 1
3**2	9	3*3

2**3	8	2*2*2
2**0	1	Bất kỳ số nào lũy thừa 0 cũng bằng 1
2.0**-3’sb1	0.5	2.0 là số thực, nên kết quả cũng là số thực
2**-3’sb1	0	2**-1=1/2, có phần nguyên là số 0
0**-1	‘bx	0 lũy thừa số âm là một số không xác định
9**0.5	3.0	Kết quả là một số thực
9.0**(1/2)	1.0	½ kết quả là 0
-3.0**2.0	9.0	

4.2.6 Biểu thức số học với tập thanh ghi (regs) và số nguyên (integer)

Một giá trị được gán cho một biến reg hoặc net được xem như là một giá trị không dấu nếu không biến reg hoặc net phải được khai báo rõ ràng là có dấu. Một giá trị được gán cho một biến integer, real hoặc realtime được xem như là một giá trị có dấu. Một giá trị được gán cho biến time được xem như là một giá trị không dấu. Giá trị có dấu, ngoại trừ chúng được gán cho biến real và realtime, sẽ sử dụng một biểu diễn hai thành phần. Giá trị gán cho biến real và realtime sẽ sử dụng biểu diễn dấu chấm động. Sự chuyển đổi giữa giá trị có dấu và không dấu sẽ giữ nguyên sự biểu diễn, chỉ thay đổi sự thể hiện.

Danh Bảng 4.9 đưa ra cách giải thích mỗi loại dữ liệu trong toán tử số học

Bảng 4.9 Loại dữ liệu trong toán tử số học

Loại dữ liệu	Giải thích
net không dấu	Không dấu
net có dấu	Có dấu, bù 2
reg không dấu	Không dấu

reg có dấu	Có dấu, bù 2
integer	Có dấu, bù 2
time	Không dấu
real, realtime	Có dấu, dấu chấm động

Theo Ví dụ 4.3 sẽ cho thấy nhiều cách khác nhau để chia "trừ 12 chia 3"- sử dụng dữ liệu loại integer và reg trong biểu thức.

Ví dụ 4.3

```
integer intA;
reg [15:0] regA;
reg signed [15:0] regS;
intA = -4'd12;
regA = intA / 3;    // kết quả của biểu thức là -4,
// intA là dữ liệu loại integer, regA bằng 65532
regA = -4'd12;      // regA bằng 65524
intA = regA / 3;    // Kết quả của biểu thức 21841,
// regA là dữ liệu loại reg
intA = -4'd12 / 3;  // kết quả của biểu thức là 1431655761.
// -4'd12 thực tế là một dữ liệu loại reg 32-bit
regA = -12 / 3;     // kết quả của biểu thức là -4,
//-12 thực tế là một dữ liệu loại integer.
regS = -12 / 3;     // kết quả của biểu thức là -4. regS là một reg có dấu
regS = -4'sd12 / 3; // kết quả của biểu thức là 1. -4'sd12 là  4.
// Theo luật chia số nguyên lấy phần dư  4/3==1.
```

4.2.7 Toán tử quan hệ

Bảng 4.10 đã liệt kê và định nghĩa toán tử quan hệ

Bảng 4.10 Toán tử quan hệ

a<b	a nhỏ hơn b
a>b	a lớn hơn b
a<=b	a nhỏ hơn hoặc bằng b
a>=b	a lớn hơn hoặc bằng b

Một biểu thức sử dụng những toán tử quan hệ này sẽ có trường giá trị là 0 nếu quan hệ là sai, hoặc có giá trị là 1 nếu nó là đúng. Nếu mỗi toán hạng trong một toán tử quan hệ chứa giá trị không xác định (x) hoặc giá trị trở kháng cao (z), thì kết quả sẽ là một bit có giá trị không xác định (x).

Khi một hoặc cả hai toán hạng của một toán tử quan hệ là không dấu, biểu thức được hiểu như là so sánh giữa hai giá trị không dấu. Nếu toán hạng không bằng nhau về chiều dài bit, thì toán hạng nhỏ hơn sẽ thêm số 0 để có độ dài bằng toán hạng lớn hơn.

Khi cả hai toán hạng là có dấu, thì hiểu thức sẽ được hiểu như là so sánh giữa hai giá trị có dấu. Nếu toán hạng không bằng nhau về chiều dài bit, thì toán hạng nhỏ hơn sẽ thêm bit dấu để có độ dài bằng toán hạng lớn hơn.

Nếu một toán hạng là số thực thì toán hạng khác sẽ được chuyển đổi về dạng số thực và biểu thức được hiểu như là một so sánh giữa hai giá trị số thực.

Tất cả các toán tử quan hệ sẽ có độ ưu tiên giống nhau. Toán tử quan hệ sẽ có độ ưu tiên thấp hơn so với toán tử số học.

Ví dụ 4.4

Ví dụ sau sẽ minh họa việ thực thi các luật về độ ưu tiên

a<foo - 1// biểu thức này giống với biểu thức a<(foo - 1)

Nhưng . . .

foo-(1<a)// biểu thức này không giống với biểu thức foo-1<a

Khi foo-(1<a) được tính toán, biểu thức quan hệ sẽ được tính toán đầu tiên, và sau đố hoặc là 0, hoặc là 1 sẽ được trừ bởi foo. Khi foo-1<a được tính toán thì giá trị của toán hạng foo sẽ trừ đi 1 sau đó đem so sánh với a.

4.2.8 Toán tử so sánh bằng

Toán tử so sánh bằng có độ ưu tiên thấp hơn so với toán tử quan hệ. Bảng 4.11 liệt kê và định nghĩa toán tử so sánh bằng

Bảng 4.11 Toán tử so sánh bằng

a===b	a bằng b, bao gồm cả x và z
a!==b	a không bằng b, bao gồm cả x và z
a==b	a bằng b, kết quả có thể là không xác định
a!=b	a không bằng b, kết quả có thể là không xác định

Cả bốn toán tử so sánh bằng sẽ có độ ưu tiên giống nhau. Bốn toán tử này so sánh từng bit của các toán hạng. Giống như toán tử quan hệ, kết quả sẽ là 0 nếu so sánh sai và 1 nếu so sánh đúng.

Nếu toán hạng không bằng nhau về chiều dài bit và nếu một hoặc cả hai toán hạng là không dấu thì toán hạng nhở hơn sẽ thêm số 0 vào trước để cho bằng kích thước của toán hạng lớn. Nếu cả hai là có dấu thì toán hạng nhỏ hơn sẽ thêm bit dấu vào trước cho bằng kích thước của toán hạng lớn.

Nếu một toán hạng là một số thực, thì toán hạng còn lại sẽ chuyển về kiểu số thực và biểu thức được xem như là phép so sánh giữa hai số thực.

Trong toán tử == và != nếu toán hạng là không xác định (x) hoặc trở kháng cao (z) thì quan hệ là không xác định, và kết quả sẽ là một bit có giá trị không xác định (x).

Trong toán tử === và !==, sự so sánh sẽ hoàn thành như là một câu lệnh case. Bit x hoặc z trong toán hạng sẽ được so sánh và sẽ cho kết quả là bằng nếu giống nhau. Kết quả của toán tử sẽ là một giá trị xác định 0 hoặc 1.

4.2.9 Toán tử logic

Toán tử logic and (&&) và or (||) là toán tử logic liên kết. Kết quả của sự tính toán so sánh logic sẽ là 1, 0 hoặc nếu kết quả không rõ ràng thì kết quả là x. Độ ưu tiên của && lớn hơn || và cả hai có độ ưu tiên thấp hơn toán tử quan hệ và toán tử so sánh bằng.

Toán tử logic thứ 3 là toán tử nghịch đảo logic 1 ngôi (!). Toán tử nghịch đảo chuyển đổi toán hạng không phải số 0 hoặc 1 thành số 0 và chuyển số 0 hoặc sai thành 1. Kết quả giá trị đúng không rõ ràng sẽ là x.

Ví dụ 4.5

Ví dụ 1 – nếu reg alpha giữ giá trị integer 237 và beta giữ giá trị là 0, thì ví dụ cho phép thực thi như mô tả:

regA=alpha && beta //regA được cài đặt là 0

regB =alpha || beta //regB được cài đặt là 1

Ví dụ 2- Biểu thức cho phép thực thi một toán tử logic và ba biểu thức con mà không cần bất kỳ dấu ngoặc đơn nào

a < size -1 && b != c && index != lastone

Tuy nhiên, nó khuyến khích sử dụng dấu ngoặc đơn vào một mục đích thực tế sẽ làm cho biểu rõ ràng hơn về độ ưu tiên, như cách viết trong ví dụ dưới đây.

(a < size -1) && (b != c) && (index != lastone)

Ví dụ 3 – Thông thường sử dụng toán tử ! trong một cấu trúc như sau:

if(!inword)

Trong một vài trường hợp, cấu trúc trên làm cho người đọc chương trình khó hiểu hơn cấu trúc: if (inword ==0)

4.2.10 Toán tử thao tác trên bit

Toán tử thao tác trên bit sẽ thực thi thao tác trên từng bit của toán hạng, đây là toán tử kết hợp từng bit trên mỗi toán hạng với bit tương ứng trên toán hạng kia để tính toán ra 1 bit kết quả. Các bảng từ 4-12 đến 14-16 sẽ cho thấy kết quả mỗi phép toán có thể trên bit.

Bảng 4.12 Toán tử &

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Bảng 4.13 Toán tử |

|	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Bảng 4.14 Toán tử ^

^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
X	x	x	x	x
Z	x	x	x	x

Bảng 4.15 Toán tử ^~,~^

^~,~^	0	1	x	z
0	1	0	x	x
1	0	1	x	x
X	x	x	x	x
Z	x	x	x	x

Bảng 4.16 Toán tử ~

~	
0	1
1	0
x	x
z	x

Khi các toán hạng không bằng nhau về chiều dài, thì toán hạng ngắn hơn sẽ thêm số 0 vào vị trí bit có ý nghĩa nhất (MSB).

4.2.11 Toán tử giảm

Toán tử giảm một ngôi sẽ thực hiện một toán tử trên bit trên một toán hạng đơn để kết quả là một bit đơn. Để thực hiện toán tử giảm and, or, xor, bước đầu tiên là toán tử sẽ áp dụng toán tử giữa bit đầu tiên với bit thứ hai của toán hạng sử dụng các bảng logic từ 4-17 đến 4-19. Bước thứ hai và

các bước con tuần tự tiếp theo sẽ áp dụng toán tử giữa 1 bit kết quả của bước phía trước với bit tiếp theo của toán hạng sử dụng các bảng logic trên. Với các toán tử giảm nand, nor, xnor, kết quả sẽ tính toán bằng cách đảo kết quả của toán tử giảm and, or, xor tương ứng.

Bảng 4.17 Toán tử giảm &

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Bảng 4.18 Toán tử giảm |

|	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Bảng 4.19 Toán tử giảm ^

^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
X	x	x	x	x
Z	x	x	x	x

Bảng 4.20 cho thấy kết quả của việc áp dụng toán tử giảm trên các toán hạng khác nhau.

Bảng 4.20 Toán tử giảm trên các toán hạng khác nhau.

Toán hạng	&	~&	|	~|`	^	~^	Chú thích
4’b0000	0	1	0	1	0	1	Tất cả các bit là 0
4’b1111	1	0	1	0	0	1	Tất cả các bit là 1
4’b0110	0	1	1	0	0	1	Số chẵn là 1
4’b1000	0	1	1	0	1	0	Số lẽ là 1

4.2.12 Toán tử dịch

Đây là hai loại toán tử dịch, toán tử dịch logic << và >>, và toán tử dịch số học <<< và >>>. Toán tử dịch trái << và <<< sẽ dịch toán hạng bên trái của chúng sang trái một số vị trí bit được đưa ra trong toán hạng bên phải. Trong cả hai trường hợp, bit ở vị trí trống sẽ được điền vào bằng số 0. Toán tử dịch phải, >> và >>>, sẽ dịc toán hạng bên trái của chúng sang phải một số vị trí bit được đưa ra trong toán hạng bên phải. Trong toán tử dịch phải logic sẽ điền vào vị trí bit trống là số 0. Trong toán tử dịch phải toán học sẽ điền vào vị trí bit trống số 0 nếu kết quả là loại không dấu và nó sẽ điền vào vị trí bit trống giá trị bit có ý nghĩa nhất của toán hạng bên trái nếu kết quả là loại có dấu. Nết toán hạng bên phải có dạng giá trị x hoặc z, thì kết quả sẽ không xác định (x). Toán hạng bên phải luôn luôn được xem như là một số không dấu và không có ảnh hưởng đến dấu của kết quả. Dấu của kết quả được xác định bằng toán hạng bên trái và số dư của biểu thức như mô tả trong 4.6.1.

Ví dụ 4.6

Ví dụ 1- Trong ví dụ này, thanh ghi result được gán giá trị nhị phân 0100, do dịch giá trị nhị phân 0001 sang trái hai vị trí và điền số 0 vào vị trí trống

```
module shift;
    reg [3:0] start, result;
    initial begin
        start = 1;
        result = (start << 2);
    end
endmodule
```

Ví dụ 2- Trong ví dụ này, thanh ghi result được gán giá trị nhị phân 1110, đó là do dịch giá trị nhị phân 1000 sang phải hai vị trí và điền bit dấu vào vị trí trống.

```
module ashift;
    reg signed [3:0] start, result;
    initial begin
        start = 4'b1000;
        result = (start >>> 2);
    end
endmodule
```

4.2.13 Toán tử điều kiện

Toán tử điều kiện, còn gọi là toán tử tam phân, sẽ được quyền liên kết và xây dựng sử dụng ba toán hạng ngăn cách bởi hai toán tử trong dạng được đưa ra trong

Cú pháp 4-1

Cú pháp 4-1

```
conditional_expression ::=
expression1 ? { attribute_instance } expression2 : expression3
expression1 ::=
expression
expression2 ::=
expression
expression3 ::=
expression
```

Việc đánh giá toán tử điều kiện sẽ bắt đầu với việc so sánh bằng giá trị logic của biểu thức 1 với số 0, số hạng điều kiện. Nếu điều kiện đánh giá là sai (0), thì biểu thức 3 sẽ được tính toán và sử dụng kết quả cho kết quả của biểu thức điều kiện. Nếu điều kiện đánh giá là đúng (1), thì biểu thức 2 sẽ được tính toán và sử dụng kết quả cho kết quả của biểu thức điều kiện. Nếu điều kiện đánh giá là giá trị không xác định (x hoặc z), thì cả hai biểu thức 2 và biểu thức 3 sẽ được tính toán, và kết quả sẽ được kết hợp, bit tới bit sử dụng Bảng 4.21 để tính toán kết quả cuối cùng nếu biểu thức 2 và biểu thức 3 không phải là số thực, trong trường hợp này kết quả là 0. Nếu kích thước của biểu thức 2 và biểu thức 3 khác nhau, toán hạng ngắn hơn sẽ tăng chiều dài cho bằng toán hạng dài và thêm số 0 vào bên trái (thứ tự cao hơn).

Bảng 4.21 Toán tử điều kiện

?:	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x

z	x	x	x	x

Ví dụ 4.7

Theo ví dụ này sẽ có 3 trạng thái bus đầu ra minh hoạ việc sử dụng toán tử điều kiện thông thường.

wire *[15:0]busa=drive_busa?data:16'bz;*

Bus data sẽ được lái vào busa khi bit drive_busa là 1. Nếu bit drive_busa không xác định, thì một giá trị không xác định sẽ được lái vào busa, nói cách khác busa không xác định.

4.2.14 Toán tử ghép nối

Toán tử ghép nối là kết quả của việc nối các bit kết quả từ một hay nhiều biểu thức lại với nhau. Toán tử ghép nối sử dụng ký hiệu ngoặc nhọn ({}) và dùng dấu phảy (,) để ngăn cách các biểu thức.

Một số hằng số không xác định kích thước sẽ không được phép sử dụng trong toán tử ghép nối. Đó là vì kích thước của mỗi toán hạng trong toán tử ghép nối cần phải tính toán cho phù hợp với kích thước của kết quả toán tử ghép nối.

Ví dụ 4.8

Ví dụ này sẽ ghép nối bốn biểu thức:

{a, b[3:0], w, 3'b101}

Nó được ước lượng để cho phép trong ví dụ:

{a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1}

Toán tử này có thể ứng dụng chỉ kết nối như là một toán tử nhân bản, với biểu thức trong kết nối phía trước là một biểu thức không âm, không z, và không x thì toán tử được gọi là toán tử nhân bản hằng, các dấu ngoặc

nhỏ được lồng vào nhau và nó biểu thị một kết nối nhiều giá trị nhân bản với nhau trong phép toán kết nối. Không giống như toán tử kết nối, biểu thức bao gồm toán tử lặp không được nằm bên trái phép gán và không có kết nối với cổng output hoặc inout.

Đây là ví dụ cho toán tử nhân bản bốn lần giá trị w

{4{w}} // tập các giá trị giống nhau {w, w, w, w}

Bên dưới là một ví dụ cho toán tử nhân bản hợp lệ

{1'bx{1'b0}}

Ví dụ tiếp theo minh họa toán tử nhân bản lồng vào toán tử kết nối

{b, {3{a, b}}} // tập các giá trị {b, a, b, a, b, a, b}

Toán tử nhân bản có thể có hằng số nhân bản là số 0. Điều này được dùng làm tham biến cho chương trình. Một phép nhân bản với hằng số nhân bản là số 0 được tính là có 1 số 0 hoặc bỏ qua. Toán tử nhân bản sẽ tiếp cận chỉ với kết nối với toán hạng có kết nối với kích thước dương.

Ví dụ 4.9

```
parameter P = 32; // Hợp lệ cho tất cả P từ 1 tới 32
assign b[31:0] = { {32-P{1'b1}}, a[P-1:0] } ;
// Không hợp lệ cho  P=32 bởi vì số 0 nhân bản xuất hiện một mình
trong toán tử kết nôi.
assign c[31:0] = { {{32-P{1'b1}}}, a[P-1:0] }
// Không hợp lệ cho P=32
initial
  $displayb({32-P{1'b1}}, a[P-1:0]);
```

Khi một biểu thức nhân bản được tính toán, toán hạng sẽ tính toán một cách chính xác thậm chí nếu toán tử nhân bản là số 0. Ví dụ:

Result = {4{func(w)}}

Sẽ tính toán như là :

Y = func(w)

Result = {y, y, y, y}

4.3 Toán hạng

Có một số loại toán hạng có thể được chỉ rõ trong các biểu thức. Loại đơn giản là một tham chiếu đến một net, biến, hoặc tham số ở dạng hoàn chỉnh của nó, đó chỉ là tên của net, biến, hoặc tham số được đưa ra. Trong trường hợp này, tất cả các bit tạo thành giá trị của net, biến hoặc tham số được sử dụng như toán hạng.

Nếu một bit duy nhất của một biến vector ***net***, vector ***reg***, ***integer***, hoặc ***time***, hoặc tham số được yêu cầu, thì bit được gọi là toán hạng ***bit-select***. Toán hạng ***part-select*** sẽ được sử dụng để tham chiếu tới một nhóm các bit gần nhau trong biến vector ***net***, vector ***reg***, ***integer*** hoặc ***time*** hoặc tham số.

Một mảng các yếu tố hoặc ***bit-select*** hoặc ***part-select*** hoặc bất kỳ mảng các phần tử có thể được tham chiếu như là một toán hạng. Một toán tử kết nối của một toán hạng khác (bao gồm cả kết nối lồng nhau) có thể được xem như là một toán hạng. Một hàm cũng là một toán hạng.

4.3.1 Vector bit-select và part-select addressing

Bit-select trích ra một bit riêng biệt từ biến vector ***net***, vector ***reg***, ***integer***, hoặc ***time***, hoặc ***parameter***. Các bit có thể được định địa chỉ bằng một biểu thức. Nếu một bit-select nằm ngoài giới hạng hoặc bit-select là x hoặc z, thì giá trị trả về được tham sẽ là x. Một bit-select hoặc part-select của một giá trị vô hướng, hoặc của một biến hoặc tham số thuộc loại real hoặc realtime sẽ không hợp lệ.

Một số bit liền kề nhau trong một biến vector net, vector reg, integer, hoặc time, hoặc tham số có thể định địa chỉ và được gọi là một part-select. Có hai loại part-select, part-select hằng số và part-select chỉ số. Part-select hằng số của một vector net hoặc reg được đưa ra theo cú pháp bên dưới:

Vect [msb_expr: lsb_expr]

Cả msb_expr và lsb_expr sẽ là biểu thức số nguyên không đổi. Biểu thức đầu có địa chỉ có ý nghĩa hơn biểu thức thứ hai

Part-select chỉ số của một biến vector net, vector reg, integer hoặc time, hoặc tham số được đưa ra theo cú pháp bên dưới:

reg [15:0] big_vect;

reg [0:15] little_vect;

big_vect[lsb_base_expr +: width_expr]

little_vect[msb_base_expr +: width_expr]

big_vect[msb_base_expr -: width_expr]

little_vect[lsb_base_expr -: width_expr]

Trong đó msb_base_expr và lsb_base_expr là hai biểu thức số nguyên, và width_expr là một biểu thức số nguyên dương không đổi. Lsb_base_expr và msb_base_expr có thể thay đổi trong thời gian chạy. Trong hai ví dụ đầu bit được chọn bắt đầu từ base và tăng dần phạm vi bit. Hai ví dụ thứ hai bit được chọn bằng đầu từ vị trí base và giảm dần phạm vi bit.

Một part-select của bất kỳ loại nào có phạm vi địa chỉ nằm ngoài vùng địa chỉ của ***net***, biến ***reg, integer, time*** hoặc ***parameter*** hoặc ***part-select*** mà có giá trị x hoặc z thì chúng sẽ có giá trị x khi đọc và sẽ không ảnh hưởng đến dữ liệu lưu trữ khi ghi. Part-select nằm ngoài phạm vi cục bộ này sẽ trả về giá trị x cho các bit nằm ngoài phạm vi khi đọc và chỉ ảnh hưởng đến các bit trong phạm vi khi ghi.

Ví dụ 4.10

reg *[31: 0] big_vect;*

reg *[0 :31] little_vect;*

reg *[63: 0] dword;*

integer *sel;*

big_vect[0 +: 8] // == big_vect[7 : 0]

big_vect[15 -: 8] // == big_vect[15 : 8]

little_vect[0 +: 8] // == little_vect[0 : 7]

little_vect[15 -: 8] // == little_vect[8 :15]

*dword[8*sel +: 8] // biến part-select với độ rộng cố định*

4.3.2 Địa chỉ mảng và phần tử nhớ

Việc khai báo mảng và bộ nhớ (mảng thanh ghi một chiều) đã được thảo luận ở phần 3.7. Trong phần này sẽ đi vào vấn đề định địa chỉ mảng.

Ví dụ 4.11

Trong ví dụ tiếp theo sẽ khai báo một bộ nhớ 1024 từ 8 bit:

reg *[7:0] men_name[0:1023];*

Cú pháp cho địa chỉ bộ nhớ sẽ bao gồm tên vùng nhớ và biểu thức địa chỉ, theo định dạng sau:

men_name[addr_expr];

Trong đó addr_expr là một biểu thức nguyên bất kỳ; vì vậy một bộ nhớ gián tiếp có thể chỉ ra như là một biểu thức đơn.Ví dụ tiếp theo minh họa cho bộ nhớ gián tiếp:

men_name[men_name[3]];

Trong Ví dụ 4.11, từ nhớ địa chỉ men_name[3] sẽ dùng làm biểu thức cho việc truy cập bộ nhớ tại địa chỉ men_name[men_name[3]]. Cũng giống như toán tử bit-select, địa chỉ trong vùng khai báo bộ nhớ mới là biểu thức địa chỉ có ảnh hương. Nếu chỉ số nằm bên ngoài vùng giớn hạn địa chỉ bộ nhớ hoặc nếu bất kỳ bit nào trong địa chỉ là z hoặc x thì giá trị tham chiếu sẽ là x.

Ví dụ 4.12

Ví dụ tiếp theo khai báo một mảng hai chiều [256:256] phần tử 8 bit và một mảng ba chiều [256:256:8] các phần tử một bit:

reg *[7:0] twod_array[0:255][0:255];*

wire *threed_ array[0:255][0:255][0:7];*

Cú pháp sau truy xuất đến mảng bao gồm tên của bộ nhớ hoặc mảng và biểu thức số nguyên cho mỗi chiều của mảng:

twod_array[addr_expr][addr_expr]

threed_array[addr_expr][addr_expr][addr_expr]

Như các ví dụ trước, addr_expr là một biểu thức số nguyên bất kỳ. Trong mảng hai chiều twod_array truy cập đến vector 8 bit, trong khi mảng ba chiều threed_array truy xuất đến các bit đơn trong mảng ba chiều.

Để biểu diễn bit-select hoặc part-seclect của phần tử mảng, từ mong muốn sẽ được chọn đầu tiên bằng cách cung cấp địa chỉ cho mỗi chiều. Một lựa chọn bit-select và part-select sẽ định địa chỉ giống như là bit-select và part-select ở net và reg.

Ví dụ 4.13

```
twod_array[14][1][3:0]      // Truy xuất 4 bit thấp của từ
twod_array[1][3][6]         // Truy xuất bit thứ 6 của từ
twod_array[1][3][sel]       // Sử dụng biến bit-select
threed_array[14][1][3:0]    // Không hợp lệ
```

4.3.3 Chuỗi

Toán hạng chuỗi sẽ được xem như hằng số bao gồm một chuỗi tuần tự các ký tự ASCII 8 bit. Bất kỳ toán tử Verilog HDL có thể thao tác trên toán hạng chuỗi. Toán tử sẽ xem bên trong chuỗi như là một giá trị số riêng lẽ.

Khi một biến lớn hơn yêu cầu để giữ giá trị cho việc gán, nội dung sau khi gán sẽ được bổ sung vào bên trái với số 0. Điều này thù hợp với việc bổ sung xảy ra trong phép gán ở những giá trị không phải chuỗi.

Ví dụ 4.14

Theo ví dụ này, ta khai báo một biến chuỗi có độ lớn đủ để chứa 14 ký tự và gán cho nó một giá trị. Ví dụ sẽ thao tác trên chuỗi sử dụng toán tử ghép nối.

```
module string_test;
reg [8*14:1]stringvar;
initial begin
        stringvar="Helloworld";
        $display("%s is stored as %h", stringvar, stringvar);
        stringvar={stringvar,"!!!"};
        $display("%s is stored as %h", stringvar, stringvar);
end
endmodule
```

Kết quả mô phỏng cho đoạn chương trình trên:

Hello world is stored as 00000048656c6c6f20776f726c64

Hello world!!! is stored as 48656c6c6f20776f726c64212121

4.3.3.1 Toán tử chuỗi

Các toán tử chuỗi thường dùng sao chép, nối chuỗi, so sánh được hỗ trợ bởi toán tử Verilog HDL. Toán tử sao chép cung cấp bằng một phép gán. Toán tử nối chuỗi cung cấp bằng toán tử ghép nối. Toán tử so sánh được cung cấp bằng toán tử so sánh bằng.

Khi thao tác trên các giá trị trong các vector reg, các reg phải có ít nhất 8*n bit (với n là số ký tự ASCII) trong thứ tự phù hợp với mã 8 bit ASCII.

4.3.3.2 Giá trị chuỗi đệm và vấn đề tiềm ẩn

Khi chuỗi được gán cho một biến, giá trị lưu trữ sẽ thêm vào bên trái giá trị số 0. Sự thêm vào này có thể ảnh hưởng đến kết quả của toán tử so sánh và toán tử nối chuỗi. Toán tử so sánh và nối chuỗi sẽ không phân biệt được giữa số 0 do kết quả của việc thêm vào và số 0 trong chuỗi chính thức (\0, ASCII NUL).

Ví dụ 4.15 sẽ minh họa vấn đề xảy ra:

Ví dụ 4.15

```
reg [8*10:1]s1,s2;
initial begin
        s1="Hello";
        s2="world!";
        if ({s1,s2}=="Helloworld!")
        $display("stringsareequal");
end
```

Việc so sánh trong Ví dụ 4.15 bị lỗi bởi vì trong quá trình gán vào biến chuỗi giá trị thêm vào các biến s1, s2 được lưu vào như bên dưới:

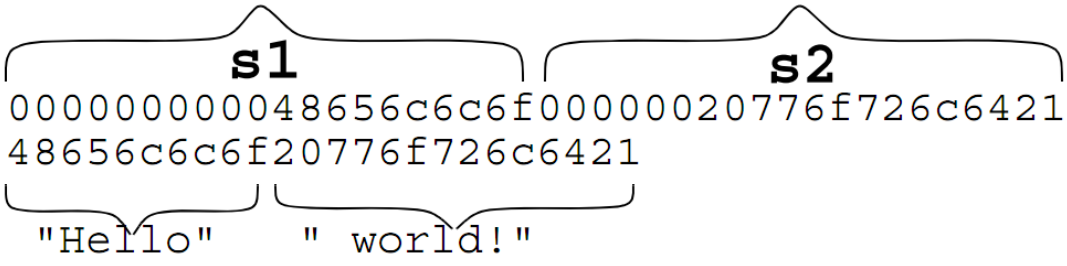
s1=000000000048656c6c6f

s2=00000020776f726c6421

Toán tử ghép nối s1 và s2 bao gồm cả các giá trị số 0 thêm vào, kết quả cho ta giá trị:

000000000048656c6c6f00000020776f726c6421

Bởi vì chuỗi “Hello world!” không bao gồm số 0 thêm vào, phép so sánh bị lỗi theo mô tả sau:



4.3.3.3 Chuỗi rỗng

Chuỗi rông (“ “) sẽ được xem như là giá trị ASCII NUL (“\0”), có giá trị là 0 và nó khác với chuỗi (“0”).

4.4 Biểu thức trì hoãn thời gian tối thiểu, trung bình, và tối đa

Biểu thức trì hoãn trong ngôn ngữ Verilog HDL được mô tả bởi ba biểu thức ngăn cách nhau bởi dấu hai châm (:) và gộp lại với nhau bằng dấu ngoặc đơn (()). Điều đó thể hiện các giá trị đại diện cho thời gian tối thiểu, trung bình và tối đa theo thứ tự. Cú pháp này được đưa ra theo cú Cú pháp 4-2.

Cú pháp 4-2

constant_expression ::=

constant_primary

| unary_operator { attribute_instance } constant_primary

| constant_expression binary_operator { attribute_instance } constant_expression

| constant_expression ? { attribute_instance } constant_expression

constant_expression

constant_mintypmax_expression ::=

constant_expression

| constant_expression : constant_expression : constant_expression

expression ::=

primary

| unary_operator { attribute_instance } primary

| expression binary_operator { attribute_instance } expression

| conditional_expression

mintypmax_expression ::=

expression

| expression : expression : expression

constant_primary ::= (From A.8.4)

number

| parameter_identifier [[constant_range_expression]]

| specparam_identifier [[constant_range_expression]]

|constant_concatenation

| constant_multiple_concatenation

| constant_function_call

| constant_system_function_call

| (constant_mintypmax_expression)

| string

primary ::=

```
number
| hierarchical_identifier [ { [ expression ] } [ range_expression ] ]
| concatenation
| multiple_concatenation
| function_call
| system_function_call
| ( mintypmax_expression )
| string
```

Biểu thức trì hoãn trong ngôn ngữ Verilog HDL thông thường có ba giá trị. Ba giá trị này cho phép thiết kế các chương trình kiểm tra với giá trị trì hõan tối thiểu, trung bình và tối đa.

Các giá trị thể hiện trong định dạng min:typ:max có thể được sử dụng trong các biểu thức. Định dạng min:typ:max có thể sử dụng ở bất kỳ biểu thức nào.

Ví dụ 4.16

Ví dụ 1 - Ví dụ này cho thấy biểu thức định nghĩa bộ ba duy nhất của giá trị trì hoãn. Trong biểu thức:

(a:b:c)+(d:e:f)

Giá trị nhỏ nhất là tổng của a+d, giá trị trung bình là tổng của b+e, giá trị lớn nhất là tổng của c+f.

Ví dụ 2 – ví dụ tiếp theo thể hiện một biểu thức điển hình sử dụng giá trị theo định dạng min:typ:max.

val-(32'd50:32'd75:32'd100)

4.5 Biểu thức độ dài bit

Kiểm soát số lượng bit được sử dụng trong việc tính toán các biểu thức là rất quan trọng nếu phù hợp với kết quả đạt được. Một vài tình huấn có giải pháp đơn giản; ví dụ, nếu một bit và toán tử được quy định trên hai thanh ghi 16 bit, thì kết quả sẽ là một giá trị 16 bit. Tuy nhiên, trong một vài tình huấn, nó không rõ ràn là có bao nhiêu bit được sử dụng trong việc tính toán biểu thức hoặc kích cỡ của kết quả là bao nhiêu.

Ví dụ, để thực hiện tính toán phép cộng số học của hai thanh ghi 16 bit cần sử dụng 16 bit, hoặc cần sử dụng 17 bit để có thể chứa cả bit tràn? Câu trả lời phụ thuộc vào loại thiết bị được mô hình và cách mà thiết bị điều khiển nhớ bit tràn. Verilog HDL sử dụng độ dài bit của toán hạng để phát hiện có bao nhiêu bit được sử dụng trong quá trình tính toán biểu thức. Các luật về độ dài bit được đưa ra ở 4.5.1. Trong trường hợp toán tử cộng, độ dài bit của toán hạng lớn hơn sẽ được sử dụng cho biến bên trái phép gán.

Ví dụ 4.17

```
reg [15:0] a, b; // thanh ghi 16 bit
reg [15:0] sumA; // thanh ghi 16 bit
reg [16:0] sumB; // thanh ghi 17 bit
sumA = a + b; // biểu thức tính toán sử dụng 16 bit
sumB = a + b; // biểu thức tính toán sử dụng 17 bit
```

4.5.1 Qui luật cho biểu thức độ dài bit

Các luận quản lý biểu thức độ dài bit được trình bài rõ rành để các tình huấn thực tế có một giải pháp tự nhiên.

Số lượng bit của một biểu thức (còn gọi là kích cỡ của biểu thức) sẽ được xác định bằng toán hạng được gọi trong biểu thức và nội dung của biểu thức đưa ra.

Một biểu thức tự xác định là biểu thức mà độ dài bit của nó được xác định duy nhất bởi tự biểu thức đó, ví dụ, biểu thức thể hiện giá trị trì hoãn.

Một biểu thức xác định toàn bộ là biểu thức mà độ dài bit của nó được xác định bằng độ dài bit của biểu thức đó và một phần của biểu thức có liên quan khác. Ví dụ, kích cỡ bit của biểu thức bên phải của phép gán phụ thuộc vào tự nó và kích cỡ của biểu thức bên trái.

Bảng 4.22 thể hiện cách các biểu thức thông thường xác định độ dài bit của kết quả biểu thức. Trong Bảng 4.22, i, j và k là các toán hạng của biểu thức, và L(i) thể hiện độ dài bit của toán hạng i.

Toán hạng nhân có thể thực hiện mà không mất bất kỳ bit tràn nào bằng cách gán kết quả đủ rộng để chứa nó.

Bảng 4.22 Biểu thức xác định độ dài bit của kết quả biểu thức

Biểu thức	Độ dài bit	Chú thích
Hằng số không xác định kích thước	Bằng độ dài bit của số integer	
Hằng số có kích thước xác định	Kích thước của hằng số	
i op j, với op là: + - * / % & | ^ ^~ ~^	max (L(i),L(j))	
op j, với op là : + - ~	L(i)	
i op j, với op là: === !== == != > >= < <=	1bit	Toán hạng có kích thước là:max (L(i),L(j))
i op j, với op là: && ||	1bit	Tất cả các toán hạng tự

		xác định
op j, với op là: & ~& | ~| ^ ~^ ^~ !	1bit	Tất cả các toán hạng tự xác định
i op j, với op là: >> << ** >>> <<<	L(i)	j tự xác định
i ? j : k	max(L(j),L(k)	i tự xác định
{i,...,j}	L(i)+..+L(j)	Tất cả các toán hạng tự xác định
{i{j,..,k}}	i * (L(j)+..+L(k))	Tất cả các toán hạng tự xác định

4.5.2 Ví dụ minh họa vấn đề về biểu thức độ dài bit

Trong suốt quá trình tính toán một biểu thức, kết quả tạm thời sẽ lấy kích cỡ của toán hạng lớn hơn (trong trường hợp toán tử gán, nó còn bao gồm cả bên trái phép gán). Sự thận trọng sẽ ngăn chặn việc mất bit dấu trong quá trình tính toán. Ví dụ bên dưới mô tả cách độ dài bit của toán tử có thể làm cho kết quả mất bit dấu.

Ví dụ 4.18 Khai báo các thanh ghi

reg *[15:0] a, b, answer; // thanh ghi 16 bit*

Mục đích để tính toán biểu thức

Answer = (a + b) >> 1l; // sẽ không thể thực thi

Ở đây a và b cộng với nhau, nên kết quả có thể xảy ra tràn, và sau đó dịch phải 1 bit để duy trì bit nhớ trong thanh ghi kết quả 16 bit.

Vấn đề xảy ra, tuy thế nhưng bởi vì tất cả các toán hạng trong biêut thức là có chiều rộng 16 bit. Vì vậy, biểu thức (a + b) cho ra một kết quả

tạm thời là một giá trị 16 bit, vì vậy bit nhớ bị mất trước khi thực thi việc tính toán toán tử dịch phải một bit.

Giải pháp là ép buộc biểu thức (a + b) thực hiện tính toán sử dụng ít nhất 17bit. Ví dụ thêm vào toán tử cộng một số integer có giá trị 0, biểu thức sẽ tính toán đúng vì nó thực thi sử dụng kích cỡ bit của integer. Theo ví dụ bên dưới thì sẽ tạo ra kết quả đúng với mục đích

Answer = (a + b + 0) >> 1; // sẽ thực thi đúng

Trong ví dụ tiếp theo:

Ví dụ 4.19

```
module bitlength();
  reg [3:0] a,b,c;
  reg [4:0] d;
  initial begin
        a = 9;
        b = 8;
        c = 1;
        $display("answer = %b", c ? (a&b) : d);
   end
  endmodule
```

Câu lệnh $display sẽ hiển thị

Answer = 01000

Bằng cách tự nó, biểu thức a&b có chiều dài là 4 bit, nhưng bởi vì trong nội dung của biểu thức điều kiện, nó sẽ sử dụng độ dài bit lớn nhất, vì vậy nên biểu thức a&b sẽ có độ dài là 5, là độ dài của d.

.

4.5.3 Ví dụ minh họa về biểu thức tự xác định

Ví dụ 4.20

```
reg [3:0] a;
reg [5:0] b;
reg [15:0] c;
initial begin
        a = 4'hF;
        b = 6'hA;
        $display("a*b=%h", a*b);  // Kích thước của biểu thức tự
        xác    định
        c = {a**b};    // biểu thức  a**b là tự xác định
                       // trong toán tử kết nối {}
        $display("a**b=%h", c);
        c = a**b;           // Kích thước của biểu thức xác định bởi c
        $display("c=%h", c);
end
```

Kết quả mô phỏng của ví dụ này:

a*b=16 // 'h96 bị cắt bỏ còn 'h16 vì kích thước của biểu thức là 6

a**b=1 // kích thước của biểu thức   4 bit (kích thước của a)

c=ac61 // kích thước của biểu thức 16 bit (kích thước của c)

4.6 Biểu thức có dấu

Điều khiển dấu của một biểu là rất quan trọng để tạo ra một kết quả phù hợp. Thêm vào đó để tuân theo các luật trong 4.6.1 tới 4.6.4, hai chức năng hệ thống sẽ sử dụng để điều khiển theo các loại khuôn khổ trong biểu thức: **$signed()** và **$usnigned().** Các hàm này sẽ tính toán các biểu thức đầu vào và trả về một giá trị có cùng kích cỡ và giá trị của biểu thức đầu vào và được định nghĩa bởi các hàm:

$signed – trả về một giá trị có dấu

$unsigned – trả về một giá trị không dấu

Ví dụ 4.21

```
reg [7:0] regA, regB;
reg signed [7:0] regS;
regA = $unsigned(-4);     // regA = 8'b11111100
regB = $unsigned(-4'sd4); // regB = 8'b00001100
regS = $signed  (4'b1100); // regS = -4
```

4.6.1 Qui định cho những loại biểu thức

Các luật cho việc xác định kết quả loại cho một biểu thức:

- Loại biểu thức chỉ phụ thuộc vào toán hạng. Nó không phụ thuộc vào vế bên trái.
- Số thập phân là có dấu.
- Số cơ số là không có dấu, ngoại trừ trường hợp có thêm ký hiệu s được sử dụng trong cơ số chỉ định (như là “4’sd12”).
- Kết quả của bit-select là không dấu, bất chấp toán hạng.
- Kết quả của part-select là không dấu, bất chấp toán hạng thậm chí nếu part-select chỉ định toàn bộ vector.

```
reg [15:0] a;
reg signed [7:0] b;
initial
        a = b[7:0]; // b[7:0] là không dấu
```

- Kếu quả của toán tử kết nối là không dấu, bất chấp toán hạng.
- Kết quả của toán tử so sánh (1,0) là không dấu, bất chấp toán hạng.
- Chuyển đổi từ số thực sang số nguyên bằng loại cưỡng bức là có đấu.

- Dấu và kích cỡ của toán hạng tự xác định được xác định bởi tự toán hạng và độc lập với yêu cầu của biểu thức
- Đối với toán hạng không tự xác định, áp dụng các luật sau:
  - ❖ Nếu bất kỳ toán hạng nào là số thực, kết quả là số thực.
  - ❖ Nếu bất kỳ toán hạng nào là không dấu, kết quả là không dấu, bất chấp toán tử.
  - ❖ Nếu tất cả toán hạng nào là có dấu, kết quả là có dấu, bất chấp toán tử, ngoại trừ trường hợp được chỉ rõ theo cách khác.

4.6.2 Những bước định giá một biểu thức

Các bước để tính toán một biểu thức:

- Xác định kích cỡ của biểu thức dựa trên các chuẩn về luật xác định kích cỡ của biểu thức bên trên.
- Xác định dấu của biểu thức sử dụng các luật ở phần 4.6
- Truyền lại các loại và kích thước của biểu thức(hoặc tự xác định biểu thức con_ trở xuống toán hạng xác định theo ngữ cảnh của biểu thức. Nói chung, toán hạng xác định theo ngữ cảnh của một toán tử sẽ giống loại và kích thước của kết quả toán tử. Tuy nhiên, có 2 ngoại lệ:
  - ❖ Nếu kết quả của toán tử là số thực và nếu nó có toán hạng xác định theo ngữ cảnh mà không phải số thực thì toán hạng sẽ đối sử như thể nếu nó là tự xác định thì nó sẽ chuyển đổi sang số thực trước khi toán tử được áp dụng
  - ❖ Toán tử quan hệ và toán tử bằng có toán hạng mà không hoàn toàn là tự xác định hoặc không hoàn toàn là xác định theo ngữ cảnh. Toán hạng sẽ ảnh hưởng lẫn nhau như là nếu chúng là toán hạng xác định theo ngữ cảnh với loại và kích

thước của kết quả (kích thước lớn nhất của 2 toán hạng) xác định theo chúng. Tuy nhiên, loại kết quả thực sự sẽ luôn là 1 bit không dấu. Loại và kích thước của toán hạng sẽ độc lập với phần còn lại của biểu thức và ngược lại.

- Khi truyền đạt tới một toán hạng đơn giản như được định nghĩa ở 5.2 thì toán hạng đó sẽ chuyển đổi truyền đạt loại và kích thước. Nếu một toán hạng được mở rộng thì nó sẽ chỉ mở rộng dấu nếu loại truyền đạt là có dấu.

4.6.3 Những bước định giá một phép gán

Các bước để tính toán một phép gán:

- Xác định kích thước của phần bên phải bằng chuẩn về luật xác định kích thước của phép gán.
- Nếu cần, mở rộng kích thước của toán tử bên phải, thực hiện mở rộng bit dấu nếu chỉ nếu phần bên phải của toán tử là có dấu.

4.6.4 Tính toán những biểu thức của hai số có dấu X và Z

Nếu một toán hạng có dấu bị thay đổi kích thước tới một kích thước có dấu lớn hơn và giá trị của bit dấu là x, thì giá trị của kết quả sẽ điền thêm Xs. Nếu bit dấu là có giá trị là z, thì giá trị của kết quả sẽ điền thêm Zs. Nếu giá trị dấu của bất kỳ bit nào là x hoặc z, thì toán tử không hợp logic bất kỳ sẽ được gọi giá trị của kết quả sẽ là z và loại phù hợp với loại của biểu thức

4.7 Những phép gán và phép rút gọn

Nếu chiều rộng của biểu thức bên phải lớn hơn chiều rộng của biểu thức bên trái trong phép gán, thì MSBs của biểu thức bên phải sẽ luôn luôn bị loại bỏ để phù hợp với kích thước của biểu thức bên trái. Quá trình thực hiện không yêu cầu cảnh báo hoặc báo cáo bất kỳ lỗi nào liên quan đến kích thước của phép gán không phù hợp hoặc bị cắt ngắn. Cắt ngắn bit dấu của biểu thức có dấu sẽ thay đổi dấu của kết quả.

Ví dụ 4.22

```
Ví dụ 1:
reg      [5:0] a;
reg signed [4:0] b;
initial begin
  a = 8'hff; // sau khi gán, a = 6'h3f
  b = 8'hff; // sau khi gán, b = 5'h1f
end
Ví dụ 2:
reg      [0:5] a;
reg signed [0:4] b, c;
initial begin
  a = 8'sh8f; // sau khi gán, a = 6'h0f
  b = 8'sh8f; // sau khi gán, b = 5'h0f
  c = -113;   // sau khi gán, c = 15
  // 1000_1111 = (-'h71 = -113) bị cắt ngắn còn ('h0F = 15)
end
```

Ví dụ 3:

```
reg      [7:0] a;
reg signed [7:0] b;
reg signed [5:0] c, d;
initial begin
  a = 8'hff;
  c = a;      // sau khi gán, c = 6'h3f
  b = -113;
  d = b;      // sau khi gán, d = 6'h0f
  end
```

4.8 Bài tập

1. Nêu các toán tử thường dùng và độ ưu tiên của chúng?
2. Nêu thứ tự tính toán trong một biểu thức logic?
3. Cách sử dụng số nguyên trong biểu thức?
4. Có bao nhiêu loại toán hạng trong Verilog? Mô tả cụ thể từng loại?
5. Các luật để xác định dấu cho kết quả của một biểu thức?
6. Các bước định giá trị của một biểu thức?
7. Cho a, b, c, d, e được khai báo như sau:

   reg [7:0] a, b; reg [8:0]c; reg [15:0] d;

   Định giá trị của các biểu thức sau:

   - a = 255; b = 255; c = a + b;

- c = 9'b0 + a + b;
- d={a,b};
- c = &b;

Chương 5. Cấu trúc phân cấp và module

5.1 Cấu trúc phân cấp

Ngôn ngữ mô tả phần cứng Verilog hỗ trợ cấu trúc phân cấp bằng cách cho phép modules được nhúng trong modules khác. Modules cấp độ cao hơn tạo thể hiện của module ở cấp độ thấp và giao tiếp với chúng thông qua các đầu vào, đầu ra và đầu vào ra 2 chiều. Các cổng vào ra có thể là vô hướng hoặc là vector.

Cấu trúc phân cấp giúp người thiết kế chia một hệ thống thiết kế ra thành các module nhỏ hơn để dễ thiết kế và kiểm soát luồng dữ liệu trong quá trình thiết kế.

Như một ví dụ cho hệ thống module phân cấp, hãy xem xét một hệ thống bao gồm các bảng mạch in (PCBs).

5.2 Module

5.2.1 Khai báo module

Trong mục này cung cấp cú pháp thông thường cho một định nghĩa module và cú pháp cho việc cài đặt module, cùng với một ví dụ về định nghĩa module và cài đặt module.

Một định nghĩa module được bao giữa bởi hai từ khóa module và endmodule. Các định danh kèm theo sau từ khóa module sẽ là tên định nghĩa của module. Danh sách các tùy chọn của tham số được định nghĩa sẽ chỉ rõ một danh sách theo thứ tự các tham số của module. Danh sách các tùy chọn của cổng hoặc khai báo cổng được định nghĩa sẽ chỉ rõ một danh sách theo thứ tự các cổng của module. Thứ tự được sử dụng trong định nghĩa danh sách các tham số trong *module-parameter-port-list* và trong

danh sách cổng có thể có ý nghĩa trong việc cài đặt các module. Các định danh trong danh sách này sẽ khai báo lại trong các câu lệnh input, output, và inout trong định nghĩa module. Khai báo cổng trong danh sách khai báo cổng sẽ không khai báo lại trong thân module. Các mục của module định nghĩa cái tạo thành module, và chúng bao gồm nhiều loại khai báo và định nghĩa khác nhau, nhiều trong số đó đã được giới thiệu.

Từ khóa ***macromodule*** có thể dùng để thay thế từ khóa ***module*** để định nghĩa một module. Một quá trình thực thi có thể chọn để giải quyết module được định nghĩa bắt đầu với thừ khóa macromodule khác nhau.

Cú pháp 5-1

```
module_declaration ::=
        {attribute_instance} module_keyword module_identifier [
        module_parameter_port_list ]
        list_of_ports ; { module_item }
endmodule
        |{ attribute_instance } module_keyword module_identifier [
        module_parameter_port_list ]
        [ list_of_port_declarations ] ; { non_port_module_item }
endmodule
module_keyword ::= module | macromodule
module_parameter_port_list ::= (From A.1.3
        # ( parameter_declaration { , parameter_declaration } )
list_of_ports ::= ( port { , port } )
list_of_port_declarations ::= ( port_declaration { , port_declaration } ) | ( )
port ::= [ port_expression ] | . port_identifier ( [ port_expression ] )
port_expression ::= port_reference | { port_reference { , port_reference } }
port_reference ::= port_identifier [ [ constant_range_expression ] ]
port_declaration ::= {attribute_instance} inout_declaration
        | {attribute_instance} input_declaration
        | {attribute_instance} output_declaration
module_item ::=  (From A.1.4)
          port_declaration ;
        | non_port_module_item
module_or_generate_item ::=
          { attribute_instance } module_or_generate_item_declaration
        | { attribute_instance } local_parameter_declaration ;
```

```
        | { attribute_instance } parameter_override
        | { attribute_instance } continuous_assign
        | { attribute_instance } gate_instantiation
        | { attribute_instance } udp_instantiation
        | { attribute_instance } module_instantiation
        | { attribute_instance } initial_construct
        | { attribute_instance } always_construct
        | { attribute_instance } loop_generate_construct
        | { attribute_instance } conditional_generate_construct
module_or_generate_item_declaration ::=
          net_declaration
        | reg_declaration
        | integer_declaration
        | real_declaration
        | time_declaration
        | realtime_declaration
        | event_declaration
        | genvar_declaration
        | task_declaration
        | function_declaration
non_port_module_item ::=
          module_or_generate_item
        |generate_region
        | specify_block
        | { attribute_instance } parameter_declaration ;
        | { attribute_instance } specparam_declaration
parameter_override ::= defparam list_of_defparam_assignments ;
```

Ví dụ 5.1 Định dạng của một module chuẩn

module tên_module (danh sách các cổng, nếu có);
Khai báo port input, output, inout; Khai báo tham số
Khai báo các loại dữ liệu (dữ liệu net, dữ liệu biến, ví dụ: wire, reg, integer)
Gọi và gán đặc tính (instantiate) module con (sub-module)
Phát biểu gán sử dụng mô hình RTL (assign)
Phát biểu gán qui trình (always, initial)

Khai báo hàm và tác vụ
Khai báo kết thúc module (endmodule)

5.2.2 Module mức cao nhất

Module mức cao nhất (top-module) là module mà nó bao gồm trong văn bản gốc, nhưng nó hầu như không có một câu lệnh cài đặt nào trong bất kỳ một module nào khác. Điều này áp dụng cả khi module cài đặt tạo ra trong khối tạo mà không phải tự nó cài đặt. Một mô hình phải có ít nhất một module mức cao nhất.

5.2.3 Gọi và gán đặc tính một module (instantiate)

Việc gọi và gán đặc tính module cho phép một module gọi và gán đặc tính một module khác ra để sử dụng. Các module không được định nghĩa lồng nhau. Nói cách khác, một module được định nghĩa sẽ không chứa mô tả thiết kế của một module khác trong cặp từ khóa module endmodule. Một module được định nghĩa lồng trong một module khác bằng cách gọi và gán đặc tính của module đó ra để sử dụng. Một câu lệnh gọi và gán đặc tính module sẽ tạo ra một hoặc nhiều bản sao của module được định nghĩa.

Ví dụ, một module bộ đếm phải cài đặt module D flip-flop để tạo ra nhiều thể hiện của flip-flop.

Cú pháp 5-2 đưa ra cú pháp chi tiếc cho việc gọi và gán đặc tính module.

Cú pháp 5-2

```
module_instantiation ::= (From A.4.1)
        module_identifier [ parameter_value_assignment ]
                module_instance { , module_instance } ;
parameter_value_assignment ::=
        # ( list_of_parameter_assignments )
```

```
list_of_parameter_assignments ::=
        ordered_parameter_assignment { , ordered_parameter_assignment }
        | named_parameter_assignment   { , named_parameter_assignment }
ordered_parameter_assignment ::=
        expression
named_parameter_assignment ::=
        . parameter_identifier ( [ mintypmax_expression ] )
module_instance ::=
        name_of_module_instance ( [ list_of_port_connections ] )
name_of_module_instance ::=
        module_instance_identifier [ range ]
list_of_port_connections ::=
        ordered_port_connection { , ordered_port_connection }
        | named_port_connection { , named_port_connection }
ordered_port_connection ::=
        { attribute_instance } [ expression ]
named_port_connection ::=
        { attribute_instance } . port_identifier ( [ expression ] )
```

Việc gọi và gán đặc tính module có thể chứa một loạt các đặc điểm kỹ thuật. Nó cho phép một mảng các thể hiện được tạo ra. Cú pháp và ngữ nghĩa của các mảng thể hiện định nghĩa cho các cổng và các cổng cơ bản áp dụng tốt cho các module.

Một hoặc nhiều thể hiện của module (bản sao nguyên bản của module) có thể đưa ra trong một câu lệnh gọi và gán đặc tính module riêng lẻ.

Danh sách các cổng kết nối sẽ cung cấp chỉ cho module được định nghĩa với cổng. Các dấu ngoặc đơn luôn luôn cần thiết. Khi một danh sách các cổng kết nối được đưa ra để sử dụng theo thức tự phương thức các cổng kết nối, phần tử đầu tiên trong danh sách sẽ kết nối với cổng đầu tiên trong khai báo cổng trong module, phần tử thứ 2 kết nối với cổng thứ 2 và cứ như thế. Phần 5.2.4.9 sẽ thảo luận rõ hơn về các luật kết nối cổng với cổng.

Một kết nối có thể tham khảo đơn giản tới một biến hoặc một định danh net, một biểu thức, hoặc một khoản trống. Một biểu thức có thể sử

dụng để cung cấp một giá trị tới một cổng vào của module. Một cổng kết nối trống sẽ trình bày tình huống nơi mà cổng đó không kết nối

Khi kết nối một cổng bằng tên, một cổng chưa được kết nối sẽ chỉ ra bằng cách bỏ nó ra trong danh sách hoặc không cung cấp biểu thức bên trong đấu ngoặc (ví dụ portname()).

Ví dụ 5.2

Ví dụ 1: Ví dụ này minh họa một mạch (module cấp độ thấp) được điều khiển bởi một dạng sóng đơn giản (module cấp độ cao hơn) nơi mà mạch được cài đặt bên trong module dạng sóng:

```
//module cấp độ thấp: module mô tả một mạch flip-flop nand
module ffnand (q, qbar, preset, clear);
output q, qbar;//khai báo 2 net đầu ra cho mạch
input preset, clear;// khai báo 2 net đầu vào cho mạch
// khai báo cổng nand 2 đầu vào và các kết nối với chúng
nand  g1 (q, qbar, preset),
      g2 (qbar, q, clear);
endmodule
// module cấp độ cao:
// dạng sóng mô tả cho flip-flop nand
module ffnand_wave;
      wire out1, out2;//đầu ra từ mạch
      reg in1, in2;//biến để điều khiển mạch
      parameter d = 10;
      // thể hiện của mạch ffnand, tên là "ff",
      // và đặc tả đầu ra của các kết nối IO bên trong
      ffnand ff(out1, out2, in1, in2);
      // định nghĩ dạng sóng để mô phỏng mạch
```

```
initial begin
    #d in1 = 0; in2 = 1;
    #d in1 = 1;
    #d in2 = 0;
    #d in2 = 1;
end
endmodule
```

Ví dụ 2: Ví dụ này tạo ra 2 thể hiện của module flip-flop ffnand được định nghĩa trong ví dụ 1. Nó kết nối chỉ với đầu ra q vào một thể hiện và chỉ một đầu ra qbar vào một thể hiện khác.

```
// dạng sóng mô tả để kiểm tra
// nand flip-flop, không có cổng đầu ra
module ffnand_wave;
    reg in1,in2;//biến để điều khiển mạch
    parameter d=10;
    // tạo hai bản sao của mạch ff nand
    // ff1 có qbar không kết nối, ff2 có q không kết nối
    ffnand ff1(out1,,in1,in2),
            ff2(.qbar(out2), .clear(in2), .preset(in1), .q());
           // ff3(.q(out3),.clear(in1),,,); is illegal
    // định nghĩ dạng sóng để mô phỏng mạch
initial begin
    #din1=0;in2=1;
    #din1=1;
    #din2=0;
    #din2=1;
end
```

endmodule

5.2.4 Khai báo port

Cổng cung cấp một phương tiện kết nối các mô tả phần cứng bao gồm module và các phần cứng nguyên thủy. Ví dụ, module A có thể khởi tạo module B, sử dụng các cổng kết nối phù hợp tới module A. Tên các cổng này có thể khác với tên của các dây nối nội và các biến được chỉ ra trong định nghĩa module B

5.2.4.1 Định nghĩa port

Cú pháp cho các cổng và danh sách cổng được đưa ra trong Cú pháp 5-3.

Cú pháp 5-3

```
list_of_ports ::= (From A.1.3)
        ( port { , port } )
list_of_port_declarations ::=
        ( port_declaration { , port_declaration } )
        | ( )
port ::=
        [ port_expression ]
        | . port_identifier ( [ port_expression ] )
port_expression ::=
        port_reference
        | { port_reference { , port_reference } }
port_reference ::=
        port_identifier [ [ constant_range_expression ] ]
port_declaration ::=
        {attribute_instance} inout_declaration
        | {attribute_instance} input_declaration
        | {attribute_instance} output_declaration
```

5.2.4.2 Liệt kê port

Cổng tham khảo cho mỗi cổng trong danh sách các cổng ở bên trên của mỗi khai báo module có thể là 1 trong số:

- Một định danh đơn giản hoặc định danh bị bỏ qua.
- Một bit-select của một vector khai báo trong module
- Một part- select của một vector khai báo trong module
- Một toán tử kết nối của bất kỳ phần nào trong 3 phần trên.

Biểu thức cổng là tùy chọn bởi vì cổng có thể được định nghĩa mà không cần bất kỳ kết nối nào trong module. Khi một cổng đã được định nghĩa, thì không có cổng nào khác được định nghĩa cùng tên.

Có hai loại cổng module, loại đầu tiên chỉ là cổng biểu thức, là loại cổng ngầm. Loại thứ hai là loại cổng trực tiếp. Điều rõ ràng chi tiết cổng định danh sử dụng kết nối với cổng của module thể hiện bằng tên và cổng biểu thức bao gồm khai báo các định danh bên trong module như miêu tả trong phần 5.2.4.3. Tên cổng kết nối sẽ không sử dụng cho cổng ngầm định nếu cổng biểu thức không là một định danh đơn giản hoặc là định danh bị bỏ qua, mà sẽ sử dụng tên cổng.

5.2.4.3 Khai báo port

Mỗi cổng định danh trong một cổng biểu thức trong danh sách của các cổng trong khai báo module cũng sẽ khai báo trong thân của module như một trong các khai báo: input, output hoặc inout (cổng hai chiều).Ở đó có thể thêm vào khai báo các loại dữ liệu khác cho các cổng đặt thù – ví dụ reg hoặc wire.Cú pháp cho việc khai báo cổng đưa ra trong

Cú pháp 5-4:

Cú pháp 5-4

```
inout_declaration ::=
        inout [ net_type ] [ signed ] [ range ] list_of_port_identifiers
input_declaration ::=
        input [ net_type ] [ signed ] [ range ] list_of_port_identifiers
output_declaration ::=
        output [ net_type ] [ signed ] [ range ]
                list_of_port_identifiers
        | output reg [ signed ] [ range ]
                list_of_variable_port_identifiers
        | output output_variable_type
                list_of_variable_port_identifiers
list_of_port_identifiers ::= (From A.2.3)
        port_identifier { , port_identifier }
```

Nếu khai báo một cổng bao gồm một net hoặc loại biến khác, thì cổng có thể khai báo lại trong khai báo net hoặc biến. Nếu một net hoặc biến khai báo như là một vector, thì đặc tả phạm vi giữa hai khai báo công phải giống hệt nhau.

5.2.4.4 Liệt kê khai báo port

Một cú pháp thay thế để giảm tới ít nhất việc sao chép dữ liệu có thể sử dụng để chỉ ra các cổng trong module. Mỗi module sẽ khai báo bên không những trong cú pháp khai báo danh sách các cổng trong phần 5.2.4.4 mà sử dụng cả danh sách các port được mô tả trong phần này.

Mỗi khai báo cổng cung cấp thông tin đầy đủ về cổng, hướng cổng, độ rộng, net, hoặc các loại biến và những mô tả đầy đủ khác về port như có dấu hoặc không dấu. Cú pháp tương tự cho khai báo dầu vào, đầu vào ra và đầu ra cũng được sử dụng ở phần đầu của module theo cách khai báo cổng, ngoài ra danh sách khai báo port là bao gồm phần đầu của module chứ không phải tách biệt (ngay sau dấu ; ở cuối phần đầu module)

Ví dụ 5.3

Trong ví dụ sau, module tên Test được đưa ra trong ví dụ trước được khai báo lại như sau:

```
module test (
        input [7:0] a,
        input signed [7:0] b, c, d,//nhiều cổng cùng chia sẻ một thuộc tính khai báo
        output [7:0] e,// mối thuộc tính phải có 1 khai báo.
        output reg signed [7:0] f, g,
        output signed   [7:0] h) ;
        // Không hợp lệ nếu có bất kỳ khai báo cổng nào trong phần thân module
endmodule
```

Các loại cổng tham chiếu của khai báo cổng module sẽ không hoàn thành sử dụng cách thức danh sách khai báo cổng của khai báo module. Cũng như khai báo cổng sử dụng trong danh sách khai báo cổng sẽ chỉ định danh đơn giản hoặc định danh trống. Chúng sẽ không có bit-select, part-select hoặc toán tử kết nối (như trong ví dụ ***complex_ports***) hoặc không có các cổng phân chia (trong ví dụ ***split_ports***), hoặc không có tên cổng (như trong ví dụ ***same_port***).

Thiết kế tự do có thể sử dụng lẫn lộn các cú pháp trong khai báo module, vì vậy việc mô tả thực thi trường hợp đặt biệt bên trên có thể thực hiện sử dụng cú pháp danh sách cổng.

5.2.4.5 Kết nối các port của module được gọi bằng danh sách theo thứ tự

Một phương pháp làm cho các kết nối giữa các biểu thức cổng được liệt kê trong thể hiện của module và cổng khai báo bên trong thể hiện module là theo thức tự danh sách. Nghĩa là biểu thức cổng liệt kê trong thể hiện của module sẽ kết nối tới cùng vị trí cổng trong danh sách cổng khi khai báo module con.

Ví dụ 5.4 minh họa một module ở mức độ cao nhất (topmod) cài đặt mo đun thứ 2 (mobB). Module mobB có cổng kết nối theo thứ tự danh sách. Kết nối thực hiện như là:

- Cổng wa trong modB định nghĩa kết nối tới bit-select v[0] trong module topmod.
- Cổng wb kết nối tới v[3].
- Cổng c kết nối tới w.
- Cổng d kết nối tới v[4].

Trong định nghĩa mobB, cổng wa và wb được khai báo là cổng vào ra trong khi cổng c và d được khai báo là cổng vào.

Ví dụ 5.4

```
module topmod;
    wire [4:0] v;
    wire a,b,c,w;
    modB b1 (v[0], v[3], w, v[4]);
endmodule
module modB (wa, wb, c, d);
    inout wa, wb;
    input c, d;
    tranif1 g1 (wa, wb, cinvert);
    not #(2, 6)n1 (cinvert, int);
    and #(6, 5)g2 (int, c, d);
  endmodule
```

Trong suốt quá trình mô phỏng của thể hiện b1 của ***modB***, cổng ***And*** g2 hoạt động đầu tiên để cung cấp một giá trị int. Giá trị ba trạng thái qua cổng not n1 cung cấp đầu ra ***cinvert***, sau đó cho hoạt động cổng tranif g1.

5.2.4.6 Kết nối các port của module được gọi bằng tên

Cách thứ 2 để kết nối các cổng của module bao gồm liên kết rõ ràng hai tên của mỗi bên trong kết nối: khai báo tên cổng từ khai báo module tơi biểu thức, ví dụ tên sử dụng trong khai báo module, theo sau bằng tên sử dụng trong thể hiện của module. Tên ghép này sau đó được đặt trong danh sách kết nối của module. Tên cổng sẽ là tên chỉ ra trong khai báo module. Tên cổng không thể là ***bit-select, part-select*** hoặc toán tử kết nối của các cổng. Nếu khai báo cổng của module là ngầm định, biểu thức cổng phải là biểu thức đơn giản hoặc là biểu thức trống, mà sẽ được sử dụng như tên cổng. Nếu khai báo cổng của module là rõ ràng, tên rõ ràng sẽ được sử dụng như tên cổng.

Biểu thức cổng có thể là một biểu thức hợp lệ bất kỳ.

Biểu thức cổng là tùy chọn vì vậy trong cài đặt module có thể báo cáo sự tồn tại của cổng mà không kết nối với bất kỳ cái gì. Các dấu ngoặc đơn vẫn phải yêu cầu có.

Hai loại kết nối cổng của module không được lẫn lộn, kết nối tới cổng đặt thù của thể hiện module sẽ hoặc tất cả theo thứ tự hoặc tất cả theo tên.

Ví dụ 5.5

Ví dụ 1: Trong ví dụ này, cài đặt module kết nối tới tín hiệu ***topA*** và ***topB*** tới cổng ***In1*** và ***Out*** định nghĩa trong module ***ALPHA***. Có một cổng cung cấp bởi module ***ALPHA*** không được sử dụng, tên là ***In2***.Có thể có các cổng không được sử dụng được đề cập trong cài đặt này.

ALPHA instance1 (.Out(topB),.In1(topA),.In2());

Ví dụ 2:

Ví dụ này định nghĩa module ***modB*** và ***topmod***, và sau đó ***topmod*** cài đặt ***modB*** sử dụng kết nối cổng theo tên.

```
module topmod;
    wire [4:0] v;
    wire a,b,c,w;
    modB b1 (.wb(v[3]),.wa(v[0]),.d(v[4]),.c(w));
endmodule
module modB(wa, wb, c, d);
    inout wa, wb;
    input c, d;
    tranif1 g1(wa, wb, cinvert);
    not #(6, 2)n1(cinvert, int);
    and #(5, 6)g2(int, c, d);
```

endmodule

Bởi vì kết nối là theo tên nên thứ tự của các cổng khai báo có thể đảo vị trí.

Nhiều kết nối cổng của thể hiện module là không cho phép, ví dụ bên dưới là không hợp lệ

Ví dụ 3: ví dụ cho thấy kết nối cổng không hợp lệ

```
module test;
  a ia (.i (a), .i (b), // không hợp lệ khi kết nối đầu ra 2 lần.
        .o (c), .o (d), // không hợp lệ khi kết nối đầu vào 2 lần.
        .e (e), .e (f)); // không hợp lệ khi kết nối đầu vào ra 2 lần.
endmodule
```

5.2.4.7 Số thực trong kết nối port

Loại dữ liệu số thực không kết nối trực tiếp với cổng. Nó sẽ kết nối gián tiếp, như ví dụ bên dưới. Hàm hệ thống ***$realtobits*** và ***$bitstoreal*** sẽ được sử dụng để kết nối qua các bit trên mô hình cổng của module.

Ví dụ 5.6

```
module driver (net_r);
        output net_r;
        real r;
        wire [64:1] net_r = $realtobits(r);
endmodule
module receiver (net_r);
        input net_r;
        wire [64:1] net_r;
        real r;
```

```
        initial assign r = $bitstoreal(net_r);
    endmodule
```

5.2.4.8 Kết nối những port không tương tự nhau

Một cổng của một module có thể được xem như là cung cấp một liên kết hoặc một kết nối giữa hai biểu tượng (ví dụ dây nối, thanh ghi, biểu thức …) một cài đặt bên trong module và một cài đặt bên ngoài module.

Kiểm tra kết nối cổng theo luật mô tả trong phần 5.2.4.9 sẽ thấy rằng, biểu tượng nhận giá trị từ một cổng (biểu tượng input của module nội và output của module ngoại) sẽ có cấu trúc biểu thức net. Các biểu tượng khác cung cấp giá trị có thể là một biểu thức bất kỳ.

Một công được khai báo là một đầu vào (đầu ra) nhưng sử dụng như là đầu ra (đầu vào) hoặc đầu vào ra có thể được cưỡng chế để vào ra. Nếu không cưỡng chế sẽ có cảnh báo xuất hiện.

5.2.4.9 Những qui định khi kết nối port

Các luật kết nối cổng trong phần này sẽ chi phối cách khai báo cổng của module và cách chúng kết nối với nhau.

Luật 1: Một cổng vào hoặc cổng vào ra phải là một net.

Luật 2: Mỗi cổng kết nối sẽ là phép gán liên tục của nguồn tới cuối cùng, nơi mà một biểu tượng kết nối là tín hiệu nguồn và những cái khác là tín hiệu chìm. Một phép gán là phép gán liên tục từ nguồn tới cuối cùng cho đầu vào hoặc đầu ra. Phép gán là không mạnh giảm kết nối bán dẫn cho cổng inout. Chỉ dây nối hoặc biểu thức có cấu trúc dây nối sẽ ẩn trong phép gán.

Một biểu thức cấu trúc dây dẫn là một biểu thức cổng trong đó toán hạng là:

- Một net vô hướng

- Một vector net.
- Một hằng số bit-select của một vector net.
- Một part-select của một vector net.
- Một toán tử kết nối của biểu thức cấu trúc net.

Theo đó các biểu tượng bên ngoài sẽ không kết nối tới đầu ra hoặc đầu vào ra của module:

- Biến.
- Biểu thức khác với những điều sau:
  - Một net vô hướng
  - Một vector net.
  - Một hằng số bit-select của một vector net.
  - Một part-select của một vector net.
  - Một toán tử kết nối của biểu thức trong danh sách trên.

Luật 3:

Nếu net ở hai bên của cổng là loại net uwire, một cảnh báo sẽ xảy ra nế net không gộp lại vào trong một net đơn như mô tả trong phần 5.2.4.10

5.2.4.10 Loại net tạo ra từ việc kết nối port không tương tự nhau

Khi các loại net khác nhau kết nối với nhau thông qua một module, thì các net của tất cả các cổng phải đưa về cho giống loại với nhau. Kết quả loại net được xác định theo bảng 5-1. Trong bảng này, net ngoại nghĩa là net chỉ ra trong thể hiện của module, net nội nghĩa là net chỉ ra trong module định nghĩa. Net mà loại của nó được sử dụng gọi là dominating net. Net mà loại của nó bị thay đổi gọi là dominated net. Nó có quyền hợp các dominating và dominatr net vào trong một net đơn, loại này có loại như là một dominating net. Kết quả của net gọi là simulated net và dominated net gọi là collapsed net.

Loại simulated net sẽ thực hiện delay để chỉ ra dominating net. Nếu dominating net là loại trireg, bất kỳ giá trị độ mạnh nào chỉ ra cho trireg sẽ áp dụng cho simulated net.

Bảng 5.1 Tổ hợp giữa net nội và net ngoại

Net nội	Net ngoại								
wire, tri	ext	ext	ext	ext	ext	ext	ext	ext	ext
wand, triand	int	ext	ext warn	ext warn	ext warn	ext warn	ext warn	ext	ext
wor, trior	int	ext warn	ext	ext warn	ext warn	ext warn	ext warn	ext	ext
trireg	int	ext warn	ext warn	ext	ext	ext	ext warn	ext	ext
tri0	int	ext warn	ext warn	int	ext	ext warn	ext warn	ext	ext
tri1	int	ext warn	ext warn	int	ext warn	ext	ext warn	ext	ext
uwire	int	int warn	int warn	int warn	int warn	int warn	ext	ext	ext
supply0	int	int	int	int	int	int	int	ext	ext warn
supply1	int	int	int	int	int	int	int	ext warn	ext
Từ khóa: ext: sử dụng net ngoại int: sử dụng net nội warn: xuất hiện cảnh báo									

Luật giải quyết loại net:

Khi 2 net kết nối với nhau bởi các cổng khác loại nhau, kết quả là một net đơn có thể là một trong:

- Loại dominating net nếu một trong hai net là dominating net, hoặc
- Loại của net ngoại tới module

Khi loại dominatin net không tồn tại, loại net ngoại sẽ được sử dụng.

Bảng loại net:

Bảng 5.1 chỉ ra loại net bị gọi bởi loại net theo luật giải quyết net.

Simulated net sẽ theo loại net chỉ ra trong bảng và trì hoãn kỹ thuật của net. Nếu simulated net được chọn là trireg, bất kỳ độ mạnh giá trị nào chỉ ra cho trireg sẽ áp dụng cho simulated net.

5.2.4.11 Kết nối những giá trị có dấu thông qua (port)

Thuộc tính dấu không được thông qua trong cấu trúc phân cấp. Trong thứ tự để có loại có dấu qua cấu trúc phân cấp, từ khóa ***signed*** phải được sử dụng trong khai báo đối tượng ở một cấp độ khác trong cấu trúc phân cấp. Bất kỳ biểu thức nào trong một cổng sẽ được xem như là bất kỳ biểu thức nào khác trong phép gán. Nó sẽ có loại, ký cỡ, đánh giá và giá trị kết quả gán tới đối tượng ở bên khác của cổng sử dụng giống luật như một phép gán.

5.3 Bài tập

1. Mô hình cấu trúc phân cấp trong Verilog?
2. Các cách khai báo, gọi và gán đặt tính cho một module?
3. Các cách khai báo port?
4. Các cách kết nối port?

5. Những quy luật khi kết nối port?

Chương 6. Mô hình thiết kế cấu trúc (Structural model)

6.1 Giới thiệu

Mô hình thiết kế cấu trúc mô tả các hệ thống dưới dạng các cổng linh kiện hay các khối linh kiện được kết nối lại với nhau để thực hiện được những chức năng mong muốn. Mô hình thiết kế cấu trúc được mô tả một cách trực quan hệ thống thiết kế số, do đó nó thực sụ gần giống với mô tả vật lí phần cứng của hệ thống.

Người thiết kế thường sử dụng mô hình thiết kế cấu trúc cho những module nhỏ cần tối ưu về timing, diện tích vì sử dụng mô hình này thì phần cứng thiết kế sau khi tổng hợp ra mạch sẽ giống với mô tả thiết kế trên Verilog. Tuy nhiên, đối với một hệ thống lớn thì việc sử dụng mô hình cấu trúc là không khả thi bởi vì sự cồng kềnh của nó khi ghép hàng ngàn hàng vạn cổng cơ bản lại với nhau cũng như tiêu tốn thời gian rất lớn cho việc chạy mô phỏng kiểm tra thiết kế.

6.2 Những linh kiện cơ bản

6.2.1 Cổng and, nand, or, nor, xor, và xnor

Khai báo thể hiện của một cổng logic nhiều đầu vào sẽ bắt đầu với một trong những từ khóa sau:

and nand nor or xor xnor

Đặc tả trì hoãn sẽ là 0, 1 hoặc 2 trì hoãn. Nếu đặc tả trì hoãn bao gồm 2 trì hoãn, trì hoãn đầu sẽ xác định đầu ra trì hoãn ở cạnh lên, trì hoãn thứ hai sẽ xác định đầu ra trì hoãn ở cạnh xuống, và trong thời gian nhỏ hơn 2 trì hoãn sẽ thiết lập đầu ra là x. Nếu chỉ có một trì hoãn được đưa ra thì sẽ

trì hoãn ở cả cạnh lên và cạnh xuống. Nếu không có đặc tả trì hoãn thì sẽ không có trì hoãn thông qua cổng.

Sáu cổng logic này có một đầu ra và một hoặc nhiều đầu vào. Tham số đầu tiên trong danh sách các tham số sẽ kết nối với đầu ra của cổng logic, các tham số khác kết nối tới đầu vào:

Bảng sự thật của các cổng này thể hiện kết quả của cổng 2 giá trị đầu vào:

Bảng 6.1 Bảng sự thật của các cổng logic

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

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	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

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	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

Các phiên bản của sáu cổng logic này có nhiều hơn 2 đầu vào sẽ mở rộng tự nhiên theo bảng trên, nhưng số lượng đầu vào ảnh hưởng tới trì hoãn truyền.

Ví dụ 6.1

Ví dụ này khai báo một cổng and 2 đầu vào:

and a1 (out, in1, in2);

Trong đó đầu vào là *in1, in2*. Đầu ra là *out*, thể hiện tên là *a1*.

6.2.2 Cổng buf và not

Khai báo thể hiện của một cổng logic nhiều đầu vào sẽ bắt đầu với một trong những từ khóa sau:

and nand nor or xor xnor

Đặc tả trì hoãn sẽ là 0, 1 hoặc 2 trì hoãn. Nếu đặc tả trì hoãn bao gồm 2 trì hoãn, trì hoãn đầu sẽ xác định đầu ra trì hoãn ở cạnh lên, trì hoãn thứ hai sẽ xác định đầu ra trì hoãn ở cạnh xuống, và nhỏ hơn 2 trì hoãn sẽ thiết lập đầu ra là x. Nếu chỉ có một trì hoãn được đưa ra thì sẽ trì hoãn ở cả cạnh lên và cạnh xuống. Nếu không có đặc tả trì hoãn thì sẽ không có trì hoãn thông qua cổng.

Hai cổng logic này có một đầu vào và một hoặc nhiều đầu ra. Tham số cuối cùng trong danh sách các tham số sẽ kết nối với đầu vào của cổng logic, các tham số khác kết nối tới đầu ra.

Bảng sự thật của các cổng này thể hiện kết quả của cổng 1 đầu vào và một đầu ra:

Bảng 6.2 Bảng sự thật của cổng buffer và cổng not

buf		not	
Đầu vào	Đầu vào	Đầu vào	Đầu vào
0	0	0	0

1	1	1	1
x	x	x	x
x	x	x	x

Ví dụ 6.2

Ví dụ sau khai báo một cổng buf 2 đầu ra:

buf b1 (out1, out2, in);

Đầu vào là in, đầu ra là out1, out2, tên thể hiện là b1

6.2.3 Cổng bufif1, bufif0, notif1, và notif0

Khai báo thể hiện của một cổng logic ba trạng thái sẽ bắt đâu với một trong các từ khóa sau:

bufif0 bufif1 notif1 notif0

Đây là bốn cổng logic thuộc loại ba trạng thái điều khiển. Bên cạnh các giá trị logic 0 và 1, đầu ra cổng này có thể là giá trị z.

Đặc tả trì hoãn sẽ là 0, 1, 2 hoặc 3 trì hoãn. Nếu đặc tả trì hoãn bao gồm 3 trì hoãn, trì hoãn đầu sẽ xác định đầu ra trì hoãn ở cạnh lên, trì hoãn thứ hai sẽ xác định đầu ra trì hoãn ở cạnh xuống, trì hoãn thứ ba sẽ xác định trì hoãn sự chuyển tiếp tới giá trị z và nhỏ nhất trong 3 trì hoãn sẽ xác định trì hoãn của chuyển tiếp tới x. Nếu đặc tả trì hoãn bao gồm 2 trì hoãn, trì hoãn đầu sẽ xác định đầu ra trì hoãn ở cạnh lên, trì hoãn thứ hai sẽ xác định đầu ra trì hoãn ở cạnh xuống, và nhỏ hơn trong 3 trì hoãn sẽ xác định trì hoãn của chuyển tiếp tới x và z. Nếu chỉ có một trì hoãn được đưa ra thì nó chỉ tới trì hoãn ở tất cả các chuyển tiếp đầu ra. Nếu không có đặc tả trì hoãn thì sẽ không có trì hoãn thông qua cổng.

Một vài tổ hợp của giá trị dữ liệu đầu vào và giá trị điều khiển đầu vào có thể gây ra cổng có hai giá trị đầu ra, mà không có tham khảo nào cho một trong hai giá trị (xem phần 7.10.2). Bảng logic cho các cổng này

bao gồm hai ký hiệu biểu diễn cho kết quả không sát định. Ký hiệu L sẽ chỉ ra một kết quả có giá trị 0 hoặc z. Giá trị H chỉ ra kết quả có giá trị 1 hoặc z. Trì hoãn trên sự chuyển tiếp tới H hoặc L sẽ xem như giống với trì hoãn chuyển tiếp tới giá trị x. Bốn cổng logic này sẽ có một đầu ra và một đầu vào dữ liệu, một đầu vào điều khiển. Tham số thứ nhất trong danh sách tham số kết nối với đầu ra, tham số thứ hai kết nối với đầu vào, tham số thứ ba kết nối với đầu vào điều khiển.

Bảng 6.3 Bảng sự thật của các cổng ba trạng thái

bufif0	CONTROL				
		0	1	x	z
	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

bufif1	CONTROL				
		0	1	x	z
	0	z	0	L	L
	1	z	1	H	H
	x	x	z	x	x
	z	x	z	x	x

notif0	CONTROL				
		0	1	x	z
	0	1	z	L	L
	1	0	z	H	H
	x	x	z	x	x
	z	x	z	x	x

notif1	CONTROL				
		0	1	x	z
	0	z	1	L	L
	1	z	0	H	H
	x	x	z	x	x
	z	x	z	x	x

Ví dụ 6.3

Ví dụ sau khai báo một thể hiện của công ***bufif1***:

bufif1 bf1 (outw, inw, controlw);

Trong đó đầu ra là *outw*, đầu vào là *inw*, đầu vào điều khiển là *controlw*, thể hiện tên là *bf1*

6.2.4 Công tắc MOS

Khai báo thể hiện của một công tắc MOS sẽ bắt đầu với một trong các từ khóa sau:

cmos nmos pmos rcmos rnmos rpmos

Công tắc cmos và rcmos được mô tả trong 7.7

Từ khóa pmos viết tắt cho transistor P-type matal-oxide semiconductor (PMOS) và từ khóa nmos là viết tắt cho transistor N-type matal-oxide semiconductor (NMOS). Transistor PMOS và NMOS có trở kháng tương đối thấp giữa cực nguồn và cực máng khi chúng dẫn. Từ khóa rpmos là viết tắt của transistor điện trở PMOS và từ khóa rnmos là viết tắt của transistor điện trở NMOS. Transistor điện trở PMOS và NMOS có trở kháng cao hơn nhiều giữa cực nguồn và cực máng khi chúng dẫn so với transistor PMOS và NMOS thường. Thiết bị tải trong mạch MOS tĩnh là ví dụ của transistor rpmos và rnmos. Bốn công tác là kênh dẫn một chiều cho dữ liệu tưởng tự như cổng bufif.

Đặc tả trì hoãn sẽ là 0, 1, 2 hoặc 3 trì hoãn. Nếu đặc tả trì hoãn bao gồm 3 trì hoãn, trì hoãn đầu sẽ xác định đầu ra trì hoãn ở cạnh lên, trì hoãn thứ hai sẽ xác định đầu ra trì hoãn ở cạnh xuống, trì hoãn thứ ba sẽ xác định trì hoãn sự chuyển tiếp tới giá trị z và nhỏ nhất trong 3 trì hoãn sẽ xác định trì hoãn của chuyển tiếp tới x. Nếu đặc tả trì hoãn bao gồm 2 trì hoãn, trì hoãn đầu sẽ xác định đầu ra trì hoãn ở cạnh lên, trì hoãn thứ hai sẽ xác định đầu ra trì hoãn ở cạnh xuống, và nhỏ hơn trong 2 trì hoãn sẽ xác định trì hoãn của chuyển tiếp tới x và z. Nếu chỉ có một trì hoãn được đưa ra thì nó chỉ tới trì hoãn ở tất cả các chuyển tiếp đầu ra. Nếu không có đặc tả trì hoãn thì sẽ không có trì hoãn thông qua cổng.

Một vài tổ hợp của giá trị dữ liệu đầu vào và giá trị điều khiển đầu vào có thể gây ra công tắc có hai giá trị đầu ra, mà không có tham khảo

nào cho một trong hai giá trị. Bảng logic cho các cổng này bao gồm hai ký hiệu biểu diễn cho kết quả không sát định. Ký hiệu L sẽ chỉ ra một kết quả có giá trị 0 hoặc z. Giá trị H chỉ ra kết quả có giá trị 1 hoặc z. Trì hoãn trên sự chuyển tiếp tới H hoặc L sẽ xem như giống với trì hoãn chuyển tiếp tới giá trị x.

Bốn cổng logic này sẽ có một đầu ra và một đầu vào dữ liệu, một đầu vào điều khiển. Tham số thứ nhất trong danh sách tham số kết nối với đầu ra, tham số thứ hai kết nối với đầu vào, tham số thứ ba kết nối với đầu vào điều khiển.

Công tắt nmos và pmos sẽ cho qua tín hiệu từ đầu vào và thông tới đầu ra của chúng với một thay đổi về độ mạnh tín hiện tron một trường hợp, thảo luận ở 7.11. Công tắt rnmos và rpmos sẽ giảm độ mạnh tính hiện truyền qua chúng, thảo luận trong phần 7.12.

Bảng 6.4 Thể hiện bảng logic của cổng truyền

pmos / rpmos	CONTROL				
		0	1	x	z
	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	z	z	z	z

cmos / rcmos	CONTROL				
		0	1	x	z
	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	z	z	z

Ví dụ 6.4

Ví dụ này khai báo một công tắc ***pmos***:

pmos p1 (out, data, control);

Trong đó đầu ra là *out*, đầu vào là *data*, đầu điều khiển là *control* và tên thể hiện là *p1*.

6.2.5 Công tắc truyền hai chiều

Khai báo thể hiện của công tắc truyền hai chiều sẽ bắt đầu với một trong các từ khóa sau:

tran ***tranif1*** ***tranif0***

rtran ***rtranif1*** ***rtranif0***

Công thức truyền hai chiều sẽ không trì hoãn tín hiệu truyền qua chúng. Khi thiết bị ***tranif(), tranif1, rtranif0*** hoặc ***rtranif1*** là tắt, chúng sẽ chặn tín hiệu; và khi chúng mở thì chúng sẽ cho tín hiệu đi qua. Thiết bị ***tran*** và ***rtran*** không thể tắt và chúng luôn luôn cho tín hiệu qua chúng.

Đặc tả trì hoãn cho các thiết bị ***tranif1, tranif0, rtranif1***, và ***rtranif0*** là 0, 1 hoặc 2 trì hoãn. Nếu đặc tả trì hoãn bao gồm 2 trì hoãn, trì hoãn đầu sẽ xác định đầu ra trì hoãn mở, trì hoãn thứ hai sẽ xác định đầu ra trì hoãn đóng, và nhỏ hơn trong 2 trì hoãn sẽ xác định trì hoãn của chuyển tiếp tới x và z. Nếu chỉ có một trì hoãn được đưa ra thì nó đặc tả cho cả trì hoãn mở và đống. Nếu không có đặc tả trì hoãn thì sẽ không có trì hoãn đóng và mở cho công tắc truyền hai chiều.

Công tắc truyền hai chiều ***tran*** và ***rtran*** sẽ không chấp nhận đặc tả trì hoãn.

Các thiết bị ***tranif1, tranif0, rtranif1*** và ***rtranif0*** có 3 tham số trong danh sách vào ra. Hai tham số đầu sẽ là hai thiết bị đầu cuối hai chiều điều khiển tín hiệu tới và đi từ thiết bị, và đầu cuối thứ 3 sẽ kết nối với đầu vào điều khiển. Thiết bị ***tran*** và ***rtran*** sẽ có danh sách các cổng gồm hai cổng hai chiều. Cả hai đầu cuối hai chiều sẽ điều khiển truyền vô điều kiện tín hiệu tới và đi từ thiết bị, cho phép tín hiện qua theo mọi hướng từ thiết bị. Thiết bị đầu cuối hai chiều có tất cả 6 thiết bị chỉ kết nối với những net vô hướng hoặc bit-select của những vector net.

Thiết bị ***tran, tranif0*** và ***tranif1***cho qua tín hiệu với thay đổi về độ mạnh chỉ trong trường hợp mô tả phần 6.11. Thiết bị ***rtran, rtranif1, rtranif0*** sẽ làm giảm độ mạnh của tín hiệu qua chúng theo luật thảo luận trong phần 6.12

Ví dụ 6.5

Ví dụ sau mô tả khai báo một thể hiện ***tranif1***:

tranif1 t1 (inout1,inout2,control);

Thiết bị đầu cuối hai chiều là *inout1* và *inout2*, đầu vào điều khiển là *control*, tên thể hiện là *t1*.

6.2.6 Công tắc CMOS

Khai báo thể hiện của một công tắc CMOS bắt đầu với một trong những từ khóa sau:

cmos rcmos

Đặc tả trì hoãn sẽ là 0, 1, 2 hoặc 3 trì hoãn. Nếu đặc tả trì hoãn bao gồm 3 trì hoãn, trì hoãn đầu sẽ xác định đầu ra trì hoãn ở cạnh lên, trì hoãn thứ hai sẽ xác định đầu ra trì hoãn ở cạnh xuống, trì hoãn thứ ba sẽ xác định trì hoãn sự chuyển tiếp tới giá trị z và nhỏ nhất trong 3 trì hoãn sẽ xác định trì hoãn của chuyển tiếp tới x. Trì hoãn trong quá trình chuyển tiếp tới giá trị H hoặc L giống như trì hoãn trong chuyển tiếp tới x. Nếu đặc tả trì hoãn bao gồm 2 trì hoãn, trì hoãn đầu sẽ xác định đầu ra trì hoãn ở cạnh lên, trì hoãn thứ hai sẽ xác định đầu ra trì hoãn ở cạnh xuống, và nhỏ hơn trong 2 trì hoãn sẽ xác định trì hoãn của chuyển tiếp tới x và z. Nếu chỉ có một trì hoãn được đưa ra thì nó chỉ tới trì hoãn ở tất cả các chuyển tiếp đầu ra. Nếu không có đặc tả trì hoãn thì sẽ không có trì hoãn thông qua cổng

Công tắc ***cmos*** và ***rcmos*** có một đầu vào dữ liệu, một đầu ra dữ liệu và hai đầu vào điều khiển. Trong danh sách các cổng vào ra, cổng vào ra

đầu tiên kết nối tới đầu ra dữ liệu, cổng vào ra thứ hai kết nối tới đầu vào dữ liệu, cổng vào ra thứ 2 kết nối tới kênh n đầu vào điều khiển, là cổng vào ra cuối cùng kết nối tới kênh p của đầu vào điều khiển.

Cổng ***cmos*** sẽ cho qua tín hiệu với thay đổi độ mạnh chỉ trong một trường hợp, mô tả 6.2.11. Cổng ***rcmos*** làm giảm độ mạnh của tín hiệu qua nó theo luật mô tả 6.2.12.

Công tắc ***cmos*** xem như là tổ hợp của công tắc ***pmos*** và công tắc ***nmos***. Công tắc ***rcmos*** xem như là tổ hợp của công tắc ***rpmos*** và công tắc ***rnmos***. Công tắc tổ hợp trong cấu hình này sẽ chia sẻ dữ liệu đầu vào và đầu ra trên cổng vào ra nhưng chúng phân biệt nhau về đầu vào điều khiển.

Ví dụ 6.6

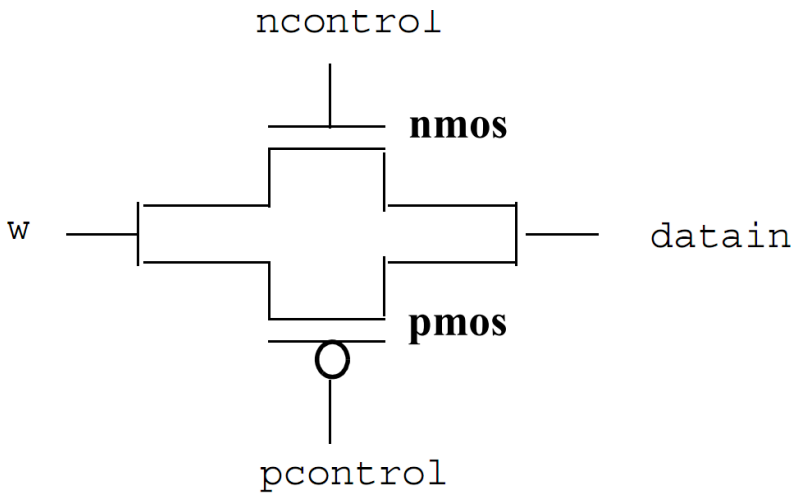
Sự tương đương một cổng cmos ghép đôi từ một cổng ***cmos*** và một cổng ***pmos*** được đưa ra trong ví dụ sau:

cmos (w, datain, ncontrol, pcontrol);

tương đương với

nmos (w, datain, ncontrol);

pmos (w, datain, pcontrol);



Hình 6.1 Cổng truyền CMOS

6.2.7 Nguồn pullup và pulldown

Khai báo thể hiện của một nguồn pullup và pulldown bắt đầu bằng một trong các từ khóa:

pullup ***pulldown***

Nguồn ***pullup*** đặt giá trị logic 1 lên net kết nối tới danh sách các cổng. Nguồn ***pulldown*** đặt giá trị 0 lên net kết nối với danh sách các cổng.

Tín hiện mà nguồn đặt lên những net có độ mạnh pull trong sự thiếu đặc tả độ mạnh. Nếu có một đặc tả độ mạnh 1 trên nguồn pullup hoặc độ mạnh 0 trên nguồn pulldown, tín hiệu sẽ có đặc tả độ mạnh. Đặc tả độ mạnh 0 trên nguồn ***pullup*** và đặc tả độ mạnh 1 trên nguồn ***pulldown***.

Không có đặc tả trì hoãn cho nguồn

Ví dụ 6.7

Ví dụ khai báo hai thể hiện nguồn ***pullup***:

pullup (strong1) p1 (neta), p2 (netb);

Trong ví dụ này, thể hiện *p1* điều khiển neta và thể hiện *p2* điều khiển *netb* với độ mạnh *strong*

6.2.8 Mô hình độ mạnh logic

Ngôn ngữ Verilog cung cấp cho mô hình chính xác của tín hiệu tranh chấp, cổng truyền hai chiều, thiết bị MOS điện trở, MOS động, chia sẻ thay đổi, và những cấu hình mạng khác phụ thuộc kỹ thuật bằng cách cho phép tín hiệu net vô hướng có giá trị không đầy đủ và có nhiều cấp độ độ mạnh khác nhau hoặc tổ hợp các cấp độ độ mạnh. Mô hình logic nhiều cấp độ độ mạnh giải quyết tổ hợp tín hiệu trong các giá trị biết và không biết để biểu diễn cho hành vi của phần cứng thực hiện chính xác hơn.

Chi tiết về độ mạnh sẽ có hai yếu tố:

- Độ mạnh phần 0 của giá trị net, gọi là ***strength0***, thiết kế như một trong các từ khóa sau:

  supply0 strong0 pull0 weak0 highz0

- Độ mạnh phần 1 của giá trị net, gọi là ***strength1***, thiết kế như một trong các từ khóa sau:

  supply1 strong1 pull1 weak1 highz1

Tổ hợp (highz0, highz1) và (highz1, highz0) là không hợp lệ

Mặt dù phần này đặc tả độ mạnh, nó hữu ích để xem xét độ mạnh như một thuộc tính nắm giữ các vùng của một chuỗi liên tục trong thứ tự để dự đoán kết quả của tín hiệu tổ hợp.

Bảng 6.5 chứng minh sự liên tục của độ mạnh. Cột bên trái là danh sách các từ khóa sử dụng trong đặc tả độ mạnh. Cột bên phải là mức độ độ mạnh tương quan.

Bảng 6.5 Độ mạnh của net

Tên độ mạnh	Cấp độ độ mạnh
supply0	7
strong0	6
pull0	5
large0	4
weak0	3
medium0	2
small0	1
highz0	0
highz1	0
small1	1
medium1	2
weak1	3

large1	4
pull1	5
strong1	6
supply1	7

Trong Bảng 6.5 có bốn độ mạnh điều khiển

supply strong pull weak

Tín hiệu với độ mạnh điều khiển sẽ truyền từ đầu ra cổng và đầu ra của phép gán liên tục.

Trong Bảng 6.5 có ba độ mạnh thay đổi do lưu trữ

large medium small

Tín hiệu với độ mạng thay đổi do lưu trữ sẽ hình thành trong loại net ***trireg***.

Có thể nghĩ rằng độ mạnh của tín hiệu trong Bảng 6.5 như vị trí trên thước tỷ lệ trong Hình 6.2.

strength0								strength1							
7 Su0	6 St0	5 Pu0	4 La0	3 We0	2 Me0	1 Sm0	0 HiZ0	0 HiZ1	1 Sm1	2 Me1	3 We1	4 La1	5 Pu1	6 St1	7 Su1

Hình 6.2 Độ mạnh các mức logic “0” và “1”

Thảo luận về tổ hợp tín hiệu sau phần này dùng hình vẽ tương tự như Hình 6.2

Nếu giá trị tín hiệu của một net là biết, tất cả độ mạnh của nó sẽ nằm trong một trong hai phần: phần ***strength0*** của thước tỷ lệ trên Hình 6.2 hoặc phần stregth1. Nếu giá trị của một net là không biết, nó sẽ có cấp độ độ mạnh cảu trong phần ***strength1*** và ***strength0***. Một net với giá trị z có cấp độ độ mạnh trong một phần con của thước tỷ lệ.

6.2.9 Độ mạnh và giá trị của những tín hiệu kết hợp

Thêm vào trong giá trị của tín hiệu, một net có thể hoặc là có độ mạnh nhiều cấp độ không rõ ràng hoặc là độ mạnh rõ ràng bao gồm một hay nhiều cấp độ. Khi tổ hợp tín hiệu, mỗi độ mạnh và giá trị sẽ xác định độ mạnh và giá trị của tín hiệu kết quả tuân theo các khái niệm trong phần 6.2.9.1tới 6.2.9.4.

6.2.9.1 Sự kết hợp giữa những tín hiệu có độ mạnh rõ ràng

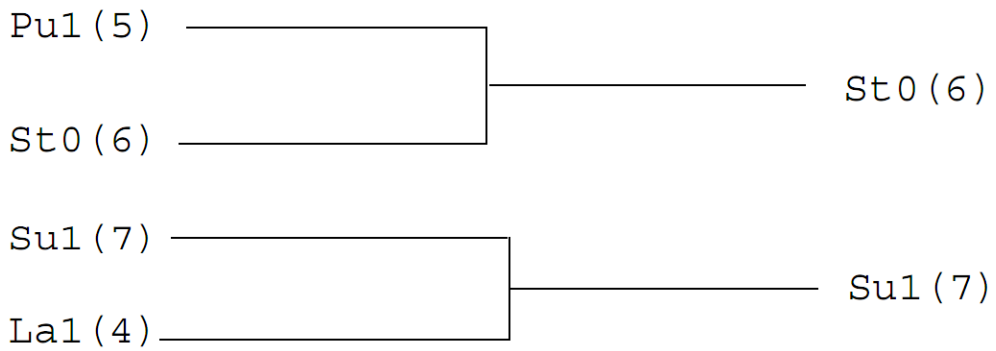
Phần này giải quyết với các tín hiệu tổ hợp mà mỗi tín hiệu có một giá trị rõ ràng và có một mức độ độ mạnh.

Nếu hai hoặc nhiều hơn các tín hiệu có độ mạnh không bằng nhau tổ hợp trong một cấu hình dây dẫn net, độ mạnh của tín hiệu sẽ là độ mạnh trội hơn trong tất cả và xác định kết quả. Tổ hợp của hai hoặc nhiều hơn các tín tín hiệu có giá trị giống nhau sẽ có kết quả giống với giá trị có độ mạnh lớn nhất trong tất cả. Tổ hợp tín hiệu đồng nhất độ mạnh và giá trị sẽ cho kết quả giống với tín hiệu đó.

Tổ hợp của các tín hiệu có giá trị không giống nhau và có độ mạnh giống nhau sẽ có thể có một trong ba kết quả. Hai kết quả xảy ra trên dây dẫn logic, kết quả thứ ba xảy ra trong trường hợp trống. Dây dẫn logic sẽ thảo luận trong 6.2.9.4. Kết quả trong sự vắng mặt của dây dẫn logic được thảo luận trong chủ đề của Hình 6.4.

Ví dụ:

Trong Hình 6.3, những con số trong dấu ngoặc đơn chỉ ra quan hệ về độ mạnh của tín hiệu. Tổ hợp của một pull1 và một strong0 kết quả là một strong0, vì nó mạnh hơn trong 2 tín hiệu.



Hình 6.3 Kết quả của việc điều khiển net bởi hai độ mạch khác nhau

6.2.9.2 Độ mạnh không rõ ràng: nguồn và sự kết hợp

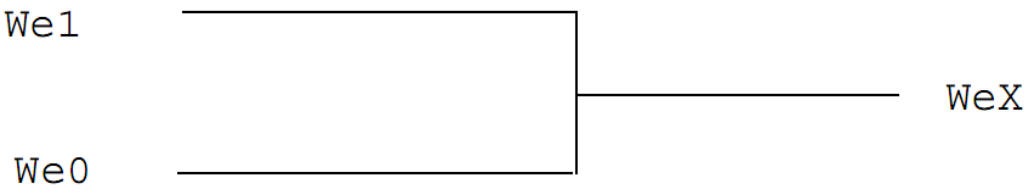
Một số phân loại tín hiệu sử lý độ mạnh không rõ ràng:

- Tín hiệu với giá trị xác định và nhiều cấp độ độ mạnh.
- Tính hiệu với một giá trị x, trong đó các cấp độ độ mạnh bao gồm cả hai phần strength1 và strength0 của thước đô độ mạnh trong Hình 6.2.
- Tín hiệu với một giá trị L, trong đó các cấp độ độ mạnh bao gồm trở kháng cao gia nhập với cấp độ độ mạnh trong phần strength0 trong thước đo độ mạnh trong Hình 6.2.
- T ín hiệu với một giá trị H, trong đó các cấp độ độ mạnh bao gồm trở kháng cao gia nhập với cấp độ độ mạnh trong phần strength1 trong thước đo độ mạnh trong Hình 6.2.

Nhiều cấu hình có thể tạo ra các tín hiệu với độ mạnh không rõ ràng. Khi hai tín hiệu bằng nhau về độ mạnh và ngược nhau về giá trị kết hợp, kết quả sẽ là giá trị x, cùng với cấp độ độ mạnh của cả hai tín hiệu và các cấp độ độ mạnh nhỏ hơn.

Ví dụ:

Trong Hình 6.4 chỉ ra một tổ hợp của tín hiệu weak với giá trị 1 và tín hiệu weak với giá trị 0 cho ra một tín hiệu có độ mạnh weak và có giá trị là x.



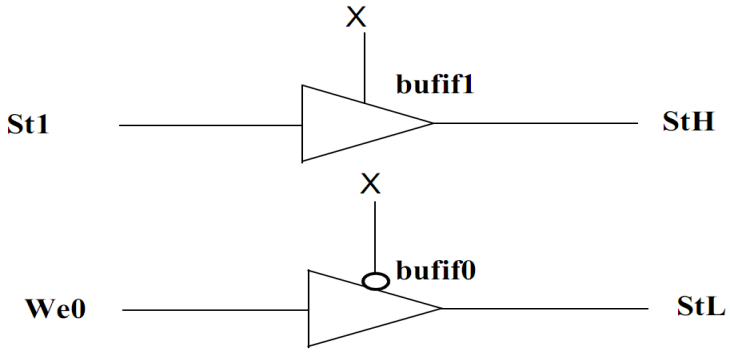
Hình 6.4 Hai tín hiệu có độ mạnh bằng nhau cùng điều khiển một net

Đầu ra của tín hiệu (WeX) được mô tả trong Hình 6.5:

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Hình 6.5 Đầu ra của tín có độ mạnh nằm trong một trong các giá trị thuộc dãy trên

Một tín hiệu có độ mạnh không rõ ràng có thể có giá trị là một dãy các giá trị có thể. Ví dụ độ mạnh của đầu ra từ một cổng điều khiển ba trạng thái với đầu vào điều khiển là không xác định như Hình 6.6:



Hình 6.6 Cổng ba trạng thái với tín hiệu điều khiển không xác định

Đầu ra của bufif1 trong Hình 6.6là một strong H, dãy giá trị đầu ra được mô tả trong Hình 6.7.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

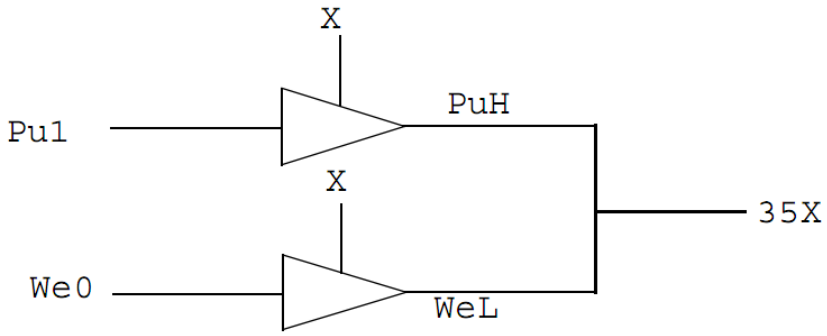
Hình 6.7 Tín hiệu ngõ ra của bufif1 có độ mạnh nằm trong khoảng như trên hình

Đầu ra của bufif0 trong Hình 6.6 là một ***strong*** L, dãy giá trị đầu ra được mô tả trong Hình 6.8.

strength0								strength1							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

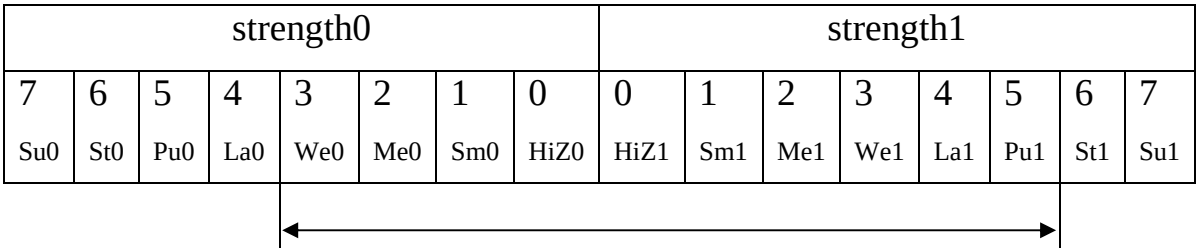
Hình 6.8 Tín hiệu ngõ ra của bufif0 có độ mạnh nằm trong khoảng như trên hình

Tổ hợp của hai tính hiệu có độ mạnh không rõ ràng sẽ cho ra kết quả là một tín hiệu có độ mạnh không rõ ràng. Kết quả của tín hiệu sẽ có một dãy các cấp độ độ mạnh mà bao gồm các cấp độ độ mạnh của các thành phần tín hiệu. Tổ hợp đầu ra từ 2 cổng điều khiển 3 trạng thái với đầu vào điều khiển không xác định được mô tả trong Hình 6.9, là một ví dụ.



Hình 6.9 Tổ hợp hai ngõ ra có độ mạnh không xác định

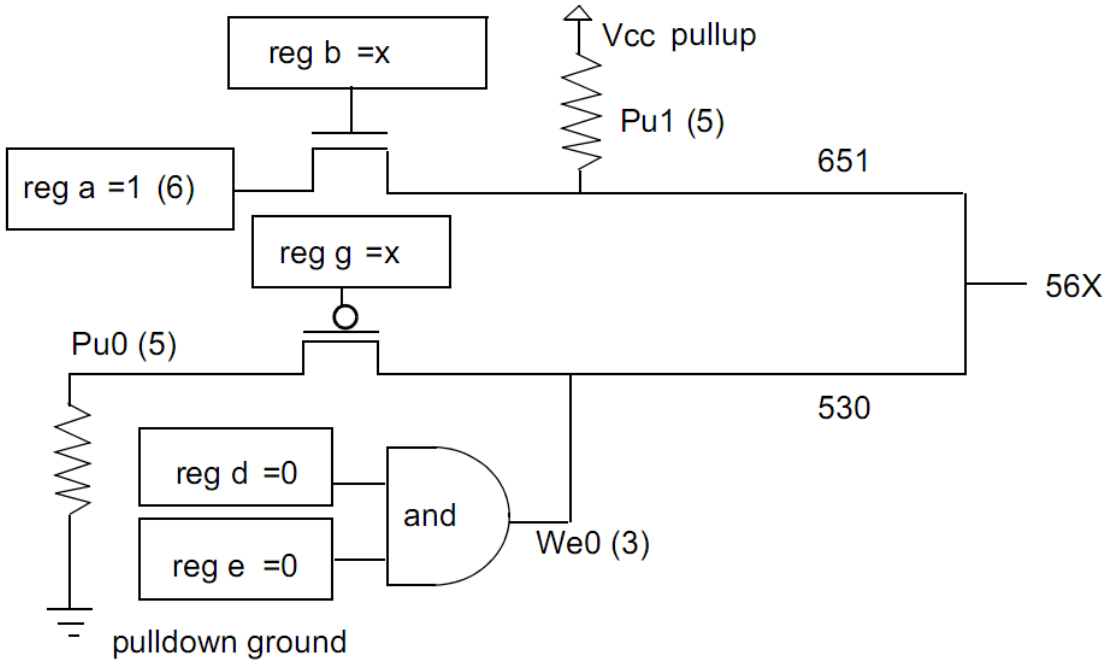
Trong Hình 6.9, tổ hợp tín hiệu có độ mạnh không rõ ràng cho ra một dãy bao gồm các mức độ cao nhất và thấp nhất của tín hiệu và tất cả các độ mạnh giữa chúng, như mô tả trong Hình 6.10.



Hình 6.10 Độ mạnh của ngõ ra trên hình 6.9

Kết quả là một giá trị x bởi vì phạm vi của nó gồm giá trị 1 và 0. Số 3 5, đi trước giá trị x, là tổ hợp của hai số. Số thứ nhất là số 3, tương ứng với mức độ strength0 cao nhất cho kết quả. Số thứ hai là 5, tương ứng với cấp độ strength1 cao nhất cho kết quả.

Mạng chuyển mạch có thể tạo ra một phạm vi độ mạnh của các giá trị giống nhau, như là tính hiệu của một cấu hình từ cao xuống thấp như Hình 6.11.



Hình 6.11 Mạng chuyển mạch

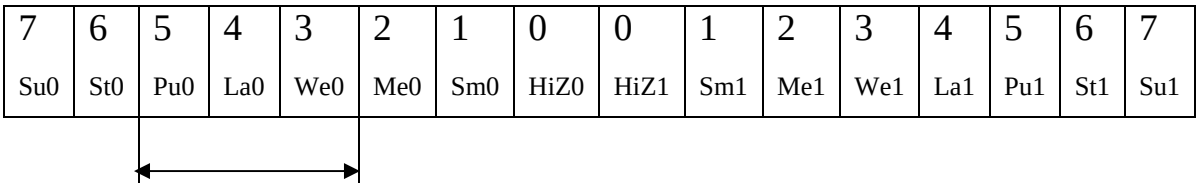
Trong Hình 6.11, tổ hợp cao của một thanh ghi, một cổng điều khiển bằng một thanh ghi có giá trị không xác định, và một pullup tạo ra một tín hiệu có giá trị 1 và phạm vi độ mạnh 651 mô tả trong Hình 6.12.

strength0								***strength1***							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Hình 6.12 Độ mạnh ngõ ra của net 651

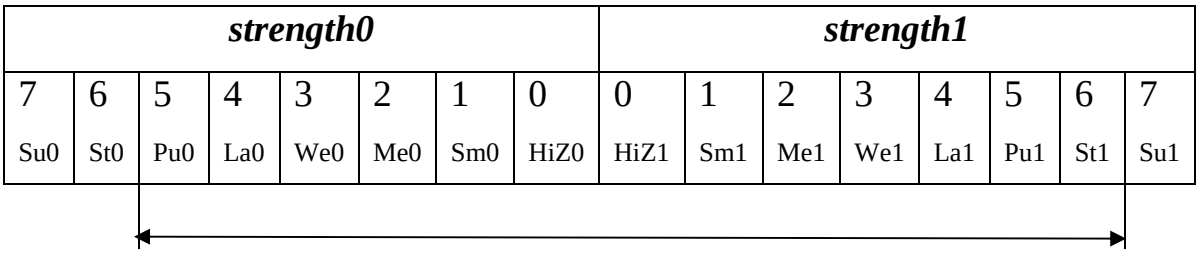
Trong Hình 6.11, tổ hợp thấp của một pulldown, một cổng điều khiển bởi một thanh ghi có giá trị không xác định, và một cổng and cho ra một tín hiệu có giá trị 0 và phạm vi độ mạnh 530 mô tả trong Hình 6.13.

strength0	***strength1***



Hình 6.13 Độ mạnh ngõ ra của net 530

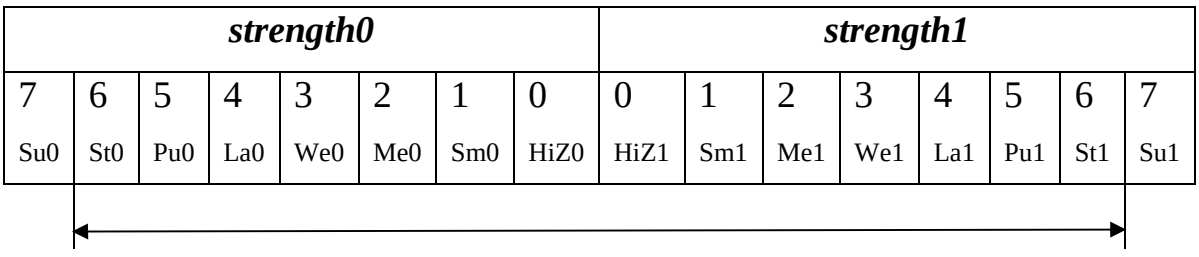
Khi tín hiệu của cấu hình từ cao xuống thấp trong Hình 6.11 tổ hợp, kết quả là một giá trị không xác định với phạm vi (56x) xác định bởi giá trị đầu và cuối là hai tín hiệu trong Hình 6.14.



Hình 6.14 Độ mạnh ngõ ra của net 56x

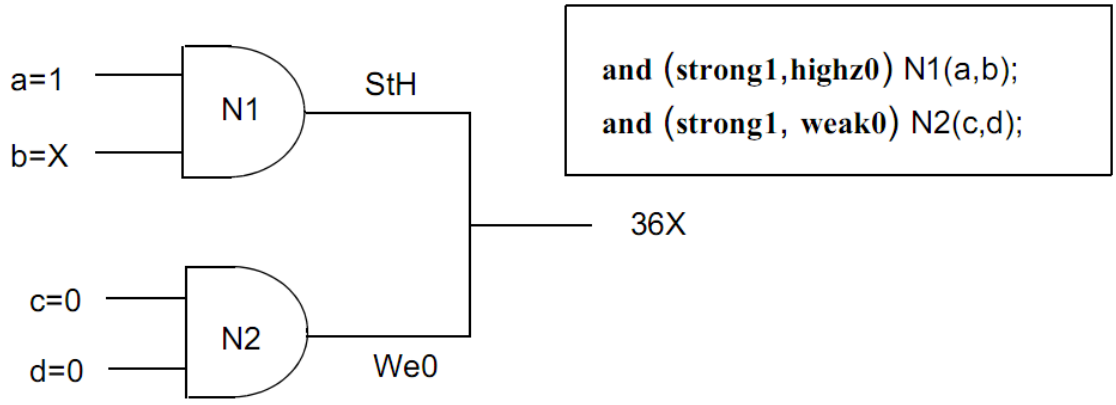
Trong Hình 6.11, thay thế pulldown trong cấu hình thấp bằng một supply0 sẽ thay đổi phạm vi của kết quả tới phạm vi (StX) mô tả trong Hình 6.15.

Phạm vi trong Hình 6.15 là strong x bởi vì nó là không xác định và các cực của cả hai thành phần là strong. Các cực của đầu ra của cấu hình thấp là strong bởi vì pmos thấp giảm độ mạnh của tín hiệu supply0. Mô hình hóa tính năng này thảo luận trong phần 6.2.10.



Hình 6.15 Độ mạnh ngõ ra của net 56x khi thay thế pulldown bởi supply0

Cổng logic tạo ra kết quả với độ mạnh không rõ ràng cũng giống như điều khiển ba trạng thái. Như trong trường hợp trong Hình 6.16. Cổng andN1 khai báo với độ mạnh highz0, và N2 khai báo với độ mạnh weak0.



Hình 6.16 Ngõ ra tạo bởi các cổng logic có tín hiệu điều khiển không rõ ràng

Hình 6.16, thanh ghi b có giá trị không xác định; vì vậy đầu vào của cổng and phía trên là strong x, cổng and phía trên có đặc tả độ mạnh bao gồm ***highz0***. Tín hiệu từ cổng **and** phía trên là một strong H bao gồm các giá trị mô tả trong Hình 6.17.

strength0								***strength1***							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Hình 6.17 Độ mạnh của tín hiệu ngõ ra StH

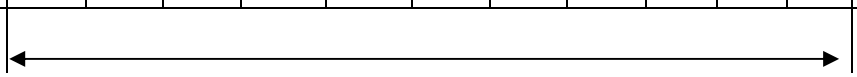
Hiz0 trong phần của kết quả bởi vì đặc tả độ mạnh cho cổng trong câu hỏi đã xác đinh độ mạnh cho đầu ra với giá trị 0. Đặc tả độ mạnh khác ngoài trở kháng cao cho giá trị 0 kết quả đầu ra trong một đầu ra cổng có giá trị x. Đầu ra của cổng **and** bên dưới là ***weak*** 0 như mô tả trong Hình 6.18.

strength0								***strength1***							
7 Su0	6 St0	5 Pu0	4 La0	3 We0	2 Me0	1 Sm0	0 HiZ0	0 HiZ1	1 Sm1	2 Me1	3 We1	4 La1	5 Pu1	6 St1	7 Su1

Hình 6.18 Độ mạnh của tín hiệu ngõ ra We0

Khi tín hiệu tổ hợp, kết quả có phạm vi (36x) như mô tả trong Hình 6.19.

strength0								***strength1***							
7 Su0	6 St0	5 Pu0	4 La0	3 We0	2 Me0	1 Sm0	0 HiZ0	0 HiZ1	1 Sm1	2 Me1	3 We1	4 La1	5 Pu1	6 St1	7 Su1



Hình 6.19 Độ mạnh của tín hiệu ngõ ra 36x

Hình 6.19 trình bày tổ hợp của một tín hiệu có độ mạnh không rõ ràng và một tín hiệu có độ mạnh rõ ràng. Tổ hợp này là chủ đề của phần 6.2.9.3.

6.2.9.3 Tín hiệu có độ mạnh không rõ ràng và tín hiệu có độ mạnh rõ ràng

Tổ hợp của một tín hiệu có độ mạnh rõ ràng và giá trị xác định với một tín hiệu có độ mạnh không rõ ràng được trình bài trong một vài trường hợp có thể. Để hiểu một tập các luật quản lý loại tổ hợp, nó cần thiết xem xét các cấp độ độ mạnh của tín hiệu có độ mạnh không rõ ràng riền mỗi nó và quan hệ với tín hiệu có độ mạnh rõ ràng. Khi tín hiệu có giá trị xác định và có độ mạnh rõ ràng kết hợp với một thành phần tín hiệu không rõ ràng, sẽ theo các luật sau:

- Các cấp độ độ mạnh của tín hiệu có độ mạnh không rõ ràng lớn hơn cấp độ độ mạnh của tín hiệu có độ mạnh rõ ràng sẽ vẫn có trong kết quả.

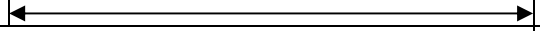
- Các cấp độ độ mạnh của tín hiệu có độ mạnh không rõ ràng nhỏ hơn cấp độ độ mạnh của tín hiệu có độ mạnh rõ ràng sẽ biến mất khỏi kết quả, chủ đề của luật c.
- Nếu các hoạt động của các luật a và luật b cho ra kết quả các cấp độ độ mạnh gián đoạn vì tính hiệu ngược nhau về giá trị, tín hiệu gián đoạn sẽ là một phần của kết quả.

Các hình sau mô tả một vài ứng dụng của các luật.

Trong Hình 6.20, các cấp độ độ mạnh của tín hiệu có độ mạnh không rõ ràng nhỏ hơn hoặc bằng cấp độ độ mạnh của tín hiệu có độ mạnh rõ ràng, biến mất khỏi kết quả, theo luật b.

Trong Hình 6.21, luật a, luật b và luật c được áp dụng. Các cấp độ độ mạnh của tín hiệu có độ mạnh không rõ ràng mà có giá trị đối ngược và độ mạnh thấp hơn độ mạnh của tín hiệu có độ mạnh rõ ràng không xuất hiện trong kết quả. Các cấp độ độ mạnh của tín hiệu có độ mạnh không rõ ràng thấp hơn độ mạnh của tín hiệu có độ mạnh rõ ràng, và có giá trị giống nhau không xuất hiện trong kết quả. Các cấp độ độ mạnh của tín hiệu có độ mạnh rõ ràng có độ mạnh cao hơn các cực của tín hiệu có độ mạnh không rõ ràng định nghĩa phạm vi của kết quả.

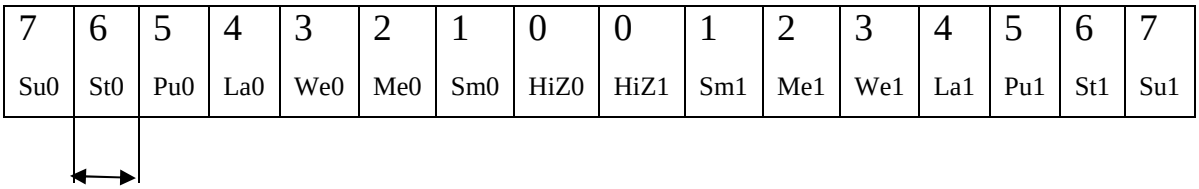
strength0								***strength1***							
7 Su0	6 St0	5 Pu0	4 La0	3 We0	2 Me0	1 Sm0	0 HiZ0	0 HiZ1	1 Sm1	2 Me1	3 We1	4 La1	5 Pu1	6 St1	7 Su1



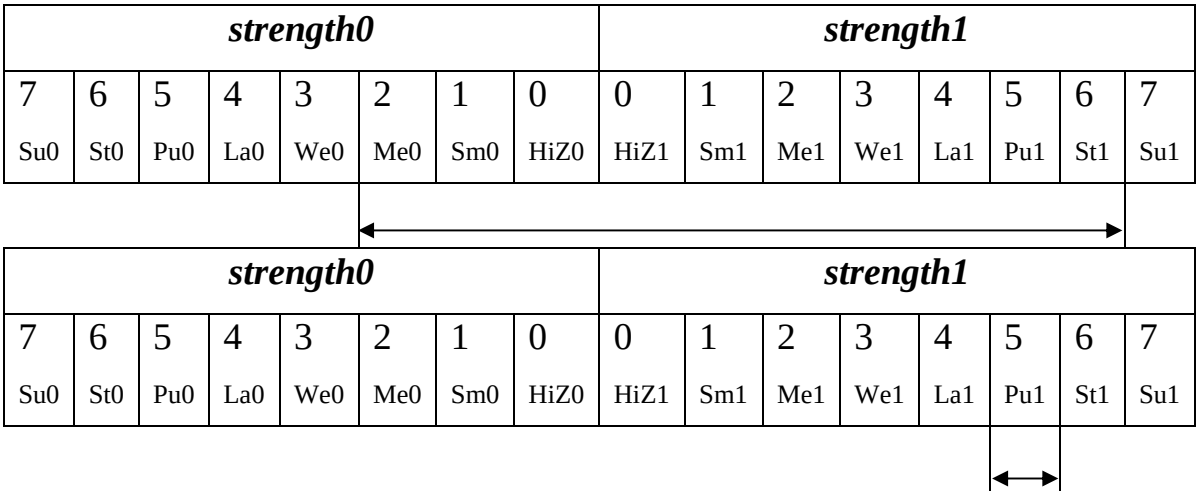
strength0								***strength1***							
7 Su0	6 St0	5 Pu0	4 La0	3 We0	2 Me0	1 Sm0	0 HiZ0	0 HiZ1	1 Sm1	2 Me1	3 We1	4 La1	5 Pu1	6 St1	7 Su1

Tổ hợp của hai tín hiệu bên trên cho ra tín hiệu kết quả bên dưới:

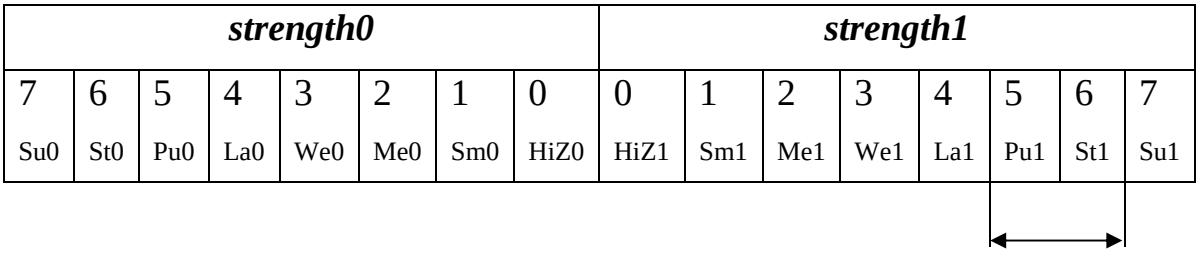
strength0	***strength1***



Hình 6.20 Độ mạnh ngõ ra tuân theo luật b



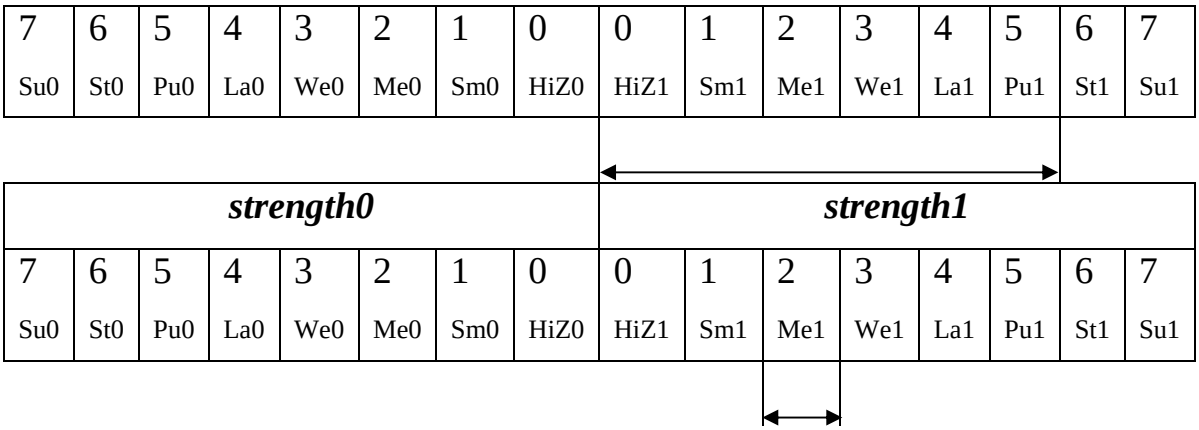
Tổ hợp của hai tín hiệu bên trên cho ra tín hiệu kết quả bên dưới:



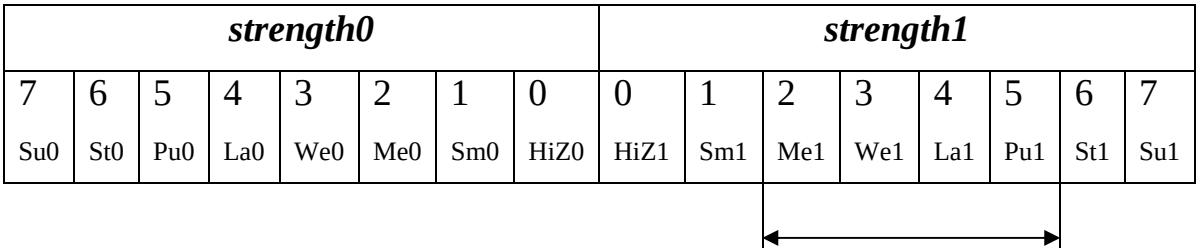
Hình 6.21 Độ mạnh ngõ ra tuân theo luật a, b và c

Trong Hình 6.22, luật a và luật b được áp dụng. Các cấp độ độ mạnh của tín hiệu có độ mạnh không rõ ràng mà có giá trị đối ngược và độ mạnh thấp hơn độ mạnh của tín hiệu có độ mạnh rõ ràng không xuất hiện trong kết quả. Các cấp độ độ mạnh của tín hiệu có độ mạnh rõ ràng có độ mạnh cao hơn các cực của tín hiệu có độ mạnh không rõ ràng định nghĩa phạm vi của kết quả.

strength0	***strength1***

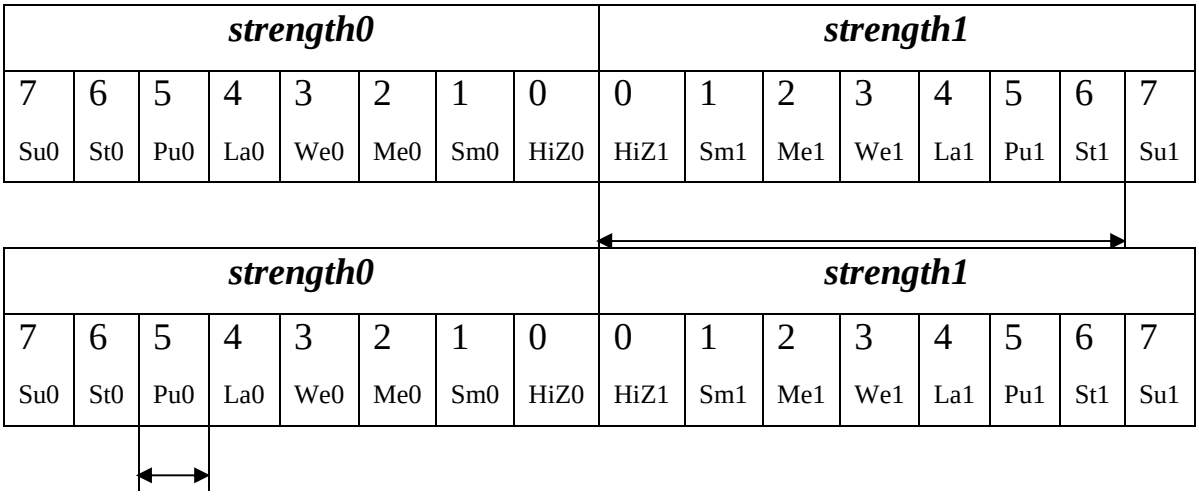


Tổ hợp của hai tín hiệu bên trên cho ra tín hiệu kết quả bên dưới:



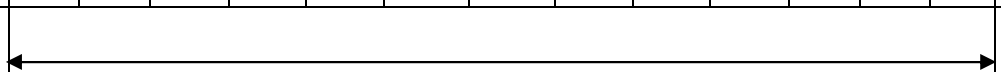
Hình 6.22 Độ mạnh ngõ ra tuân theo luật a, b

Trong Hình 6.23, luật a, luật b và luật c được áp dụng. Các cực lớn của độ mạnh của tín hiệu có độ mạnh không rõ ràng lớn hơn cấp độ độ mạnh của tín hiệu có độ mạnh rõ ràng. Kết quả là một phạm vi được định nghĩa bởi độ mạnh lớn nhất trong phạm vi độ mạnh của tín hiệu có độ mạnh không rõ ràng và cấp độ độ mạnh của tín hiệu có độ mạnh rõ ràng.



Tổ hợp của hai tín hiệu bên trên cho ra tín hiệu kết quả bên dưới:

strength0								***strength1***							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1



Hình 6.23 Độ mạnh ngõ ra tuân theo luật a, b và c

6.2.9.4 Loại wired logic net

Các loại net ***triand, wand, trior*** và ***wor*** sẽ giải quyết xung đột khi có nhiều điều khiển có cùng một độ mạnh. Loại net sẽ giải quyết giá trị tín hiệu bằng cách xem tín hiệu như đầu vào của hàm logic.

Ví dụ:

Xem xét tổ hợp của hai tín hiệu có độ mạnh rõ ràng trong Hình 6.24.

strength0								***strength1***							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

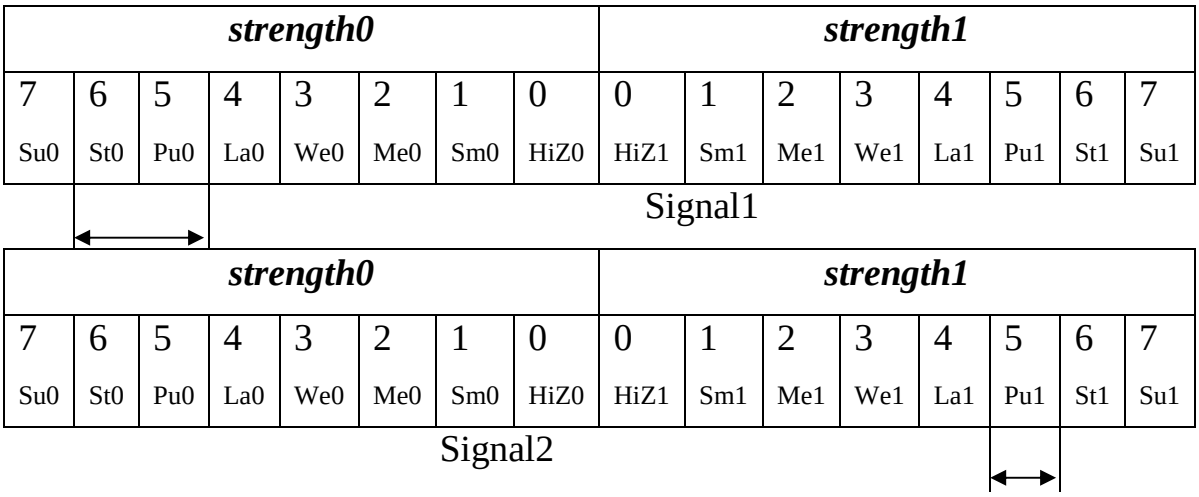
strength0								***strength1***							
7	6	5	4	3	2	1	0	0	1	2	3	4	5	6	7
Su0	St0	Pu0	La0	We0	Me0	Sm0	HiZ0	HiZ1	Sm1	Me1	We1	La1	Pu1	St1	Su1

Hình 6.24 Ngõ ra tổ hợp bởi tín hiệu ngõ vào có độ mạnh rõ ràng

Tổ hợp của tín hiệu trong Hình 6.24, sử dụng dây dẫn logic ***and***, cho ra một kết quả với giá trị giống với kế quả tạo ra bởi cổng ***and*** với giá trị của hai tín hiệu đầu vào. Tổ hợp của tín hiệu sử dụng dây dẫn logic ***or*** tạo ra kết quả với giá trị giống với kết quả được tạo ra bởi cổng ***or*** với giá trị của hai tín hiệu đầu vào. Độ mạnh của kết quả giống với độ mạnh của tổ hợp tín hiệu trong cả hai trường hợp. Nếu giá trị của tín hiệu bên trên thay

đổi cả hai tín hiệu trong Hình 6.24 thành giá trị 1, thì kết quả của cả hai loại logic là 1.

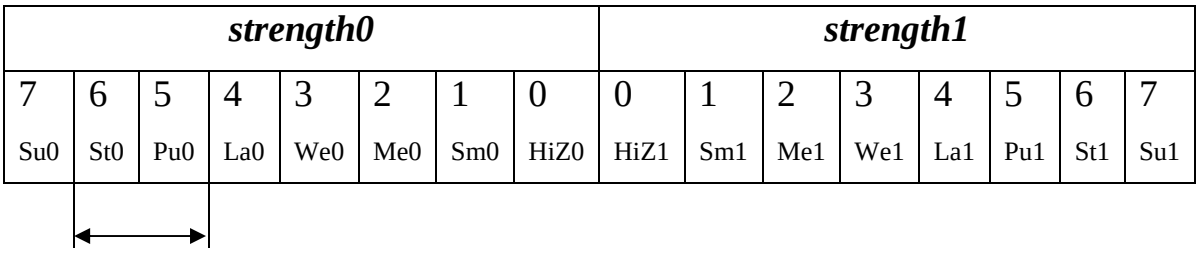
Khi tín hiệu có độ mạnh không rõ ràng tổ hợp trong dây dẫn logic, nó cần phải xem xét các kết quả của tất cả các tổ hợp của mỗi cấp độ độ mạnh trong tín hiệu đầu với mỗi cấp độ độ mạnh trong tín hiệu thứ 2, như trong Hình 6.25.



Tổ hợp các cấp độ độ mạnh cho cổng and theo hình dưới đây:

Signal1		***Signal2***		***Kết quả***	
Độ mạnh	*Giá trị*	*Độ mạnh*	*Giá trị*	*Độ mạnh*	*Giá trị*
5	0	5	1	5	0
6	0	5	1	6	0

Kết quả của tín hiệu:

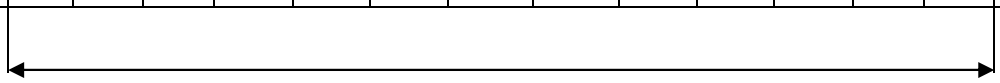


Tổ hợp các cấp độ độ mạnh cho cổng or theo hình dưới đây:

Signal1		***Signal2***		***Kết quả***	
Độ mạnh	*Giá trị*	*Độ mạnh*	*Giá trị*	*Độ mạnh*	*Giá trị*
5	0	5	1	5	1
6	0	5	1	6	0

Kết quả của tín hiệu:

strength0								***strength1***							
7 Su0	6 St0	5 Pu0	4 La0	3 We0	2 Me0	1 Sm0	0 HiZ0	0 HiZ1	1 Sm1	2 Me1	3 We1	4 La1	5 Pu1	6 St1	7 Su1



Hình 6.25 Tổ hợp các mức logic có độ mạnh khác nhau

6.2.10 Sự suy giảm độ mạnh bằng những linh kiện không trở

Các công tắc ***nmos, pmos*** và ***cmos*** sẽ cho qua độ mạnh từ dữ liệu đầu vào tới dữ liệu đầu ra, ngoại trừ độ mạnh của ***supply*** sẽ giảm xuống độ mạnh ***strong***.

Các công tắc ***tran, tranif0, tranif1*** sẽ không ảnh hưởng tới độ mạnh tín hiệu qua các cổng đầu cuối hai chiều, ngoại trừ độ mạnh ***supply*** sẽ giảm xuống độ mạnh ***strong***.

6.2.11 Sự suy giảm độ mạnh bằng những linh kiện trở

Các thiết bị ***rnmos, rpmos, rtran, rtranif1, rtranif0*** sẽ giảm độ mạnh của tín hiệu đi qua chúng theo Bảng 6.6

Bảng 6.6 Độ mạnh những linh kiện trở

Độ mạnh đầu vào	***Độ mạnh giảm***
Supplydrive	Pulldrive
Strongdrive	Pulldrive

Pulldrive	Weak drive
Largecapacitor	Mediumcapacitor
Weak drive	Mediumcapacitor
Mediumcapacitor	Smallcapacitor
Smallcapacitor	Smallcapacitor
Highimpedance	Highimpedance

6.2.12 Độ mạnh của loại net

Các loại net ***tri0, tri1, supply0*** và ***supply1*** sẽ tạo ra tín hiệu với các cấp độ độ mạnh cụ thể. Khai báo một ***trireg*** có thể đặc tả một trong hai cấp độ độ mạnh khác cấp độ độ mạnh mặc định.

6.2.12.1 Độ mạnh của net tri0, tri1

Loại net ***tri0*** mô hình một mạng kết nối tới một thiết bị điện trở kéo xuống. Trong sự vắng mặt của một nguồn đè, như vậy tín hiệu có giá trị 0 và độ mạnh ***pull***.Loại net ***tri1*** mô hình một mạng kết nối tới một thiết bị điện trở kéo xuống. Trong sự vắng mặt nguồn đề, như vậy tín hiệu có giá trị 1 và độ mạnh ***pull***.

6.2.12.2 Độ mạnh của trireg

Loại net ***trireg*** mô hình một nút lưu trữ thay đổi. Độ mạnh của kết quả từ một net trireg mà nó thay đổi trạng thái lưu trữ (đó là, một điều khiển thay đổi net và sau đó đi đến trở kháng cao) sẽ là một trong ba trạng độ mạnh: ***large, medium***, hoặc ***small***. Đặc tả độ mạnh kết nối với một net ***trireg*** cụ thể sẽ đặc tả bởi người dùng định nghĩa net. Mặt định là ***medium***. Cú pháp của đặc tả này được mô tả trong 3.4.1(thay đổi độ mạnh).

6.2.12.3 Độ mạnh của net supply0, supply1

Loại net ***supply0*** mô hình một kết nối với đất. Loại net ***supply0*** mô hình một kết nối tới nguồn cung cấp. Loại net ***supply0*** và ***supply1*** là các ***supply*** điều khiển độ mạnh.

6.2.13 Độ trì hoãn cổng (gate) và net

Trì hoãn cổng và net cung cấp một phương tiện mô tả chính xác hơn trì hoãn thông qua một mạch. Trì hoãn cổng đặc tả trì hoãn truyền tín hiệu từ bất kỳ đầu vào nào của cổng tới đầu ra cổng. Tối đa ba giá trị trên một đầu ra trình diễn bởi trì hoãn cạnh tăng, cạnh giảm, và tắt đặc tả như thấy từ 6.2.1 tới 6.2.8.

Trì hoãn net tham khảo thời gian lấy được từ bất kỳ điều khiển nào trên net thay đổi giá trị theo thời gian khi giá trị của net cập nhật và truyền qua hơn nữa. Tối đa có ba giá trị trên net có thể đặc tả.

Cả cổng và net, mặt định trì hoãn sẽ là 0 khi không có đặc tả trì hoãn được đưa ra. Khi một giá trị trì hoãn được đưa ra, thì giá trị này sẽ sử dụng cho tất cả các trì hoãn truyền chính xác cho cổng hoặc net. Khi hai trì hoãn được đưa ra, trì hoãn thứ nhất sẽ đặc tả cho trì hoãn cạnh lên, và trì hoãn thứ hai đặc tả cho trì hoãn cạnh xuống. Trì hoãn khi tín hiệu thay đổi tới trở kháng cao hoặc không xác định sẽ thấp hơn hai giá trị trì hoãn này.

Ba đặc tả trì hoãn :

- Trì hoãn thứ nhất tham khảo tới chuyển tiếp tới giá trị 1 (trì hoãn cạnh lên).
- Trì hoãn thứ hai tham khảo tới chuyển tiếp tới giá trị 0 (trì hoãn cạnh xuống).
- Trì hoãn thứ ba tham khảo tới chuyển tiếp tới giá trị trở kháng cao.

Khi một giá trị thay đổi tới giá trị không xác định, trì hoãn sẽ là nhỏ nhất trong ba trì hoãn. Độ mạnh của tín hiệu đầu vào sẽ không ảnh hưởng tới trì hoãn truyền từ đầu vào tới đầu ra.

Bảng 6.7 tổng hợp trì hoãn truyền từ - tới được lựa chọn đặc tả 2 và 3 trì hoãn.

Bảng 6.7 Độ trì hoãn trên net

Từ giá trị:	***Tới giá trị:***	***Trì hoãn nếu có***	
		2 trì hoãn	***3 trì hoãn***
0	1	d1	d1
0	x	min(d1,d2)	min(d1,d2,d3)
0	z	min(d1,d2)	d3
1	0	d2	d2
1	x	min(d1,d2)	min(d1,d2,d3)
1	z	min(d1,d2)	d3
x	0	d2	d2
x	1	d1	d1
x	z	min(d1,d2)	d3
z	0	d2	d2
z	1	d1	d1
z	x	min(d1,d2)	min(d1,d2,d3)

Ví dụ 6.8

Ví dụ 1: Ví dụ này đặc tả một, hai và ba trì hoãn:

and *#(10) a1 (out, in1, in2);// chỉ có một trì hoãn*

and *#(10,12) a2 (out, in1, in2);// trì hoãn cạnh lên và cạnh xuống*

bufif0 *#(10,12,11) b3 (out, in, ctrl);// trì hoãn cạnh lên, cạnh xuống, và tắt*

Ví dụ 2: Ví dụ này đặc tả một module mạch lật đơn giản với ba trạng thái đầu ra, nơi trì hoãn riêng được đưa ra cho từng cổng. Trì hoãn truyền từ đầu vào tới đầu ra của một module sẽ được tích lũy, và nó phụ thuộc vào phần tín hiệu đi qua mạng.

```
module tri_latch (qout, nqout, clock, data, enable);
    output qout, nqout;
    input clock, data, enable;
    tri qout, nqout;
    not #5          n1 (ndata,data);
    nand #(3,5)     n2 (wa,data,clock),
                     n3 (wb,ndata,clock);
    nand #(12,15)   n4 (q,nq,wa),
                    n5 (nq,q,wb);
    bufif1 #(3,7,13) q_drive(qout,q,enable),
                    nq_drive(nqout,nq,enable);
endmodule
```

6.2.14 Độ trì hoãn min:typ:max

Cú pháp cho trì hoãn trên một cổng nguyên thủy (bao gồm UPD), net và lệnh gán liên tục sẽ cho phép ba giá trị trì hoãn cho cạnh lên, cạnh xuống và tắc. Giá trị tối đa, trung bình và tối thiểu của mỗi giá trị trì hoãn sẽ đặc tả như là các biểu thức cách nhau bởi dấu hai châm (:). Chúng không yêu cầu qua hệ (ví dụ: min<=typ<=max) giữa các biểu thức trì hoãn tối thiểu, trung bình và tối đa. Đó có thể là ba biểu thức bất kỳ.

Ví dụ 6.9 cho thầy giá trị min:typ:max cho trì hoãn cạnh lên, cạnh xuống và tắc.

Ví dụ 6.9

```
module iobuf (io1, io2, dir);
    . . .
    bufif0 #(5:7:9, 8:10:12, 15:18:21) b1 (io1, io2, dir);
    bufif1 #(6:8:10, 5:7:9, 13:17:19) b2 (io2, io1, dir);
    . . .
endmodule
```

Cú pháp trì hoãn điều khiển trong lệnh thủ tục (9.7) cũng cho phép các giá trị tối thiểu, trung bình và tối đa. Đó là các đặc tả bởi các biểu thức ngăn cách bởi dấu hai chấm (:). Ví dụ 6.10 minh hoạ lý thuyết này:

Ví dụ 6.10

```
parameter min_hi = 97, typ_hi = 100, max_hi = 107;
reg clk;
always begin
    #(95:100:105) clk = 1;
    #(min_hi:typ_hi:max_hi) clk = 0;
end
```

6.2.15 Độ phân rã điện tích của net trireg

Giống như với net, đặc tả trì hoãn trong ***trireg*** net khai báo cũng bao gồm ba giá trị trì hoãn. Hai trì hoãn đầu đặc tả trì hoãn chuyển tiếp tới trạng thái logic 1 và 0 khi net ***trireg*** được điều khiển bởi một điều khiển. Trì hoãn thứ ba đặc tả cho thời gian phân rã điện tích thể hiện cho trì hoãn chuyển tiếp tới trạng thái logic z. Thời gian phân rã điện tích đặc tả trì hoãn giữa khi điều khiển ***trireg*** net tắc và khi nó thay đổi lưu trữ trong khoảng không xác định.

Một net ***trireg*** không cần đặc tả trì hoãn tắc vì net ***trireg*** không bao giờ thực hiện chuyển tiếp tới trạng thái logic z. Khi điều khiển của một net ***trireg*** thực hiện chuyển tiếp từ trạng thái logic 1, 0, hoặc x tới tắc, net ***trireg*** sẽ nhớ lại trạng thái logic 1, 0 hoặc x trước đó khi mà điều khiển còn mở. Giá trị z sẽ không truyền từ điều khiển của một net trireg tới một trireg. Một net ***trireg*** có thể chỉ giữ một trạng thái logic x khi x là trạng thái logic khởi đầu của net ***trireg*** hoặc khi net ***trireg*** bị ép buộc sang trạng thái z khi sử dụng câu lệnh ***force*** (trong phần 9.3.2).

Một đặc tả trì hoãn cho độ phân rã điện tích một hình một nút lưu trữ nạp không là lý tưởng, ví dụ một nút lưu trữ nạp sẽ nạp dòng rỉ ra ngoài thông qua thiết bị xung quanh và các kết nối.

Các quá trình phân rã điện tích và đặc tả trì hoãn cho phân rã điện tích được mô tả trong phần 6.2.15.1 và 6.2.15.2.

6.2.15.1 Quá trình phân rã điện tích

Phân rã điện tích là nguyên nhân của chuyển tiếp từ 1 hoặc 0 lưu trữ trong net ***trireg*** tới giá trị không xác định sau đặc tả trì hoãn. Quá trình phân rã điện tích bắt đầu khi điều khiển của net trireg tắc và net net ***trireg*** bắt đầu giữ điện tích. Quá trình phân rã điện tích sẽ kết thúc theo hai điều kiện bên dưới:

Đặc tả trì hoãn bởi thời gian phân rã điện tích trôi qua và net ***trireg*** thực hiện chuyển tiếp từ 1 hoặc 0 tới x.

Điều khiển của một net ***trireg*** mở và truyền 1, 0 hoạc x vào net ***trireg***.

6.2.15.2 Đặc tả trì hoãn của thời gian phân rã điện tích

Giá trị trì hoãn thứ 3 trong khai báo một net ***trireg*** đặc tả thời gian phân rã điện tích. Một bộ ba giá trị trì hoãn đặc tả trong một khai báo net trireg được mô tả:

#(d1,d2,d3)// (trì hoãn cạnh lên, trì hoãn cạnh xuống , thời gian phân rã điện tích)

Thời gian phân rã điện tích đặc tả trong một khai báo net ***trireg*** sẽ đi trước bởi một đặc tả trì hoãn cạnh tăng hoặc cạnh giảm.

Ví dụ 6.11

Ví dụ 1*:*

Ví dụ sau mô tả một đặc tả của thời gian phân rã điện tích trong một khai báo net ***trireg***:

trireg *(large) #(0,0,50) cap1;*

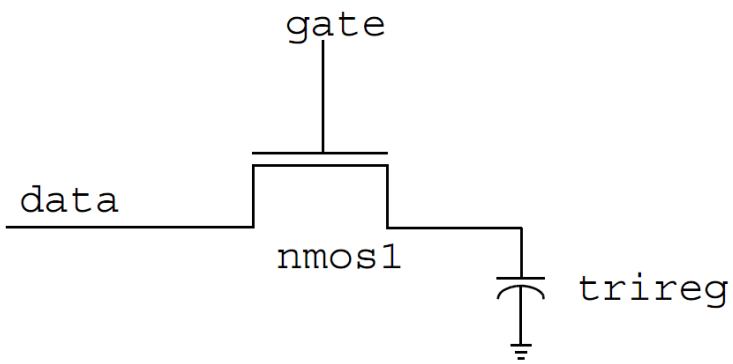
Ví dụ này khai báo một net ***trireg*** tên cap1. Net ***trireg*** này lưu trữ một điện tích ***large***. Đặc tả trì hoãn cho cạnh tăng là 0, cho trì hoãn cạnh giảm là 0, và thời gian phân rã điện tích là 50 đơn vị thời gian.

Ví dụ 2*:*

Ví dụ tiếp theo trình diễn một file nguồn mô tả bao gồm một khai báo net ***trireg*** với một đặc tả thời gian phân rã điện tích. Hình 6-26 thể hiện thi sơ đồ mạch của nguồn mô tả:

```
module capacitor;
        reg data, gate;
        // khai báo trireg với thời gian phân rã điện tích là 50 đơn vị.
         trireg (large) #(0,0,50) cap1;
        nmos nmos1 (cap1, data, gate); // nmos điều khiển trireg
        initial begin
                $monitor("%0d data=%v gate=%v cap1=%v", $time,
```

```
                    data, gate, cap1);
                    data = 1;
                    // Chốt điều khiển đầu vào bằng công tắc nmos
                     gate = 1;
                    #10 gate = 0;
                    #30 gate = 1;
                    #10 gate = 0;
                    #100 $finish;
            end
endmodule
```



Hình 6.26 Mô hình phần cứng của đoạn mô tả trong ví dụ trên

6.3 Những phần tử cơ bản người dùng tự định nghĩa (UDP)

Phần này mô tả một mô hình kỹ thuật để tăng thêm vào tập các cổng nguyên thủy được định nghĩa trước bằng cách thiết kết và đặc tả những yếu tố cơ bản mới gọi là UDPs. Thể hiện của UDPs mới đó có thể sử dụng giống hệt như là cổng nguyên thủy để biểu diễn các mô hình mạch điện.

Có hai loại quan hệ được thể hiện trong một UDP:

- Mạch tổ hợp – mô hình bởi một UDP tổ hợp.
- Mạch tuần tự - mô hình bởi một UDP tuần tự.

Một UDP tổ hợp sử dụng giá trị đầu vào của nó để xác định giá trị đầu ra tiếp theo của nó. Một UDP tuần tự sử dụng giá trị của đầu vào của nó và giá trị của đầu ra hiện tại để xác định giá trị đầu ra tiếp theo của nó. UDP tuần tự cung cấp môt cách để mô hình mạch điện tuần tự là các flip-flop và mạch chốt. UDP tuần tự có cả quan hệ kích hoạt theo mức và theo cạnh.

Mỗi UDP có chính xác một đầu ra, có thể có một trong ba trạng thái 1, 0 hoặc x. Giá trị ba trạng thái z không được hỗ trợ. Trong mạch UDPs tuần tự, đầu ra luôn luôn có giá trị là trạng thái nội bộ.

Giá trị z truyền tới đầu vào của một UDP được xem như là một giá trị x.

6.3.1 Định nghĩa phần tử cơ bản UDP

Định nghĩa UDP là một module độc lập, chúng giống như một cấp độ định nghĩa module trong cú pháp phân cấp. Chúng có thể nằm bất cứ đâu trong văn bản nguồn, có thể cả trước hoặc sau thể hiện của chúng trong module gọi chúng. Chúng không nằm giữa cặp từ khóa ***module*** và ***endmodule***.

Thực có thể giới hạn số định nghĩ UDP tối đa trong một mô hình, nhưng chúng cho phép tới ít nhất là 256.

Cú pháp thông thường của một định nghĩa UDP như cú Cú pháp 6-1:

Cú pháp 6-1

```
udp_declaration ::=
        { attribute_instance }  primitive udp_identifier ( udp_port_list ) ;
        udp_port_declaration { udp_port_declaration }
        udp_body
        endprimitive
        | { attribute_instance } primitive udp_identifier (
        udp_declaration_port_list ) ;
        udp_body
```

endprimitive

udp_port_list ::= (From A.5.2)

output_port_identifier , input_port_identifier{, input_port_identifier}

udp_declaration_port_list ::=

udp_output_declaration, udp_input_declaration{,

dp_input_declaration }

udp_port_declaration ::=

udp_output_declaration ;

| udp_input_declaration ;

| udp_reg_declaration ;

udp_output_declaration ::=

{ attribute_instance } **output** port_identifier

| { attribute_instance } **output reg** port_identifier [= constant_expression]

udp_input_declaration ::=

{ attribute_instance } **input** list_of_port_identifiers

udp_reg_declaration ::=

{ attribute_instance } **reg** variable_identifier

udp_body ::= (From A.5.3)

combinational_body | sequential_body

combinational_body ::=

table combinational_entry { combinational_entry } **endtable**

combinational_entry ::=

level_input_list : output_symbol ;

sequential_body ::=

[udp_initial_statement] **table** sequential_entry { sequential_entry }

endtable

udp_initial_statement ::=

initial output_port_identifier = init_val ;

init_val ::= **1'b0 | 1'b1 | 1'bx | 1'bX | 1'B0 | 1'B1 | 1'Bx | 1'BX | 1 | 0**

sequential_entry ::=

seq_input_list : current_state : next_state ;

seq_input_list ::=

level_input_list | edge_input_list

level_input_list ::=

level_symbol { level_symbol }

edge_input_list ::=

{ level_symbol } edge_indicator { level_symbol }

edge_indicator ::=

(level_symbol level_symbol) | edge_symbol

current_state ::= level_symbol

```
next_state ::=output_symbol | -
output_symbol ::= 0 | 1 | x | X
level_symbol ::= 0 | 1 | x | X | ? | b | B
edge_symbol ::= r | R | f | F | p | P | n | N | *
```

6.3.1.1 Tiêu đề UDP

Một định nghĩa UDP có hai hình thức xen kẽ nhau. Hình thức thứ nhất bắt đầu với từ khóa primitive, theo sau là một định danh, sẽ là tên của UDP. Nó kéo theo sau là một danh sách các cổng nằm trong dấu ngoặc đơn và được phân cách nhau bởi dấu phảy, cuối cùng là một dấu chấm phảy. Tiêu đề của định nghĩa UDP sẽ theo sau bởi danh sách các cổng và bảng trạng thái. Định nghĩa UDP kết thúc bởi từ khóa UDP.

Hình thức thứ hai bắt đầu với từ khóa primitive, theo sau là một định danh, sẽ là tên của UDP. Nó kéo theo sau là một danh sách các cổng nằm trong dấu ngoặc đơn và được phân cách nhau bởi dấu phảy, cuối cùng là một dấu chấm phảy. Tiêu đề của định nghĩa UDP sẽ theo sau bởi bảng trạng thái. Định nghĩa UDP kết thúc bởi từ khóa UDP.

UDPs có nhiều cổng đầu vào và có chính xác một cổng đầu ra; cổng hai chiều vào ra không được phép dùng trong UDPs. Tất cả các cổng trong UDP là vô hướng, cổng vector không được phép dùng.

Đầu ra sẽ cổng đầu tiên trong danh sách cổng.

6.3.1.2 Khai báo cổng (port) UDP

UDPs bao gồm các khai báo cổng đầu vào và đầu ra. Khai báo cổng đầu ra bắt đầu bằng từ khóa ***output***, theo sau là tên cổng đầu ra.Khai báo cổng đầu vào bắt đầu bằng từ khóa ***input***, theo sau là tên của một hoặc nhiều cổng đầu vào.

UDPs tuần tự bao gồm khai báo một ***reg*** cho cổng đầu ra thêm vào khai báo đầu ra, khi khai báo UDP sử dụng cách khai báo thứ nhất của tiêu đề UDP hoặc như phần khai báo đầu ra. UDPs tổ hợp không bao gồm khai báo ***reg***. Giá trị ban đầu của đầu ra có thể đặc tả bởi câu lện initial trong UDP tuần tự (6.3.1.3)

Quá trình thực thi có thể giới hạn số đầu vào của UDP, như chúng cho phép ít nhất 9 đầu vào cho UDP tuần tự và 10 đầu vào cho UDP tổ hợp.

6.3.1.3 Khai báo khởi tạo UDP tuần tự

Câu lệnh khởi tạo của một UDP tuần tự đặc tả giá trị đầu ra khi quá trình mô phỏng bắt đầu. Câu lệnh này bắt đầu với từ khóa ***initial***. Câu lện này theo sau sẽ là câu lệnh gán mà một bit đơn nguyên được gán vào cổng đầu ra.

6.3.1.4 Bảng khai báo UDP

Bảng trạng thái định nghĩa quan hệ của một UDP. Nó bắt đầu với từ khóa ***table*** và kết thúc với từ khóa ***endtable***. Mỗi dòng của bảng kết thúc bằng một dấu chấm phảy.

Mỗi dòng của bảng được tạo ra sử dụng một loạt các ký tự (mô tả trong trong Bảng 6.8), trong đó cho thấy giá trị đầu vào trạng thái đầu ra. Ba trạng thái 1, 0 và x được hỗ trợ. Trạng thái z bị loại thực thi trong UDPs. Một số ký tự đặt biệt được định nghĩa để biểu diễn các trạng thái có thể của mạch tổ hợp. Chúng được mô tả trong Bảng 6.8.

Thứ tự của các trường trạng thái đầu vào trong mỗi dòng của bảng theo thứ tự trong danh sách cổng trong định nghĩa tiêu đề UDP. Nó không quan hệ với thứ tự khai báo cổng.

UDPs tổ hợp có một trường trên đầu vào và một trường cho đầu ra. Trường đầu vào ngăn cách với đầu ra bởi đấu hai chấm (:). Mỗi dòng định nghĩa đầu ra cho một tổ hợp đặt biệt các giá trị đầu vào.

UDPs tuần tự có thêm vào một trường giữa trường đầu vào và đầu ra. Đó là trường thể hiện trạng thái hiện tại của UDP và tương ứng với giá trị đầu ra hiện tại. Nó giới hạn bởi dấu hai chấm (:). Mỗi dòng định nghĩa đầu ra dựa trên trạng thái hiện tại, tổ hợp chi tiết của các giá trị đầu vào, và một chuyển tiếp giá trị hiện tại tới đầu vào. Một dòng như mô tả bên dưới là không hợp lệ:

(01) (10) 0 : 0 : 1 ;

Nếu tất cả các giá trị đầu vào là x, thì trạng thái đầu ra sẽ là x.

Không cần thiết để đặc tả rõ ràng mỗi tổ hợp đầu vào có thể. Khi tất cả tổ hợp các giá trị đầu vào là không đặc tả rõ ràng kết quả mặc định cho đầu ra là x.

Không hợp lệ nếu tổ hợp của đầu vào giống nhau, bao gồm cả cạnh đặc tả cho đầu vào khác.

6.3.1.5 Giá trị Z trong UDP

Giá trị z trong bảng trạng thái không được hỗ trợ và nó xem như là không hợp lệ. Giá trị z qua cổng đầu vào tới UDP được xem như là giá trị x.

6.3.1.6 Tổng hợp các ký hiệu

Để tăng sự rõ ràng và dễ viết của bảng trạng thái, một vài ký hiệu đặt biệt được cung cấp. Bảng 6.8 tổng hợp nghĩa của tất cả các giá trị ký hiệu hợp lệ trong phần bảng của định nghĩa UDP.

Bảng 6.8 Giá trị ký hiệu hợp lệ trong phần bảng của định nghĩa UDP

Ký hiệu	Giải thích	Ghi chú
0	Logic 0	
1	Logic 1	
x	Không xác định	Cho phép trong các trường đầu vào và đầu ra của tất cả UDPs và trường trạng thái hiện tại trong UDPs tuần tự
?	Lặp lại các giá trị 0,1 và x	Không cho phép trong trường đầu ra
b	Lặp lại các giá trị 0,1	Cho phép trong các trường đầu vào của tất cả UDPs và trường trạng thái hiện tại trong UDPs tuần tự. Không cho phép trong trường đầu ra
-	Không thay đổi	Chỉ cho phép trong trường đầu ra của UDPs tuần tự.
(vw)	Giá trị thay đổi từ v tới w	v và w có thể là một giá trị bất kỳ trong 0, 1, x, ?, b và chỉ cho phép trong trường đầu vào.
*	Như (??)	Bất kỳ thay đổi nào trong đầu vào
r	Như (01)	Cạnh tăng của đầu vào
f	Như (10)	Cạnh giảm của đầu vào
p	Lặp lại các giá trị (01), (0x) và (x1)	Cạnh dương của điện thế đầu vào
n	Lặp lại các giá trị (01), (0x) và (x0)	Cạnh âm của điện thế đầu vào

6.3.2 UDP tổ hợp

Trong UDPs tổ hợp, trạng thái đầu ra được xác định chỉ như là một hàm của các trạng thái đầu vào hiện tại. Bất kỳ thay đổi nào của trạng thái đầu vào, UDP cũng ước lượng và trạng thái đầu ra sẽ được thiết lập theo giá trị chỉ định bởi dòng trong bảng trạng thái phù hợp với tất cả các trạng thái đầu vào. Tất cả các tổ hợp của đầu vào không đặc tả rõ ràng sẽ điều khiển trạng thái đầu ra tới giá trị x.

Ví dụ 6.12 định nghĩa một bộ dồn kênh với hai đầu vào dữ liệu và một đầu vào điều khiển:

Ví dụ 6.12

```
primitive multiplexer (mux, control, dataA, dataB);
    output mux;
    input control, dataA, dataB;
    table
    // bảng điều khiển mux dataA dataB
    01 0 : 1 ;
    01 1 : 1 ;
    01 x : 1 ;
    00 0 : 0 ;
    00 1 : 0 ;
    00 x : 0 ;
    10 1 : 1 ;
    11 1 : 1 ;
```

```
        1x 1 : 1 ;
        10 0 : 0 ;
        11 0 : 0 ;
        1x 0 : 0 ;
        x0 0 : 0 ;
        x1 1 : 1 ;
        endtable
endprimitive
```

Mục nhập đầu tiên trong

Ví dụ 6.12 có thể được giải thích như sau: Khi control bằng 0, dataA bằng 1 và dataB bằng 0 thì đầu ra mux bằng 1.

Tổ hợp đầu vào 0xx(control =0, dataA=x, dataB=x) là không rõ ràng. Nếu tổ hợp này xảy ra trong suốt quá trình mô phỏng thì giá trị cổng đầu ra sẽ là x.

Sử dụng dấu hỏi châm ?, mô tả một bộ dồn kênh có thể được viết tắt:

Ví dụ 6.13

```
primitive multiplexer (mux, control, dataA, dataB);
        output mux;
        input control, dataA, dataB;
        table
        //bảng điều khiển mux dataA dataB
        01?:1 ;// ? = 0 1 x
        00?:0 ;
        1?1:1 ;
        1?0:0 ;
```

```
        x00:0 ;
        x11:1 ;
        endtable
endprimitive
```

6.3.3 UDP tuần tự tích cực mức

Hành vi của mạch tuần tự tích cực mức cũng giống với hành vi của mạnh tổ hợp, ngoại trừ đầu ra khai báo là một loại ***reg*** và thêm vào một trường trong mỗi mục của bảng. Trường mới này sẽ biểu diễn trạng thái hiện tại của UDP. Trường đầu ra trong UDP tuần tự biểu diễn trạng thái tiếp theo.

Xem xét ví dụ mạch chốt:

Ví dụ 6.14

```
primitive latch (q, clock, data);
        output q; reg q;
        input clock, data;
        table
        // clock data q q+
        01 : ? : 1 ;
        00 : ? : 0 ;
        1? : ? : - ;// - = không thay đổi
        endtable
endprimitive
```

Mô tả này khác với UDP tổ hợp ở hai điều. Điều thứ nhất, định danh đầu ra q có thêm khai báo là **reg** chỉ ra đó là trạng thai nội bộ q. Giá trị đầu ra của UDP luôn giống với trạng thái hộp bộ. Điều thứ hai là một trường

cho trạng thái hiệu tại, được thêm vào phân cách với đầu vào và đầu ra bằng dấu hai chấm.

6.3.4 UDP tuần tự tích cực cạnh

Trong hành vi tích cực mức, giá trị của đầu vào và trạng thái hiện tại là đủ để xác định giá trị đầu ra. Hành vi tích cực cạnh khác ở chỗ thay đổi ở đầu ra gây nên bởi một chuyển tiếp của đầu vào. Điều này làm cho bảng trạng thái thành bảng chuyển tiếp.

Mỗi mục của bảng có một đặc tả chuyển tiếp ít nhất một giá trị đầu vào. Chuyển tiếp này đặc tả bởi một cặp giá trị trong dấu ngoặc đơn như là (01) hoặc ký hiệu chuyển tiếp như là r. Mục như thế này là không hợp lệ:

(01)(01)0:0:1 ;

Tất cả các chuyển tiếp mà không ảnh hưởng đến trạng thái đầu ra được quy định rõ ràng. Nếu không, quá trình chuyển đổi đó làm cho giá trị đầu ra chuyển sang z. Tất cả các chuyển tiếp không rõ ràng được mặt định giá trị đầu ra là x.

Nếu hành vi của UDP tích cực cạnh của đầu vào bất kỳ, trạng thái đầu ra mong muốn sẽ đặc tả cho tất cả các cạnh của tất cả các đầu vào.

Ví dụ 6.15 sau mô tả một flip-flopD tích cực cạnh lên:

Ví dụ 6.15

```
primitive d_edge_ff (q, clock, data);
    output q; reg q;
    input clock, data;
    table
    // clock dataqq+
    // đầu ra thu được trên cạnh tăng của clock
    (01) 0:?:0;
```

```
        (01) 1:?:1;
        (0?) 1:1:1;
        (0?) 0:0:0;
        // bỏ qua cạnh âm của lock
        (?0) ?:?:-;
        // bỏ qua sự thay đổi dữ liệu khi clock không thay đổi
         ?(??):?:-;
        endtable
endprimitive
```

Thuật ngữ (01) biểu diễn cho quá trình chuyển tiếp của giá trị đầu vào. Cụ thể, (01) biểu diễn một chuyển tiếp từ 0 tới 1. Dòng đầu tiên trong bảng định nghĩa UDP trước được hiểu như sau: khi clock thay đổi giá trị từ 0 tới 1 và dữ liệu bằng 0, đầu ra sẽ là 0 bất kể giá trị hiện hành.

Chuyển tiếp của clock từ 0 tới x với dữ liệu bằng 0 và trạng thái hiện tại là 1 thì kết quả đầu ra q sẽ là x.

6.3.5 Mạch hỗn hợp giữa UDP mạch tích cực mức và UDP tích cực cạnh

Định nghĩa UDP cho phép trộn lẫn lộn giữa cấu trúc tích cực mức và tích cực cạnh trong cùng một bảng. Khi đầu vào thay đổi, trường hợp tích cực cạnh xử lý đầu tiên, theo sau là trường hợp tích cực mức. Vì vậy, khi trường hợp tích cực cạnh và tích cực mức cho ra các giá trị đầu ra khác nhau, thì kết quả sẽ quyết định bởi trường hợp tích cực mức.

Ví dụ 6.16

```
primitive jk_edge_ff (q, clock, j, k, preset, clear);
        output q; reg q;
        input clock, j, k, preset, clear;
```

```
        table
        // clockjkpcstate output/next state
         ???01: ? : 1 ; // preset logic
         ???*1: 1 : 1 ;
         ???10: ? : 0 ; // clear logic
         ???1*: 0 : 0 ;
         r0000: 0 : 1 ; // normal clockingcases
         r0011: ? : - ;
         r0111: ? : 0 ;
         r1011: ? : 1 ;
         r1111: 0 : 1 ;
         r1111: 1 : 0 ;
         f????: ? : - ;
         b*???: ? : - ; // chuyển tiếp j và k
         b?*??: ? : - ;
        endtable
endprimitive
```

Trong ví dụ này, biến logic preset và clear là tích cực mức. Bất kỳ khi nào tổ hợp preset và clear là 01, thì đầu ra sẽ là 1. Tương tự, khi tổ hợp preser và clear là 10, thì giá trị đầu ra sẽ là 0.

Các logic còn lại là tích cực cạnh với clock. Trong trường hợp clock thông thường, flip-flop tích cực với cạnh lên của clock, như chỉ định r trong trường clock trong bảng bên trên. Trường hợp không tích cực với cạnh xuống của clock được chỉ ra bởi một dấu gạch (-) trong trường đầu ra (Bảng 6.8) cho mỗi mẫu với f là giá trị của clock. Nhớ rằng đầu ra mong muốn cho các chuyển tiếp đầu vào để tránh giá trị không mong muốn x ở

đầu ra. Hai mẫu cuối cùng chỉ ra chuyển tiếp trong đầu vào j và k không thay đổi đầu ra khi đồng hồ ổn định ở mức thấp hoặc cao.

6.3.6 Gọi sử dụng UDP

Cú pháp để tạo một thể hiện UDP được mô tả như

Cú pháp 6-2.

Cú pháp 6-2

```
udp_instantiation ::=
        udp_identifier [ drive_strength ] [ delay2 ]
                udp_instance { , udp_instance } ;
udp_instance ::=
        [ name_of_udp_instance ] ( output_terminal , input_terminal
                { , input_terminal } )
name_of_udp_instance ::=
        udp_instance_identifier [ range ]
```

Thể hiện của UDPs được đặc tả bên trong module giống như là một cổng (xem 6.2) Tên thể hiện là tùy chọn giống như cổng. Các cổng kết nối theo thứ tự khai báo trong định nghĩa UDP. Chỉ có hai trì hoãn có thể đặc tả bởi gì giá trị z không được hỗ trợ cho UDPs. Một loạt các tùy chọn chó thể được chỉ định bởi một mảng các thể hiện UDP. Luật kết nối các cổng cũng yêu cầu giống như với cổng trong phần 6.2.

Ví dụ 6.17 mô tả việc tạo một thể hiện của flip-flop loại D d_edge_ff (định nghĩa trong phần 6.3.2)

Ví dụ 6.17

```
module flip;
        reg clock, data;
        parameter p1 = 10;
        parameter p2 = 33;
        parameter p3 = 12;
```

```
        d_edge_ff #p3 d_inst (q, clock, data);
        initial begin
            data = 1;
            clock = 1;
            #(20 * p1) $finish;
        end
        always #p1 clock = ~clock;
        always #p2 data = ~data;
endmodule
```

6.4 Mô tả mạch tổ hợp và mạch tuần tự sử dụng mô hình cấu trúc

6.4.1 Mô tả mạch tổ hợp

Ta có thể sử dụng mô hình cấu trúc để mô tả thiết kế cho tất cả các loại mạch tổ hợp như multiplexer, decoder, encoder, adder,…

Ví dụ 6.18 Mô tả cổng XOR

```
module xor_gate ( out, a, b );
 input    a, b;
 output   out;
 wire     abar, bbar, t1, t2;
 not invA (abar, a);
 not invB (bbar, b);
 and and1 (t1, a, bbar);
```

```
  and and2 (t2, b, abar);
  or  or1 (out, t1, t2);
endmodule
```

Ví dụ 6.19 Mô tả mạch MUX2

```
module mux2 (in0, in1, select, out);
   input in0,in1,select;
   output out;
   wire s0,w0,w1;
   not (s0, select);
   and (w0, s0, in0),
       (w1, select, in1);
   or  (out, w0, w1);
endmodule // mux2
```

Ví dụ 6.20 Mô tả mạch decoder2-4

```
module decoder2_to_4 ( y0, y1, y2, y3, s1, s0 );
// Inputs and outputs
output y0, y1, y2, y3;
input s1, s0;
// Internal wires
wire s1n, s0n;
// Create complements of s1 and s0
not ( s1n, s1 );
not ( s0n, s0 );
and ( y0, s1n, s0n );
and ( y1, s1n, s0 );
and ( y2, s1, s0n );
and ( y3, s1, s0 );
endmodule
```

Ví dụ 6.21 Mô tả mạch encoder8-3

```
module encoder8_3( A , D );
    output[2:0] A;
    input[7:0] D;
    or( A[0], D[1], D[3], D[5], D[7] );
    or( A[1], D[2], D[3], D[6], D[7] );
    or( A[2], D[4], D[5], D[6], D[7] );
endmodule
```

Ví dụ 6.22 Mô tả mạch adder 1 bit

```
module adder1 (s, cout, a, b, cin);
  output s, cout;
  input a, b, cin;

  xor (t1, a, b);
  xor (s, t1, cin);
  and (t2, t1, cin),
      (t3, a, b);
  or  (cout, t2, t3);
endmodule
```

Ví dụ 6.23 Mô tả mạch adder 4 bit từ mạch adder 1 bit

```
module adder4 (sum, carry, inA, inB);
  output [3:0] sum;
  output carry;
  input [3:0] inA, inB;
  adder1 a0 (sum[0], c0,    inA[0], inB[0], 1'b0);
  adder1 a1 (sum[1], c1,    inA[1], inB[1], c0);
  adder1 a2 (sum[2], c2,    inA[2], inB[2], c1);
  adder1 a3 (sum[3], carry, inA[3], inB[3], c2);
endmodule
```

6.4.2 Mô tả mạch tuần tự

Ta có thể sử dụng mô hình cấu trúc để mô tả thiết kế cho tất cả các loại mạch tuần tự như latch, flipflop, register, counter,…

Ví dụ 6.24 Mô tả mạch SR latch

```
module clockedSR_latch(Q, Qbar, Sbar, Rbar, clk);
//Port declarations
output Q, Qbar;
input Sbar, Rbar, clkbar;
wire X, Y;
// Gate declarations
not a(clkbar, clk);
or r1(X, Sbar, clkbar);
or r2(Y, Rbar, clkbar);
nand n1(Q, X, Qbar);
nand n2(Qbar, Y , Q);
endmodule
```

Ví dụ 6.25 Mô tả mạch D latch

```
module clockedD_latch(Q, Qbar, D, clk);
//Port declarations
output Q, Qbar;
input D, clk;
wire X, Y, clkbar, Dbar;
// Gate declarations
not a1(clkbar, clk);
not a2(Dbar, D);
or r1(X, Dbar, clkbar);
or r2(Y, D, clkbar);
nand n1(Q, X, Qbar);
nand n2(Qbar, Y , Q);
```

```
endmodule
```

Ví dụ 6.26 Mô tả mạch D flipflip

```
module edge_dff(q, qbar, d, clk, clear);
output q,qbar;
input d, clk, clear;
wire s, sbar, r, rbar,cbar;
not (cbar, clear);
not (clkbar, clk);
// Input latches
nand (sbar, rbar, s);
nand (s, sbar, cbar, clkbar);
nand (r, rbar, clkbar, s);
nand (rbar, r, cbar, d);
// Output latch
nand (q, s, qbar);
nand (qbar, q, r, cbar);
endmodule
```

Ví dụ 6.27 Mô tả D flipflop master-slaver

```
// Negative edge-triggered D-flipflop with 2 D-latches in master-slave relation
module edge_dff(q, qbar, d, clk, clear);
output q, qbar;
input d, clk, clear;
wire q1;
clockedD_latch master(q1, , d, clk, clear); // master D-latch
not(clkbar, clk);
clockedD_latch slave(q, qbar, q1, clkbar, clear, writeCtr); // slave D-latch
endmodule
```

Ví dụ 6.28 Mô tả bộ đếm 4 bit

```
module counter(Q , clock, clear);
output [3:0] Q;
```

```
input clock, clear;
// Instantiate the T flipflops
t_ff tff0(Q[0], clock, clear);
t_ff tff1(Q[1], Q[0], clear);
t_ff tff2(Q[2], Q[1], clear);
t_ff tff3(Q[3], Q[2], clear);
endmodule
```

Ví dụ 6.29 Mô tả thanh ghi 4 bit

```
// Register module - 4-bit register
module register4bits( dataOut, dataIn, enable, clock, clear );
// Inputs and outputs
output [3:0] dataOut;
input [3:0] dataIn;
input enable, clock, clear;
// 4 D-flipflops
edge_dff ff0( dataOut[0], dataIn[0], enable, clock, clear );
edge_dff ff1( dataOut[1], dataIn[1], enable, clock, clear );
edge_dff ff2( dataOut[2], dataIn[2], enable, clock, clear );
edge_dff ff3( dataOut[3], dataIn[3], enable, clock, clear );
endmodule
```

6.5 Bài tập

1. Bạn hiểu thế nào về mô hình cấu trúc (structural model)?
2. Nêu một số cổng logic cơ bản mà bạn biết và các gọi chúng trong mô hình cấu trúc bằng ngôn ngữ Verilog?
3. Các mô hình độ mạnh logic trong Verilog?
4. Sự kết hợp độ mạnh logic của những tín hiệu không rõ ràng trong Verilog như thế nào?

5. Nêu các đặt tả trì hoãn cổng và net?

6. UDP là gì? Nêu các loại UDP cơ bản?

7. Cách khai báo và sử dụng một UDP cơ bản?

8. Tạo một UDP theo công thức boolean sau:

   out = (a1 & a2 & a3) | (b1 & b2)

Chương 7. Mô hình thiết kế hành vi (Behavioral model)

7.1 Khái quát

Mô hình thiết kế ở mức độ hành vi sẽ mô tả hệ thống theo cách mà nó hành xử thay vì kết nối các linh kiện ở mức thấp lại với nhau. Hay nói cách khác, mô hình thiết kế hành vi chỉ mô tả mối quan hệ giữa các tín hiệu ngõ ra với các tín hiệu ngõ vào mà không cần quan tâm đến cấu trúc phần cứng bên trong nó. Mô hình hành vi có hai mức độ mô tả khác nhau, một là mô tả ở mức độ RTL hay còn gọi là phép gán nối tiếp hay cũng có thể gọi là phép gán liên tục, hai là mô tả ở mức độ giải thuật (algorithmic).

7.2 Phép gán nối tiếp hay phép gán liên tục - mô hình thiết kế RTL (continuous assignment)

7.2.1 Giới thiệu

Phép gán nối tiếp – mô hình thiết kế RTL, thông thường mô tả luồng dữ liệu bên trong những hệ thống giống như luồng dữ liệu giữa những thanh ghi. Phép gán nối tiếp – mô hình thiết kế RTL đa số được sử dụng trong việc thiết kế mạch tổ hợp.

Phép gán nối tiếp – mô hình thiết kế RTL, dùng để gán một giá trị đến net, net ở đây có thể là net đơn hoặc một mảng (vector) các net, do đó biểu thức bên trái phép gán nối tiếp phải có dữ liệu là net, không thể là loại dữ liệu thanh ghi (register). Phép gán này được thực hiện ngay khi có sự thay đổi giá trị ở bên phải của phép gán. Phép gán nối tiếp – mô hình thiết kế RTL, cung cấp một phương pháp để mô hình mạch tổ hợp mà không cần mô tả sự kết nối giữa các cổng với nhau, mà thay vào đó nó mô tả biểu thức

logic để điều khiển net. Hay nói cách khác, phép gán nối tiếp điều khiển net theo như cách mà các cổng linh kiện điều khiển net, trong đó biểu thức bên phải phép gán có thể được xem như là một mạch tổ hợp để điều khiển net một cách liên tục.

Ví dụ 7.1

```
assign m = 1'b1;
assign a = b & c;
assign #10 a = 1'bz;
```

7.2.2 Phép gán nối tiếp khi khai báo net

Verilog cho phép một phép gán nối tiếp được đặt trên cùng phát biểu khai báo net

Ví dụ 7.2

```
wire (strong1, pull0) mynet = enable ; //khai báo và gán
wire a = b & c; //khai báo và gán
```

Tuy nhiên với cách gán này, do một net chỉ được khai báo một lần dẫn đến net này chỉ nhận được giá trị từ một phép gán duy nhất. Nếu muốn một net có thể nhận giá trị từ nhiều phép gán khác nhau thì ta phải dùng phát biểu phép gán nối tiếp tường minh.

7.2.3 Phát biểu phép gán nối tiếp tường minh

Trong phát biểu phép gán nối tiếp tường minh, ta dùng một cách tường minh một phép gán nối tiếp với từ khóa *assign* để gán giá trị cho net sau khi net đã được khai báo.

Những phép gán trên các net sẽ được thực hiện liên tục một cách tự động. Hay nói cách khác, bất cứ khi nào giá trị của một toán hạng bên biểu

thức phía phải của phép gán thay đổi thì toàn bộ giá trị các net ở bên trái phép gán sẽ cập nhật ngay lại giá trị. Nếu giá trị mới khác với giá trị trước đó thì giá trị mới sẽ được gán vào net bên trái phép gán.

Ví dụ 7.3

```
wire a; //khai báo
parameter Zee = 1'bz;
assign a = Zee; //gán 1
assign a = b & c; //gán 2
```

Những dạng hợp lệ của biểu thức bên trái của phép gán nối tiếp phải là loại dữ liệu net:

- Net (có thể là net đơn – scalar hoặc một mảng các net – net vector)
- Bit bất kì được chọn trong net vector
- Một phần các bit bất kì được chọn trong net vector
- Một phần các bit bất kì có chỉ số (index) được chọn trong net vector.
- Biểu thức nối {} giữa các biểu thức hợp lệ ở trên.

Ví dụ 7.4

```
module adder (sum_out, carry_out, carry_in, ina, inb);
    output [3:0] sum_out;
    output carry_out;
    input [3:0] ina, inb;
    input carry_in;
    wire carry_out, carry_in;
    wire [3:0] sum_out, ina, inb;
    assign {carry_out, sum_out} = ina + inb + carry_in;
endmodule
```

Trong ví dụ trên, phép gán nối tiếp được sử dụng để mô hình một mạch cộng 4 bit có nhớ. Ở đây ta không thể dùng phép gán nối tiếp khi khai báo net bởi vì ta sử dụng các net này trong phép nối {} phía bên trái phép gán.

Ví dụ 7.5

```
module select_bus(busout, bus0, bus1, bus2, bus3, enable, s);
    parameter n = 16;
    parameter Zee = 16'bz;
    output [1:n] busout;
    input [1:n] bus0, bus1, bus2, bus3;
    input enable;
    input [1:2] s;
    tri [1:n] data; // khai báo net
    // khai báo net với phép gán nối tiếp
    tri [1:n] busout = enable ? data : Zee;
    // phát biểu với 4 phép gán nối tiếp
    assign
    data = (s == 0) ? bus0 : Zee,
    data = (s == 1) ? bus1 : Zee,
    data = (s == 2) ? bus2 : Zee,
    data = (s == 3) ? bus3 : Zee;
endmodule
```

Trong ví dụ trên ta thấy net data có thể nhận giá trị từ nhiều phép gán nối tiếp khác nhau.

7.2.4 Tạo độ trì hoãn (delay) cho phép gán

Để tạo độ trì hoãn tính từ khi giá trị của một toán hạng bên phải phép gán thay đổi cho đến khi giá trị net bên tay trái được cập nhật sự thay đổi giá trị đó thì ta có thể mô tả độ trì hoãn đó ngay trên phép gán. Nếu biểu thức bên tay trái là một net đơn (scalar) thì việc mô tả độ trì hoãn hoàn toàn giống như mô tả độ trì hoãn cho các cổng linh kiện, đó có thể là thời gian lên (rising), thời gian xuống (falling) hay thời gian để đạt trạng thái tổng trở cao (high impedance) cho net đó.

Nếu biểu thức bên tay trái là một mảng các net (net vector) thì qui luật sau sẽ quyết định phép gán có độ trì hoãn loại nào.

- Nếu biểu thức bên phải phép gán tạo ra sự thay đổi từ trạng thái khác 0 đến trạng thái 0, thì độ trì hoãn thời gian xuống (falling) sẽ được sử dụng
- Nếu biểu thức bên phải phép gán tạo ra sự thay đổi từ trạng thái bất kì đến trạng thái tổng trở cao z thì thời gian trì hoãn turn-off sẽ được sử dụng.
- Tất cả các trường hợp còn lại thì trì hoãn thời gian lên (rising) sẽ được sử dụng.

Việc mô tả độ trì hoãn trong một phép gán nối tiếp ngay trên khai báo net sẽ được xử lí khác với việc mô tả độ trì hoãn của một net rồi sau đó mới thực hiện phép gán nối tiếp. Một giá trị độ trì hoãn có thể được cung cấp cho một net trong khai bao net như trong ví dụ sau:

```
wire #10 wireA;
assign wireA = 1'bz;
assign wireA = b;
```

Cú pháp này được gọi là mô tả độ trì hoãn của net, có nghĩa là bất kì sự thay đổi giá trị nào từ biểu thức bên phải phép gán mà được cung cấp đến wireA đều sẽ bị trì hoãn trong 10 đơn vị thời gian trước khi phép gán có hiệu lực. Như vậy trong ví dụ trên, cả hai phép gán đều đợi 10 đơn vị thời gian sau thì giá trị bên phải phép gán mới được cập nhật cho net wireA.

Còn đối với phép gán nối tiếp được mô tả ngay trong khai báo net thì độ trì hoãn là một phần của phép gán nối tiếp chứ không phải là độ trì hoãn của net, do đó nó sẽ không phải là độ trì hoãn của các phép gán khác trên net đó. Hơn nữa, nếu bên trái phép gán là một mảng các net (vector net) thì thời gian trì hoãn lên, thời gian trì hoãn xuống sẽ không được cung cấp đến những bit riêng lẻ nếu phép gán nằm ngay trong phần khai báo net.

wire #10 wireA = 1’bz;

Trong trường hợp mà một toán hạng bên phải phép gán thay đổi trước khi sự thay đổi của giá trị trước đó có đủ thời gian để truyền đến net bên trái phép gán thì những bước sau sẽ diễn ra.

- Giá trị của biểu thức bên phải phép gán được tính.
- Nếu giá trị được tính đó khác với giá trị đang truyền đến net bên trái phép gán thì giá trị đang truyền này sẽ hủy.
- Nếu giá trị được tính đó bằng với giá trị đang truyền đến net bên trái phép gán thì giá trị đang truyền vẫn cứ tiếp tục.

7.2.5 Độ mạnh phép gán

Độ mạnh điều khiển của phép gán có thể được mô tả bởi người sử dụng. Việc mô tả độ mạnh này chỉ hợp lệ cho những phép gán đến những net đơn như sau

wire	tri	trireg
wand	triand	tri0

wor trior tri1

Những phép gán nối tiếp có điều khiển độ mạnh có thể được mô tả ngay trong khai báo net hoặc đứng một mình trong phép gán nối tiếp với từ khóa **assign**. Việc mô tả độ mạnh, nếu được cung cấp thì phải theo ngay sau từ khóa (từ khóa cho loại dữ liệu hay từ khóa **assign**) và đứng trước mô tả độ trì hoãn. Bất cứ khi nào mà phép gán nối tiếp điều khiển net thì độ mạnh của giá trị sẽ mô phỏng như là đã được mô tả.

Một mô tả độ mạnh điều khiển sẽ bao gồm một giá trị độ mạnh để mô tả cho một net được gán giá trị 1 và một giá trị độ mạnh để mô tả cho một net được gán giá trị 0.

Những từ khóa sau sẽ mô tả độ mạnh của phép gán 1:

supply1 strong1 pull1 weak1 highz1

Những từ khóa sau sẽ mô tả độ mạnh của phép gán 0:

supply0 strong0 pull0 weak0 highz0

Thứ tự của sự mô tả hai độ mạnh trên là tùy ý. Hai nguyên tắc sau sẽ ràng buộc việc sử dụng sự mô tả độ mạnh điều khiển.

- Những mô tả độ mạnh (highz1, highz0) và (highz0, highz1) sẽ được xem như là không hợp lệ.
- Nếu độ mạnh điều khiển không được mô tả thì độ mạnh mặc định sẽ là (strong1, strong0).

7.3 Phép gán qui trình - mô hình thiết kế ở mức độ thuật toán (procedural assignment)

Phép gán qui trình – mô hình thiết kế ở mức độ thuật toán sử dụng một chuỗi các lệnh cụ thể của những phát biểu để định nghĩa chuỗi các phép toán trong hệ thống. Việc mô tả này giống như việc mô tả chương trình sử dựng ngôn ngữ cấp cao khác chẳng hạn như C. Phép gán qui trình

– mô hình thiết kế ở mức độ thuật toán đa số được sử dụng trong việc thiết kế mạch tuần tự. Phép gán qui trình sẽ cung cấp khả năng trừu tượng cần thiết để mô tả một hệ thống phức tạp ở mức cao chẳng hạn như hệ thống vi xử lí hoặc thực thi việc kiểm tra định thời phức tạp, mà ta khó có thể thực hiện được chúng nếu sử dụng mô hình cấu trúc hoặc mô hình RTL (phép gán nối tiếp)

Phép gán qui trình được dùng để cập nhật giá trị vào cho loại dữ liệu biến (variable) như **reg**, **integer**, **time**, **real**, **realtime** và **memory**. Phép gán này không mất thời gian, mà thay vào đó biến sẽ lưu giữ giá trị của phép gán cho đến khi có một phép gán qui trình kế tiếp cho biến đó xuất hiện.

Khác với phép gán liên tục (continuous assignment) đó là điều khiển net và cập nhật giá trị cho net bất cứ khi nào giá trị của một toán hạng trong biểu thức bên phải phép gán thay đổi, còn phép gán qui trình cập nhật giá trị cho biến dưới sự điều khiển của luồng cấu trúc qui trình xung quanh nó.

Bên phải của phép gán qui trình có thể là bất kì một biểu thức nào, bất kì loại dữ liệu nào dùng để tính toán ra một giá trị. Bên trái phép gán là một biến nhận giá trị gán từ phía phải phép gán. Biến bên trái phép gán chỉ có thể là một trong những dạng sau:

- Một biến đơn hay một mảng biến (variable vector) có loại dữ liệu biến là reg, integer, real, realtime, hoặc time.
- Một bit được chọn trong một mảng biến (variable vector) có loại dữ liệu biến là reg, integer, hoặc time. Các bit còn lại sẽ không bị tác động.
- Một phần các bit được chọn trong một mảng biến (variable vector) có loại dữ liệu biến là reg, integer, hoặc time. Các bit còn lại sẽ không bị tác động.
- Một từ (word) trong bộ nhớ.

- Sự kết hợp dùng phép {} để nối các dạng ở trên lại với nhau. Việc kết hợp dùng phép {} này khiến việc phân chia các phần kết quả của giá trị biểu thức bên phải và gán những phần này vào những phần khác nhau của các biến trong phép {} bên tay trái theo thứ tự rõ ràng.

Chú ý: Phép gán đến một biến có kiểu dữ liệu biến là reg sẽ khác so với phép gán đến biến có kiểu dữ liệu biến là real, realtime, time, hay integer khi mà số bít bên phải phép gán ít hơn so với bên trái phép gán. Phép gán đến reg sẽ không "sign-extend".

Phép gán qui trình xuất hiện bên trong những khối qui trình (procedure) như là **always**, **initial** , **task**, **function** và những từ khóa này có thể được xem như là sự kích khởi của các phép gán qui trình.

Các khối qui trình **always** và khối qui trình **initial** bắt đầu theo những luồng hoạt động độc lập.

Các khối qui trình **task** và **function**, ta sẽ xem xét trong chương sau.

Ta xét một ví dụ đơn giản hoàn chỉnh về mô hình thiết kế qui trình procedure.

Ví dụ 7.6

```
module behave;
    reg [1:0] a, b;
    initial begin
    a = 'b1;
    b = 'b0;
    end
    always begin
    #50 a = ~a;
```

```
        end
        always begin
        #100 b = ~b;
        end
endmodule
```

Trong quá trình chạy mô phỏng thiết kế này, tất cả các luồng dữ liệu được mô tả bởi khối initial và always sẽ cùng thực thi chạy mô phỏng tại thời điểm zero. Luồng dữ liệu bên trong khối initial chỉ thực thi một lần, còn luồng dữ liệu trong khối always sẽ thực thi lập lại khi luồng dữ liệu (phép gán) chạm đến từ khóa 'end'. Cần chú ý ở đây đó là sự lập lại của luồng dữ liệu (hay có thể nói là sự lập lại của các phép gán) bên trong khối always là hoàn toàn khác với sự lập lại của các mã lệnh (instructions) trong ngôn ngữ software. Các phép gán trong khối always sẽ được synthesize ra một mạch phần cứng, mạch này có chức năng hoạt động lập lại, ví dụ một mạch đếm lên chẳng hạn.

Trong mô tả thiết kế trên, hai dữ liệu biến thanh ghi a và b được khởi tạo giá trị lần lượt là 1 và 0 ngay tại thời điểm zero. Các phép gán trong khối initial sau khi khởi tạo giá trị cho các biến xong thì sẽ không thực thi lại bất kì lần nào nữa trong suốt quá trình chạy mô phỏng. Khối cấu trúc initial chứa một block begin-end (đây còn được gọi là sequential block) của phát biểu. Trong khối begin-end này, biến dữ liệu thanh ghi a được khởi tạo trước, sau đó đến b. Khối cấu trúc always cũng bắt đầu thực thi tại thời điểm zero, nhưng giá trị của các biến dữ liệu thanh ghi không thay đổi cho đến khi khoảng thời gian trì hoãn được mô tả bởi "delay controls" (bắt đầu bởi #) được trôi qua. Như vậy, theo như mô tả thiết kế, trong quá trình chạy mô phỏng, giá trị biến thanh ghi dữ liệu a sẽ bị đảo sau khoảng thời gian là 50 đơn vị thời gian và biến thanh ghi dữ liệu b bị đảo sau khoảng thời gian

là 100 đơn vị thời gian. Khối cấu trúc alays cũng chứa một block begin-end (đây còn được gọi là sequential block). Bởi vì, luồng dữ liệu (các phép gán) trong khối cấu trúc always được lập lại khi luồng dữ liệu chạm –end trong khối begin-end nên với mô tả thiết kế trên thì sẽ có hai dạng xung tuần hoàn được tạo ra bởi a và b. Một là xung có chu kì là 100 đơn vị thời gian được tạo ra bởi a, một là xung khác có chu kì là 200 đơn vị thời gian được tạo ra bởi b. Hai khối cấu trúc always như trong mô tả thiết kế trên sẽ thực thi đồng thời trong suốt quá trình chạy mô phỏng, bởi vì trong thực tế sau quá trình synthesis, thì hai khối always này sẽ tạo ra hai mạch phần cứng độc lập và khi được cung cấp nguồn điện hoạt động thì cả hai sẽ hoạt động đồng thời.

7.3.1 Phép gán khai báo biến

Phép gán khai báo biến là một trường hợp đặc biệt của phép gán qui trình, dùng để gán một giá trị cho biến. Nó cho phép khởi tạo một giá trị cho biến trong cùng phát biểu khai báo biến. Phép gán có thể là một biểu thức hằng số. Phép gán này không mất thời gian, mà thay vào đó, một dữ liệu biến giữ giá trị cho đến khi có một phép gán kế tiếp được gán đến nó. Những phép gán khai báo biến đến một mảng là không được phép. Những phép gán khai báo biến chỉ được phép ở mức độ module. Nếu một biến mà được gán hai giá trị khác nhau, một là ở khối initial, một là trong phép gán khai báo biến thì thứ tự của phép gán không được xác định.

Ví dụ 7.7

Ví dụ a: Khai báo một biến thanh ghi 4 bit và gán giá trị giá trị cho nó là 4.

reg[3:0] a = 4'h4;

Phép gán khai báo biến trên tương đương với :

reg[3:0] a;

initial a = 4'h4;

Ví dụ b: Phép gán sau là không được phép

reg[3:0] array [3:0] = 0;

Ví dụ c: Khai báo hai biến có kiểu dữ liệu biến là integer, biến đầu được gán giá trị 0.

integer i =0, j;

Ví dụ d: Khai báo hai biến có kiểu dữ liệu biến là số thực real, gán giá trị lần lượt là 2.5 và 3,000,000.

real r1 = 2.5, n300k = 3E6;

Ví dụ e: Khai báo một biến có kiểu dữ liệu biến là time, một có kiểu dữ liệu biến là realtime với giá trị khởi tạo

time t1 = 25;

realtime rt1 = 2.5;

7.3.2 Phép gán qui trình kín (blocking assignment)

Một phép gán được gọi là phép gán blocking assignment khi mà giá trị bên trái của phép gán này đã được gán bởi biểu thức bên phải (nghĩa là giá trị của biểu thức bên trái đã được xác định –valid) thì phép gán kế tiếp nó trong sequential block (begin-end) mới được thực thi. Hay nói cách khác, phép gán blocking assignment thực thi xong thì phép gán kế tiếp nó trong sequential block (begin-end) mới được thực thi. Tuy nhiên, phép gán blocking assignment bên trong parallel block (fork-join) sẽ không cản trở việc thực hiện các phép gán kế tiếp nó.

Theo phép gán qui trình kín, "=" là toán tử gán. Để điều khiển việc định thời (timing) cho phép gán có thể dùng một điều khiển trì hoãn (delay) (ví dụ: #6) hay một điều khiển sự kiện (ví dụ: @(posedge clk)). Giá trị của biểu thức bên phải phép gán sẽ được gán vào bên trái phép gán. Nếu phép

gán đòi hỏi một giá trị, giá trị này sẽ được gán đến ngay tại thời điểm được khai báo bởi một trong hai điều khiển định thời ở trên.

Toán tử gán "=" dùng trong phép gán qui trinh kín cũng được dùng trong phép gán nối tiếp (phép gán liên tục, continuous assignment).

Ví dụ 7.8 mô tả phép gán qui trình kín:

Ví dụ 7.8

```
rega = 0;
rega[3] = 1; // gán tới a bit
rega[3:5] = 7; // gán tới một phần của mảng
mema[address] = 8'hff; // phép gán đến một phần tử nhớ.
{carry, acc} = rega + regb; // gán đến một phép nối (concatenation)
```

7.3.2.1 Mạch tổ hợp với phép gán qui trình kín

Những phép gán qui trình kín thường được sử dụng trong thiết kế mạch tổ hợp (combination circuit). Tuy nhiên, ta vẫn có thể sử dụng phép gán qui trình kín để mô tả mạch tuần tự (sequential circuit) nhưng không phổ biến.

Khi dùng phép gán qui trình kín để thiết kế mạch tổ hợp, ta cần chú ý những yêu cầu mang tính bắt buộc sau:

- Tất cả các tín hiệu input phải được đặt trong "sensitive list".
- Tín hiệu ngõ ra phải được gán trong tất cả các luồng điều khiển.

Ví dụ 7.9 Mạch tổ hợp chọn kênh, mô tả thiết kế này đã đáp ứng được hai yêu cầu mang tình bắt buộc ở trên.

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

```
    reg f;
    always @(sel or b or c)
        if (sel == 1)
            f= b;
        else
            f = c;
endmodule
```

Điều kiện 1: Ba tín hiệu input là sel, b, c đều nằm trong "sensitive list" đó là @ (sel or b or c)

Điều kiện 2: Tín hiệu output f nằm trong cả hai luồng điều khiển (sel==1) và (sel ==0).

Ta cần chú ý ở đây, nếu trong mô tả thiết kế mạch tổ hợp ta không tuân theo hai điều kiện trên thì thiết kế sẽ không sai về cú pháp nhưng chức năng của thiết kế sẽ không như ta mong muốn.

Ví dụ 7.10

```
module mux (f, g, a, b, c);
    output f, g;
    input a, b, c;
    reg f;
    always @(a or b or c)
        if (a == 1)
            f= b;
        else
            g = c;
endmodule
```

Trong Ví dụ 7.10, điều kiện 1 được đáp ứng tuy nhiên điều kiện 2 đã không được đáp ứng, trong luồng điều khiển đầu tiên (a==1) chỉ có output f được gán, vậy output g không có giá trị xác định, tiếp đến luồng điều khiển kế tiếp (a==0) chỉ có output g được gán, vậy output f không có giá trị xác định. Như vậy đây không thể là một mô tả thiết kế cho mạch tổ hợp bởi vì trong mạch tổ hợp khi tín hiệu input xác định thì tất cả các giá trị output cũng phải xác định.

7.3.3 Phép gán qui trình hở (non-blocking assignment)

Phép gán qui trình hở cho phép các tất cả các phép gán trong khối sequential block (begin-end) được thực thi gán một cách độc lập mà không phụ thuộc vào quá trình gán của phép gán trước chúng. Hay có thể nói cách khác, tất cả các phép gán trong sequential block (begin-end) sẽ được gán đồng thời ngay tại một thời điểm mà không cần quan tâm đến thứ tự cũng như sự phụ thuộc vào các phép gán trước đó.

Theo phép gán qui trình hở “<=” là toán tử gán của phép gán qui trình hở. Để điều khiển việc định thời (timing) cho phép gán có thể dùng một điều khiển trì hoãn (delay) (ví dụ: #6) hay một điều khiển sự kiện (ví dụ: @(posedge clk)). Nếu phép gán đòi hỏi một giá trị từ bên phải phép gán, giá trị này sẽ được gán đến cùng tại thời điểm biểu thức bên phải được xác định. Thứ tự của việc xác định giá trị giữa phép gán và biểu thức bên phải sẽ không được xác định nếu việc điều khiển việc định thời không được mô tả.

Toán tử của phép gán hở “<=” giống như toán tử quan hệ nhỏ hơn hoặc bằng “<=”. Trình biên dịch sẽ quyết định nó thuộc loại toán tử nào dựa vào bối cảnh sử dụng nó. Khi “<=” được dùng trong một biểu thức, nó sẽ được xem như là một toán tử quan hệ, và khi nó được dùng trong phép gán qui trình hở thì nó được xem như là một toán tử gán.

Phép gán qui trình hở được tính theo hai bước:

- Bước 1: Trình mô phỏng xác định các giá trị của biểu thức bên phải của tất cả các phép gán qui trình hở trong khối sequential block (begin-end) và chờ chuẩn bị thực thi phép gán giá trị vừa xác định khi có một điều khiển định thời (#delay, @, wait()) hoặc sau phép gán cuối cùng trong sequential block (begin-end) xảy ra.
- Bước 2: Khi điều khiển định thời (#delay, @, wait()) hoặc sau phép gán cuối cùng trong sequential block (begin-end) xảy ra, trình mô phỏng sẽ gán các giá trị đã được xác định sẵn trong bước 1 vào bên trái các phép gán một cách đồng thời.

Ví dụ 7.11 Mô tả hai bước trong phép gán qui trình hở

```
module evaluates2 (out);
    output out;
    reg a, b, c;
    initial begin
    a = 0;
    b = 1;
    c = 0;
    end
    always c = #5 ~c;
    always @(posedge c) begin
    a <= b; // tính toán, chờ gán
    b <= a; // thực hiện hai phép gán
    end
endmodule
```

Trong Ví dụ 7.11:

Bước 1: Trình mô phỏng xác định các giá trị của biểu thức bên phải của tất cả các phép gán qui trình hở trong khối sequential block (begin-end) và chờ chuẩn bị thực thi phép gán giá trị vừa xác định khi có cạnh lên xung clock. Như vậy tại bước này a vẫn mang giá trị 0, b vẫn mang giá trị 1.

Bước 2: Tại cạnh lên xung clock, trình mô phỏng sẽ gán các giá trị đã được xác định sẵn trong bước 1 vào bên trái các phép gán một cách đồng thời. Như vậy, sau bước này, a được gán giá trị mới từ b có sẵn trước đó là 1 còn b được gán giá trị mới từ a có sẵn trước đó là 0.

Ví dụ 7.12 So sánh phép gán qui trình kín và qui trình hở trong việc khởi tạo giá trị.

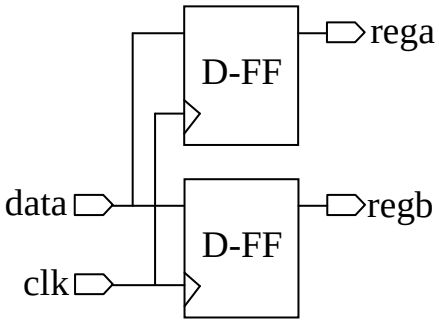
```
//non_block1.v
module non_block1;
    reg a, b, c, d, e, f;
    //phép gán kín
    initial begin
    a = #10 1; // a sẽ được gán = 1 tại t = 10
    b = #2 0; // b sẽ được gán = 0 tại t = 12
    c = #4 1; // c sẽ được gán = 1 tại t =  16
    end
    //phép gán hở
    initial begin
    d <= #10 1; // d sẽ được gán = 1 tại t = 10
    e <= #2 0; // e sẽ được gán = 0 tại t = 2
    f <= #4 1; // f sẽ được gán = 1 tại t = 4
    end
endmodule
```

Ta thấy rằng, khi sử dụng phép gán qui trình hở, do tất cả phép gán chỉ được thực thi đồng thời tại bước 2 do đó thứ tự của các phép gán qui trình hở không quan trong.

Ví dụ 7.13 So sánh phép gán qui trình kín và qui trình hở trong mô tả mạch tuần tự.

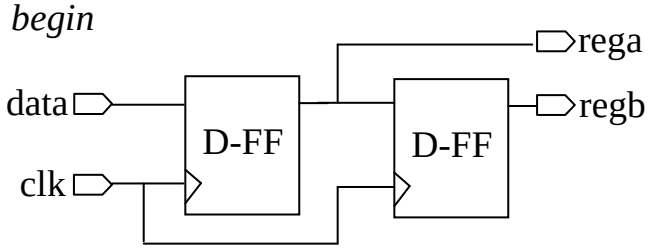
- Ví dụ 3: Blocking assignments

```
always @(posedge clk) begin
rega = data ;
regb = rega ;
end
```



- Blocking assignments

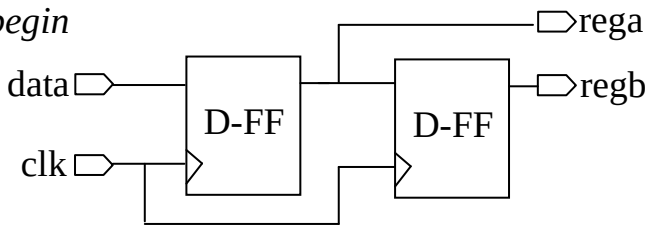
```
always @(posedge clk)   begin
regb = rega ;
rega = data ;
end
```



Trong phép gán qui trinh kín, thì thứ tự giữa các phép gán sẽ ảnh hưởng đến kết quả synthesis của phần cứng tạo ra do trong phép gán qui trinh kín, phép gán đứng sau chỉ được thực thi khi giá trị của phép gán trước nó đã được xác định.

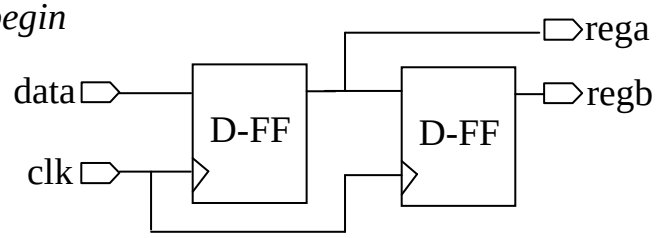
- Non-blocking assignments

```
always @(posedge clk) begin
rega <= data ;
regb <= rega ;
end
```



- Non-blocking assignments

```
always @(posedge clk) begin
regb <= rega ;
rega <= data ;
end
```



Trong phép gán qui trinh hở, thì thứ tự giữa các phép gán sẽ không ảnh hưởng đến kết quả synthesis của phần cứng tạo ra, do trong phép gán qui trinh hở các phép gán được thực thi một cách đồng thời.

Ví dụ 7.14

```
//non_block1.v
module non_block1;
    reg a, b;
    initial begin
    a = 0;
    b = 1;
    a <= b;
    b <= a;
    end
    initial begin
    $monitor ($time, ,"a = %b b = %b", a, b);
    #100 $finish;
    end
endmodule
```

Giá trị cuối cùng của phép gán : a = 1; b =0

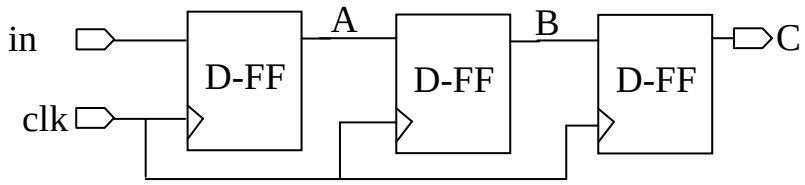
7.3.3.1 Mạch tuần tự với phép gán qui trình hở

Những phép gán qui trình hở thường được sử dụng trong thiết kế mạch tuần tự (sequential circuit).

Ví dụ 7.15 Mô tả thiết kế mạch ghi dịch (shifter)

```
module shifter (in, A,B,C,clk);
    input in, clk;
    input A,B,C;
    reg A, B, C;
    always @ (posedge clk) begin
        B <= A;
        A <= in;
        C <= B;
    end
endmodule
```

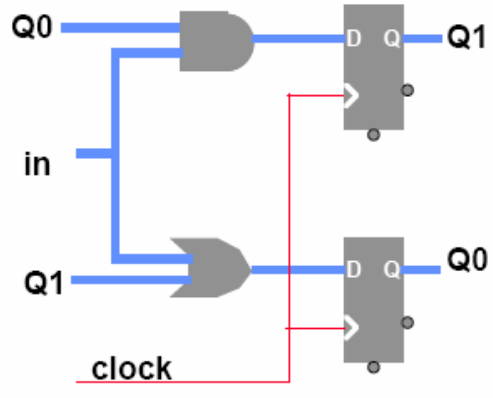
Điều chú ý trong ví dụ trên đó là, thứ tự của các phép gán qui trình hở là không quan trọng.



Hình 7.1 Phần cứng thiết kế của mô tả

Ví dụ 7.16 Mô tả thiết kế một máy trạng thái

```
module fsm (Q1, Q2, in, clock);
        output Q1, Q2;
        input clock, in;
        reg Q1, Q2;
        always @posedge clock) begin
                Q1 <= in & Q0;
                Q0 <= in | Q1;
        end
endmodule
```



7.4 Phát biểu có điều kiện

Phát biểu (expression) có điều kiện (phát biểu if-else) được sử dụng để đưa ra một quyết định xem một phát biểu (statement), một phép gán có được thực thi hay không. Cú pháp của phát biểu có điều kiện như sau

Cú pháp

```
conditional_statement ::=
if ( expression ) statement_or_null [ else statement_or_null ]
statement_or_null ::= statement | ;
```

Theo Cú pháp, nếu biểu thức (expression) được xác định là đúng (nghĩa là một giá trị khác không) thì phát biểu (statement) sẽ được thực thi. Nếu biểu thức (expression) được xác định là sai (nghĩa là một giá trị bằng 0, x hoặc z) thì phát biểu không được thực thi. Nếu trong phát biểu có điều kiện mà có thêm phát biểu else và giá trị của biểu thức (expression) là sai thì phát biểu else sẽ được thực thi.

Bởi vì giá trị số học của biểu thức if sẽ được kiểm tra xem có phải là 0 hay không, biểu thức có thể được viết ngắn gọn.

Ví dụ 7.17 Ba phát biểu sẽ mô tả cùng một logic

```
if (expression)
if (expression !=0)
if (expression == 1)
```

Bởi vì phần else trong phát biểu if-else là tùy chọn, có thể có hoặc không nếu không cần thiết nên có thể gây hiểu lầm khi phần else bị bỏ đi trong các phát biểu mà các mệnh đề if liên tiếp nhau. Để không bị bối rối trong suy nghĩ ta cần nhớ rằng phần else luôn là một phần của mệnh đề if gần nhất với nó. Ta xem xét 3 ví dụ sau:

Ví dụ 7.18

```
if (index > 0)
        if (rega > regb)
                result = rega;
else
        result = regb;
```

Với mô tả thiết kế như trên, do người viết không cẩn thận dẫn đến người đọc dễ hiểu lầm là phát biểu else là một phần của phát biểu if (index >0). Nhưng phát biểu else ở đây là một phần của phát biểu if (rega > regb) vì nó gần nhất với else.

Ví dụ 7.19

```
if (index > 0)
        if (rega > regb)
                result = rega;
        else
                result = regb;
```

Với cách mô tả rõ ràng như trên thì sẽ hạn chế được những hiểu lầm trong thiết kế.

Để kiểm soát chặt chẽ hơn, khi mô tả ta nên thêm begin-end block vào phát biểu if-else.

Ví dụ 7.20

```
if (index > 0)
begin
        if (rega > regb)
                result = rega;
end
else result = regb;
```

Trong trường hợp này, phát biểu else sẽ là một phần của phát biểu if (index > 0)

7.4.1 Cấu trúc If-else-if

Cấu trúc If-else-if rất thường xuất hiện trong mô tả thiết kế. Cú pháp của nó như sau:

Cú pháp

```
if_else_if_statement ::=
if (expression) statement_or_null
{ else if (expression) statement_or_null }
else statement
```

Chuỗi phát biểu if-else-if này là cách phổ biến nhất trong việc mô tả để đưa ra nhiều quyết định. Biểu thức sẽ được tính theo thứ tự, nếu bất kì biểu thức nào là đúng thì phát biểu đi kèm với nó sẽ được thực thi và sau

đó nó sẽ thoát ra khỏi chuỗi phát biểu. Mỗi phát biểu có thể là một phát biểu đơn hay một khối các phát biểu nằm trong begin-end block.

Phần phát biểu else cuối cùng trong cấu trúc if-else-if sẽ được thực thi khi mà không có điều kiện nào đáp ứng được các điều kiện ở trên nó. Phần else thường được sử dụng để tạo ra các giá trị mặc định, hoặc dùng trong quá trình kiểm tra lỗi.

Trong mô tả thiết kế sau sử dụng phát biểu if-else để kiểm tra biến index để đưa ra quyết định xem một trong ba biến thanh ghi *modify_segn* có phải được cộng vào địa chỉ ô nhớ hay không, và việc tăng này phải được cộng vào biến thanh ghi index. Mười dòng đầu tiên khai báo biến thanh ghi và các tham số.

Ví dụ 7.21

```
// khai báo kiểu dữ liệu biến và khai báo tham số
reg [31:0] instruction, segment_area[255:0];
reg [7:0] index;
reg [5:0] modify_seg1, modify_seg2, modify_seg3;
parameter segment1 = 0, inc_seg1 = 1, segment2 = 20, inc_seg2 =
2, segment3 = 64, inc_seg3 = 4, data = 128;
// kiểm tra chỉ số biến
if (index < segment2) begin
        instruction = segment_area [index + modify_seg1];
        index = index + inc_seg1;
end
else if (index < segment3) begin
        instruction = segment_area [index + modify_seg2];
        index = index + inc_seg2;
end
```

```
else if (index < data) begin
        instruction = segment_area [index + modify_seg3];
        index = index + inc_seg3;
end
else
        instruction = segment_area [index];
```

7.5 Phát biểu Case

Phát biểu case là phát biểu tạo ra nhiều sự lựa chọn, nó kiểm tra xem một biểu thức có phù hợp với một biểu thức khác hay không. Cú pháp của phát biểu case như sau:

Cú pháp

```
case_statement ::=
        | case ( expression ) case_item { case_item } endcase
        | casez ( expression ) case_item { case_item } endcase
        | casex ( expression ) case_item { case_item } endcase
case_item ::=
expression { , expression } : statement_or_null
| default [ : ] statement_or_null
```

Phát biểu defaule có thể lựa chọn có hoặc không có. Sử dụng nhiều phát biểu defaut trong một phát biểu case là không hợp lệ.

Một ví dụ đơn giản dùng phát biểu case để giải mã biến thanh ghi rega để tạo ra giá trị cho biến thanh ghi result như sau:

Ví dụ 7.22

```
reg [15:0] rega;
reg [9:0] result;
```

```
case (rega)
        16'd0: result = 10'b0111111111;
        16'd1: result = 10'b1011111111;
        16'd2: result = 10'b1101111111;
        16'd3: result = 10'b1110111111;
        16'd4: result = 10'b1111011111;
        16'd5: result = 10'b1111101111;
        16'd6: result = 10'b1111110111;
        16'd7: result = 10'b1111111011;
        16'd8: result = 10'b1111111101;
        16'd9: result = 10'b1111111110;
        default result = 'bx;
endcase
```

Những biểu thức trong *case item* sẽ được tính toán và so sánh theo thứ tự mà chúng được cho. Theo thứ tự từ trên xuống, nếu một trong các biểu thức trong *case item* phù hợp với biểu thức trong dấu ngoặc đơn () của case thì phát biểu mà kết hợp với *case item* đó sẽ được thực thi. Nếu tất cả các so sánh đều không đúng thì phát biểu mà kết hợp với *default item* sẽ được thực thi. Nếu *default item* không được mô tả trong phát biểu case và tất cà các so sánh đều không đúng thì không có bất kì phát biểu nào được thực thi.

Khác ở cú pháp, phát biểu case khác với cấu trúc if-else-if ở hai điểm quan trọng sau:

- Biểu thức có điều kiện trong if-else-if phổ biến hơn việc so sánh biểu thức với nhiều biểu thức khác trong phát biểu case.
- Phát biểu case cung cấp một kết quả rõ ràng khi biểu thức có giá trị là x hoặc z.

Trong việc so sánh biểu thức của phát biểu case, việc so sánh chỉ thành công khi mọi bit giống nhau chính xác một cách tương ứng theo các giá trị 0, 1, x, và z. Kết quả là, sự cẩn trọng khi mô tả thiết kế sử dụng phát biểu case là cần thiết. Độ dài bit của tất cả các biểu thức sẽ phải bằng nhau để việc so sánh bit-wise giữa các bit có thể được thực hiện. Độ dài của tất cả biểu thức trong *case item* cũng như biểu thức trong () của case phải bằng với độ dài lớn nhất của biểu thức của case và *case item*.

Chú ý: Độ dài mặc định của x và z bằng với độ dài mặc định của một số nguyên (integer).

Lí do của việc cung cấp khả năng so sánh biểu thức của case đó là giúp xử lí những giá trị x và z, điều này cung cấp một cơ chế để phát hiện ra những giá trị này và có thể kiểm soát được thiết kế khi sự xuất hiện của chúng.

Ví dụ sau minh họa việc sử dụng phát biểu case để xử lí những giá trị x và z một cách thích hợp.

Ví dụ 7.23

```
case (select[1:2])
        2'b00: result = 0;
        2'b01: result = flaga;
        2'b0x,
        2'b0z: result = flaga ? 'bx : 0;
        2'b10: result = flagb;
        2'bx0,
        2'bz0: result = flagb ? 'bx : 0;
        default result = 'bx;
endcase
```

Trong Ví dụ 7.23, trong case item thứ 3, nếu select[1] là 0 và flaga là 0 thì select[2] có là x hoặc z thì result sẽ là 0.

Trong Ví dụ 7.24 sẽ minh họa một cách khác để sử dụng phát biểu case để phát hiện x và z

Ví dụ 7.24

```
case (sig)
        1'bz: $display("signal is floating");
        1'bx: $display("signal is unknown");
        default: $display("signal is %b", sig);
endcase
```

7.5.1 Phát biểu Case với "don't care"

Hai loại khác của phát biểu case được cung cấp cho phép xử lí những điều kiện "don't care " trong việc so sánh case. Một đó là xử lí giá trị tổng trở cao (z) như là "don't care", hai đó là xử lí những giá trị tổng trở cao (z) và giá trị không xác định (x) như là "don't care".

Những giá trị "don't care" (giá trị z cho casez, z và x cho casex) trong bất kì bit nào của biểu thức trong case hay trong case item sẽ đều được xem như những điều kiện "don't care" trong suốt quá trình so sánh, và vị trí của bit đó sẽ không được xem xét. Những điều kiện "don't care" trong biểu thức case có thể được điều khiển một cách linh động, bit nào nên được so sánh tại thời điểm nào.

Cú pháp của số học cho phép sử dụng dấu chấm hỏi (?) để thay thế cho z trong những phát biểu case. Điều này cung cấp một định dạng thuận tiện cho việc mô tả những bit "don't care" trong phát biểu case.

Ví dụ 7.25 dùng phát biểu casez. Nó minh họa một mạch giải mã lệnh, ở đó những giá trị có trọng số lớn nhất chọn tác vụ (task) nào cần

được gọi. Nếu bit có trọng số lớn nhất của ir là 1 thì tác vụ instruction1 được gọi mà không cần quan tâm đến giá trị của các bit khác trong ir.

Ví dụ 7.25

```
reg [7:0] ir;
casez (ir)
8'b1???????: instruction1(ir);
8'b01??????: instruction2(ir);
8'b00010???: instruction3(ir);
8'b000001??: instruction4(ir);
endcase
```

Ví dụ 7.26 dùng phát biểu casex. Nó minh họa một case mà ở đó những điều kiện "don't care" được điều khiển một cách linh hoạt trong quá trình mô phỏng. Trong case này, nếu r = 8'b01100110, thì tác vụ stat2 sẽ được gọi.

Ví dụ 7.26

```
reg [7:0] r, mask;
mask = 8'bx0x0x0x0;
casex (r ^ mask)
        8'b001100xx: stat1;
        8'b1100xx00: stat2;
        8'b00xx0011: stat3;
        8'bxx010100: stat4;
endcase
```

7.5.2 Phát biểu case với biểu thức hằng số

Biểu thức hằng số có thể được dùng trong biểu thức của case. Giá trị của biểu thức hằng số sẽ được so sánh với biểu thức của *case item*.

Ví dụ 7.27 Minh họa việc sử dụng mạch mã hóa ưu tiên 3bit

```
reg [2:0] encode ;
case (1)
        encode[2] : $display("Select Line 2") ;
        encode[1] : $display("Select Line 1") ;
        encode[0] : $display("Select Line 0") ;
        default $display("Error: One of the bits expected ON");
endcase
```

Chú ý rằng, biểu thức trong case là một biểu thức hằng số (1). Các case item là biểu thức (bit-selects) và sẽ được so sánh với biểu thức hằng số. Như vậy, chỉ khi một trong các bit của encode bằng 1 thì biểu thức đi kèm với nó mới được thực thi.

Ví dụ 7.28

```
reg [2:0] encode ;
case (0)
        encode[2] : $display("Select Line 2") ;
        encode[1] : $display("Select Line 1") ;
        encode[0] : $display("Select Line 0") ;
        default $display("Error: One of the bits expected ON");
endcase
```

Với ví dụ trên, chỉ khi một trong các bit của encode bằng 0 thì biểu thức đi kèm với nó mới được thực thi.

7.6 Phát biểu vòng lập

Trong Verilog hỗ trợ bốn loại phát biểu lặp vòng. Những phát biểu này cung cấp phương tiện để kiểm soát một phát biểu phải cần thực thi bao nhiêu lần, có thể là một lần, nhiều lần hoặc có thể là không lần.

forever: Thực thi một phát biểu liên tục

repeat: Thực thi một phát biểu với một số lần cố định. Nếu biểu thức có giá trị là không xác định (x) hoặc tổng trở cao, nó sẽ được xem như có giá trị zero và không có phát biểu nào được thực thi.

while: Thực thi một phát biểu cho đến khi một biểu thức trở thành sai (false). Nếu biểu thức có giá trị bắt đầu đã là sai (false) thì phát biểu sẽ không được thực thi lần nào.

for: Điều khiển việc thực thi những phát biểu kết hợp với nó bởi ba bước sau:

- Thực thi một phép gán một cách bình thường dùng để khởi tạo giá trị một thanh ghi để điều khiển số lần cần thực thi lập lại.
- Xác định giá trị biểu thức – nếu kết quả là zero thì vòng lặp for tháo ra, nếu kết quả khác không thì vòng lặp for sẽ thực thi những phát biểu kết hợp với nó, sau đó thực hiện tiếp bước c. Nếu giá trị biểu thức là không xác định (x) hay tổng trở cao (z), nó sẽ được xem như zero.
- Thực thi một phép gán thông thường dùng để cập nhật giá trị cho thanh ghi điều khiển lập vòng, sau đó lập lại bước b.

Cú pháp cho những phát biểu lập vòng:

Cú pháp

```
looping_statements ::=
        forever statement
```

| **repeat** (expression) statement

| **while** (expression) statement

| **for** (reg_assignment ; expression ; reg_assignment) statement

Ví dụ 7.29 Phát biểu repeat – mạch nhân sử dụng toán tử add và shift.

```
parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;
begin : mult
    reg [longsize:1] shift_opa, shift_opb;
    shift_opa = opa;
    shift_opb = opb;
    result = 0;
    repeat (size) begin
        if (shift_opb[1])
            result = result + shift_opa;
        shift_opa = shift_opa << 1;
        shift_opb = shift_opb >> 1;
    end
end
```

Ví dụ 7.30 Phát biểu while đếm số logic 1 của một giá trị trong thanh ghi a.

```
begin : count1s
    reg [7:0] tempreg;
    count = 0;
    tempreg = rega;
    while (tempreg) begin
```

```
            if (tempreg[0])
                count = count + 1;
            tempreg = tempreg >> 1;
        end
    end
```

Ví dụ 7.31 Mô tả thiết kế mạch tạo bit chẵn lẻ, sử dụng vòng lặp while.

```
    `timescale 1ns/100ps
    module parity_gen (a, p);
    parameter SIZE = 8;
    input [SIZE-1:0] a;
    output reg p;
    reg im_p;
    integer indx;
    always @( a ) begin
        im_p = 0;
        indx = 0;
        while (indx < SIZE) begin
            im_p = im_p ^ a[indx];
            indx = indx + 1;
        end
        p = im_p;
    end
    endmodule
```

Ví dụ 7.32 Vòng lập for: Phát biểu for sẽ cho kết quả như pseudo-code sau mà dựa trên vòng lặp while.

```
    begin
        initial_assignment;
```

```
        while (condition) begin
            statement
            step_assignment;
        end
    end
```

Vòng lặp for thực thi cùng chức năng trên nhưng chỉ cần hai dòng như "pseudo code" sau:

```
    for (initial_assignment; condition; step_assignment)
        statement
```

Ví dụ 7.33 Mô tả thiết kế mạch tổ hợp priority-encoder dùng vòng lặp for

```
`timescale 1ns/100ps
module priority_encoder (i0, i1, i2, i3, y0, y1, f);
    input i0, i1, i2, i3;
    output reg y1, y0, f;
    wire [3:0] im = { i3, i2, i1, i0 };
    reg [2:0] indx;
    always @(im) begin
        { y1, y0 } = 2'b00;
        f = 1'b0;
        for (indx=0; indx<4; indx=indx+1) begin
            if ( im[indx] ) begin
                { y1, y0 } = indx;
                f = 1'b1;
            end
        end
    end
endmodule
```

7.7 Điều khiển định thời (procedural timing controls)

Verilog HDL hỗ trợ ba phương pháp điều khiển định thời tường minh khi những phát biểu qui trình xuất hiện. Loại đầu tiên đó là điều khiển trì hoãn "delay control", loại thứ hai đó là điều khiển sự kiện "event control". Loại thứ ba đó là phát biểu "wait".

- Một điều khiển trì hoãn, được nhận diện bắt đầu với kí hiệu #
- Một điều khiển sự kiện, được nhận diện bắt dầu với kí hiệu @
- Phát biểu wait, hoạt động của nó là sự kết hợp giữa điều khiển sự kiện và vòng lặp while nhưng đối lập về chức năng.

Ví dụ 7.34

```
#150 regm = regn;
@(posedge clock) regm = regn;
@(a, b, c)   y = (a & b) | (b & c) | (a & c);
```

Việc mô tả thời gian trì hoãn cho cổng và net sử dụng trong mô phỏng đã được đề cập trong các phần trên, trong phần này chỉ thảo luận về ba phương pháp điều khiển định thời trong các phép gán qui trình.

7.7.1 Điều khiển trì hoãn (delay control)

Một phát biểu qui trình theo sau một điều khiển trì hoãn sẽ bị trì hoãn việc thực thi một khoảng thời gian được mô tả trong điều khiển trì hoãn. Nếu biểu thức trì hoãn có giá trị là không xác định hoặc tổng trở cao, nó sẽ được xem như có độ trì hoãn bằng 0. Nếu biểu thức trì hoãn có giá trị âm, nó sẽ xem như đó là khoảng thời gian có giá trị số nguyên không dấu bù 2.

Ví dụ 7.35

Ví dụ 1: Thực thi phép gán sau 10 đơn vị thời gian

 # 10 rega = regb;

Ví dụ 2: Biểu thức trì hoãn

#d rega = regb; // d được định nghĩa như là một tham số

#((d+e)/2) rega = regb;// độ trì hoãn là giá trị trung bình của d và e

#regr regr = regr + 1; // độ trì hoãn là giá trị trong regr

7.7.2 Điều khiển sự kiện (event control)

Việc thực thi một phát biểu qui trình có thể được đồng bộ với sự thay đổi giá trị trên một net hay một thanh ghi hoặc sự xuất hiện của một khai báo sự kiện. Những sự thay đổi giá trị trên net hay trên thanh ghi có thể được xem như một sự kiện dùng để kích hoạt sự thực thi của một phát biểu. Điều này giống như việc dò tìm một sự kiện không tường minh. Sự kiện có thể được dựa trên hướng của sự thay đổi đó là hướng lên 1 (posedge) hay hướng xuống 0 (negedge). Verilog HDL hỗ trợ ba loại điều khiển sự kiện:

- Sự kiện có thể được dò tìm khi có bất kì sự chuyển trạng thái nào xảy ra trên net hoặc thanh ghi. Được mô tả bởi **@** (net) hay **@** (reg).
- Sự kiện negedge có thể được dò tìm khi có sự chuyển trạng thái từ 1 xuống x, z, hoặc 0, và từ x hoặc z xuống 0. Được mô tả bởi **@** (**negedge** net) hay **@** (**negedge** reg)
- Sự kiện posedge có thể được dò tìm khi có sự chuyển trạng thái từ 0 lên x, z, hoặc 1, và từ x hoặc z lên 1. Được mô tả bởi **@** (**posedge** net) hay **@** (**posedge** reg)

From \ To	0	1	x	z
0	No edge	posedge	posedge	posedge
1	negedge	No edge	negedge	negedge
x	negedge	posedge	No edge	No edge
z	negedge	posedge	No edge	No edge

Nếu giá trị của biểu thức nhiều hơn 1 bit, sự chuyển trạng thái cạnh sẽ được dò tìm trên bit có trọng số thấp nhất của giá trị đó. Sự thay đổi giá trị trong bất kì toán hạng nào mà không có sự thay đổi giá trị trên bit có trọng số thấp nhất của biểu thức thì sự chuyển trạng thái cạnh không thể được dò thấy.

Ví dụ 7.36 Minh họa những phát biểu điều khiển sự kiện

@r rega = regb; // được điều khiển bởi bất kì sự thay đổi giá trị trên thanh ghi r.

@(a, b, c) rega = regb; // tương đương với @(a ***or*** *b* ***or*** *c) rega = regb*

// được điều khiển bởi bất kì sự thay đổi nào của các tín hiệu a, b hoặc c.

*@(****posedge*** *clock) rega = regb; // được điều khiển bởi cạnh lên xung clock.*

*@(****posedge*** *clk_a* ***or posedge*** *clk_b* ***or*** *trig) rega = regb;*

// được điều khiển bởi cạnh lên tín hiệu clk_a hoặc cạnh xuống tín hiệu clk_b hoặc có bất kì sự thay đổi nào xảy ra trên tín hiệu trig.

forever *@(****negedge*** *clock) rega = regb; // được điều khiển bởi cạnh xuống xung clock*

Ví dụ 7.37

```
`timescale 1ns/100ps
module maj3 (input a, b, c, output reg y);
always
        @(a, b, c)  // tương đương với @(a or b or c)
        begin
            y = (a & b) | (b &c) | (a & c);
        end
endmodule
Ví dụ trên có thể được mô tả gọn hơn như sau:
module maj3 (input a, b, c, output reg y);
always @(a, b, c) begin    // tương đương với @(a or b or c)
    y = (a & b) | (b &c) | (a & c);
end
endmodule
```

Do mô tả thiết kế trên chỉ có một phép gán qui trình, nên ở đây ta không cần dùng khối **begin-end**. Ví dụ trên có thể được rút gọn hơn như sau:

Ví dụ 7.38

```
`timescale 1ns/100ps
module maj3 (input a, b, c, output reg y);
    always
        @(a, b, c)
        y = (a & b) | (b & c) | (a & c);
    // có thể viết như sau: @(a, b, c)        y = (a & b) | (b & c) | (a & c);
endmodule
```

Trong Ví dụ 7.38, điều khiển sự kiện được đặt trước phát biểu để hình thành một phát biểu qui trình bằng cách bỏ đi dấu chấm phẩy. Điều này có nghĩa dấu chấm phẩy sau @(a, b, c) có thể có hoặc không. Khi hai phát biểu trên được ghép lại thì chỉ một phát biểu được thực thi.

7.7.3 Phát biểu "wait"

Việc thực thi một phát biểu qui trình có thể được trì hoãn cho đến khi một điều kiện trở thành đúng (true). Điều này đạt được bằng sử dụng phát biểu **wait**, đây là một dạng đặc biệt của điều khiển sự kiện. Mặc định của phát biểu **wait** là tích cực mức, điều này trái ngược với phát biểu điều khiển sự kiện là tích cực cạnh.

Phát biểu wait sẽ tính giá trị của điều kiện, nếu giá trị sai (false), những phát biểu qui trình theo sau nó sẽ bị đóng lại không thực thi cho đến khi giá trị đó trở thành đúng (true) thì mới thực thi những phát biểu qui trình đó và thoát ra khỏi phát biểu wait để tiếp tục các phát biểu kế tiếp, điều này đối lập với hoạt động của phát biểu vòng lặp while, trong phát biểu vòng lặp while, giá trị của điều kiện nếu đúng (true) thì những phát biểu qui trình theo sau nó sẽ thực thi lặp lại liên tục trong vòng lặp cho đến khi giá trị của điều kiện trở thành sai (false) thì thoát ra khỏi vòng lặp while để tiếp tục các phát biểu kế tiếp. Cú pháp của những phát biểu wait được mô tả như sau:

Cú pháp

```
wait_statement::=
        wait (expression) statement_or_null
```

Ví dụ 7.39

```
begin
```

```
        wait (!enable) #10 a = b;
        #10 c = d;
    end
```

Nếu giá trị enable là 1, phát biểu wait sẽ trì hoãn việc thực thi của phát biểu kế tiếp nó (#10 a = b;) cho đến khi giá trị của enable là 0. Nếu enable đã có giá trị sẵn là 0 khi khối begin-end bắt đầu thì phép gán "a = b;" sẽ được gán sau khoảng trì hoãn 10 đơn vị thời gian và không có thêm trì hoãn nào xuất hiện.

Ví dụ 7.40

```
    always
    begin
        wait (var1 ==1);
            a = b;
    end
```

Ví dụ 7.40 mô tả một thiết kế thực hiện chức năng khi var bằng 1 thì a sẽ liên tục cập nhật giá trị từ b.

Ví dụ 7.41

```
    always
    begin
        @var
        wait (var1 ==1);
            a = b;
    end
```

Ví dụ trên mô tả một thiết kế thực hiện chức năng khi var chuyển trạng thái lên 1 thì a sẽ cập nhật giá trị từ b. Tương tự như mô tả sau:

```
always @var1
    if (var1 ==1)
        a = b;
```

7.8 Phát biểu khối

Trong khai báo qui trình, Verilog HDL có hỗ trợ việc phát biểu khối. Phát biểu khối là việc nhóm hai hay nhiều phát biểu cùng với nhau để chúng có thể hoạt động theo cùng cú pháp như là một phát biểu đơn. Có hai loại khối trong Verilog HDL.

- Khối tuần tự, hay còn được gọi là khối begin-end
- Khối song song, hay còn được gọi là khối fork-join

Khối tuần tự được giới hạn bởi hai từ khóa begin và end. Những phát biểu qui trình trong khối tuần tự sẽ được thực thi một cách tuần tự theo thứ tự đã được cho.

Khối song song được giới hạn bởi hai từ khóa fork và join. Những phát biểu qui trình trong khối song song sẽ được thực thi một cách đồng thời

7.8.1 Khối tuần tự

Một khối tuần tự sẽ có những đặc điểm sau:

- Những phát biểu được thực thi theo thứ tự, cái này xong đến cái kia.
- Giá trị trì hoãn của một phát biểu là thời gian mô phỏng việc thực thi của phát biểu ngay trước phát biểu hiện tại.
- Điều khiển sẽ thoát ra khỏi khối sau khi phát biểu cuối cùng được thực thi.

Ví dụ 7.42

```
begin
        areg = breg;
        creg = areg; // creg lưu trữ giá trị của breg
end
```

Ví dụ 7.43 Điều khiển trì hoãn có thể được dùng trong khối tuần tự để phân biệt hai phép gán theo thời gian

```
begin
    areg = breg;
    @(posedge clock) creg = areg; // phép gán bị trì hoãn cho đến
end                               //  khi có cạnh lên xung clock.
```

Ví dụ 7.44 Kết hợp khối tuần tự và điều khiển trì hoãn có thể được sử dụng để mô tả dạng sóng.

```
parameter d = 50;
reg [7:0] r;
begin // một dạng sóng được điều khiển bởi thời gian trì hoãn tuần
tự
#d r = 'h35;
#d r = 'hE2;
#d r = 'h00;
#d r = 'hF7;
#d -> end_wave;//kích hoạt một sự kiện có tên là end_wave
end
```

7.8.2 Khối song song (fork-join)

Một khối song song có những đặc điểm sau:

- Những phát biểu sẽ được thực thi đồng thời

- Giá trị trì hoãn của mỗi phát biểu là thời gian mô phỏng được tính từ khi luồng điều khiển bước vào khối fork-join cho đến phát biểu đó.
- Điều khiển trì hoãn được sử dụng để cung cấp thời gian tuần tự cho các phép gán.
- Điều khiển sẽ thoát ra khỏi khối khi phát biểu có thứ tự cuối cùng theo thời gian (giá trị trì hoãn lớn nhất) được thực thi.

Những điều khiển định thời trong khối fork-join không phải theo thứ tự tuần tự theo thời gian.

Ví dụ 7.45 sẽ mô tả dạng sóng giống như trong ví dụ trên nhưng dùng khối song song thay vì dùng khối tuần tự. Dạng sóng được tạo ra trên một thanh ghi sẽ giống nhau trong cả hai trường hợp.

Ví dụ 7.45

```
fork
#50 r = 'h35;
#200 r = 'hF7; // không cần theo thứ tự
#100 r = 'hE2;
#250 -> end_wave; // điều khiển thoát ra
#150 r = 'h00;
join
```

7.8.3 Tên khối

Cả hai khối tuần tự và song song đều có thể được đặt tên bằng cách thêm : name_of_block ngay sau từ khóa begin hay fork.

Tên của khối phục vụ cho các mục đích sau:

- Nó cho phép những thanh ghi nội được khai báo cho khối

- Nó cho phép khối được tham chiếu trong những phát biểu chẳng hạn như phát biểu disable.

Tất cả các thanh ghi là tĩnh, nghĩa là một vị trí duy nhất tồn tại trong tất cả các thanh ghi và rời khối hoặc tiến vào khối sẽ không ảnh hưởng đến giá trị chứa trong nó.

Tên khối sẽ là tên định dạng duy nhất cho tất cả các thanh ghi tại bất kì thời điểm mô phỏng nào.

7.9 Cấu trúc qui trình

Tất cả qui trình trong Verilog HDL phải đều được mô tả bên trong một trong bốn cấu trúc sau:

- Cấu trúc initial
- Cấu trúc always
- Task
- Function

Cấu trúc initial và always được cho phép hoạt động tại thời điểm bắt đầu mô phỏng. Cấu trúc initial chỉ thực thi một lần và hoạt động của nó sẽ ngừng khi phát biểu hoàn thành. Trái lại, cấu trúc always sẽ thực thi lập lại, hoạt động của nó sẽ chỉ ngừng khi quá trình mô phỏng bị thoát ra. Điều này không ngụ ý đến thứ tự của việc thực thi giữa hai cấu trúc intial và always. Cấu trúc initial không cần lên kế hoạch trước và cần thực thi trước cấu trúc always. Ở đây cũng không có sự giới hạn nào về số lượng cấu trúc initial và always có thể được mô tả trong một module.

Tác vụ (task) và hàm (function) sẽ được cho phép hoạt động từ một hoặc nhiều vị trí trong những qui trình khác nhau. Task và function sẽ được thảo luận chi tiết trong chương sau.

7.9.1 Cấu trúc initial

Cú pháp của cấu trúc initial

initial_construct::=

initial statement

Ví dụ 7.46 Dùng cấu trúc initial cho việc khởi tạo biến tại thời điểm bắt đầu mô phỏng.

```
initial begin
    areg = 0; // khởi tạo một thanh ghi
    for (index = 0; index < size; index = index + 1)
        memory[index] = 0; //khởi tạo giá trị cho phần tử nhớ
end
```

Một công dụng thường gặp của cấu trúc initial đó là mô tả dạng sóng để chạy mô phỏng cho thiết kế

Ví dụ 7.47

```
initial begin
inputs = ‘b000000; //khởi tạo tại thời điểm zero
#10 inputs = ‘b011001; // 10 đơn vị thời gian đầu
#10 inputs = ‘b011011; // 10 đơn vị thời gian thứ hai
#10 inputs = ‘b011000; // 10 đơn vị thời gian thứ ba
#10 inputs = ‘b001000; // 10 đơn vị thời gian cuối
end
```

7.9.2 Cấu trúc always

Cấu trúc always cho phép lập lại liên tục trong suốt quá trình mô phỏng.

Cú pháp của cấu trúc always

```
always_construct ::=
        always statement
```

Cấu trúc always do đặc tính vòng lặp tự nhiên của nó nên chỉ hữu dụng khi được dùng kết hợp với một số dạng điều khiển định thời khác. Nếu một cấu trúc always không có điều khiển định thời, nó sẽ tạo ra một vòng lặp vô hạn. Ví dụ sau sẽ tạo ra một vòng lặp vô hạn không trì hoãn

always areg = ~areg;

Cung cấp một điều khiển định thời vào đoạn mô tả trên sẽ tạo ra một mô tả hữu dụng như ví dụ sau:

always #half_period areg = ~areg;

7.10 Máy trạng thái (state machine)

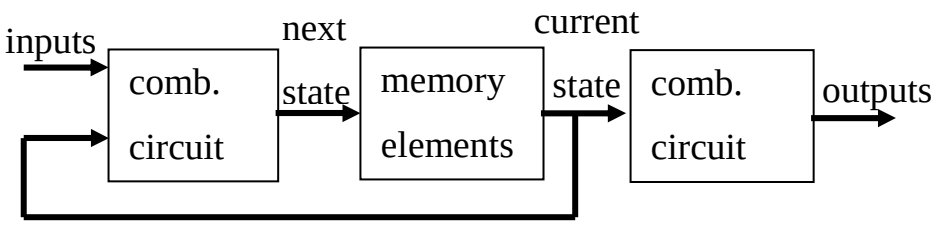
Máy trạng thái là máy trong đó tất cả các output được đồng bộ với xung clock của hệ thống.

Sử dụng những cấu trúc mô tả Verilog HDL mà ta đã thảo luận cho đến lúc này có thể mô tả cho bất kì loại máy trạng thái nào. Trong phần này, ta sẽ thảo luận để có thể dùng những cấu trúc Verilog đã học mô tả một cách tường minh, khoa học một máy trạng thái hữu hạn. Ta bắt đầu với mô hình máy trạng thái Moore, sau đó là mô hình máy trạng thái Mealy.

7.10.1 Máy trạng thái Moore

Máy trạng thái hữu hạn Moore là máy trong đó ngõ ra của hệ thống chỉ phụ thuộc vào trạng thái hiện tại của hệ thống chứ không phụ thuộc vào

ngõ vào hệ thống. Hệ thống máy trạng thái hữu hạn Moore được mô tả như hình dưới:



Hình 7.2 Hệ thống máy trạng thái hữu hạn Moore

Với sơ đồ khối trên, ta có thể mô tả ngắn gọn nguyên lí máy trạng thái Moore như sau:

Next state = F (current state, inputs)

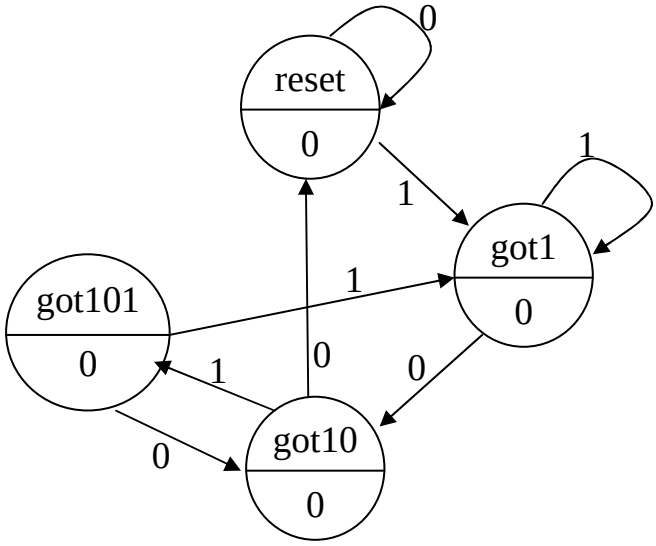
Outputs = G (current state)

Dựa vào sơ đồ khối trên, để mô tả một cách tường minh một máy trạng thái hữu hạn Moore dùng Verilog HDL, ta sẽ mô tả như sau:

- Khối mạch tổ hợp dùng cấu trúc always để kiểm soát sự chuyển đổi giữa các trạng thái để tạo ra trạng thái kế tiếp.
- Khối mạch tuần tự dùng để tạo ra trạng thái hiện tại.
- Khối mạch tổ hợp khác dùng phép gán assign để tạo ra giá trị ngõ ra của hệ thống.

Hình 7.3 mô tả lưu đồ máy trạng thái hữu hạn Moore có chức năng dò tìm chuỗi 101 liên tục dùng Verilog HDL. Hệ thống máy trạng thái dò tìm chuỗi 101 liên tục từ ngõ vào của nó, khi chuỗi 101 được phát hiện thì ngõ ra sẽ lên 1 và duy trì giá trị này cho đến hết một chu kì xung clock. Như được mô tả trong lưu đồ máy trạng thái, khi máy đạt đến trạng thái got101 thì ngõ ra sẽ bật lên 1.

Ví dụ 7.48 mô tả code Verilog cho hệ thống này. Chúng ta sử dụng một khai báo parameter để gán giá trị đến các máy trạng thái. Máy trạng thái của ta có bốn trạng thái do đó cần sử dụng 2 bit để khai báo trạng thái.



Hình 7.3 Lưu đồ máy trạng thái hữu hạn Moore có chức năng dò tìm chuỗi 101 liên tục.

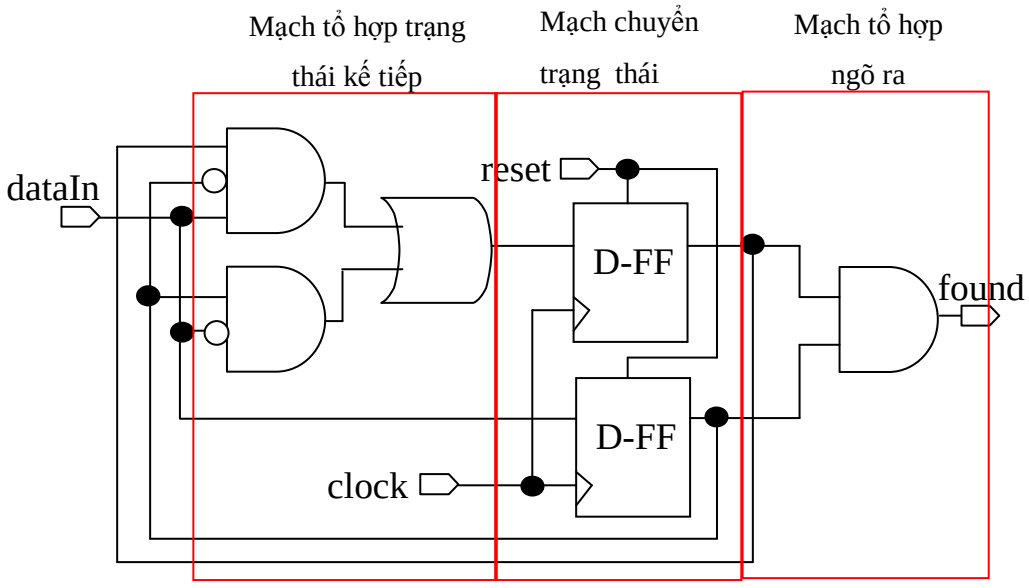
Ví dụ 7.48 Mô tả thiết kế máy trạng thái Moore

```
module Moore101Detector (dataIn, found, clock, reset);
//Khai báo cổng ngõ vào, ngõ ra
  input          dataIn;
  input          clock;
  input          reset;
  output         found;
//Khai báo biến nội
  reg [1:0]    state;
  reg [1:0]    next_state;
//Khai báo trạng thái
  parameter reset = 2'b00;
  parameter got1   = 2'b01;
```

```
    parameter got10   = 2'b10;
    parameter got101   = 2'b11;
  //Mạch tổ hợp của trạng thái kế tiếp
  always @(state or dataIn)
      case (state)
          reset:
          if (dataIn)
              next_state = got1;
          else
              next_state = reset;
      got1:
          if (dataIn)
              next_state = got1;
          else
              next_state = got10;
      got10:
          if (dataIn)
              next_state = got101;
          else
              next_state = reset;
      got101:
          if (dataIn)
              next_state = got1;
          else
              next_state = got10;
      default:
              next_state = reset;
```

```
    endcase // case(state)
  //Mạch chuyển trạng thái
  always @(posedge clock)
      if (reset == 1)
          state <= reset;
      else
          state <= next_state;
  //Mạch tổ hợp ngõ ra
  assign found = (state == got101) ? 1: 0;
  endmodule
```

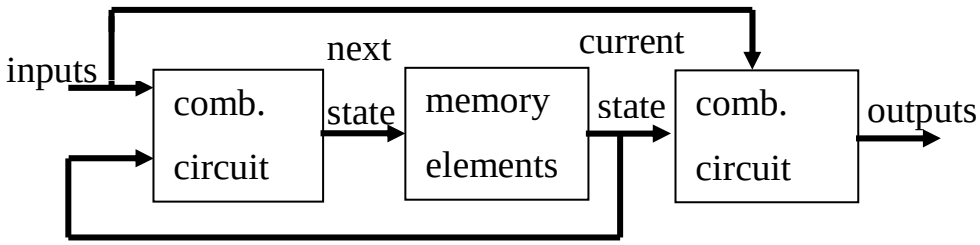
Mạch sau khi được tổng hợp sẽ được như trên Hình 7.4. Mạch sẽ gồm ba phần đúng như hệ thống máy trạng thái Moore: mạch tổ hợp trạng thái kế tiếp, mạch chuyển trạng thái và mạch tổ hợp ngõ ra, trong đó giá trị ngõ ra chỉ phụ thuộc vào trạng thái hiện tại.



Hình 7.4 Mạch phát hiện chuỗi 101 sau tổng hợp sử dụng FSM Moore

7.10.2 Máy trạng thái Mealy

Máy trạng thái hữu hạn Mealy là máy trong đó ngõ ra của hệ thống phụ thuộc vào cả trạng thái hiện tại của hệ thống và ngõ vào của hệ thống. Hệ thống máy trạng thái hữu hạn Mealy được mô tả như hình dưới:



Hình 7.5 Hệ thống máy trạng thái hữu hạn Mealy

Với sơ đồ khối trên, ta có thể mô tả ngắn gọn nguyên lí máy trạng thái Mealy như sau:

Next state = F (current state, inputs)

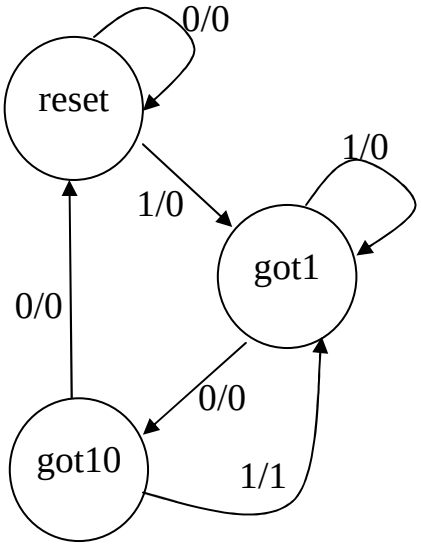
Outputs = G (current state, inputs)

Dựa vào sơ đồ khối trên, để mô tả một cách tường minh một máy trạng thái hữu hạn Mealy dùng Verilog HDL, ta sẽ mô tả như sau:

- Khối mạch tổ hợp dùng cấu trúc always để kiểm soát sự chuyển đổi giữa các trạng thái để tạo ra trạng thái kế tiếp.
- Khối mạch tuần tự dùng để tạo ra trạng thái hiện tại.
- Khối mạch tổ hợp khác dùng phép gán assign để tạo ra giá trị ngõ ra của hệ thống.

Hình 7.6 mô tả lưu đồ máy trạng thái hữu hạn Mealy có chức năng dò tìm chuỗi 101 liên tục dùng Verilog HDL (giống như minh họa trong máy trạng thái Moore). Hệ thống máy trạng thái dò tìm chuỗi 101 liên tục từ ngõ vào của nó, khi chuỗi 101 được phát hiện thì ngõ ra sẽ lên 1 và duy trì giá trị này cho đến hết một chu kì xung clock. Như được mô tả trong lưu đồ máy trạng thái, khi máy đạt đến trạng thái got101 thì ngõ ra sẽ bật lên 1.

Ví dụ 7.49 mô tả code Verilog cho hệ thống trên. Chúng ta sử dụng một khai báo parameter để gán giá trị đến các máy trạng thái. Máy trạng thái của ta có ba trạng thái nên cần sử dụng 2 bit để khai báo trạng thái.



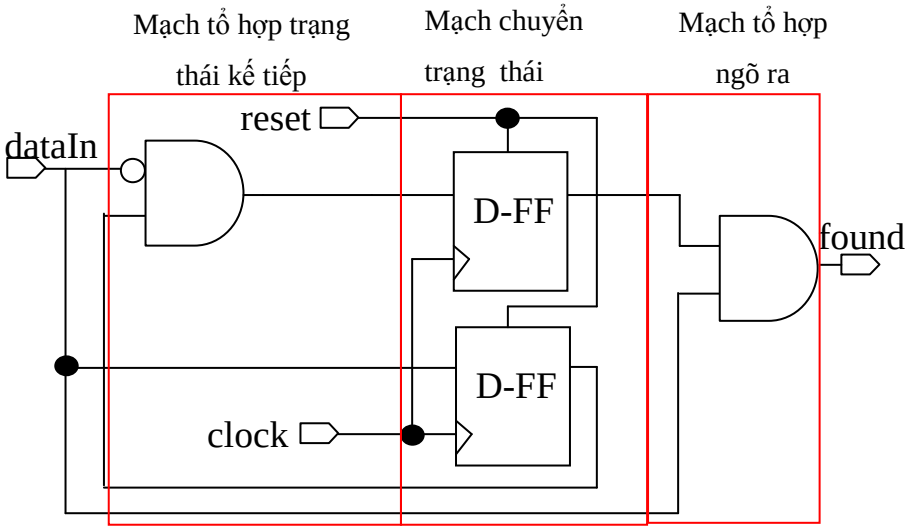
Hình 7.6 Lưu đồ máy trạng thái hữu hạn Mealy có chức năng dò tìm chuỗi 101 liên tục.

Ví dụ 7.49

```
module Mealy101Detector (dataIn, found, clock, reset);
//Khai báo cổng ngõ vào, ngõ ra
  input          dataIn;
  input          clock;
  input          reset;
  output         found;
//Khai báo biến dữ liệu nội
  reg [1:0]    state;
  reg [1:0]    next_state;
//Khai báo trạng thái
  parameter reset = 2'b00;
  parameter got1   = 2'b01;
  parameter got10   = 2'b10;
```

```
//Khai báo mạch tổ hợp trạng thái kế tiếp
always @(state or dataIn)
    case (state)
        reset:
            if (dataIn)
                next_state = got1;
            else
                next_state = reset;
        got1:
            if (dataIn)
                next_state = got1;
            else
                next_state = got10;
        got10:
            if (dataIn)
                next_state = got1;
            else
                next_state = reset;
        default:
            next_state = reset;
    endcase // case(state)
//Mạch chuyển trạng thái
always @(posedge clock)
    if (reset == 1)
        state <= reset;
    else
        state <= next_state;
```

```
        //Mạch tổ hợp ngõ ra
        assign found = (state == got10 && dataIn == 1) ? 1: 0;
endmodule // Mealy101
```



Hình 7.7 Mạch phát hiện chuỗi 101 sau tổng hợp sử dụng FSM Mealy

Hình 7.7. Mạch gồm ba phần đúng như hệ thống máy trạng thái Mealy: mạch tổ hợp trạng thái kế tiếp, mạch chuyển trạng thái và mạch tổ hợp ngõ ra, trong đó ngõ ra phụ thuộc vào giá trị ngõ vào và trạng thái hiện tại.

7.11 Bài tập

1. Nêu sự khác biệt giữa mô hình cấu trúc và mô hình hành vi trong mô tả phần cứng Verilog HDL. Nêu ưu điểm và khuyết điểm của từng mô hình.
2. Nêu sự khác biệt giữa phép gán qui trình kín và phép gán qui trình hở.
3. Trong mô hình hành vi, nêu sự khác biệt giữa phép gán liên tục (còn gọi là continuous assignment hay phép gán RTL) với phép gán qui trình.

4. Tại sao ta phải sử dụng hàm và tác vụ trong mô tả phần cứng Verilog HDL?
5. Nêu sự khác biệt giữa hàm, tác vụ và module trong mô tả phần cứng Verilog HDL.
6. Nêu những ràng buộc khi sử dụng phép gán qui trình kín trong mô tả mạch tổ hợp.
7. Nêu sự khác biệt giữa phát biểu *case* và *casex*.
8. Nêu sự khác nhau giữa máy trạng thái Moore và máy trạng thái Mealy trong mô tả phần cứng Verilog HDL. Nêu ưu điểm và khuyết điểm của mỗi loại. Trình bày sự tương ứng giữa các phần trong mạch phần cứng và các phần trong mô tả Verilog khi sử dụng phương pháp máy trạng thái Moore hoặc Mealy.
9. Nêu những phương pháp điều khiển định thời trong mô tả phần cứng Verilog HDL ?
10. Có mấy loại cấu trúc qui trình trong mô tả phần cứng Verilog HDL ? Nêu sự khác biệt giữa cấu trúc initial và cấu trúc always ? Tại sao cấu trúc initial không thể tổng hợp ra mạch phần cứng ?
11. Có mấy loại phát biểu khối (block assignment) trong mô tả phần cứng Verilog HDL ? Nêu sự khác biệt giữa chúng.
12. Viết lại đoạn mô tả thiết kế sử dụng phép gán hở cho đoạn mô tả thiết kế sau để chức năng không đổi

```
initial
begin
    a = #delay1 b;
    c = #delay2 d;
end
```

13. Thay thế ba khối initial như dưới bởi một khối duy nhất

```
initial
```

```
        a = #delay1 b;
initial
        c = #delay2 d;
initial
begin
        e <= #delay3 f;
        k <= #delay4 g;
end
```

14. Đoạn code sau thực hiện chức năng gì? Viết lại đoạn mô tả lại sử dụng phép gán kín

```
always @(posedge clock) begin
        a <= b;
        b <= a;
end
```

15. Đoạn code sau thực hiện chức năng gì? Viết lại đoạn mô tả lại sử dụng phép gán kín

```
always @(posedge clock)
        #0 a <= b + c;
always @(posedge clock)
        b <= a;
```

16. Vẽ dạng sóng của tín hiệu d trên toàn bộ thời gian mô phỏng

```
`timescale 1ns/100ps
module test;
reg b,c,d;
initial begin
        b=1'b1;
        c=1'b0;
        #10 b=1'b0;
end
```

```
initial d = #25(b|c);
endmodule
```

17. Vẽ dạng sóng của a, b và c trong 100ns đầu tiên của quá trình mô phỏng.

```
module test;
wire a, b;
reg c;
assign #60 a = 1 ;
initial begin
        #20 c = b;
        #20 c = a;
        #20;
end
endmodule
```

18. Kiểm tra xem đoạn mô tả sau có thực hiện chức năng tìm giá trị lớn nhất được không?

```
reg [3:0] mx;
reg [3:0] lst [0:3];
always @ (lst[0] or lst[1] or lst[2] or lst[3]) begin
     mx <= 4b'0000;
     if ( lst[0] >= mx ) mx <= lst[0];
     if ( lst[0] >= mx ) mx <= lst[1];
    if ( lst[0] >= mx ) mx <= lst[2];
    if ( lst[0] >= mx ) mx <= lst[3];
end
```

19. Kiểm tra xem hai đoạn mô tả thiết kế sau có cùng chức năng hay không?

Mô tả 1:

```
module addr_dcd (addr_in, decoded_addr);
parameter ADDRESS = 8;
parameter RAMSIZE = 256;
input [ADDRESS – 1 : 0] addr_in;
output      [RAMSIZE - 1 : 0] decoded_addr;
integer     i;
reg   [RAMSIZE – 1 : 0] decoded_addr;
always @ (addr_in) begin
    for( i=0; i < RAMSIZE ; i=i+1)
        decoded_addr[ i ] = (addr_in == i);
end
endmodule
```

Mô tả 2:

```
module addr_dcd (addr_in, decoded_addr);
parameter ADDRESS = 8;
parameter RAMSIZE = 256;
input [ADDRESS – 1 : 0] addr_in;
output      [RAMSIZE - 1 : 0] decoded_addr;
integer     i;
reg   [RAMSIZE – 1 : 0] decoded_addr;
always @ (addr_in) begin
    for( i=0; i < RAMSIZE ; i=i+1)
        decoded_addr[ i ] <= (addr_in == i);
end
endmodule
```

Chương 8. Tác vụ (task) và hàm (function)

Task và function cung cấp khả năng thực thi các thủ tục chung từ nhiều nơi khác nhau trong một mô tả thiết kế. Chúng cung cấp một phương tiện để chia nhỏ những mô tả thiết kế lớn, phức tạp thành những phần nhỏ hơn để dễ dàng trong việc đọc và gỡ rối các mô tả thiết kế nguồn.

Như ta đã biết, trong mô hình cấu trúc ta có thể gọi (instantiate) một sub-module thực hiện một chức năng nào đó ra sử dụng ở bất kì đâu mà không cần phải mô tả lại thiết kế của module đó. Tuy nhiên, trong mô hình hành vi (behavioural model), ta không thể gọi module ra giống như vậy được. Do đó, để giải quyết yêu cầu là có thể sử dụng một mô tả chức thiết kế có chức năng nào đó nhiều lần trong mô hình hành vi mà không cần phải mô tả lại thì ta sẽ sử dụng hàm (function) hoặc tác vụ (task).

Phần này sẽ thảo luận sự khác nhau giữa task và function, mô tả cách định nghĩa và gọi task và function, các ví dụ mô tả cho mỗi phần.

8.1 Phân biệt giữa tác vụ (task) và hàm (function)

Các quy tắc sau đây phân biệt task và function:

- Một function sẽ thực hiện trong một đơn vị thời gian mô phỏng, task có thể bao gồm câu lệnh điều khiển thời gian.
- Một function không thể kích hoạt một task, một task có thể kích hoạt các task và function khác nhau.
- Một function phải có ít nhất một đối số đầu vào và không có đối số đầu ra hoặc đầu vào ra, một task có thể không có hoặc có nhiều đối số bất kỳ.
- Một function sẽ trả về một giá trị duy nhất, còn task sẽ không có giá trị trả về.

Mục đích của một function là đáp ứng giá trị đầu vào bằng cách trả về một giá trị đơn. Một task có thể hỗ trợ nhiều mục đích và có thể tính toán nhiều giá trị kết quả. Tuy nhiên, chỉ có đối số đầu ra hoặc đầu vào ra thông qua giá trị kết quả từ việc gọi một task. Một function được sử dụng như là một toán hạng của một biểu thức. Giá trị của toán hạng là giá trị trả về của hàm.

Ví dụ:

Hoặc là task hoặc là function định nghĩa để chuyển đổi byte trong một từ 16 bit. Task có thể trả về một giá trị từ đã chuyển đổi trong đối số đầu ra; vì vậy chương trình có thể gọi task *switch_bytes* như sau:

switch_bytes (old_word, new_word);

Task *switch_bytes* sẽ lấy các byte trong *old_word*, đảo ngược các byte và đặt các byte đã đảo ngược trong *new_word*.

Function đảo ngược từ sẽ trả về từ đã đảo ngược như là một giá trị trở về của function. Vì vậy có thể gọi function *switch_bytes* như sau:

new_word = switch_bytes (old_word);

8.2 Tác vụ và kích hoạt tác vụ

Một task sẽ được kích hoạt bằng một câu lệnh định nghĩa các giá trị đối số để thông qua task và các biến để nhận kết quả trả về. Trình điều khiển sẽ truyền lại để kích hoạt tiến trình sau khi task hoàn thành. Vì vậy, nếu một task có điều khiển thời gian bên trong nó, thời gian kích hoạt một task có thể khác với thời gian điều khiển quá trình trả về. Một task có thể kích hoạt một task khác, và trong quá trình hoạt động vẫn có thể kích hoạt các task khác nữa, nó không giới hạn số lượng task được kích hoạt. Bất kể có bao nhiêu task được kích hoạt, trình điều khiển sẽ không trả về cho đến khi tất cả các task đã kích hoạt hoàn thành.

8.2.1 Khai báo tác vụ

Cú pháp để định nghĩa một task được mô tả trong Cú pháp 8-1.

Cú pháp 8-1

```
task_declaration ::=
        task [ automatic ] task_identifier ;
        { task_item_declaration }
        statement_or_null
        endtask
        | task [ automatic ] task_identifier ( [ task_port_list ] ) ;
        { block_item_declaration }
        statement_or_null
        endtask
task_item_declaration ::=
        block_item_declaration
        | { attribute_instance } tf_ input_declaration ;
        | { attribute_instance } tf_output_declaration ;
        | { attribute_instance } tf_inout_declaration ;
task_port_list ::=
        task_port_item { , task_port_item }
task_port_item ::=
        { attribute_instance } tf_input_declaration
        | { attribute_instance } tf_output_declaration
        | { attribute_instance } tf_inout_declaration
tf_input_declaration  ::=
        input [ reg ] [ signed ] [ range ] list_of_port_identifiers
        | input task_port_type list_of_port_identifiers
tf_output_declaration ::=
```

```
        output [ reg ] [ signed ] [ range ] list_of_port_identifiers
        | output task_port_type list_of_port_identifiers
tf_inout_declaration ::=
        inout [ reg ] [ signed ] [ range ] list_of_port_identifiers
        | inout task_port_type list_of_port_identifiers
task_port_type ::=
        integer | real | realtime | time
block_item_declaration ::=
        { attribute_instance } reg [ signed ] [ range ]
        list_of_block_variable_identifiers ;
        | { attribute_instance } integer list_of_block_variable_identifiers ;
        | { attribute_instance } time list_of_block_variable_identifiers ;
        | { attribute_instance } real list_of_block_real_identifiers ;
        | { attribute_instance } realtime list_of_block_real_identifiers ;
        | { attribute_instance } event_declaration
        | { attribute_instance } local_parameter_declaration ;
        | { attribute_instance } parameter_declaration ;
list_of_block_variable_identifiers ::=
        block_variable_type { , block_variable_type }
list_of_block_real_identifiers ::=
        block_real_type { , block_real_type }
block_variable_type ::=
        variable_identifier { dimension }
block_real_type ::=
        real_identifier { dimension }
```

Có hai cú pháp khai báo task:

Cú pháp thứ nhất sẽ bắt đầu với từ khóa task, theo sau là từ khóa tùy chọn **automatic**, theo sau là tên task và tiếp theo là dấu chấm phảy (;), và kết thúc với từ khóa endtask. Từ khóa **automatic** khai báo một task tự động lõm vào, với tất cả các task được khai báo phân bố động cho mỗi mục task hiện tại. Khai báo các yếu của task bao gồm:

- Đối số đầu vào
- Đối số đầu ra.
- Đối số đầu vào ra.
- Tất cả các kiểu dữ liệu có thể khai báo trong một khối thủ tục.

Cú pháp thứ hai bắt đầu với từ khóa **task,** theo sau là tên của task và danh sách các cổng của task nằm trong dấu ngoặc đơn. Danh sách các cổng của task có thể không có hoặc có nhiều các cổng ngăn cách nhau bởi đấu phảy. Và có một dấu chấm phảy sau dấu ngoặc đơn. Tiếp theo là phần thân của task và kết thúc bằng từ khóa endtask.

Trong cả hai cú pháp, khai báo các cổng giống với cú pháp được định nghĩa bởi tf_input_declaration, tf_output_declaration, tf_inout_declaration được mô tả trong Cú pháp 8-1bên trên.

Task mà không chứa từ khóa tùy chọn automatic là một task tĩnh, với tất cả các mục khai báo sẽ được phân bố cố định. Những mục này sẽ chia sẽ được chia sẻ thông qua tất cả các sử dụng của task thực thi hiện tại. Task bao gồm từ khóa automatic sẽ là một task động. Tất cả các mục khai báo trong task động sẽ được phân bố động trong mỗi lần task được gọi. Các mục của task động không thể truy cập theo cấu trúc phân cấp. Task động có thể được gọi sử dụng thông qua tên phân cấp.

8.2.2 Kích hoạt tác vụ và truyền đối số

Câu lệnh kích hoạt task sẽ thông qua các đối số như một danh sách các biểu thức nằm trong dấu ngoặc đơn ngăn cách với nhau bởi dấu phảy. Cú pháp kích hoạt task được mô tả trong Cú pháp 8-2.

Cú pháp 8-2

```
task_enable ::=
        hierarchical_task_identifier [ ( expression { , expression } ) ] ;
```

Nếu định nghĩa một task không có đối số, danh sách đối số sẽ không được cung cấp trong câu lệnh kích hoạt task. Ngược lại, nếu định nghĩa task có đối số, thì sẽ có một danh sách các biểu thức theo thứ tự tương ứng với kích thước và thức tự của danh sách các đối số trong định nghĩa task. Một biểu thức rỗng không được xem là một đối số trong câu lệnh kích hoạt task.

Nếu một đối số trong task được khai báo là input, thì biểu thức tương ứng với đối số đó là một biểu thức bất kỳ. Trình tự đánh giá một biểu thức trong danh sách các đối số là không được định nghĩa trước. Nếu một đối số trong task được khai báo là output hoặc inout, thì biểu thức tương ứng sẽ giới hạn là một biểu thức phù hợp với biểu thức bên trái trong thủ tục gán (phần 9.2). Các mục sau đây đáp ứng yêu cầu này:

- Các biến reg, integer, real, realtime, và time.
- Bộ nhớ tham chiếu.
- Các biến kết nối của reg, integer, và time.
- Kết nối của bộ nhớ tham chiếu
- Các biến bit-selects và part-selects của reg, integer, và time.

Việc thực thi câu lệnh kích hoạt task sẽ thông qua giá trị input từ danh sách các biểu thức trong câu lệnh kích hoạt phù hợp với đối số của

task. Việc thực thi sẽ trả về giá trị từ task thông qua các giá trị từ các đối số loại output hoặc inout của task tương ứng với biến trong câu lệnh kích hoạt task. Tất cả các đối số trong task sẽ thông qua các giá trị hơn là tham chiếu (là một con trỏ đến giá trị).

Ví dụ 8.1 mô tả cấu trúc cơ bản của định nghĩa một task với năm đối số:

Ví dụ 8.1

```
task my_task;
input a, b;
inout c;
output d, e;
begin
    . . .  // các câu lệnh thực thi nhiệm vụ của task.
    . . .
    c = foo1; // gán trạng thái ban đầu cho thanh ghi kết quả.
    d = foo2;
    e = foo3;
end
endtask
```

Hoặc sử dụng hình thức thứ 2 của khai báo task, task có thể định nghĩa như sau:

```
task my_task (input a, b, inout c, output d, e);
begin
    . . .  // các câu lệnh thực thi nhiệm vụ của task.
    . . .
    c = foo1; // gán trạng thái ban đầu cho thanh ghi kết quả.
    d = foo2;
```

```
        e = foo3;
    end
    endtask
```

Câu lệnh sau cho phép kích hoạt task:

my_task (v, w, x, y, z);

Các đối số trong câu lệnh kích hoạt task (v, w, x, y, và z) tương ứng với các đối số (a, b, c, d, và e) trong định nghĩa task. Trong thời gian kích hoạt task, các đối số input và inout (a, b, và c) nhận các giá trị thông qua các đối số v, w, và x. Như vậy việc thực thi lời gọi kích hoạt task sẽ tương ứng thực hiện lệnh gán sau:

a = v;

b = w;

c = x;

Tiếp theo trong tiến trình của task, theo định nghĩa của my_task sẽ đặt giá trị kết quả tính toán vào c, d, và e. Khi task hoàn thành, lệnh gán sau sẽ trả về giá trị tính toán tới lời gọi thực thi task:

x = c;

y = d;

z = e;

Ví dụ 8.2 sẽ mô tả việc sử dụng task trong chương trình đèn giao thông tuần tự:

Ví dụ 8.2

```
module traffic_lights;
reg clock, red, amber, green;
parameter   on = 1, off = 0, red_tics = 350,
            amber_tics = 30, green_tics = 200;
```

```
        // trạng thái ban đầu.
        initial red = off;
        initial amber = off;
        initial green = off;
        always begin // điều khiển đèn tuần tự.
                red = on;  // bật đèn đỏ
                light(red, red_tics); // đợi
                green = on; // bật đèn xanh
                light(green, green_tics); // đợi.
                amber = on; // bật đèn vàng
                light(amber, amber_tics); // đợi
        end
        // task sẽ đợi  'tics' trong cạnh lên của clock trước khi tắc tín hiệu đèn
        task light;
        output color;
        input [31:0] tics;
        begin
                repeat (tics) @ (posedge clock);
                color = off; // tắt đèn.
        end
        endtask
        always begin // tạo dạng sóng cho đồng hồ.
                #100 clock = 0;
                #100 clock = 1;
        end
endmodule
```

8.2.3 Sử dụng bộ nhớ tác vụ và sự kích hoạt đồng thời

Một task có thể kích hoạt đồng thời nhiều lần. Tất cả các biến của một task động được sao chép trên mỗi task được gọi đồng thời để lưu trữ trạng thái cụ thể của việc gọi đó. Tất cả các biến của task tĩnh sẽ cố định trong đó mỗi biến đơn sẽ tương ứng với một biến nội bộ trong module gọi thể hiện, bất kể số lượng các task được kích hoạt đồng thời. Tuy nhiên, task tĩnh trong các thể hiện khác nhau trong một module sẽ lưu trữ tách biệt với những thể hiện khác.

Khai báo biến trong task tĩnh bao gồm các loại đối số input, output và inout, sẽ giữ lại giá trị của chúng giữa các lần gọi. Chúng được khởi tạo bởi giá trị khởi tạo mặc định.

Khai báo biến trong task động, bao gồm đối số loại output sẽ khởi tạo bởi giá trị khởi tạo mặc định bất cứ khi nào việc thực thi đi vào vùng của nó, đối số loại input hoặc inout sẽ khởi tạo thông qua giá trị từ biểu thức tương ứng với danh sách đối số trong câu lệnh khởi tạo task.

8.3 Hàm và việc gọi hàm

Mục đích của một function là để trả về một giá trị được sử dụng trong một biểu thức. Phần tiếp theo của chương này sẽ mô tả các định nghĩa và sử dụng function.

8.3.1 Khai báo hàm

Cú pháp để định nghĩa một function được đưa ra trong Cú pháp 8-3.

Cú pháp 8-3

function_declaration ::=
    function [**automatic**] [function_range_or_type]

function_identifier ;
function_item_declaration { function_item_declaration }
function_statement
endfunction
| **function** [**automatic**] [function_range_or_type]
function_identifier (function_port_list) ;
{ block_item_declaration }
function_statement
endfunction

function_item_declaration ::=
block_item_declaration
| { attribute_instance } tf_input_declaration ;

function_port_list ::=
{ attribute_instance } tf_input_declaration
{ , { attribute_instance }tf_input_declaration }

tf_input_declaration ::=
input [**reg**] [**signed**] [range] list_of_port_identifiers
| **input** task_port_type list_of_port_identifiers

function_range_or_type ::=
[**signed**] [range]
| **integer**
| **real**
| **realtime**
| **time**

block_item_declaration ::= (From A.2.8)
{ attribute_instance } **reg** [**signed**] [range]
list_of_block_variable_identifiers ;

```
        | { attribute_instance } integer list_of_block_variable_identifiers ;
        | { attribute_instance } time list_of_block_variable_identifiers ;
        | { attribute_instance } real list_of_block_real_identifiers ;
        | { attribute_instance } realtime list_of_block_real_identifiers ;
        | { attribute_instance } event_declaration
        | { attribute_instance } local_parameter_declaration ;
        | { attribute_instance } parameter_declaration ;
list_of_block_variable_identifiers ::=
        block_variable_type { , block_variable_type }
list_of_block_real_identifiers ::=
        block_real_type { , block_real_type }
block_variable_type ::=
        variable_identifier { dimension }
block_real_type ::=
        real_identifier { dimension }
```

Một function được định nghĩa bắt đầu với từ khóa function, theo sau bằng từ khóa tùy chọn automatic, theo sau là tùy chọn *function_range_or_type* của giá trị trả về từ function, tiếp theo là tên của function, tiếp theo sau hoặc là dấu chấm phảy hoặc là danh sách các cổng của function nằm trong dấu ngoặc đơn, và sau đó kết thúc với từ khóa endfunction.

Việc sử dụng *function_range_or_type* trong function là tùy chọn. Một function không có đặc tả *function_range_or_type* sẽ mặc định trả về một giá trị vô hướng. Nếu sử dụng, *function_range_or_type* sẽ đặc tả giá trị trở về của function là real, integer, time, realtime hoặc một vector (dấu tùy chọn) với dãy phạm vi [n:m] bit.

Một function sẽ có ít nhất một khai báo đầu vào.

Từ khóa *automatic* khai báo một function chứa nó là một function động, với tất cả các khai báo phân bố tự động cho mỗi hàm được gọi. Các mục của function động không thể truy cập bằng cấu trúc phân cấp. Function động có thể gọi sử dụng tên phân cấp.

Đầu vào của function được khai báo theo một trong hai cách. Cách thứ nhất, theo sau tên hàm là một dấu chấm phảy. Sau dấu chấm phảy là một hoặc nhiều đầu vào được khai báo lẫn lộn với khai báo các khối mục kèm theo. Sau đó là khai báo các mục của function, đó là các câu lệnh hành vi và kết thúc bởi từ khóa *endfunction*.

Cách thứ hai là theo sau tên hàm là một dấu mở ngoặc đơn, tiếp theo là một hoặc nhiều khai báo đầu vào, ngăn cách nhau bởi dấu phảy, tiếp theo là dấu đóng ngoặc đơn và dấu chấm phảy. Sau dấu chấm phảy là có hoặc không có khác mục khối khai báo, tiếp đến là các câu lệnh hành vi và cuối cùng kết thúc bằng từ khóa endfunction.

Ví dụ 8.3 mô tả một định nghĩa function getbyte, sử dụng dãy chỉ định:

Ví dụ 8.3

```
function [7:0] getbyte;
input [15:0] address;
begin
    // chương trình lấy byte từ word
    . . .
    getbyte = result_expression;
end
endfunction
```

Hoặc sử dụng cách thứ hai để định nghĩa function, function getbyte có thể được định nghĩa như sau:

```
function [7:0] getbyte (input [15:0] address);
begin
    // chương trình lấy byte từ word
    . . .
    getbyte = result_expression;
end
endfunction
```

8.3.2 Trả về một giá trị từ hàm

Trong định nghĩa function sẽ khai báo một biến tường minh, bên trong của hàm, với tên cùng với tên hàm. Biến này hoặc mặc định là một thanh ghi 1 bit hoặc có loại cùng với loại được đặc tả trong định nghĩa function. Định nghĩa function khởi tạo giá trị trả về từ function bằng cách gán kết quả của function tới một biến nội có tên giống với tên của function.

Là không hợp lệ nếu khai báo một đối tượng khác có tên giống với tên function trong phạm vi mà function được khai báo. Bên trong function có một biến ngầm định là tên của function, biến này để sử dụng cho các biểu thức bên trong function. Vì vậy là không hợp lệ khi khai báo một đối tượng khác có tên giống với tên của function trong phạm vi bên trong function.

Dòng lệnh sau minh họa cho việc gọi hàm trong ví dụ phần 8.3.1.

getbyte = result_expression;

8.3.3 Việc gọi hàm

Một lời gọi function là một toán hạng trong một biểu thức. Cú pháp của lời gọi function được mô tả trong

Cú pháp 8-4.

Cú pháp 8-4

```
function_call ::=
        hierarchical_function_identifier{ attribute_instance } (
        expression { , expression } )
```

Trình tự đánh giá các đối số để gọi hàm không được định nghĩa.

Ví dụ sau tạo một từ bằng cách kết nối kết quả của hai lời gọi hàm getbyte (định nghĩa trong phần 8.3.1).

word = control ? {getbyte(msbyte), getbyte(lsbyte)}:0;

8.3.4 Những qui tắc về hàm

Function có nhiều hạn chế hơn so với task. Các quy tắc sau đây chi phối việc sử dụng chúng:

- Một định nghĩa function không bao gồm các câu lệnh điều khiển thời gian, đó là các câu lệnh có chứa #, @ hoặc wait.
- Function không thể kích hoạt task.
- Định nghĩa function phải có ít nhất một đối số đầu vào.
- Định nghĩa function sẽ không có bất kì khai báo đối số đầu ra hoặc đầu vào ra.
- Function sẽ không có bất kỳ câu lệnh gán nonblocking hoặc thủ tục gán liên tục.
- Function sẽ không có sự kiện triggers.

Ví dụ 8.4 định nghĩa một hàm gọi là factorial nó trả về một giá trị integer. Hàm factorial sẽ được gọi đi gọi lại và kết quả được in ra.

Ví dụ 8.4

```
module tryfact;
// định nghĩa hàm
function automatic integer factorial;
input [31:0] operand;
integer i;
if (operand >= 2)
        factorial = factorial (operand - 1) * operand;
else
        factorial = 1;
endfunction
// kiểm tra hàm
integer result;
integer n;
initial begin
        for (n = 0; n <= 7; n = n+1) begin
                result = factorial(n);
                $display("%0d factorial=%0d", n, result);
        end
end
endmodule
```

Kết quả của việc mô phỏng:

0 factorial=1

1 factorial=1

2 factorial=2

3 factorial=6

4 factorial=24

5 factorial=120

```
6 factorial=720
7 factorial=5040
```

8.3.5 Sử dụng những hàm hằng số

Lời gọi hàm hằng số được sử dụng để hỗ trợ việc xây dựng các tính toán phức tạp cảu các giá trị ở thời gian xây dựng. Một lời gọi hàm hằng số là một lời gọi hàm của một hàm hằng số nằm trong module gọi nó nơi mà các đối số của hàm là các biểu thức hằng số. Hàm hằng số là một tập hợp con của các hàm Verilog thông thường, chúng phải đáp ứng được các ràng buộc sau:

- Chúng không chứa cấu trúc phân cấp.
- Bất kỳ lời gọi hàm nào bên trong hàm hằng số sẽ làm một hàm hằng số trong nằm trong module hiện tại.
- Là hợp lệ để gọi bất kỳ một hàm hệ thống nào được cho phép trong biểu thức hằng số, gọi các hàm hệ thống khác là không hợp lệ.
- Tất cả các task hệ thống trong hàm hệ thống sẽ được bỏ qua.
- Tất cả các tham số được sử dụng bên trong hàm phải được định nghĩa trước khi sử dụng lời gọi hàm hằng số.
- Tất cả các định danh không phải là tham số hoặc hàm sẽ được khai báo tại hàm hiện tại.
- Nếu chúng sử dụng bất kỳ giá trị tham số nào ảnh hưởng trực tiếp hoặc gián tiếp đến câu lệnh defparam, kết quả không được định nghĩa. Điều này có thể tạo ra một lỗi hoặc các hàm hằng số trả về một giá trị không xác định.
- Chúng không được khai báo bên trong một khối tạo.
- Chúng không tự sử dụng hàm hằng số trong bất kỳ hoàn cảnh nào yêu cầu một biểu thức hằng số.

- Một lời gọi hàm hằng số sẽ được đánh giá trong thời gian xây dựng. Việc thực thi của chúng không ảnh hưởng đến giá trị khởi tạo của các biến sử dụng hoặc trong thời gian mô phỏng hoặc giữa nhiều lời gọi một hàm trong thời gian xây dựng. Trong mỗi trường hợp, biến sẽ được khởi tạo nhưng quá trình mô phỏng thông thường.

Ví dụ 8.5 sẽ định nghĩa một hàm gọi là clogb2 trả về một số nguyên với giá trị cao nhất của log 2.

Ví dụ 8.5

```
module ram_model (address, write, chip_select, data);
  parameter data_width = 8;
  parameter ram_depth = 256;
  localparam addr_width = clogb2(ram_depth);
  input [addr_width - 1:0] address;
  input write, chip_select;
inout [data_width - 1:0] data;
//định nghĩa hàm clogb2
 function integer clogb2;
  input [31:0] value;
  begin
    value = value - 1;
    for (clogb2 = 0; value > 0; clogb2 = clogb2 + 1)
          value = value >> 1;
  end
  endfunction
  reg [data_width - 1:0] data_store[0:ram_depth - 1];
  //phần còn lại của  ram_model
```

Thể hiện của *ram_model* này với tham số được gán như bên dưới:

ram_model #(32,421) ram_a0(a_addr,a_wr,a_cs,a_data);

8.4 Bài tập

1. Tại sao phải dùng task và function trong khi Verilog HDL đã hỗ trợ module?
2. Phân biệt task và function?
3. Cách khai báo và sử dụng task?
4. Cách khai báo và sử dụng function?
5. Sử dụng function cần tuân theo những quy tắt nào?

Chương 9. Kiểm tra thiết kế

Một hệ thống được thiết kế dùng Verilog phải được mô phỏng và kiểm tra xem thiết kế đã đúng chức năng chưa trước khi tạo ra phần cứng. Trong quá trình chạy mô phỏng này, những lỗi thiết kế và sự không tương thích giữa những linh kiện dùng trong thiết kế có thể được phát hiện. Chạy mô phỏng một thiết kế đòi hỏi việc tạo ra một dữ liệu ngõ vào kiểm tra và quá trình quan sát kết quả sau khi chạy mô phỏng, dữ liệu ngõ vào dùng để kiểm tra này có thể được tạo bằng hai cách, một là tạo dạng sóng (waveform) bằng tay sử dụng trình waveform editor, tuy nhiên cách này chỉ khả thi cho những thiết kế nhỏ với số lượng tín hiệu ngõ vào ít, còn đối với một thiết kế hệ thống lớn, phức tạp với nhiều tín hiệu ngõ cần hàng ngàn chu kì để kiểm tra thì ta phải sử dụng testbench để mô tả dữ liệu ngõ vào kiểm tra. Testbench sử dụng cấu trúc mức cao của Verilog để tạo ra dữ liệu kiểm tra, quan sát đáp ứng ngõ ra, và cả việc bắt tay giữa những tín hiệu trong thiết kế. Bên trong testbench, hệ thống thiết kế cần chạy mô phỏng sẽ được gọi ra (instantiate) trong testbench. Dữ liệu testbench cùng với hệ thống thiết kế sẽ tạo ra một mô hình mô phỏng mà sẽ được sử dụng bởi một công cụ mô phỏng Verilog.

Chương này sẽ thảo luận về việc sử dụng ngôn ngữ Verilog để kiểm tra việc thiết kết module. Chúng ta sẽ thấy rằng thời gian và thủ tục hiển thị sẽ trở nên quan trọng hơn khi tiếp xúc với module testbench. Chương này cho thấy cách cấu trúc của ngôn ngữ Verilog được sử dụng để áp dụng dữ liệu cho module trong quá trình kiểm tra (module under test (MUT)), và cách module đáp ứng sẽ được hiển thị và đánh dấu. Trong phần đầu của chương này sẽ thảo luận về dữ liệu ứng dụng và theo dõi đáp ứng. Trong

phần cuối sẽ thảo luận về kỹ thuật chèn để kiểm tra thiết kế nhằm tạo giải pháp tốt nhất cho việc thiết kể module.

9.1 Testbench

Môi trường mô phỏng Verilog cung cấp một công cụ đồ họa hoặc văn bản hiển thị để hiển thị các kết quả mô phỏng. Một số môi trường mô phỏng đi xa hơn, nó cung cấp một công cụ đồ họa cho việc chỉnh sửa đầu vào dữ liệu kiểm tra tới thiết kế module trong quá trình kiểm tra. Như các công cụ biên tập dạng sóng, chúng thường phù hợp cho những thiết kế nhỏ. Đối với các thiết kế lớn với nhiều bus và dữ liệu điều khiển thì biên tập dạng sóng trở nên phức tạp. Một vấn đề trong biên tập dạng sóng là mỗi môi trường mô phỏng sử dụng một tập các thủ tục khác nhau để chỉnh sưa dạng sóng, vì vậy khi chuyển sang một môi trường mô phỏng mới đòi hỏi phải học lại một tập các thủ tục chỉnh sửa dạng sóng mới.

Vấn đề này có thể được giải quyết bằng cách sử dụng testbenches của Verilog. Một testbenches Verilog là một module có chứa thể hiện của MUT, áp dụng các dữ liệu kiểm tra vào nó và quan sát đầu ra. Bởi vì testbenches là một chương trình Verilog nên nó có thể chạy trên nhiều môi trường mô phỏng khác nhau. Một module và testbenches tương ứng của nó từ một mô hình mô phỏng trong đó MUT được kiểm tra với các dữ liệu đầu vào giống nhau thì sẽ giống nhau bất chấp môi trường mô phỏng.

9.1.1 Kiểm tra mạch tổ hợp

Phát triển một testbenches cho một mạch tổ hợp là một bước tiến đáng kể, tuy nhiên việc chọn dữ liệu kiểm tra và cách để kiểm tra phụ thuộc vào từng MUT và các chức năng của nó.

Chúng ta sẽ xem xét một ví dụ minh họa kiểm tra mạch tổ hợp qua module *alu_4bit*. Phần đầu của module và các khai báo các cổng được thể hiện như sau:

Ví dụ 9.1

```
module alu_4bit (a, b, f, oe, y, p, ov, a_gt_b, a_eq_b, a_lt_b);
    input [3:0] a, b;
    input [1:0] f;
    input oe;
    output [3:0] y;
    output p, ov, a_gt_b, a_eq_b, a_lt_b;
        // . . .
endmodule
```

Module có các đầu vào a, b và đầu vào chức năng f, đầu ra y và các đầu ra kèm theo p, ov, a_gt_b, a_eq_b, a_lt_b.

Một testbenches cho *alu_4bit* được định nghĩa như sau:

Ví dụ 9.2

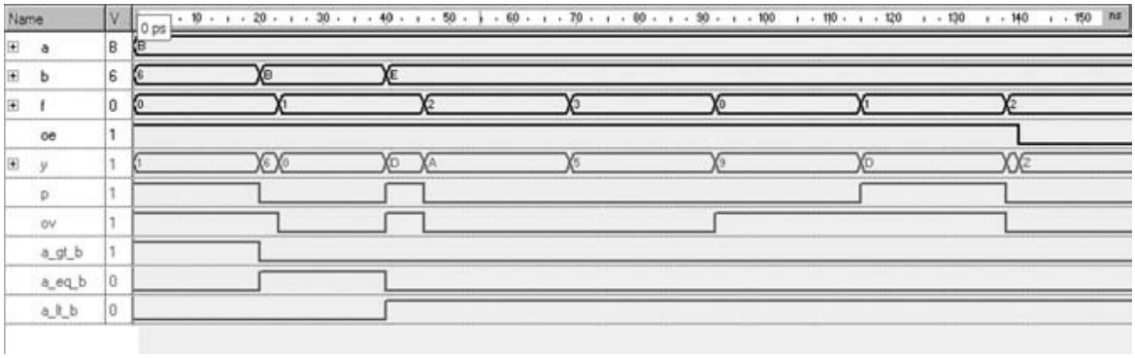
```
module test_alu_4bit;
    reg [3:0] a=4'b1011, b=4'b0110;
    reg [1:0] f=2'b00;
    reg oe=1;
    wire [3:0] y;
    wire p, ov, a_gt_b, a_eq_b, a_lt_b;
    alu_4bit cut( a, b, f, oe, y, p, ov, a_gt_b, a_eq_b,
                a_lt_b );
        initial begin
            #20 b=4'b1011;
            #20 b=4'b1110;
```

```
                #20 b=4'b1110;
                #80 oe=1'b0;
                #20 $finish;
            end
        always #23 f = f + 1;
endmodule
```

Các biến tương ứng với các đầu vào và đầu ra của MUT được khai báo trong testbench. Các biến kết nối tới đầu vào được khai báo là reg, các biến kết nối với đầu ra được khai báo là wire. Thể hiện của alu_4bit sẽ liên kết với các reg và wire cục bộ tới các cổng của MUT.

Biến liên kết với đầu vào của module alu_4bit phải được khai báo giá trị ban đầu. Áp dụng của dữ liệu tới đầu vào dữ liệu b và đầu ra cho phép oe được khai báo trong câu lệnh initial. Trong 60ns đầu, sau mỗi 20ns một giá trị mới được gán cho b. Khối initial sau đó đợi 80ns, vô hiệu hóa alu bằng cách đặt oe = 0. Và sau đó đợi 20ns để hoàn thành mô phỏng, đợi 20ns là để sự thay đổi đầu vào cuối cùng ảnh hưởng tới đầu ra.

Áp dụng dữ liệu tới đầu vào chức năng f của aly_4bit trong khối lệnh always. Bắt đầu với giá trị ban đầu bằng 0, f tăng lên 1 sau mỗi 23ns. Câu lệnh $finish trong khối initial kết thúc tại vị trí 160ns, tại thời điểm này tất cả các khối thủ tục đang hoạt đông dừng lại và quá trình mô phỏng chấm dứt. Hình 9.1 hiển thị kết quả quá trình mô phỏng module alu_4bit



Hình 9.1 Kết quả quá trình mô phỏng module alu_4bit

9.1.2 Kiểm tra mạch tuần tự

Để kiểm tra một mạch tuần tự cần gọi đồng bộ giữa mạch đồng hồ và các dữ liệu đầu vào khác. Chúng ta sử dụng mạch sau để ví dụ trong phần này, mạch có một đầu vào clock, một đầu vào reset, đầu vào dữ liệu và đầu ra:

Ví dụ 9.3

```
module #(parameter [3:0] poly=0) misr (input clk, rst,
        input [3:0] d_in, outputreg [3:0] d_out );
    always @( posedge clk )
    if( rst )
        d_out =4'b0000;
    else
        d_out = d_in ^ ({4{d_out[0]}} & poly) ^
        {1'b0,d_out[3:1]};
endmodule
```

Trong mạch này có tham số poly để chỉ định ký hiệu và dữ liệu nén . Với mỗi chu kỳ đồng hộ một ký hiệu mới được tính toán với dữ liệu mới và nằm trong thanh ghi dữ liệu của misr.

Đoạn chương trình sau đây mô tả một testbench cho module misr. Các biến tương ứng với các cổng của MUT được khai báo trong testbench. Khi misr được gọi thể hiện, các biến này sẽ kết nối với các cổng thật sự của nó. Thể hiện của misr cũng bao gồm chi tiết về các tham số poly của nó.

Ví dụ 9.4

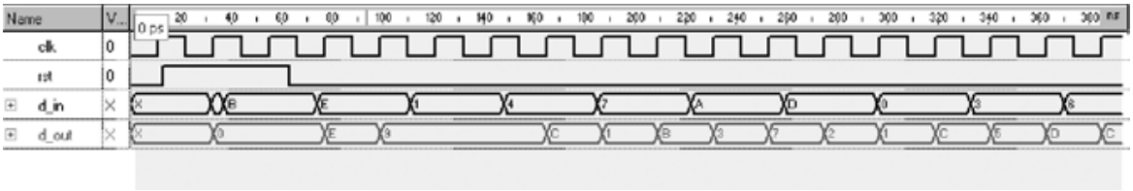
```
module test_misr;
    reg clk=0, rst=0;
    reg [3:0] d_in;
    wire [3:0] d_out;
    misr #(4'b1100) MUT ( clk, rst, d_in, d_out );
        initial begin
            #13 rst=1'b1;
            #19 d_in=4'b1000;
            #31 rst=0'b0;
            #330 $finish;
        end
    always #37 d_in = d_in + 3;
    always #11 clk = ~clk;
endmodule
```

Trong khối *initial* của testbench tạo ra một pulse lên vào lúc rst mà bắt đầu tại 13ns và kết thúc tại 63ns. Thời gian như vậy là sự lựa chọn để bao gồm ít nhất một cạnh lên của đồng hồ, vì vậy sự đồng bộ của đầu vào rst có thể khởi tạo trong thanh ghi misr. Đầu vào dữ liệu *d_in* bắt đầu là x, và trong khởi là 4'b1000 khi rst là 1.

Ngoài các khối khởi tạo, module test_misr còn bao gồm 2 khối *always* để tạo dữ liệu cho *d_in* và *clk*. Clock cho ra một tín hiệu tuần hoàn với chu kỳ 11ns. Đầu vào misrd_in gán một giá trị mới sau mỗi chu kỳ

37ns. Để giảm trường hợp các đầu vào thay đổi cùng một lúc, chúng ta sử dụng số nguyên tố cho thời gian của đầu vào mạch tuần tự.

Hình 9.2 mô tả một kết quả của testbench:



Hình 9.2 Một kết quả của testbench

9.2 Kĩ thuật tạo testbench

Các kỹ thuật viết chương trình Verilog khác nhau để tạo dữ liệu kiểm tra và xem xét đáp ứng của mạch được thảo luận trong phần này. Chúng ta sử dụng module máy trạng thái Moore để phát hiện chuỗi 101 làm ví dụ cho phần này. Module này được định nghĩa như sau:

Ví dụ 9.5

```
module moore_detector (input x, rst, clk, output z );
    parameter [1:0] a=0, b=1, c=2, d=3;
    reg [1:0] current;
    always @( posedge clk )
        if ( rst )  current = a;
        else case ( current )
            a : current = x ? b : a ;
            b : current = x ? b : c ;
            c : current = x ? d : a ;
            d : current = x ? b : c ;
            default : current = a ;
        endcase
    assign z = (current==d) ? 1'b1 : 1'b0;
```

```
endmodule
```

9.2.1 Dữ liệu kiểm tra

Một testbench cho module moore_detector được mô tả như sau:

Ví dụ 9.6

```
module test_moore_detector;
        reg x, reset, clock;
        wire z;
        moore_detector MUT ( x, reset, clock, z );
        initial begin
                clock=1'b0; x=1'b0; reset=1'b1;
        end
        initial #24 reset=1'b0;
        always #5 clock=~clock;
        always #7 x=~x;
endmodule
```

Module testbench là một module không có các cổng kết nối ra ngoài. Bên trong module, có 4 khối thủ tục cung cấp dữ liệu cho việc kiểm tra máy trạng thái. Các biến kết nối với MUT và sử dụng vào vế bên trái của các khối thủ tục phải được khai báo là **reg**.

Thay vì khởi tạo các biến **reg** khi chúng khai báo, chúng ra có thể sử dụng khối initial để khởi tạo các biến reg. Điều quan trọng khi khai khởi tạo biến, giống như clock là giá trị cũ của chúng được sử dụng để xác định giá trị mới của chúng. Nếu không như vậy, clock sẽ bắt đầu với giá trị X và cho đến khi hoàn thành, giá trị của nó cũng không thay đổi. Khối always tạo chu kỳ tín hiện với chu kỳ 10ns trên một clock.

Theo sau khối thủ tục always tạo clock, một khối always khác sẽ tạo chu kỳ tín hiệu cho x với chu kỳ 14ns. Dạng sóng tạo ra cho X có thể có hoặc không thể kiểm tra máy trạng thái của chúng ra đúng tuần tự 101. Tuy nhiên, chu kỳ của clock và X có thể thay đổi điều này. Với thời gian sử dụng ở đây, đầu ra của moore_detector lên 1 sau 55ns và sau mỗi 70ns tiếp theo.

9.2.2 Điều khiển mô phỏng

Một testbench khác cho mạch moore_detector bên trên được mô tả như sau:

Ví dụ 9.7

```
module test_moore_detector;
        reg x=0, reset=1, clock=0;
        wire z;
        moore_detector MUT ( x, reset, clock, z );
        initial #24 reset=1'b0;
        always #5 clock=~clock;
        always #7 x=~x;
        initial #189 $stop;
endmodule
```

Mặc dù, cấu trúc Verilog được sử dụng khác nhau, dữ liệu và clocj áp dụng cho MUT trong testbench cũng giống với testbench bên trên. Tuy nhiên, nếu trình mô phỏng cho testbench trước không có ngắt, hoặc điểm dừng, nó sẽ chạy mãi mãi. Trong testbench này giải quyết vấn đề này bằng cách thêm vào một khối khác khối initial sẽ dừng quá trình mô phỏng sau 189ns. Các task điều khiển trình mô phỏng là $stop và $finish. Thời gian đầu tiên theo chu trình của một khối thủ tục được tiếp cận như một task,

trình mô phỏng sẽ dừng lại hoặc hoàn thành. Một task $stop thì có thể nối lại, nhưng một task $finish thì không thể nối lại.

Một testbench khác được mô tả như sau:

Ví dụ 9.8

```
module test_moore_detector;
        reg x=0, reset=1, clock=0;
        wire z;
        moore_detector MUT ( x, reset, clock, z );
        initial begin
            #24 reset=1'b0;
            #165 $finish;
        end
        always #5 clock=~clock;
        always #7 x=~x;
endmodule
```

Trong testbench này tích hợp trong khối initial cả tín hiệu reset tích cực thấp và tín hiện điều khiển thời gian trong một khối initial. Thời gian điều chỉnh chấm dứt quá trình mô phỏng tại thời gian 189ns giống như testbench bên trên.

9.2.3 Thiết lập giới hạn dữ liệu

Thay vì cài đặt giới hạn thời gian mô phỏng, một testbench có thể đặt một giới hạn trên số dữ liệu đặt vào đầu vào của MUT. Điều sẽ cũng có thể ngăn chặn việc mô phỏng không dừng lại.

Chương trình sau mô tả một testbench cho module moore_detector MUT.

Ví dụ 9.9

```
module test_moore_detector;
        reg x=0, reset=1, clock=0;
        wire z;
        moore_detector MUT ( x, reset, clock, z );
        initial #24 reset=1'b0;
        initial repeat(13) #5 clock=~clock;
        initial repeat(10) #7 x=$random;
endmodule
```

Testbench này sử dụng $radom để tạo dữ liệu ngẫu nhiên cho đầu vào x của mạch, câu lệnh repeat trong khối initial tạo ra tín hiện clock 13 lần sau mỗi 5ns, và x nhận dữ liệu ngẫu nhiên 13 lần sau mỗi 7ns. Thay vì dùng một bộ tính quyết định dữ liệu để đảm bảo tính quyết định các trạng thái, thì dữ liệu ngẫu nhiên được sử dụng ở đây. Chiến lượt này làm cho nó dễ dàng hơn để tạo dữ liệu, nhưng phân tích đầu ra của mạch sẽ khó khăn hơn, do đầu vào không đoán trước. Trong các mạch lớn, dữ liệu ngẫu nhiên được sử dụng nhiều cho đầu vào dữ liệu hơn tín hiệu điều khiển. Testbench trong phần này sẽ dừng sau 70ns.

9.2.4 Cung cấp dữ liệu đồng bộ

Trong các ví dụ trước testbench cho MUT sử dụng thời gian độc lập cho clock và dữ liệu. Trong trường hợp nhiều bộ dữ liệu được áp dụng, việc đồng bộ hóa dữ liệu với đồng hồ hệ thống trở nên khó khăn. Hơn nữa, việc thay đổi tần số clock sẽ yêu cầu thay đổi thời gian của tất cả các dữ liệu đầu vào của module đang kiểm tra.

Testbench sau được viết cho module moore_detector, sử dụng một câu lệnh để điều khiển sự đồng bộ giữa dữ liệu áp dụng vào X với clock tạo ra trong testbench. Tín hiệu clock này tạo ra trong câu lệnh initial sử dụng cấu trúc repeat. Một câu lệnh initial khác được sử dụng để tạo dữ liệu ngẫu nhiên cho X. Như ta thấy trong câu lệnh initial này, vòng lặp mãi mãi sẽ lặp đi lặp lại câu lệnh này để sử dụng ở đây. Vòng lặp này đợi cho tới cạnh lên của clock, và sau khi cạnh lên của clock 3ns, một dữ liệu ngẫu nhiên được tạo ra cho X. Trạng thái dữ liệu sau cạnh lên của clock sẽ sử dụng bởi moore_detector trên đầu cạnh tiếp theo của clock. Kỹ thuật này của dữ liệu đảm bảo rằng sự thay đổi dữ liệu và clock không trùng nhau.

Ví dụ 9.10

```
module test_moore_detector;
        reg x=0, reset=1, clock=0;
        wire z;
        moore_detector MUT ( x, reset, clock, z );
        initial #24 reset=0;
        initial repeat(13) #5 clock=~clock;
        initial forever @(posedge clock) #3 x=$random;
endmodule
```

Có trì hoãn 3ns được sử dụng ở đây làm nó có thể sử dụng testbench giống nhau trong mô phỏng sau khi tổng hợp thiết kế tốt như mô tả hành vi trong các phần trên.Trong quá trình mô phỏng sau khi tổng hợp, mô hình các thành phần sẽ thực sự trì hoãn như các giá trị được sử dụng, trì hoãn trong testbench cho phép truyền tín hiệu kiểm tra hoàn thành trước khi áp dụng tín hiệu khiểm tra mới.

9.2.5 Tương tác testbench

Trong quá trình tạo testbench tiếp theo chúng ta sử dụng một máy trạng thái khác. Đó là máy trạng thái moore phát hiện chuỗi 1101 với trạng thái bắt đầu (start) và reset (rst) điều khiển đầu vào. Nết start là 0 trong khi tìm kiếm chuỗi 1101, máy trạng thái sẽ reset về trạng thái khởi đầu. Như ta thấy trong mạch sau có 5 trạng thái, và đầu ra của nó lên 1 khi nó bắt đầu trạng thái e.

Ví dụ 9.11

```
module moore_detector (input x, start, rst, clk, output z );
        parameter a=0, b=1, c=2, d=3, e=4;
        reg [2:0] current;
        always @( posedge clk )
        if ( rst )  current <= a;
        else if ( ~start ) current <= a;
            else case ( current )
                a : current <= x ? b : a ;
                b : current <= x ? c : a ;
                c : current <= x ? c : d ;
                d : current <= x ? e : a ;
                e : current <= x ? c : a ;
                default: current <= a;
            endcase
        assign z = (current==e);
endmodule
```

Testbench cho máy trạng thái là một testbench tương tác một.

```
module test_moore_detector;
        reg x=0, start, reset=1, clock=0;
```

```
        wire z;
        moore_detector MUT ( x, start, reset, clock, z );
        initial begin
            #24 reset=1'b0; start=1'b1;
            wait(z==1'b1);
            #11 start=1'b0;
            #13 start=1'b1;
            repeat(3) begin
                #11 start=1'b0;
                #13 start=1'b1;
                wait(z==1'b1);
            end
            #50 $stop;
        end
        always #5 clock=~clock;
        always #7 x=$random;
endmodule
```

Trong khối initial, testbench giao tiếp với MUT. Đầu vào X và clock được tạo bởi 2 khối always. Một tín hiệu có chu kỳ liên tục được tạo ra cho clock và một dữ liệu có chu kỳ ngẫu nhiên được gán cho x.

Đầu tiên, giá trị 0 và 1 được đặt cho reset và start để đưa máy trạng thái vào trạng thái bắt đầu. Theo sau đó, một câu lệnh wait để đợi z lên 1 như là kết quả đáp ứng của MUT tới giá trị x và clock. Sau sự kiện này, biến start được gán bằng 0 và sau đó là 1 sau 13ns để khởi động lại máy. Theo sau vòng kích hoạt đầu tiên này, một câu lệnh repeat lặp lại tiến trình bắt đầu máy trạng thái và độ x lên 1 ba lần nữa. Sau 50ns testbench dừng quá trình mô phỏng bằng một task $stop.

9.2.6 Tạo những khoảng thời gian ngẫu nhiên

Chúng ta thấy cách hàm #random có thể sử dụng để tạo một dữ liệu ngẫu nhiên. Phần này chúng ta sẽ thảo luận cách dùng ngẫu nhiên thời gian đợi để gán giá trị cho x.

Testbench sau dùng để kiểm tra module phát hiện chuỗi 1101 sử dụng $random để điều khiển trì hoãn. Như ta thấy, câu lệnh initial tên running áp dụng dữ giá trị phù hợp lên biến reset và start để hệ thống bắt đầu tìm kiếm chuỗi 1101. Trong khối thủ tục này sử dụng lệnh gán không chặn (nonblocking) tạo ra các giá trị trì hoãn trong câu lệnh gán để được coi như là một giá trị thời gian tuyệt đối.

Ví dụ 9.12

```
module test_moore_detector;
        reg x, start, reset, clock;
        wire z;
        reg [3:0] t;
        moore_detector MUT ( x, start, reset, clock, z );
        initial begin:running
            clock <= 1'b0; x <= 1'b0;
            reset <= 1'b1; reset <= #7 1'b0;
            start <= 1'b0; start <= #17 1'b1;
            repeat (13) begin
                @( posedge clock );
                @( negedge clock );
            end
            start=1'b0;
            #5;
            $finish;
```

```
            end
            always #5 clock=~clock;
            always begin
                t = $random;
                #(t) x=$random;
            end
endmodule
```

Sau khi đặt máy trạng thái vào trạng thái running, testbench đợi 13lần để hoàn thành xung clock trước khi nó đặt lại đầu vào start và hoàn thành mô phỏng. Như ta thầy, khối always đồng thời với khối running liên tục tạo xung clock 5ns. Cùng đồng thời với các khối này là một khối always khác tạo dữ liệu ngẫu nhiên cho t, và sử dụng t để trì hoãn lệnh gán ngẫu nhiên cho x. Cả khối này tạo dữ liệu cho đầu vào x cho đến khi câu lệnh **$finish** trong khối running được thực hiện.

9.3 Kiểm tra thiết kế

Trong phần trước đã thảo luận các kỹ thuật kiểm tra để kiểm tra một thiết kế Verilog. Trong phần này sẽ bàn tới một vài phương thức để tạo dữ liệu kiểm tra và áp dụng kiểm tra, và hỗ trợ một vài cách để xem xét và kiểm tra kết quả kiểm tra. Tạo các kích thích và phân tích đáp ứng của một thiết kế đòi hỏi một phần nỗ lực đáng kể của người thiết kết phần cứng. Tìm hiểu các kỹ thuật kiểm tra đúng là tốt, nhưng thiết kế tự động làm các thủ tục này sẽ rất hữu dụng cho các kỹ sư.

Hình thức kiểm tra thiết kế là một cách tự động thiết kế kiểm tra bằng cách loại bỏ testbenches và các vấn đề liên quan đến việc tạo dữ liệu và quan sát các đáp ứng của mạch. Trong hình thức kiểm tra thiết kế, người

thiết kế sẽ thiết kế một thuộc tính để kiểm tra thiết kế của anh ta. Công cụ kiểm tra thiết kế hình thức không thực hiện mô phỏng, nhưng đưa ra câu trả lời có/không có cho mỗi thuộc tính của thiết kế đang được kiểm tra. Mặc dù phương pháp kiểm tra thiết kế giúp tìm ra nhiều lỗi thiết kế nhưng hầu như nhiều thiết kế vẫn cần phát triển testbench và mô phỏng để xác nhận chương trình Verilog của họ thực hiện đúng chức năng mong đợi. Nói cách khác, tất cả các câu trả lời là "có" cho tất cả các thuộc tính kiểm tra bởi công cụ kiểm tra thiết kế hình thức là chưa đủ. Thay vì bỏ qua việc tạo dữ liệu và quan sát đáp ứng, một bước theo hướng tự động hóa thiết kế xác nhận là giảm hoặc bỏ qua các nỗ lực cần thiết cho phân tích kết quả đáp ứng. Kỷ thuật chèn để kiểm tra thiết kế được sử dụng theo hướng này, nó sẽ thêm giám sát để thiết kế cải thiện khả năng quan sát đáp ứng. Trong khi thiết kế đang được mô phỏng với dữ liệu testbench, trình quan sát được chèn vào đại diện cho một số thuộc tính của thiết kế liên tục kiểm tra có đúng với thiết kế hành vi bằng các xác minh những thuộc tính đó. Nếu dữ liệu mô phỏng dẫn đến điều kiện chỉ ra một trình giám sát chèn vào là không phù hợp với hành vi thiết kế, trình giám sát sẽ cảnh báo tới người thiết kế vấn đề đó.

Như đã đề cập, chúng ta vẫn phải phát triển một testbench và thiết kế cẩn thần các đầu vào kiểm tra cho thiết kế cần kiểm tra là cần thiết cho kỹ thuật chèn để kiểm tra thiết kế. Nhưng trong nhiều trường hợp, kỹ thuật chèn tự động kiểm tra để chắc chắn sự kiện xảy ra trong thiết kế là đúng như mong đợi. Điều này có ý nghĩa làm giảm sự cần thiết cho việc xử lý một danh sách dài các đầu ra và dạng sóng.

9.4 Kĩ thuật chèn (assertion) dùng để kiểm tra thiết kế

Không giống như mô phỏng với một testbench hoặc con người để giải thích kết quả, trong kỹ thuật chèn để kiểm tra kết quả chương trình giám sát chịu trách nhiệm phát hành một thông báo nếu có một điều gì đó xảy ra không như mong đợi. Trong Verilog, các trình giám sát là các module, và chúng được khởi tạo trong một thiết kế để kiểm tra các thuộc tính của thiết kế. Thể hiện của một module chèn (assertion module) không được xem như là một module phần cứng. Thay vào đó, loại thể hiện này giống như một thủ tục luôn luôn hoạt động và liên tục kiểm tra các sự kiện trong module thiết kế.

Thiết lập hiện tại của một trình giám sát chèn (assertion monitor) có sẵn trong một thư viện được biết đến như là một thư viện kiểm tra mở (OVL). Người thiết kế có thể phát triển các cài đặt riêng của họ vào trong module chèn. Những gì tồn tại trong trình giám sát kiểm tra giá trị của tín hiệu, quan hệ của một vài tín hiệu với các cái khác, sự tuần tự của các sự kiện, và các mô hình dự kiến trên vector hoặc nhóm tín hiệu. Để sử dụng kỹ thuật chèn, người thiết kế biên dịch OVL và thư viện có sẵn trong thiết kế cần kiểm tra. Khi một thiết kế được phát triển, kỹ thuật chèn sẽ thay thế các điểm cần thiết trong thiết kế đẻ kiểm tra các chức năng cần thiết. Khi thiết kế được mô phỏng như một thành phần đơn chuẩn, hoặc trong một cấu trúc phân cấp của một thiết kế lớn, trình giám sát kiểm tra tín hiệ cho các giá trị ngoại lệ. Nếu tín hiệu không có giá trị ngoại lệ bằng sự giám sát, trình giám sát chèn vào sẽ hiển thị một thông điệp và thời gian sự khác biệt (vi phạm của các thuộc tính) đã xảy ra. Thông thường, thông điệp xuất hiện trong vùng báo cáo của mô phỏng, bản dịch hoặc khung hiển thị điều khiển (console).

9.4.1 Lợi ích của kỹ thuật chèn kiểm tra.

Cách để tìm nơi để chèn một trình giám sát sao cho có lợi nhất sẽ được thảo luận trong phần này.

Kỷ luật thiết kế: Khi một người thiết kế đặt một nơi để chèn trong thiết kế, người thiết kế cần phải tự yêu cầu bản thân mình xem xét cẩn thận thiết kế và trích xuất các thuộc tính cần thiết.

Khả năng quan sát: Chèn thêm các trình giám sát vào các điểm cần giám sát của thiết kế làm sao cho dễ quan sát nhất.

Quá trình kiểm tra chính thức sẵn sàng: Các chương trình chèn tương ứng với các thuộc tính được sử dụng trong công cụ kiểm tra chính thức. Có chèn vào một trình giám vào một thiết kế, sẵn sàng cho nó kiểm tra bằng công cụ kiểm tra thiết kế chính thức.

Thực thi chú thích: Trình giám sát chèn vào có thể xem như là chú thích để chú thích một vài tính năng hoặc hành vi của thiết kế. Các chú thích này tạo ra một thông điệp khi hành vi của chúng được giải thích là vi phạm.

Tự thiết kế kín: Một thiết kế với kỹ thuật chèn giám sát đã mô tả thiết kế và các thủ tục kiểm tra nó trong một module Verilog.

9.4.2 Thư viện thiết kế mở (OVL)

OVL có sẵn từ tổ chức Accellera (http://www.accellera.org/activities/ovl/) và các tổ chức EDA khác. Hướng dẫn tham khảo ngôn ngữ, hướng dẫn sử dụng, và thư viện chương trình của Verilog và VHDL cũng có sẵn trong các tổ chức EDA này. Danh sách các chương trình chèn có sẵn được liệt kê trong danh sách sau:

- assert_change
- assert_decrement

- assert_even_parity
- assert_frame
- assert_implication
- assert_never
- assert_next
- assert_no_transition
- assert_odd_parity
- assert_one_hot
- assert_quiescent_state…

Một assertion được đặt trong chương trình giống như một thể hiện của module. Cấu trúc sau mô tả một khai báo thể hiện module assertion.

Cú pháp 9-1

```
assert_name
    #(static_parameters)
        instance_name
            (dynamic_arguments);
```

Nó bắt đầu với tên của module assertion, theo sau là các tham số tĩnh (satatic_parameters) như là kích thước của vector hoặc các tùy chọn. Sau đó là tên duy nhất bất kỳ của thể hiện, và phần cuối cùng là một trình giám sát chèn vào bao gồm các tham chiếu, giám sát tín hiệu và các đối số động khác. Các đối số động này là các cổng của module và cũng được xem như là các cổng của assertion module.

9.4.3 Sử dụng kỹ thuật chèn giám sát.

- **assert_always :** Cú pháp để sử dụng kỹ thuật chèn giám sát được mô tả như sau:

Cú pháp 9-2

```
assert_always
    #( severity_level, property_type,
       msg, coverage_level )
          instance_name ( clk, reset_n, test_expr )
```

Lệnh này sẽ liên tục chèn kiểm tra test_expr để chắc chắn nó luôn luôn đúng trên mỗi cạnh của clock. Nếu biểu thức kiểm tra sai, một thông điệp tương ứng sẽ được hiển thị.

Ví dụ 9.13

```
module BCD_Counter (input rst, clk, outputreg [3:0] cnt);
    always @(posedge clk)
        begin
            if (rst || cnt >= 10) cnt = 0;
            else cnt = cnt + 1;
        end
    assert_always #(1, 0, “Err: Non BCD Count”, 0)
    AA1 (clk, 1’b1, (cnt >= 0) && (cnt <= 9));
endmodule
```

Testbench để kiểm tra module:

```
module BCD_Counter_Tester;
    reg r, c;
    wire [3:0] count;
        BCD_Counter UUT (r, c, count);
    initial begin
        r = 0; c = 0;
    end
```

```
    initial repeat (200) #17 c= ~c;
    initial repeat (03) #807 r= ~r;
endmodule
```

- **assert_change :** lệnh chèn này giám sát kiểm tra trong một số chu kỳ đồng hồ sau sự kiện bắt đầu, biểu thức kiểm tra sẽ thay đổi, cú pháp được sử dụng như sau:

Cú pháp 9-3

```
assert_change
        #( severity_level, width, num_cks,
        action_on_new_start, property_type,
        msg, coverage_level )
                instance_name ( clk, reset_n, start_event,
        test_expr )
```

- **assert_one_hot:** kỹ thuật chèn giám sát này dùng để kiểm tra chỉ một bit trong n bit của biểu thức kiểm tra là 1 trong khi trình giám sát vẫn đang hoạt động, cú pháp như sau:

Cú pháp 9-4

```
assert_one_hot
        #( severity_level, width, property_type,
        msg, coverage_level )
                instance_name ( clk, reset_n, test_expr )
```

- **assert_cycle_sequence:** kỹ thuật chèn này dùng để kiểm tra máy trạng thái, cú pháp như sau:

Cú pháp 9-5

```
assert_cycle_sequence
        #( severity_level, num_cks, necessary_condition,
        property_type,
        msg, coverage_level )
                instance_name ( clk, reset_n, event_sequence )
```

- **assert_next:** kỹ thuật chèn này sử dụng cú pháp bên dưới và kiểm tra các sự kiện xảy ra tại các chu kỳ đồng hồ ở giữa sự kiện bắt đầu và kết thúc:

Cú pháp 9-6

```
assert_next
        #( severity_level, num_cks, check_overlapping,
        check_missing_start, property_type,
        msg, coverage_level )
                instance_name ( clk, reset_n, start_event,
        test_expr )
```

9.5 Bài tập

1. Tại sao phải kiểm tra thiết kế? Có mấy cách để kiểm tra thiết kế, nêu cụ thể?
2. Phân biệt hai hình thức kiểm tra thiết kế sử dụng testbench và verification?
3. Các kỹ thuật để tạo một testbench hiệu quả ?

4. Viết testbench cho mạch tổ hợp và mạch tuần tự có gì khác nhau?
5. Thế nào là chèn để kiểm tra thiết kế (assertion verification)? Lợi ích của kỹ thuật này?
6. Thư viện mởi OVL bao gồm những câu lệnh chèn cơ bản nào và các sử dụng chúng?

PHỤ LỤC

DANH SÁCH TỪ VIẾT TẮT

CAD	:	Computer Aided Design.
HDL	:	Hardware Description Language
EDA	:	Electronic Design Automation
ASIC	:	Application Specific Integrated Circuit.
PLD	:	Programmable Logic Device
RTL	:	Register Transfer Level
SDF	:	Standard Delay Format
PLI	:	Programming Language Interface
UDP	:	User Defined Primitives
CMOS	:	Complementary Metal-Oxide-Semiconductor
OVL	:	Open Verification Library
IC	:	Integrated Circuit
FPLD	:	Field Programmable Logic Device
FPGA	:	Field Programmable Gate Array
RTL	:	Register Transfer Level
OVI	:	Open Verilog International
IEEE	:	Institute of Electrical and Electronics Engineers
ASCII	:	American Standard Code for Information Interchange
SDF	:	Standard Delay Format
PCB	:	Printed Circuit Board
MUT	:	Module Under Test