Verilog Coding for Logic Synthesis

Author : Weng Fook Lee

Publisher : John Wiley and Sons

Sách dày 308 trang, với nội dung trình bày về Verilog đồng thời về thiết kế ASIC với cách làm này tôi cũng muốn đưa vào một số từ ngữ chuyên môn mà có thể nhiều bạn còn chưa rõ.

Những ký hiệu dùng trong bản dịch nháp.

[.....] : những ghi chú của tôi nhằm giải thích cho một từ nào đó hay một khái niệm nào đó.

Những file thiết kế hay testbench sau này sẽ được up lên forum.

Chúng ta bắt đầu nhé.

Lời nói đầu

Độ phức tạp của mạch tích hợp đã gia tăng rất lớn trong hơn 10 năm qua.Vào thập niên 80, thiết kế một con chip chứa vài triệu transistor đã khó tưởng tượng nổi.Ngày nay những mạch tích hợp có vài triệu transistor là điều thường thấy.Sự gia tăng tính phức tạp trong mạch tích hợp chủ yếu là do kết quả của việc tích hợp nhiều chức năng trên một chip đơn.Với những thay đổi cơ bản này, phương pháp thiết kế thông thường bằng sơ đồ mạch (schematic) đã trở thành một cản trở cho những kỹ sư thiết kế. Thực sự quá khó khăn cho kỹ sư thiết kế để có thể "vẽ bằng tay" một số lượng lớn những sơ đồ mạch cần thiết cho một chức năng mạch mong muốn. Thêm vào đó, mạch tích hợp được đẩy ra thị trường với tốc độ nhanh làm co lại khung thời gian ra thị trường của chip (time to market). Người thiết kế bị đặt dưới áp lực thiết kế nhiều chip phức tạp hơn với tốc độ nhanh hơn.

Thử tưởng tượng kỹ sư thiết kế cần phải vẽ hàng triệu transistor trong sơ đồ mạch. Công việc này gần như là không thể.

Điều này đòi hỏi một phương pháp hiệu quả và năng suất hơn là cho phép người thiết kế tạo ra sơ đồ mạch với số lượng lớn các cổng trong một khung thời gian hợp lý. Nhu cầu này dẫn đến sự pháp triển của ngôn ngữ mô tả phần cứng. (Hardware Description Language) (HDL)

Phương pháp mới này cho phép người thiết kế mã hóa chức năng logic của một mạch trong HDL. Mã này sau đó sẽ được tổng hợp (synthesize) thành những cổng logic sử dụng công cụ tổng hợp (synthesizer).

Có hai kiểu ngôn ngữ mô tả phần cứng được dùng trong công nghiệp: Verilog và VHDL. Quyển sách này chỉ trình bày về Verilog.

Quyền sách này được viết dành cho sinh viên và kỹ sư học viết mã Verilog tổng hợp được (synthesizable Verilog code).Chương 1 giới thiệu việc dùng VHDL và Verilog. Chương 2 mô tả luồng thiết kế vi mạch ứng dụng đặc biệt (ASIC). Đồ thị luồng (flow charts) và mô tả được dùng để giúp người đọc hiểu rõ về ASIC.

Chương 3 trình bày những khái niệm cơ bản về mã Verilog. Chương này chỉ ra cho người đọc việc sử dụng số (numbers), ghi chú (comments) và những kiểu dự liệu trong Verilog. Việc sử dụng những primitive mức độ cổng cũng như do người dùng định nghĩa cũng sẽ được giải thích trong chương này.

Chương 4 trình bày những thói quen thiết kế và phong cách viết mã được dùng cho tổng hợp. Cách đặt tên (naming convention), phân vùng thiết kế, ảnh hưởng của những vòng lặp định thì, tạo tín hiệu xung đồng hồ, sử dụng reset và danh sách nhạy (sentivity list). Khái niệm về câu lệnh blocking (khóa) và non-blocking sẽ được trình bày chi tiết. Ví dụ và dạng sóng được dùng xuyên suốt để giúp người đọc hiểu những khái niệm này.Chương 4 cũng trình bày những ví dụ về phong cách viết mã thông dụng cho những toán tử Verilog. Khái niệm về "chốt suy ra" (latch inference), mảng bộ nhớ và máy trạng thái cũng có trong chương này.Thiết kế máy trạng thái bao gồm đặc tả thiết kế (design specifications), giản đồ trạng thái để chỉ ra chức năng của máy trạng thái, mã tổng hợp được cho máy trạng thái cùng với testbench để kiểm tra chức năng của máy trạng thái.

Chương 5 chỉ cho người đọc làm thế nào dự án thiết kế mạch định thì lập trình được được thực hiện. Chương này bắt đầu với đặc tả cho mạch định thì lập trình được (programmable timer). Sau đó sẽ là vi kiến trúc (microarchitecture) được tạo ra từ

đặc tả trên được dẫn ra. Giản đồ sẽ giúp người đọc dễ hiểu hơn chức năng cần thiết kế. Mã Verilg, testbench để kiểm tra và mô phỏng mạch cùng với dạng sóng ra (waveform) cũng được thêm vào.

Chương 6 trình bày về những khối logic lập trình được dành cho giao tiếp ngoại vi (peripheral interface). Chương này bắt đầu bằng đặc tả thiết kế rồi vi kiến trúc cùng với giản đồ hòng giúp bạn đọc hiểu rõ vấn đề hơn. Mã Verilog, testbench và dạng sóng ra sẽ được dùng để mô tả mạch trong chương này.

Sách này gồm nhiều ví dụ và được viết với tác phong thực tiễn. Có cả thảy 91 ví dụ giúp bạn đọc hiểu những khái niệm cũng như phong cách lập trình đang được trình bày. Bắt đầu với những mã Verilog đơn giản tiền dần vì dụ thiết kế thực tế phức tạp.Chương 4 trình bày thiết kế máy trạng thái về hệ thống đèn giao thông thông minh. Chương 5 nói về mạch định thì lập trình được bắt đầu với đặc tả thiết kế rồi vi kiến trúc, mã verilog và cuối cùng là testbench.Thiết kề này chỉ ra cho người đọc làm thế nào mã Verilog được viết kiểm tra nhưng không thể tổng hộ thành mạch thực được.

Nhằm giúp bạn đọc hiểu hơn về việc làm thế nào những thiết kế thực tế được thiết kế : giản đồ, dạn sóng và những giải thích chi tiết kết quả mô phỏng được đưa vào.

Giới thiệu

Kể từ đầu thập niên 80 khi mà sơ đồ mạch được giới thiệu như một cách hiểu quả để thiết kế những mạch có mức độ tích hợp quy mô lớn (VLSI), nó đã trở thành phương pháp được người thiết kế của thế giới VLSI lựa chọn.

Tuy nhiên, việc dùng phương pháp này đạt đến giới hạn vào cuối thập niên 90 khi có ngày càng nhiều chức năng logic và đặc tính được tích hợp vào một chip đơn. Ngày nay, đa số những mạch tích hợp có chức năng đặc biệt (ASIC) gồm nhiều hơn một triệu transistor. Thiết kế những mạch lớn như thế này dùng sơ đồ mạch (schematic capture) rất mất thời gian và không còn hiệu quả nữa. Do đó, cần một cách thiết kế hiệu quả hơn. Phương pháp mới này phải tăng được hiệu suất của người thiết kế va cho phép thiết kế dễ dàng thậm chí khi làm việc với những mạch lớn.

Từ sự đòi hỏi này nảy sinh việc chấp nhận rộng rãi ngôn ngữ mô tả phần cứng (HDL). HDL cho phép người thiết kế mô tả chức năng của một mạch logic trong một ngôn ngữ dễ hiểu. Việc mô tả sau đó sẽ được mô phỏng dùng testbench. Sau khi

mô tả HDL đã được kiểm tra về mặt chức năng (functionality) nó sẽ được tổng hợp thành những cổng logic bằng những công cụ tổng hợp.

Phương pháp này giúp người thiết kế thiết kế mạch trong thời gian ngắn hơn. Thời gian được tiết kiệm vì người thiết kế không cần quan tâm đến những phức tạp bên trong tồn tại trong một mạch cụ thể. Phương pháp thiết kế mới này đã được dùng rộng rãi trong lãnh vực thiết kế ASIC. Nó cho phép người thiết kế thiết kế một số lớn những cổng logic để thực hiện đặc tính và chức năng logic mong muốn của chip ASIC.

Những ngôn ngữ mô tả phần cứng được dùng rộng rãi trong công nghiệp ASIC là Verilog và VHDL. Mỗi cái có ưu khuyết điểm riêng. Phong cách viết mã cho hai ngôn ngữ này có những điểm tương đồng cũng như khác biệt.

Chương 2 : LUỒNG THIẾT KẾ ASIC

Thiết kế ASIC dựa vào một luồng thiết kế sử dụng ngôn ngữ mô tả phần cứng.Đa số những công cụ thiết kế điện tử tự động (EDA) được dùng cho luồng thiết kế ASIC đều tương thích với Verilog và VHDL.

Trong luồng này, thiết kế cũng như thực thi (implementation) một mạch logic đều được mã hóa dùng Verilog hay VHDL. Mô phỏng (simulation) được thực thi để kiểm tra chức năng của mạch logic.Tiếp theo sẽ là tổng hợp. Tổng hợp (synthesis) là quá trình biến đổi mã HDL thành cổng logic. Sau khi tổng hợp, bước tiếp theo là APR (auto place route) [sắp xếp các linh kiện và cách thức nối dây giữa chúng]. APR sẽ được giải thích chi tiết hơn trong phần 2.6.

Hình vẽ 2.1 chỉ ra một giản đồ của một luồng thiết kế ASIC bắt đầu với đặc tả của một thiết kế ASIC đến mã hóa RTL (Register transfer level) [viết mã Verilog để mô tả mạch] và cuối cùng là làm mẫu thử chip thật trên silicon (tapeout) [Đôi khi người ta còn gọi là Prototype].

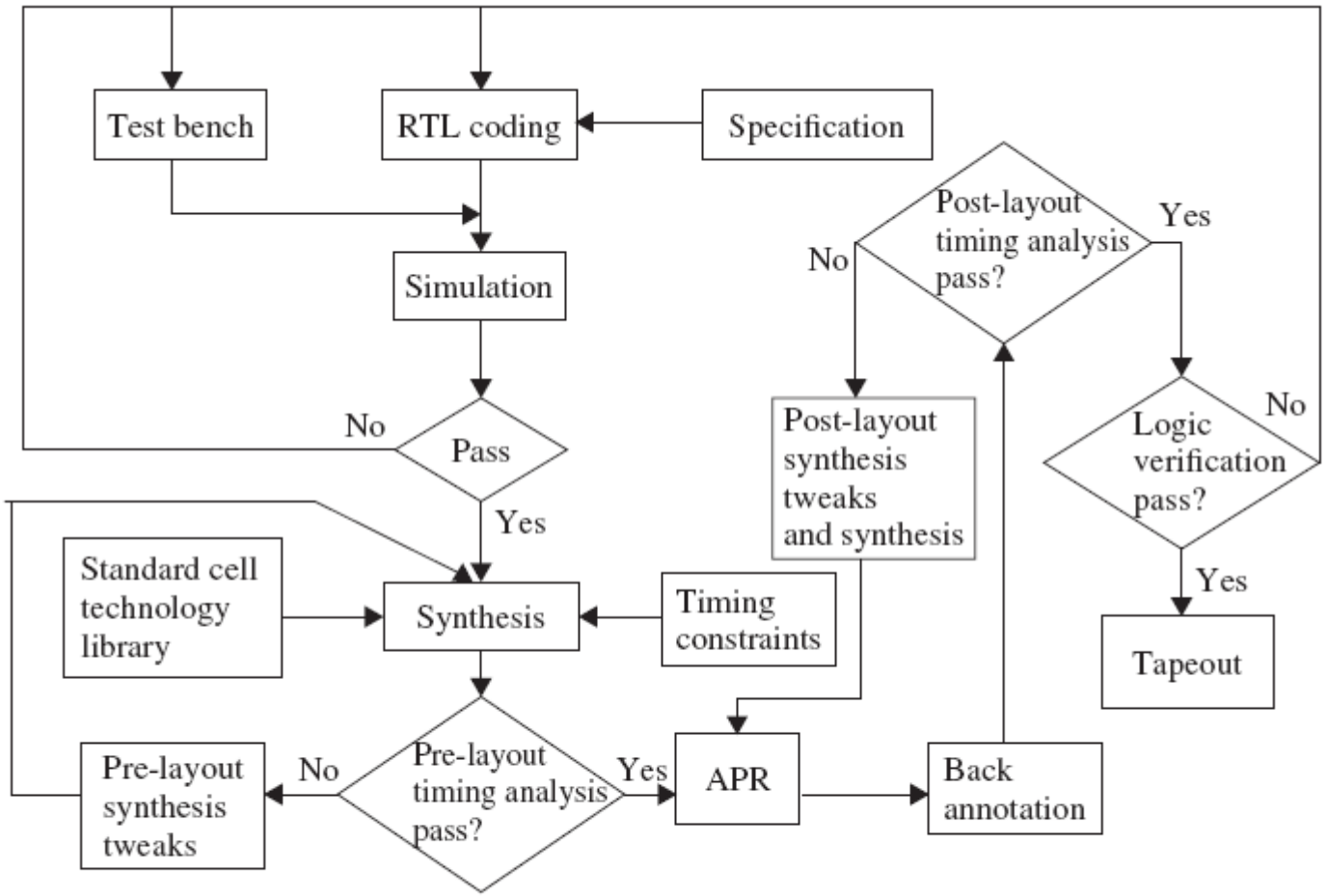


FIGURE 2.1. Diagram showing an ASIC design flow. Sections 2.1 to 2.9 explain each section of the ASIC flow in detail.

2.1 : Đặc tả (Specifications)

Hình 2.2 chỉ ra những bước đầu của luồng ASIC : đặc tả của một thiết kế. Đây là bước đầu tiên của luồng thiết kế ASIC. Việc thiết kế một chip ASIC bắt đầu từ đây.

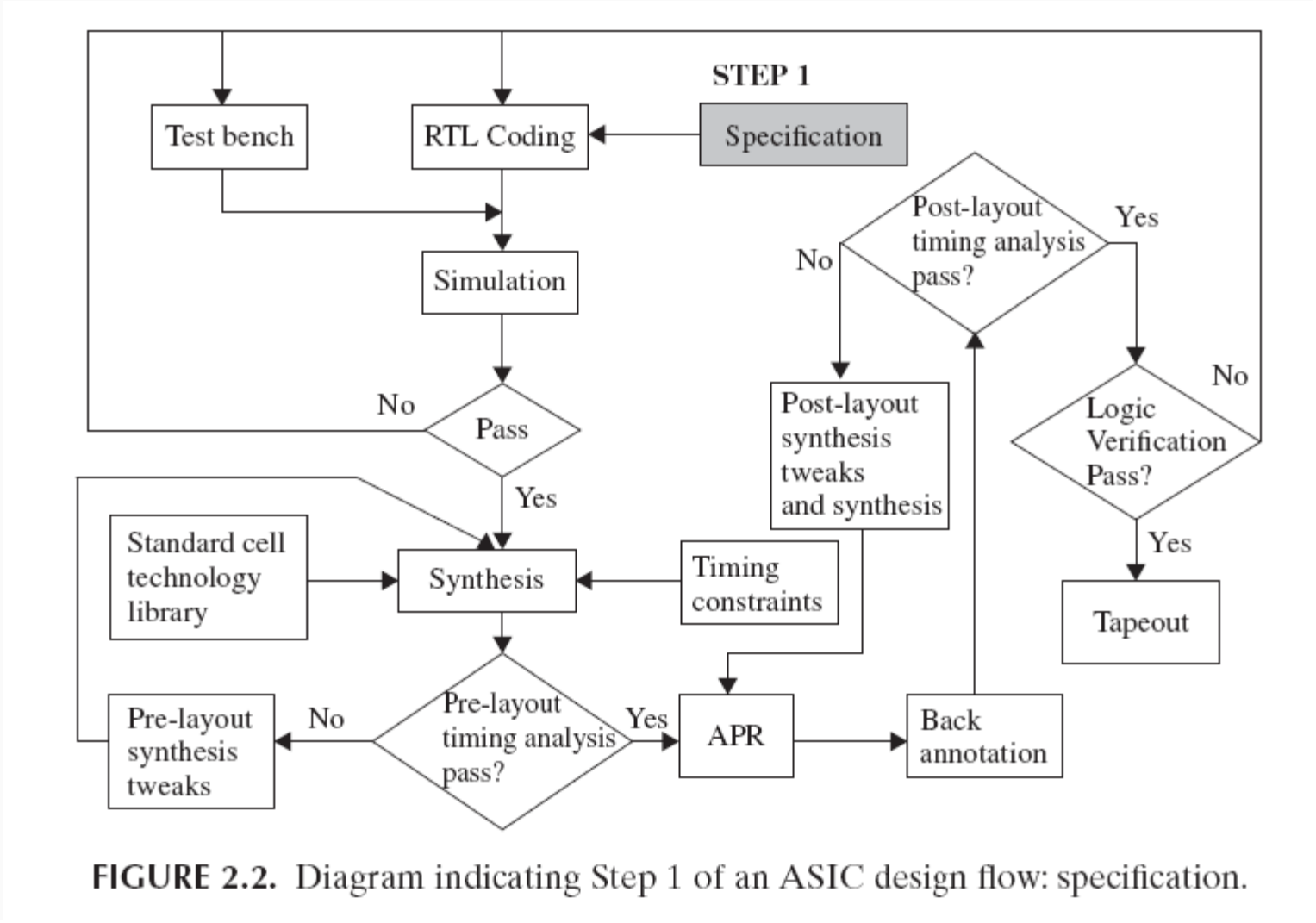


FIGURE 2.2. Diagram indicating Step 1 of an ASIC design flow: specification.

Đặc tả là phần quan trọng nhất của luồng thiết kế ASIC. Trong bước này, đặc tính và chức năng của chip ASIC được định nghĩa. Lên kế hoạch cho chip (Chip Planning) cũng được làm trong bước này. [Liên quan đến thời gian hoàn thành dự án, chi phí, diện tích chip ...]

Trong quá trình này, kiến trúc và vi kiến trúc được dẫn ra từ những đòi hỏi về chức năng và đặc tính (features). Việc dẫn ra này (derivation) đặc biệt quan trọng bởi kiến trúc của một thiết kế đóng vai trò quan trọng trong việc quyết định khả năng về hiệu suất (performance) của thiết kế. Bao gồm mức tiêu thụ công suất (power consumption), mức điện áp, những giới hạn về định thì (timing restrictions) và những tiêu chuẩn về hiệu suất. Từ danh sách này, kiến trúc chip sẽ được phác thảo nháp (draft). Kiến trúc này sau đó phải được xem xét tất cả những đòi hỏi về định thì, điện áp, tốc độ và hiệu suất của thiết kế. Những mô phỏng về kiến trúc cần được thực hiện trên kiến trúc nháp để bảo đảm rằng nó thỏa mãn tất cả những đặc tả mong muốn.

Trong quá trình mô phỏng kiến trúc, định nghĩa kiến trúc sẽ phải thay đổi nếu kết quả mô phỏng cho thấy nó không đáp ứng những yêu cầu về đặc tả.Một khi tất cả những yêu cầu đặc tả được thỏa mãn, vi kiến trúc (microarchitecture) được phác thảo và định nghĩa để cho phép thực thi kiến trúc từ một điểm thiết kế nào đó.

Vi kiến trúc là chìa khóa cho phép giai đoạn thiết kế được bắt đầu. Vi kiến trúc chính là điểm tiếp giáp của kiến trúc (Architecture) và mạch thực tế. Nó cũng cho phép biến đổi nhưng khái niệm về kiến trúc thành những thực thi thiết kế khả dĩ

2.2 : Mã hóa RTL

Hình vẽ 2.4 chỉ ra bước thứ hai trong luồng thiết kế ASIC. Đây là điểm khởi đầu của giai đoạn thiết kế. Vi kiến trúc được biến đổi thành một thiết kế bằng cách mô tả nó dưới dạng ngôn ngữ RTL. [Thực chất chỉ là mô tả lại sơ đồ khối mạch, chức năng nào đó bằng ngôn ngữ HDL]

Như đã đề cập trong phần 2.1 (Bước 1 của luồng thiết kế ASIC) kiến trúc và vi kiến trúc được dẫn ra từ đặc tả. Trong bước 2, vi kiến trúc, thực thi thật sự của mạch, được viết bằng mã RTL tổng hợp được. [Những mô tả có thể tổng hợp thành phần cứng được, vì trong ngôn ngữ như Verilog có rất nhiều thành phần chỉ dùng cho mô phỏng mà không tổng hợp được.]

Có rất nhiều cách để tạo ra mã RTL. Một số nhà thiết kế dùng công cụ nhập thiết kế đồ họa. Những công cụ đồ họa này cho phép người thiết kế dùng những giản đồ nút (bubble diagram), đồ thị luồng [mô tả máy trạng thái] hay bảng sự thật (truth table) để thực thi vi kiến trúc sau đó sẽ tạo ra mã Verilog hay VHDL. Tuy nhiên, một số nhà thiết kế lại thích viết mã RTL thay vì dùng công cụ đồ họa. Cả hai phương pháp đều kết thúc với kết quả là mã RTL tổng hợp được mà có thể mô tả chức năng logic của đặc tả.

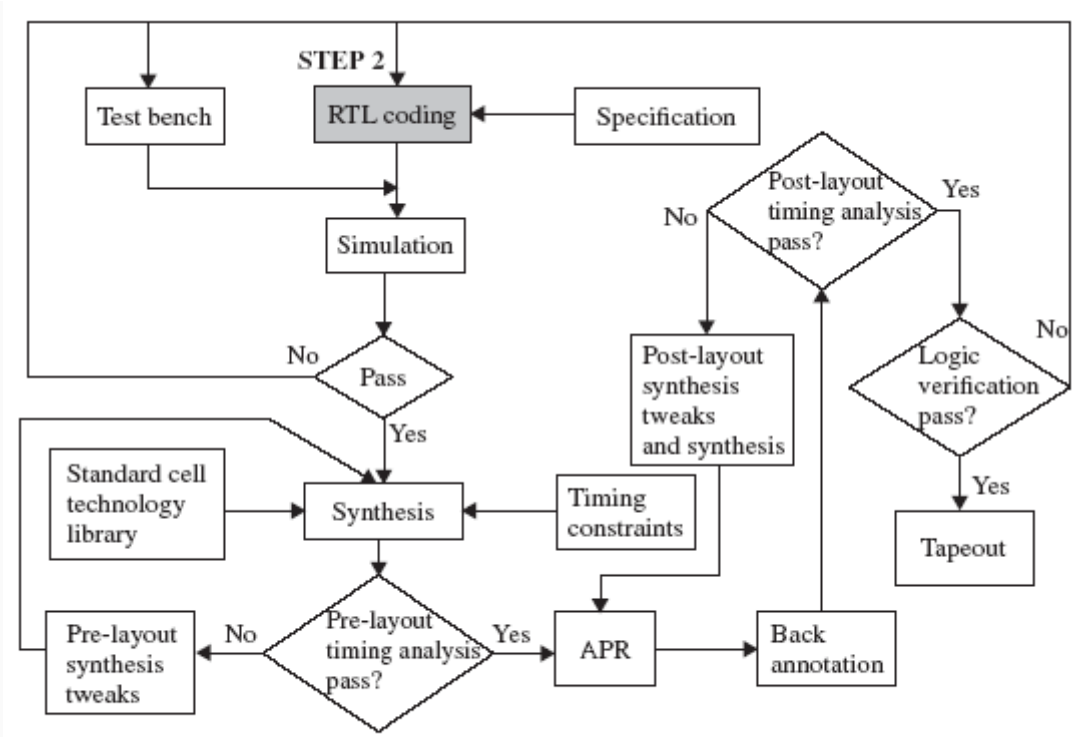


FIGURE 2.4. Diagram indicating Step 2 of an ASIC design flow: RTL coding.

2.2.1 : Những kiểu mã Verilog : RTL, Hành vi (Behavioral) và cấu trúc (strutural)

Phần 2.2 đã trình bày về mã RTL. Trong ngôn ngữ Verilog, có ba kiểu mã. Đối với đa số trường hợp tổng hợp thì mã RTL tổng hợp được sẽ được dùng.

2.3 : Testbench và mô phỏng

Hình 2.5 chỉ ra bước 3 trong luồng thiết kế ASIC liên quan đến việc tạo testbench. Những mã này sẽ dược dùng để mô phỏng mã RTL.

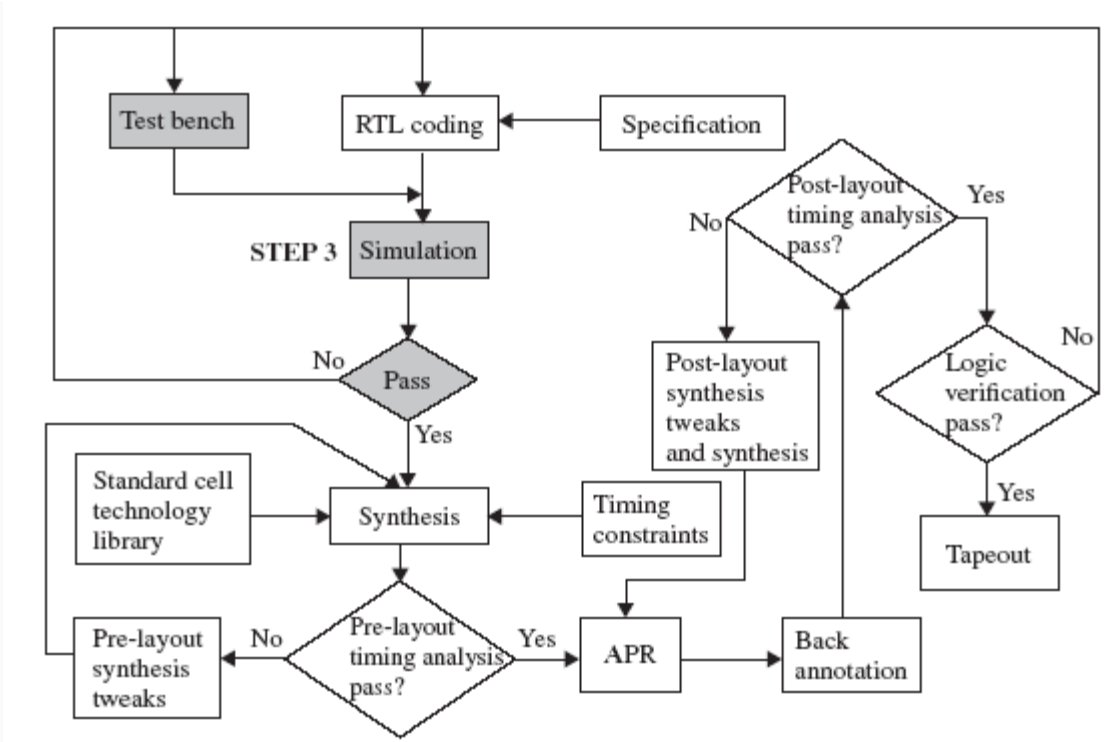


FIGURE 2.5. Diagram indicating Step 3 of an ASIC design flow: test bench and simulation.

Testbench về cơ bản là một môi trường đóng xung quanh (wraparound) bao lấy một thiết kế, cho phép thiết kế được mô phỏng. Nó đẩy tập hợp xác định những kích thích (stimulus) vào đầu vào (inputs) của thiết kế, kiểm tra hay xem xét ngõ ra (outputs) của thiết kế để bảo đảm rằng dạng sóng hay vector kết quả (pattern) phù hợp với mong muốn của người thiết kế

Mã RTL và testbench được mô phỏng dùng bộ mô phỏng HDL. Nếu mã RTL được viết bằng Verilog thì cần một bộ mô phỏng Verilog.Nếu viết bằng VHDL thì tất nhiên cần một bộ mô phỏng VHDL. Verilog XL của Cadence, VCS của Synopsys và ModelSim của Mentor Graphic's là những bộ mô phỏng phổ biến nhất trên thế giới hiện nay. NCSim của Cadence và ModelSim của Mentor Graphic's có khả năng mô phỏng cả Verilog và VHDL.Scirocco của Synopsys là một ví dụ của bộ mô phỏng VHDL. Ngoài những công cụ kể trên còn rất nhiều những bộ mô phỏng khác. Bất chấp bộ mô phỏng nào được dùng, kết quả cuối cùng là kiểm tra (verification) mã RTL của thiết kế dựa vào testbench được viết.

Nếu người thiết kế thấy rằng dạng sóng đầu ra hay vector kết quả (pattern) trong quá trình mô phỏng không khớp với cái mà họ chờ đợi thì thiết kế cần phải được gỡ rối (debug). Sự sai lệch có thể là do lỗi trong testbench hay là bug trong mã RTL. Người thiết kế cần xác định và sửa những lỗi này bằng cách chỉnh lại testbench (nếu nguyên nhân là do Testbench) hay thay đổi mã RTL nếu sai nằm ở mã RTL.

Sau khi hoàn tất việc thay đổi, người thiết kế phải chạy mô phỏng lại. Điều này sẽ được thực thi liên tục trong một vòng lặp cho đến khi người thiết kế thỏa mãn với kết quả mô phỏng. Điều này có nghĩa là mã RTL đã mô tả đúng hành vi logic của thiết kế.

2.4 : Tổng hợp

Hình 2.6 chỉ ra bước thứ 4 trong luồng thiết kế ASIC đó là tổng hợp. Trong bước này, mã RTL được tổng hợp. Đây là quá trình mà trong đó mã RTL được biến đổi thành cổng logic. Cổng logic được tổng hợp sẽ có cùng chức năng giống như đã được mô tả trong mã RTL.

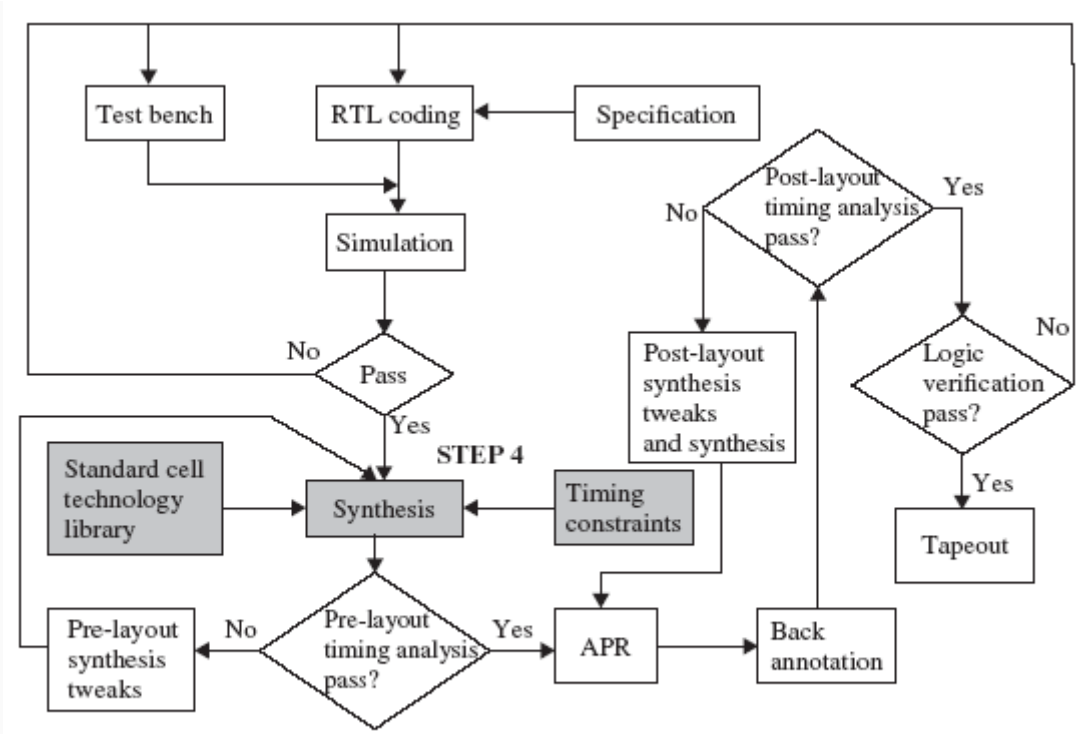


FIGURE 2.6. Diagram indicating Step 4 of an ASIC design flow: synthesis.

Trong bước 4, một công cụ tổng hợp dùng để biến đổi mã RTL thành cổng logic.Hai công cụ được dùng phổ biến trong công nghiệp là Design Compiler của Synopsys và Ambit của Cadence.

Quá trình tổng hợp cần hai tập tin đầu vào khác để thực hiện việc biến đổi từ RTL thành cổng logic. Tập tin đầu vào đầu tiên mà công cụ tổng hợp phải có trước khi thực hiện việc biến đổi là tập tin "thư viện công nghệ" (technology library file). Đó là tập tin thư viện chứa những cell chuẩn. [Cell dùng để chỉ rất nhiều mức mạch khác nhau, một cổng AND chẳng hạn nhưng có khi cả một khối RAM hay ALU cũng gọi là cell]. Trong quá trình tổng hợp chức năng logic của mã RTL được biến đổi thành những cổng logic sử dụng những cell sẵn có trong tập tin thư viện công nghệ. Tập tin đầu vào thứ hai là "tập tin giới hạn" (

constraints file) giúp quyết định việc tối ưu mạch logic tổng hợp.Tập tin này thường chứa những thông tin về định thì, yêu cầu tải và thuật toán tối ưu mà công cụ tổng hợp cần để tối ưu thiết kế thậm chí cả những nguyên tắc thiết kế cũng được xem xét trong quá trình tổng hợp.

Bước 4 là một bước rất quan trọng trong luồng thiết kế ASIC. Bước này bảo đảm việc tổng hợp được tùy biến nhằm có được kết quả tối ưu nhất có thể. Dựa vào bản tối ưu hóa cuối cùng, nếu những yêu cầu về hiệu suất hay tận dụng diện tích vẫn không nằm trong khoảng cho phép người thiết kế phải xem xét lại từ kiến trúc đến vi kiến trúc của thiết kế. Người thiết kế phải đánh giá lại kiến trúc cũng như vi kiến trúc đã đáp ứng những yêu cầu về diện tích và hiệu suất hay chưa?

Nếu vẫn chưa đáp ứng yêu cầu thì việc định nghĩa lại kiến trúc hay vi kiến trúc là việc làm bắt buộc tuy nhiên việc làm này sẽ dẫn đến việc phải bắt đầu lại từ đầu, một hành động rất mất thời gian. Thậm chí nếu việc thay đổi kiến trúc hay vi kiến trúc vẫn không mang lại kết quả mong muốn thì việc phải nghĩ đến là sửa chữa specs.

2.5 : Phân tích định thì tiền layout

Khi tổng hợp được hoàn thành trong bước 4, cơ sở dữ liệu cùng với những thông tin về định thì từ bước 4 được dùng để phân tích định thì tĩnh (static timing analysis). Trong bước 5, phân tích định thì là tiền layout vì cơ sở dữ liệu không chứa thông tin về layout.(Hình 2.7)

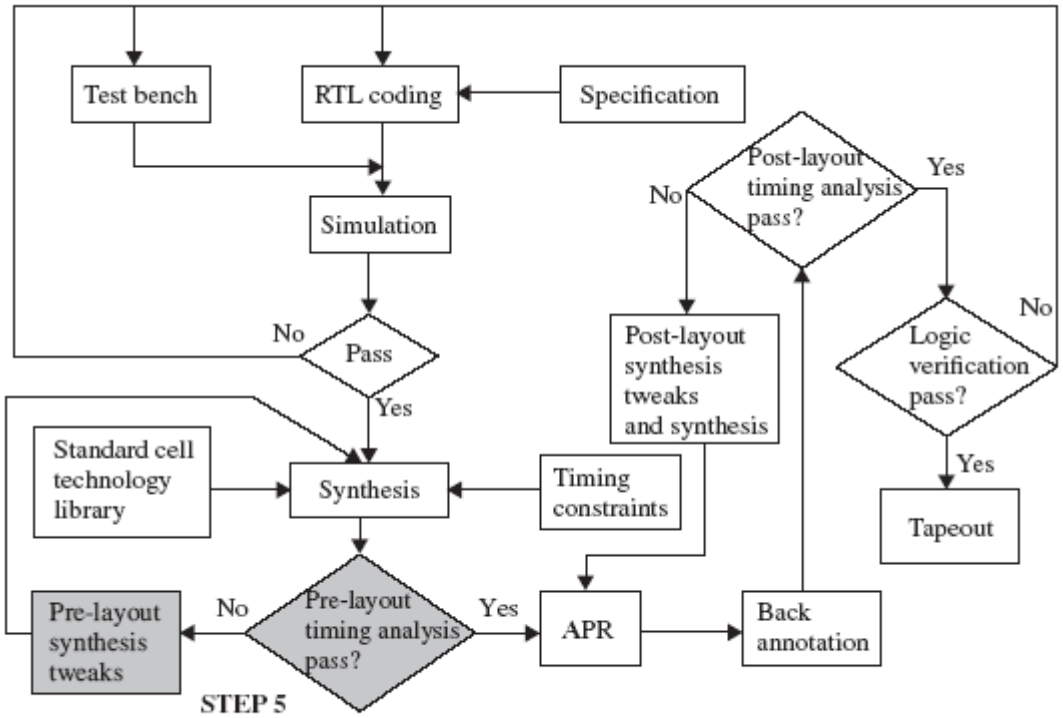


FIGURE 2.7. Diagram indicating Step 5 of an ASIC design flow: pre-layout timing analysis.

Mô hình định thì được xây dựng và phân tích định thì của nó được thực hiện trên thiết kế. Thường thì, phân tích định thì được thực hiện trên tất cả các khía cạnh của điện áp và nhiệt độ. [Mức điện áp cấp nguồn và nhiệt độ trong đó mạch hoạt động] Việc làm này nhằm bắt tất cả những vi phạm định thì trong thiết kế khi sử dụng trong những giải nhiệt độ hay điện áp xác định.

Bất kì một vi phạm định thì nào (timing violations) chẳng hạn vi phạm về thời gian setup và hold sẽ phải sửa bởi người thiết kế. Cách thông dụng nhất để sửa những vi phạm định thì này là thực hiện tùy biến tổng hợp (synthesis tweaks) để sửa

những đường sai định thì.

Cách thông thường để sửa vi phạm thời gian giữ (hold time violation) là đưa thêm những cell trễ (delay) vào trong đường mà thời gian giữ bị vi phạm. Để sửa vi phạm thời gian setup là giảm tổng trễ trên đường vi pham định thì loại này.

Những tùy biến tổng hợp này sau đó sẽ được dùng để tổng hợp lại thiết kế và việc phân tích định thì tĩnh sẽ được làm lại lần nữa.

Bước 5 trong luồng thiết kế ASIC đôi khi thay đổi tùy thuộc vào dự án thiết kế. Một số dự án thiết kế sẽ nhảy tới bước 6 dù cho có những vi phạm trong việc phân tích định thì tiền layout. Lý do là đây chỉ là tiền layout những ký sinh liên kết nối (interconnect parasitics) được dùng trong phân tích định thì chỉ mang tính ước lượng và có thể không chính xác.

Một phương pháp thông dụng hơn trong bước 5 là sửa những sai phạm về định thì vượt trên một giá trị định trước nào đó. Người thiết kế thiết lập một giá trị x nano giây cho phép vi phạm định thì. Đường nào vi phạm vượt hơn x nano giây sẽ phải sửa còn những đường vi phạm ít hơn sẽ không sửa.Điều này là do những ký sinh dùng trong định thì không chính xác do chưa có thông tin back annotation [thông tin truy vấn ngược về định thì]

2.6 : Sắp xếp và nối dây tự động (Auto Place and Route)

Một khi phân tích định thì tiền layout hoàn thành, cơ sở dữ liệu tổng hợp cùng với những thông tin về định thì từ tổng hợp được dùng cho APR.(Hình 2.8).

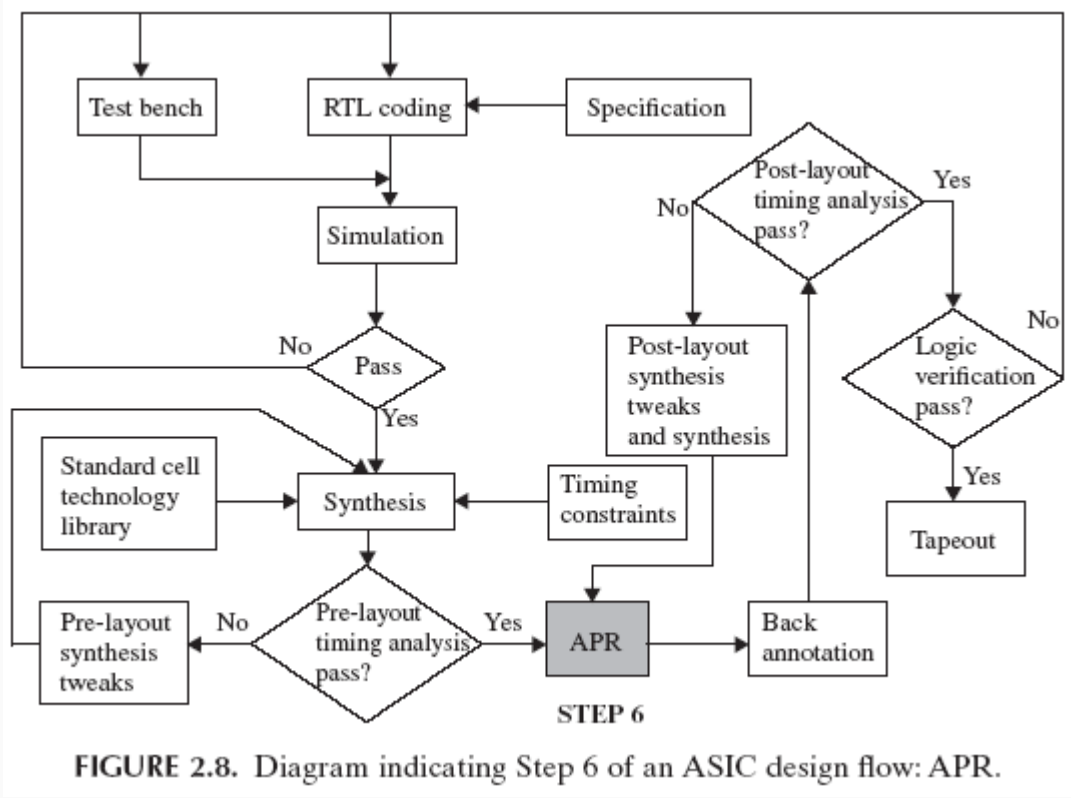


FIGURE 2.8. Diagram indicating Step 6 of an ASIC design flow: APR.

Trong bước này, những cổng logic đã được tổng hợp sẽ được sắp xếp và nối dây. Quá trình này có rất nhiều sự linh hoạt mà người thiết kế có thể dùng để sắp đặt cổng logic của mỗi module con (submodule) dựa vào kế hoạch làm nền đã được định từ trước. (predefined floor plan)

Đa số thiết kế có những đường găng (critical paths) rất chặt về mặt định thì. Những đường này có thể được xác định bởi người thiết kế bằng đường có mức ưu tiên cao (high priority paths). Công cụ APR sẽ nối những đường có mức ưu tiên cao trước nhằm đạt đến việc định tuyến tối ưu.

APR cũng là bước liên quan đến việc tổng hợp cây đồng hồ (clock tree) [Sự phân bố xung clock trong hệ thống]. Đa số những công cụ APR có thể thực hiện việc định tuyến "cây đồng hồ" với những thuật toán đặc biệt được xây dựng sẵn. Đây là

một phần quan trọng của luồng APR bởi vì việc xây dựng "cây đồng hồ" là rất tiên quyết bởi nếu được định tuyến đúng sẽ tránh được hiện tượng sai lệch clock (clock skew).

2.7 : Back Annotation

Back Annotation là bước trong luồng thiết kế ASIC ở đó ký sinh RC trong layout được trích ra. (Hình 2.9).

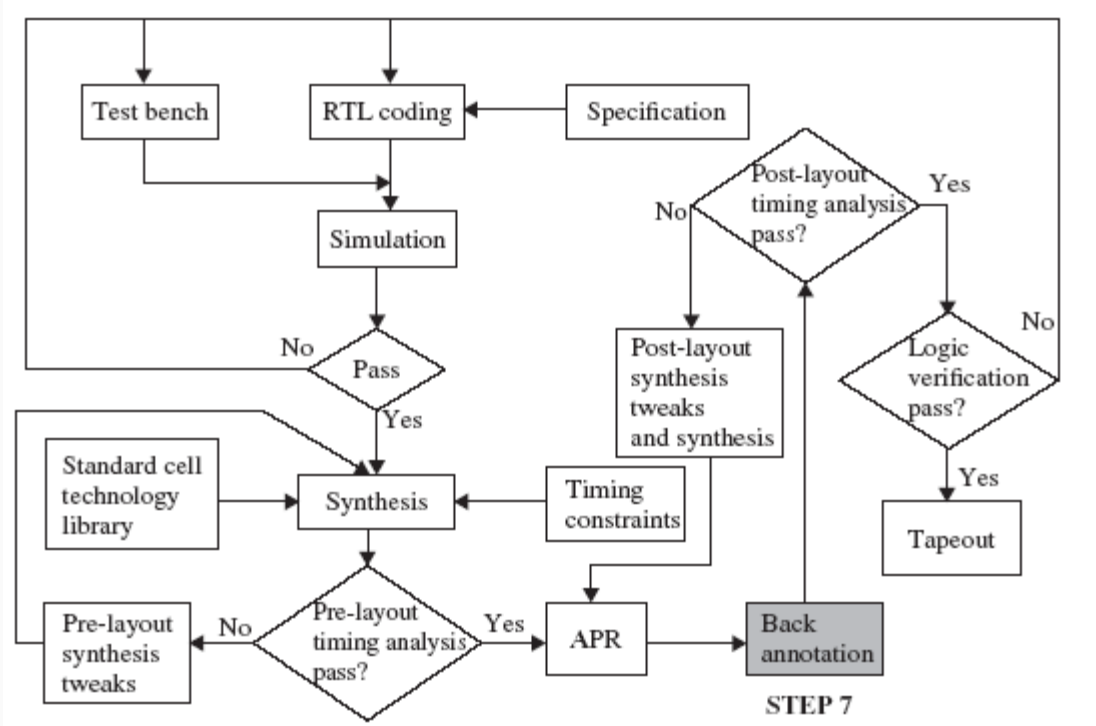


FIGURE 2.9. Diagram indicating Step 7 of an ASIC design flow: back annotation.

Đường trễ được tính từ những ký sinh RC này. Đối với những thiết kế thấp hơn micro mét rất nhiều, những ký sinh này có thể tạo ra sự gia tăng đường trễ đáng kể. Những đường định tuyến dài sẽ làm tăng trễ liên kết nối cho một đường nào đó. Một cách tiềm tàng, điều này làm cho những đường trước đây (trong bước định thì tiền layout) không găng trở thành găng

về định thì. Nó cũng làm cho những đường trước đây thỏa mãn những yêu cầu về định thì trở thành đường găng và không còn thỏa mãn yêu cầu định thì nữa.

Back Annotation là một bước quan trọng làm cầu nối cho sự khác biệt giữa tổng hợp và layout.Trong quá trình tổng hợp, những ràng buộc thiết kế được dùng bởi công cụ tổng hợp để tạo ra mạch logic mong muốn. Tuy nhiên, những ràng buộc này mang tính ước lượng được áp vào mỗi thiết kế. Những ràng buộc thực gây ra bởi ký sinh RC có thể phản ánh đúng hay sai những ràng buộc trước đó. Nhiều khả năng những ước lượng này là không chính xác. Kết quả là điều này gây ra sự sai khác giữa tổng hợp và layout. Back annotation là bước cầu nối giữa chúng

2.8 : Phân tích định thì sau layout

Phân tích định thì sau layout là một bước quan trọng trong luồng thiết kế ASIC cho phép "bắt" những vi phạm thời gian giữ/xác lập (hold/setup time violation) thực tế. (Hình 2.10)

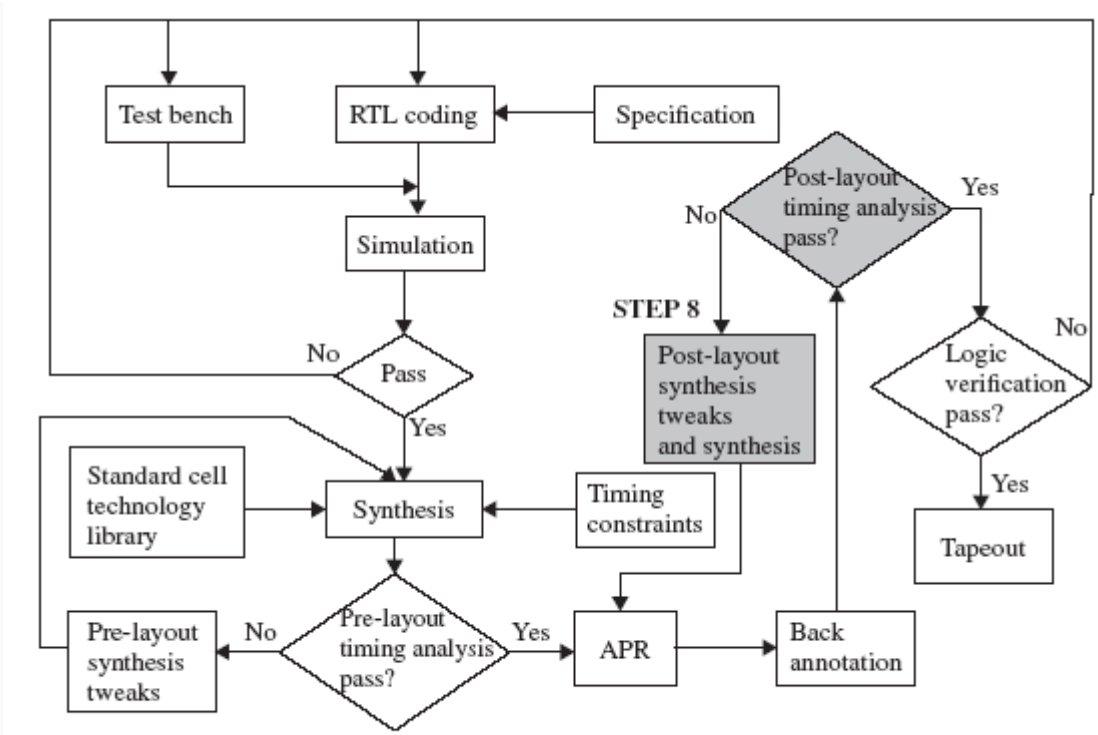


FIGURE 2.10. Diagram indicating Step 8 of an ASIC design flow: post-layout timing analysis.

Bước này tương tự như phân tích định thì tiền layout nhưng có thêm những thông tin về layout.

Trong bước này, thông tin trễ liên kết nối tổng (net interconnection delay) từ back annotation được đưa vào công cụ phân tích định thì để thực hiện phân tích định thì sau layout. Bất kì vi phạm xác lập (setup time) cần phải sửa bằng cách tối ưu hóa những đường sai xác lập để giảm đường trễ. Những vi phạm thời gian giữ (hold time) được sửa bằng cách đưa thêm những bộ đệm vào để tăng đường trễ lên.

Tùy biến tổng hợp sau layout được dùng sửa chữa những sai phạm định thì này trong quá trình tổng hợp lại. Điều này cho

phép tối ưu hóa những đường "thất bại".

Khi tổng hợp sau layout hoàn tất, APR, back annotation và phân tích định thì được thực hiện lại trong một vòng lặp cho đến khi không còn một vi phạm nào về định thì. Thiết kế bây giờ đã sẵn sàng cho việc kiểm tra logic

2.9 : Kiểm tra Logic (Logic Verification)

Khi phân tích định thì sau layout đã hoàn tất, bước tiếp theo là kiểm tra logic. Hình 2.11 mô tả điều này.

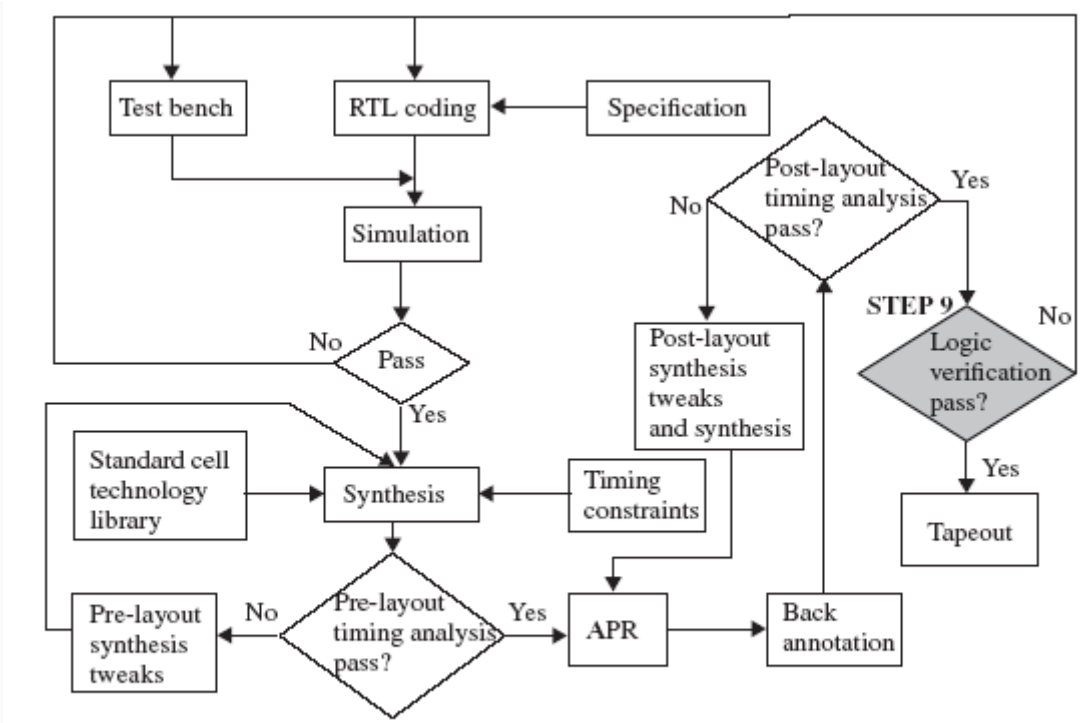


FIGURE 2.11. Diagram indicating Step 9 of an ASIC design flow: logic verification.

Bước này đóng vai trò chốt chặn cuối cùng bảo đảm thiết kế đúng chức năng. Trong bước này, thiết kế được mô phỏng lại sử dụng testbench đã có sẵn trong bước 3 nhưng có thêm thông tin định thì có được từ layout.

Mặc dù thiết kế đã được kiểm tra trong bước 3, thiết kế vẫn có thể không vượt qua được "kì sát hạch cuối cùng" này trong bước 9. Sự thất bại này có thể là do glitch [xung xuất hiện do sự chuyển đổi trạng thái của một tín hiệu gây ra cho đầu ra] hay điều kiện chạy đua (race conditions) do ký sinh layout. Nếu thất bại xảy ra người thiết kế phải đi ngược lại bước 2 (RTL coding) hay bước 8.

Khi thiết kế cuối cùng đã qua được vòng kiểm tra logic nó được đưa đi làm mẫu thử (tapeout).

Chương 3 : Mã Verilog

3.1: Giới thiệu những khái niệm cơ bản của Verilog

Verilog là một ngôn ngữ mô tả phần cứng được dùng rộng rãi trong thiết kế mạch số. Nó cũng được dùng để mô hình hóa cả mạch tương tự. Bất chấp Verilog được dùng cho mạch số hay mạch tương tự những khái niệm cơ bản của nó vẫn áp dụng đúng.

Khi một người thiết kế viết mã Verilog điều quan trọng là phải biết một số những ký hiệu cơ bản được dùng trong Verilog.

3.1.1 : Cú pháp Verilog

Verilog là một ngôn ngữ mô tả phần cứng (HDL) cho phép người dùng mô tả thiết kế phần cứng. Giống như tất cả các ngôn ngữ khác có những cú pháp phải tuân thủ khi viết mã Verilog.

Tất cả những cú pháp Verilog bắt đầu với việc khai báo module. Một module thực sự là một "chiếc hộp đen" hay "đơn vị" chứa thiết kế. Khai báo module phải bao gồm cổng giao tiếp của module. (module's interface ports)

module **design_module_name***(interface_port_list)*;

trong đó **design_module_name** là tên của module và *interface_port_list* là một danh sách chứa tất cả ngõ vào, ngõ ra hay ngõ ra/vào (inouts) của module. Mỗi cổng được phân tách bằng dấu phẩy (,).

Kiểu của cổng giao tiếp cũng được khai báo. Đó có thể là ngõ vào (inputs), ngõ ra (outputs) hay inout cho những cổng hai chiều.

```
module DUT (A, B, C, D, E);
input A, B, C;
inout D;
output B;
```

Nếu một cổng nào đó có nhiều hơn 1 bit ta phải dùng kí hiệu "[" và "]" để chỉ ra độ rộng của bus.

```
module DUT (A, B, C, D, E);
input [3:0] A, B;
input C;
inout [7:0] D;
output B;
```

3.1.2 : Ghi chú

Khi viết mã HDL cho một thiết kế, sử dụng ghi chú sẽ hình thành một thói quen tốt cho những nhà thiết kế mạch số. Chỉ ra cho người đọc cái mà mã Verilog thể hiện điều gì là một phương pháp tốt, đồng thời nó cũng là một dạng tài liệu tốt dành cho việc tham khảo sau này.

Verilog cho phép ghi chú một hàng hay nhiều hàng. Ghi chú một hàng sử dụng kí hiệu **//** trong đó những ghi chú nhiều hàng mở đầu với kí hiệu **/*** và kết thúc với kí hiệu ***/**.Ví dụ :

```
// This is a single line comment in Verilog

//* This is a multiple line comments in Verilog. Notice that it begins with a certain symbol and ends with a certain symbol*//
```

[Tất nhiên bạn hoàn toàn có thể viết comment bằng tiếng Việt không dấu.Ví lý do hiển thị trong bài post nên tôi dùng kí hiệu //* và *// thực tế bạn phải dùng là /* và */]

3.1.3 : Biểu diễn số

Verilog cho phép một dải rất rộng các loại số khác nhau được dùng trong khi viết mô tả. Người thiết kế có thể chọn dùng số thực, số nguyên, số nhị phân, số trong thang thời gian, số có dấu và số không dấu.

1. Số thực được khai báo trong Verilog dùng từ khóa *real*.Verilog cho phép số được khai báo ở dạng thập phân hay định dạng khoa học. [số có dấu chấm động]. Số thực cũng có thể được khai báo với giá trị âm.

```
module real_example();
real a,b,c;
initial
begin
a=3.141593;
b=314e-3;
c=-1.11;
end
endmodule
```

2.Số nguyên được khai báo trong Verilog dùng từ khóa *integer*, cũng có giá trị âm và dương cho kiểu số này.

```
module example();
integer i,j,k;
initial
begin
```

```
i = 150;
j = -150;
k = -32;
end
endmodule
```

3.Số có cơ số về cơ bản là những số nguyên nhưng được khai báo dùng một cơ số nhất định. Chúng có thể là bát phân (octal), thập lục phân (hexadecimal), thập phân (decimal) hay nhị phân(binary). Số có cơ số được khai báo trong định dạng sau:

```
<integer_name> = <bit_size>'<base_number><value>;
```

trong đó

- <integer_name> là tên của số nguyên mà ta cần dùng.
- <bit_size> là số bit nhị phân dùng để biểu diễn số nguyên.
- <base_number> là cơ số. Đó là *o* nếu là số bát phân, *h* nếu là số thập lục phân, *d* cho số thập phân hay *b* cho số nhị phân.
- <value> giá trị của số nguyên.

```
module example ();
integer i,j,k,l;
initial
begin
i = 5'b10111; // this is a binary number
j = 5'o24; // this is an octal number
k = 8'ha9; // This is a hex number
l = 5'd24; // This is a decimal number
end
endmodule
```

4.Số biểu diễn thang thời gian

Thời gian mô phỏng trong Verilog được khai báo với từ khóa *time*. Đơn vị cho thời gian được khai báo trong bộ định hướng biên dịch (compiler directive).Khai báo thang thời gian tuân thủ định dạng sau:

`timescale <reference_time>/<precision>;

trong đó : <reference_time> và <precision> phải là những giá trị nguyên như 1,10 hay 100. Tuy nhiên đơn vị thời gian được phép khai báo cùng với những giá trị nguyên này là fs (femto giây), ps (pico giây), ns(nano giây), us(micro giây), ms(mili giây) và s (giây).

```
module example ();
`timescale 100 us/1ns; // this is for
//reference of 100 us and precision of 1ns
```

[có nghĩa là mỗi giá trị nguyên sau này các bạn sử dụng sẽ được nhân với 100 us. Ví dụ bạn khai báo trễ là #20 ~~~> tức là thời gian trễ thực sự là 20 *100 = 20000 us. Nếu trong quá trình khai báo bạn muốn sử dụng số lẻ chẳng hạn #2.125 thí số lẻ tối đa bạn có thể dùng là 3 chữ số vì 1 us = 1000 ns]

```
time t;
initial
begin

t = $time; // $time is Verilog system function //that get the current simulation time
end
endmodule
```

3.1.4 Kiểu dữ liệu trong Verilog

Verilog cho phép hai kiểu dữ liệu là *reg* và *net*. *Reg* (viết tắt của register) là một phần tử lưu trữ (storage element) nó cho phép giá trị được lưu trữ trong kiểu giữ liệu này. Những giá trị này sẽ được lưu trữ trong kiểu dữ liệu này cho đến khi được thay bằng một giá trị khác. *Reg* chỉ được dùng trong những câu lệnh của khối *always* hay *initial*.

Kiểu dữ liệu thuộc nhóm *net* được dùng nhiều nhất là kiểu *wire* thường dùng để biểu diễn kết nối net. Nó giống như một đường dây nối trong phần cứng do đó, giá trị trên *wire* luôn được cập nhật liên tục.

Trong quá trình mô phỏng, nếu không có giá trị nào được gán vào những đối tượng được khai báo kiểu *reg* thì giá trị mặc định là không xác định hay X. Tương tự, nếu không có giá trị nào được gán cho những đối tượng kiểu *wire* thì giá trị mặc định là trạng thái thứ ba (tri - state) hay Z (Hi - Z).

Ví dụ 3.1 : chỉ ra một ví dụ đơn giản dùng *wire* trong khi ví dụ 3.2 tương tự 3.1 nhưng khác ở chỗ đó là khai báo dành cho bus 4 bit.

Ví dụ 3.1 : Mã Verilog dùng cho khai báo kiểu wire

```
module example (inputA, inputB, inputC, outputA);
input inputA, inputB, inputC;
output outputA;
wire temp;
assign temp = inputB | inputC;
assign outputA = inputA & temp;
endmodule
```

Ví dụ 3.1 : Mã Verilog dùng cho khai báo kiểu wire với bus 4 bit

```
module example (inputA, inputB, inputC, outputA);
input [3:0] inputA, inputB, inputC;
output [3:0] outputA;
```

```
wire [3:0] temp;
assign temp = inputB | inputC;
assign outputA = inputA & temp;
endmodule
```

Ví dụ 3.3 Chỉ ra một phương pháp thông dụng sử dụng *reg* trong đó vị dụ 3.4 tương tự 3.3 nhưng dùng cho bus 8 bit.

Ví dụ 3.3 : Khai báo dùng *reg*

```
module example (inputA, inputB, inputC, outputA);
input inputA, inputB, inputC;
output outputA;
reg outputA,temp;
always @ (inputA, inputB, inputC)
begin

if (inputA)
temp = 1'b0;
else
begin
if (inputB & inputC)

temp = 1'b1;

else

temp = 1'b0;

end
```

```
end

// more source code here

always @ (temp or inputC or inputA)

begin

if (temp)

outputA = inputC;

else

outputA = inputA;

end

endmodule
```

Ví dụ 3.4 : Khai báo dùng *reg* với bus 8 bit

```
module example (inputA, inputB, inputC, outputA);
input [7:0] inputA, inputB, inputC;
output [7:0]outputA;
reg [7:0] outputA,temp;
always @ (inputA, inputB, inputC)
```

```
begin

if (inputA)
temp = 8'b1111_0000; // dấu gạch dưới dùng để phân tách số cho dễ đọc
else
begin
if (inputB & inputC)

temp = 1'b1010_0101;

else

temp = 1'b1010_1111;

end

end

// more source code here

always @ (temp or inputC or inputA)

begin

if (temp == 8'b1101_1011)

outputA = inputC;

else
```

```
outputA = inputA;

end

endmodule
```

Ngoài khai kiểu *reg* và *wire* còn có 10 kiểu dữ liệu khác được dùng trong Verilog.

- 1.**Supply1** như tên đã chỉ ra, nó được dùng cho những *net* được kết nối đến nguồn VCC. Ta dùng từ khóa **supply1** khi khai báo giá trị *net* này. Ví dụ : **supply1** VCC.
- 2.**Supply0** dùng cho những *net* nào sẽ nối với đất (ground). Ta dùng từ khóa **supply0** khi khai báo *net* nào có kiểu này. Ví dụ : **supply0** GND.
- 3.**tri** là một kiểu *net* dùng để khai báo một *net* có nhiều hơn 1 driver muốn láy nó. [driver ở đây bạn có thể tưởng tượng là những lái xe, còn kiểu *net* ở đây giống như một chiếc xe hơi mà những tài xế này cùng ngồi trên cabin và ai cũng có thể cầm lái nó]. Trong ví dụ 3.5 ta sẽ thấy *net* **temp** được lái bởi nhiều driver.

Ví dụ 3.5 : Mã Verilog dùng khai báo ba trạng thái

```
module example ( inputA, inputB, inputC, outputA);

input inputA, inputB, inputC;

output outputA;

tri temp;

assign temp = inputA & ~inputB;

// Verilog code here

assign temp = inputA | ~inputB;
```

assign outputA = temp & inputC;

endmodule

Kiểu **tri** có thể tổng hợp được. Tuy nhiên, ta nên tránh dùng kiểu **tri** khi viết mã Verilog. Nếu một nút (node) nào đó được lái bởi nhiều driver khác nhau thì nút đó chỉ nên được lái bởi những bộ lái ba trạng thái. [Tức là trong một thời điểm nhất định chỉ có một driver lái tín hiệu mà thôi tùy thuộc vào tín hiệu cho phép trong những ngõ ba trạng thái. Ta sẽ bàn về vấn đề này sau].

- 4.**trior** cũng là loại *net* có nhiều ngõ lái đầu vào. Tuy nhiên nó khác kiểu **tri** bởi nó là một kiểu *net* kết nối OR. Điều này có nghĩa là nếu bất kì một driver nào trong số các driver đang lái kiểu net này ở mức logic 1 thì lập tức kiểu *net* này sẽ cho giá trị logic 1. **trior** không tổng hợp được và không dùng trong mã tổng hợp khi viết mô tả.
- 5. **triand** cũng dùng như loại *net* cho phép nhiều driver. Tuy nhiên, nó khác kiểu **tri** ở chỗ nó thuộc loại *net* kết nối AND. Điều náy có nghĩa là bất kì một trong số các driver đang lái nó có giá trị logic 0 thì lập tức kiểu *net* này có giá trị logic 0. **triand** cũng không tổng hợp được và không dùng trong viết mã tổng hợp.
- 6.**trireg** cũng là loại **net** cho phép nhiều driver nhưng khác ở chỗ nó là kiểu **net** mang tính dung (capacitive). Điều này có nghĩa là **net** này có khả năng lưu trữ giá trị. Nếu những driver đang lái **net** này có giá trị Hi-Z hay tổng trở cao thì net **trireg** vẫn giữ giá trị trước đó. Và cũng giống như những kiểu net trước, đây cũng là loại net không tổng hợp được.
- 7.**tri1** là loại net cho phép nhiều driver tuy nhiên khác với **tri** ở chỗ **tri1** sẽ giữ mức logic 1 nếu những driver đang lái nó ở trạng thái Hi-Z hay tổng trở cao.Đây cũng là một loại net nữa không tổng hợp được.
- 8.**tri0** Giống như **tri1** tuy nhiên net sẽ giữ mức logic 0 nếu các driver lái nó đếu đang ở trạng thái Hi-Z hay tổng trở cao.
- 9.**wand** dùng cho net có cấu hình nối dây kiểu AND trong đó nếu bất kì driver nào trong những driver đang lái **wand** có mức logic 0 thì **wand** sẽ ở mức logic 0.**wand** có thể tổng hợp được.
- 10.**wor** giống như **wand** ngoại trừ nếu bất kì driver nào trong số những driver đang lái **wor** có mức logic 1 thì **wor** lập tức có mức 1.Đây là loại **wor** tổng hợp được.

Lưu ý : Khi viết mã tổng hợp kiểu dữ liệu thường dùng cho nhóm kiểu **net** là **wire**. Những kiểu dữ liệu **wor**,**wand** hay **tri** đều có thể tổng hợp được nhưng ta nên tránh dùng những kiểu net này trong khi viết mã tổng hợp. Những loại net **trior**,**trireg**,**triand**,**tri1**, **tri0** đều không thể tổng hợp được.

Trong Verilog **net** hay **reg** có một trong bốn giá trị sau :

- **1** --- biểu diễn mức logic 1.
- **0** --- biểu diễn mức logic 0.
- **X** --- biểu diễn trạng thái bất kì. (Don't care)
- **Z** --- biểu diễn trạng thái tổng trở cao (Hi - Z).

Đối với những **net** cho phép nhiều driver lái nó thì mỗi driver có thể lái một trong 4 giá trị được chỉ ra ở trên, vậy giá trị nào sẽ được lái vào net ?

Giả sử net C được lái bởi hai driver A và B. Cả hai driver này đếu có thể lái 1 trong bốn giá trị 0,1,X và Z và do đó có thể có đến 16 tổ hợp có thể các giá trị trên các driver. Giá trị cuối cùng trên net C sẽ phụ thuộc vào kiểu mà net C được khai báo.

Ta sẽ dựa vào những bảng tóm tắt giá trị ra :

TABLE 3.1. Table indicating value on net C (net type *tri*) for different net values on drivers A and B

tri		*A*			
		1	0	Z	X
	1	*1*	*X*	*1*	*X*
	0	*X*	*0*	*0*	*X*
B	Z	*1*	*0*	*Z*	*X*
	X	*X*	*X*	*X*	*X*

TABLE 3.2. Table indicating value on net C (net type *trior*) for different net values on drivers A and B

Trior		*A*			
		1	0	Z	X
	1	*1*	*1*	*1*	*1*
	0	*1*	*0*	*0*	*X*
B	Z	*1*	*0*	*Z*	*X*
	X	*1*	*X*	*X*	*X*

TABLE 3.3. Table indicating value on net C (net type *triand*) for different net values on drivers A and B

Triand		*A*			
		1	0	Z	X
	1	*1*	*0*	*1*	*X*
	0	*0*	*0*	*0*	*0*
B	Z	*1*	*0*	*Z*	*X*
	X	*X*	*0*	*X*	*X*

TABLE 3.4. Table indicating value on net C (net type *trireg*) for different net values on drivers A and B

Trireg		*A*			
		1	0	Z	X
	1	*1*	*X*	*1*	*X*
	0	*X*	*0*	*0*	*X*
B	Z	*1*	*0*	*Previous value*	*X*
	X	*X*	*X*	*X*	*X*

TABLE 3.5. Table indicating value on net C (net type *tri1*) for different net values on drivers A and B

Tril		*A*			
		1	0	Z	X
	1	*1*	*X*	*1*	*X*
	0	*X*	*0*	*0*	*X*
B	Z	*1*	*0*	*1*	*X*
	X	*X*	*X*	*X*	*X*

TABLE 3.6. Table indicating value on net C (net type *tri0*) for different net values on drivers A and B

Tri0		*A*			
		1	0	Z	X
	1	*1*	*X*	*1*	*X*
	0	*X*	*0*	*0*	*X*
B	Z	*1*	*0*	*0*	*X*
	X	*X*	*X*	*X*	*X*

TABLE 3.7. Table indicating value on net C (net type *wand*) for different net values on drivers A and B

Wand		*A*			
		1	0	Z	X
	1	*1*	*0*	*1*	*X*
	0	*0*	*0*	*0*	*0*
B	Z	*1*	*0*	*Z*	*X*
	X	*X*	*0*	*X*	*X*

TABLE 3.8. Table indicating value on net C (net type *wor*) for different net values on drivers A and B

	Wor	*A*			
		1	0	Z	X
	1	*1*	*1*	*1*	*1*
	0	*1*	*0*	*0*	*X*
B	Z	*1*	*0*	*Z*	*X*
	X	*1*	*X*	*X*	*X*

3.1.5 : Sức tín hiệu (Signal Strength)

Phần 3.1.3 đã trình bày chi tiết về những kiểu khai báo khác nhau của một **net** cũng như việc dùng **reg** trong Verilog. Mỗi kiểu **net** hay **reg** có thể có một trong các giá trị 0, 1, X, Z.Mặc dù giá trị của net hay reg bị giới hạn bởi 4 giá trị đó, ta vãn có dùng 8 mức sức tín hiệu (signal strength) khác nhau. Mức "mạnh" (strength) của một **wire** hay **reg** thường được dùng trong trường hợp có tranh chấp (contention) xảy ra.

Lưu ý : *Trong khí viết mã tổng hợp thì rất ít khi "tính mạnh" (strength) được sử dụng do strength được dùng để giải quyết vấn đề tranh chấp trong mạch logic. Tuy nhiên, khi viết mã tổng hợp thì tốt nhất là tránh để xảy ra tranh chấp trong mạch thiết kế. Một ví dụ có "tranh chấp" sẽ được trình bày chi tiết ở chương 5.*

Ví dụ 3.6 chỉ ra một cách đơn giản để gán mức strength vào ngõ ra của thiết kế :

```
module example ( inputA, inputB, inputC, outputA, outputB);
```

```
input inputA, inputB, inputC;

output outputA, outputB;

and (strong1, weak0) and_gate_instance (outputA, inputA, inputB);

or (weak1, weak0) or_gate_instance (outputB, inputB, inputC);

endmodule
```

Trong quá trình tổng hợp, công cụ tổng hợp sẽ bỏ qua những gán strength.Ví dụ này sẽ được tổng hợp thành một cổng AND và một cổng OR.

3.2 Verilog gate - level primitive

[primitive là những khối đã được xây dựng sẵn trong Verilog như cổng NOT, AND, NOR ... người dùng chỉ cần nắm cú pháp khai báo của nó là có thể sử dụng được trong thiết kế của mình mà không cần phải khai báo trước]

Verilog cho phép những primitive mức cổng (gate-level) được thể hiện trong mô tả thiết kế. Những primitive này là những thành phần có sẵn trong Verilog và không cần đòi hỏi bất kì một thiết lập đặc biệt nào.

Một số primitive có thể tổng hợp được còn một số thì không. Danh sách những primitive mức cổng có thể dùng trong Verilog được trình bày dưới đây :

1.**pmos** Primitive này biểu diễn một transistor PMOS có hai ngõ vào và 1 ngõ ra.Cú pháp khai báo như sau :

```
pmos pmos_instance (output_signal, input_signal, gate_signal);
```

trong đó **pmos_instance** là tên của thể hiện biểu diễn transistor pmos, **output_signal** là tên của **net** được nối vào ngõ ra

của pmos transistor, **input_signal** là tên của **net** nối với ngõ vào của transistor pmos và **gate_signal** là tên của net nối vào cực cổng (Gate) của transistor pmos.

TABLE 3.10. Truth table for *pmos* transistor primitive

Input_signal	Gate_signal	Output_signal
0	0	0
0	1	Z
1	0	1
1	1	Z

2.**nmos** là từ khóa dùng khai báo một transistor nmos với cú pháp sau :

nmos pmos_instance (output_signal, input_signal, gate_signal);

với những giải thích về tín hiệu nối vào các ngõ của nmos giống như với pmos

TABLE 3.11. Truth table for *nmos* transistor primitive

Input_signal	Gate_signal	Output_signal
0	0	Z
0	1	0
1	0	Z
1	1	1

3.**cmos** primitive này dùng để biểu diễn một transistor cmos cổng truyền (passgate) với mô hình dưới đây :

cmos cmos_instance (output_signal, input_signal, NGate_signal, PGate_signal);

TABLE 3.12. Truth table for *cmos* passgate primitive

Input_signal	NGate_signal	PGate_signal	Output_signal
0	0	0	0
0	0	1	Z
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	Z
1	1	0	1
1	1	1	1

trong đó output_signal là tên của net nối vào ngõ ra của cmos cổng truyền này, input_signal là tên net nối vào ngõ vào của cmos, NGate_signal là tên net nối vào phần điều khiển bên N của cmos và PGate_signal dành cho điều khiển bên P.

4.**rpmos** hoạt động giống như **pmos** nhưng có tính trở hơn pmos do đó ngõ ra của rpmos có sức tín hiệu yếu hơn pmos.

5.**rnmos** hoạt động giống như **pmos** nhưng có tính trở hơn nmos do đó ngõ ra của rpmos có sức tín hiệu yếu hơn nmos.

6.**rcmos** Cũng giống như cmos nhưng vì có tính trở nên ngõ ra có sức tín hiệu yếu hơn cmos.

7.**pullup** như tên đã chỉ ra pullup dùng để biểu diễn một nút "kéo lên" với cú pháp:

pullup pullup_instance (signal_name);

trong đó signal_name là tên của tín hiệu bị "kéo lên".

8.**pulldown** tương tự như pullup.Bạn đọc tự suy ra cách sử dụng primitive này.

9.**tran** Primitive này biểu diễn một khóa hai chiều cho phép dữ liệu chuyển động theo cả hai chiều giữa hai net. Mô hình hóa như sau :

tran tran_instance (netA, netB);

10.**rtran** Giống như **tran** ngoại trừ do có tính trở nên sức tín hiệu ngõ ra yếu hơn tín hiệu ra của **tran**.

11.**tranif0** hoạt động giống như **tran** có điều nó chỉ cho phép truyền dữ liệu nếu tín hiệu Gate_control ở mức logic 0. Mô hình như sau :

tranif0 tranif0_instance (netA, netB, Gate_control);

12.**tranif1** hoạt động giống như **tran** có điều nó chỉ cho phép truyền dữ liệu nếu tín hiệu Gate_control ở mức logic 1. Mô hình như sau :

tranif1 tranif1_instance (netA, netB, Gate_control);

13.**rtranif0** hoạt động giống tranif0 ngoại trừ do có tính trở nên ngõ ra có sức tín hiệu yêu hơn so với tranif0.

14.**rtranif1** hoạt động giống tranif1 ngoại trừ do có tính trở nên ngõ ra có sức tín hiệu yêu hơn so với tranif1.

15.**notif0** primitive này biểu diễn một cổng ba trạng thái **đảo**. Nó có 2 ngõ vào và 1 ngõ ra có thể mô hình như sau :

notif0 notif0_instance (output_signal, input_signal, control_signal);

TABLE 3.13. Truth table for notif0 tri-state inverter primitive

Input_signal	Control_signal	Output_signal
0	0	1
0	1	Z
1	0	0
1	1	Z

16.**notif1** primitive này biểu diễn một cổng ba trạng thái **đảo**. Nó có 2 ngõ vào và 1 ngõ ra có thể mô hình như sau :

notif1 notif1_instance (output_signal, input_signal, control_signal);

TABLE 3.14. Truth table for notif1 tri-state inverter primitive

Input_signal	Control_signal	Output_signal
0	0	Z
0	1	1
1	0	Z
1	1	0

17.**bufif0** primitive này biểu diễn một cổng ba trạng thái. Nó có 2 ngõ vào và 1 ngõ ra có thể mô hình như sau :

bufif0 bufif0_instance (output_signal, input_signal, control_signal);

TABLE 3.15. Truth table for bufif0 tri-state buffer primitive

Input_signal	Control_signal	Output_signal
0	0	0
0	1	Z
1	0	1
1	1	Z

18.**bufif1** primitive này biểu diễn một cổng ba trạng thái. Nó có 2 ngõ vào và 1 ngõ ra có thể mô hình như sau :

bufif1 bufif1_instance (output_signal, input_signal, control_signal);

TABLE 3.16. Truth table for bufif1 tri-state buffer primitive

Input_signal	Control_signal	Output_signal
0	0	Z
0	1	0
1	0	Z
1	1	1

19.**buf** Primitive này dùng để biểu diễn một bộ đệm. Với cú pháp

buf buf_instance (output_signal, input_signal);

20. **not** Primitive này dùng để biểu diễn một cổng đảo. Nó có một ngõ vào và 1 hay nhiều ngõ ra.

not not_instance (output_signal1, output_signal2, ..., input_signal);

21. **and** Primitive này dùng để biểu diễn cổng AND. Nó có hai hay nhiều ngõ vào và 1 ngõ ra.Được mô hình như sau :

and and_instance (output_signal, input_signal1, input_signal2,...);

22.**nand** là primitive dùng để biểu diễn một cổng nand với 2 hay nhiều ngõ vào và 1 ngõ ra.

nand nand_instance (output_signal, input_signal1, input_signal2,...);

23.**nor** là primitive dùng biểu diễn một cổng NOR và cũng có một ngõ ra cùng hai hay nhiều ngõ vào.

nor nor_instance (output_signal, input_signal1, input_signal2 ...);

24.**or**, **xor**,**xnor** đều là những primitive dùng biểu diễn cổng logic với 2 hay nhiều ngõ vào và 1 ngõ ra.

3.3 Primitive do người dùng định nghĩa

Trong phần 3.2 ta đã trình bày về những primitive mức cổng được xây dựng sẵn trong Verilog. Ngoài những primitive này, người thiết kế cũng có thể tạo ra những primitive của riêng mình, những primitive này được gọi là primitive do người dùng định nghĩa. (User-defined primitive)

Nói chung, primitive do người dùng định nghĩa (UDP) là một module và được mô tả bởi người dùng. Module này được dùng trong Verilog bằng cách thể hiện nó. Có hai kiểu UDP mà người thiết kế có thể tạo ra, UDP tổ hợp và UDP tuần tự.

3.3.1 : UDP tổ hợp

UDP tổ hợp mô tả một module có tính chất tổ hợp. Điều này có nghĩa là module UDP chứa những phần tử logic tổ hợp để tạo ra ngõ ra. Ví dụ 3.7 chỉ ra cú pháp để mô tả một UDP tổ hợp

Ví dụ 3.7 :

```
primitive <primitive_UDP_name> (<output_port_list>,<input_port_list>);
output <output_port_list>;
input <input_port_list>;
table

<Truth_table_format_description_of_combinational_U DP_functionality>;
```

```
endtable
endprimitive
```

trong đó :

<primitive_UPD_name> là tên của primitive đang được định nghĩa.
<output_port_list> là tên của những cổng ngõ ra của UPD primitive.Lưu ý là primitive chỉ có một ngõ ra và ngõ ra này chỉ có độ rộng 1 bit.
<input_port_list> là tên của những ngõ vào. Lưu ý là mỗi ngõ vào cũng chỉ có độ rộng 1 bit.

<Truth_table_format_description_of_combinational_f unctionality> là bảng sự thật mô tả chức năng của primitive được định nghĩa.

Một ví dụ về bảng sự thật được minh họa bằng bảng 3.7

TABLE 3.17. Table showing functionality of UDP module UDP_GATE

InputA	InputB	InputC	OutputA
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Mã Verilog miêu tả primitive được định nghĩa bằng bảng 3.7

```
primitive udp_gate (outputA, inputA, inputB, inputC);

output outputA;

input inputA, inputB, inputC;

table

// inputA inputB inputC outputA

0 0 0 : 0;
0 0 1 : 0;
0 1 0 : 1;
0 1 1 : 1;
1 0 0 : 0;
1 0 1 : 0;
1 1 0 : 1;
1 1 1 : 1;

endtable

endprimitive

// to create a module that instantiates the UDP primitive

module example (input1, input2, input3, output1);
input input1, input2, input3;
output output1;
```

wire output1;

upd_gate udp_gate_instance (output1, input1, input2, input3);

endmodule

3.3.2 : UDP tuần tự

UDP tuần tự mô tả một module có bản chất tuần tự. Điều này có nghĩa là module UDP chứa một phần tử nhớ (storage element) mà có thể lưu một giá trị.

Cú pháp định nghĩa UDP tuần tự giống như UDP tổ hợp ngoại trừ trong đó có khai báo **reg**.

Bảng 3.18 trình bày bảng sự thật của một con chốt (latch) mà UDP định nghĩa.

TABLE 3.18. Table showing functionality of UDP module UDP_LATCH

data	clock	Q
0	0	<previous_value>
0	1	0
1	0	<previous_value>
1	1	1

primitive udp_latch (Q, data, clock);
output Q;
input data, clock;

```
reg Q; // Declare Q with a reg type to store values

initial

Q = 0;

table

// data clock Q(current) Q(next)

0 0 : ? : -;

0 1 : ? : 0;

1 0 : ? : -;

1 1 : ? : 1;

endtable
endprimitive

// create a module that instantiate the sequential udp above

module example (qout, indata, inclock);

input indata, inclock;

wire qout;
```

```
udp_latch (qout, indata, inclock);

endmodule
```

[Ví dụ minh họa trong bảng trên trình bày mô tả của một con chốt tích cực theo mức logic 1 của xung clock, tức là khi xung clock giữ giá trị logic 1 thì ngõ ra Q sẽ được cập nhật giá trị mới ở ngõ vào, còn nếu không thì ngõ ra Q vẫn giữ giá trị cũ]

Bảng 3.19 và ví dụ dưới đây sẽ trình bày một phần tử logic kích bằng cạnh giống như một Flip Flop

TABLE 3.19. Table showing functionality of UDP module UDP_POS_FLOP

data	clock	Q
0	rising edge	0
1	rising edge	1
1	others	<previous_value>
0	others	<previous_value>

```
primitive udp_pos_flop ( Q, data, clock);

output Q;

input data, clock;

reg Q;

initial
```

```
Q = 0;
table

// data clock Q(current) Q(next)
0 (01) : ? : 0;
1 (01) : ? : 1;
// no change in output values

0 (0x) : ? : -;
0 (0x) : ? : -;

// no change for negative edge

0 (10) : ? : -;
0 (10) : ? : -;

// no change for change in data

(??) ? : ? : -;
endtable
endprimitive

// to create a module that instantiates the sequential UDP primitive

module example ( qout, indata, inclock);

input indata, inclock;
```

```
output qout;

wire qout;

udp_pos_flop (qout, indata, inclock);

endmodule
```

[Lưu ý : rất ít khi ta dùng UDP trong viết mô tả Verilog. Nếu cần dùng một dùng đối tượng nào kiểu primitive ta sẽ khai báo nó dưới dạng một module hoàn chỉnh và thể hiện module này trong module mà ta cần dùng.]

Chương 4 : Phong cách Coding --- Phương pháp tổng hợp

Phong cách coding đóng vai trò quan trọng trong luồng thiết kế ASIC. Những mã HDL "tồi" thường không mang lại sự thuận lợi trong việc tối ưu hóa. Giá trị logic được tạo ra từ những công cụ tổng hợp phụ thuộc rất lớn vào cách mà mã được viết. Những mã viết "tồi" cũng sẽ dẫn đến những mạch logic không tối ưu như người ta vẫn nói : " Rác vào thì rác ra" (Garbage in, Garbage out)

Có những nguyên tắc chung cần tuân thủ khi đi vào phong cách lập trình. Bằng cách tuân theo những nguyên tắc này, sẽ có một phong cách viết mã tốt và xuyên suốt thống nhất.Điều này dẫn đến kết quả tổng hợp sẽ tối ưu.

4.1 : Cách đặt tên (Naming Convention)

Đối với một dự án thiết kế, một thông lệ đặt tên tốt là một điều cần thiết. Cách đặt tên thường thường là phần không được xem trọng khi viết mã HDL. Có được một cách đặt tên tốt dường như không quan trọng nhưng nếu không có một cách đặt tên nào có thể gây ra nhiều vấn đề cho những bước thiết kế sau đó đặc biệt là trong quá trình tích hợp chip. Việc kết nối tất cả các tín hiệu giữa các module của toàn bộ chip sẽ rất khó khăn nếu tên của tín hiệu không thống nhất.

Bằng cách định nghĩa một chính sách đặt tên, một tập hợp những quy định được áp dụng khi người thiết kế đặt tên các cổng

của một module. Nếu mỗi module trong cùng một chip tuân thủ tập hợp quy định thì sẽ dễ kết nối các tín hiệu trong chip hơn.

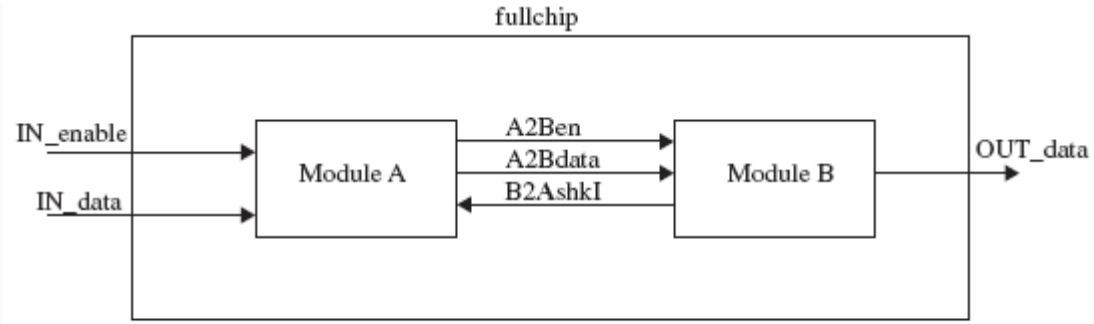


FIGURE 4.1. Diagram showing two submodules connected on a fullchip level.

Hình trên là giản đồ chỉ ra một chip chứa hai module, module A và module B. Đối với chip này, giả sử ta tuân thủ những nguyên tắc đặt tên sau :

1. Ba kí tự đầu tiên của chip phải là chữ viết hoa.
2. Kí tự đầu tiên phải biểu diễn tên của module mà tín hiệu là ngõ ra từ đó.
3. Kí tự thứ 3 phải biểu diễn tên của module mà từ đó tín hiệu là ngõ vào tới nó.
4. Kí tự thứ 2 phải là con số 2.
5. Kí tự thứ 4 trở đi là tên của tín hiệu và phải viết thường.
6. Tín hiệu truyền tới ngõ ra ở mức chip phải có 4 kí tự đầu là "OUT_".Tên tín hiệu sau 4 kí tự này phải là chữ thường.
7. Tín hiệu truyền từ ngõ vào ở mức chip tới một module nào đó trong chip phải có 3 chữ cái đầu là "IN_" và tên tín hiệu sau 3 kí tự này phải là chữ viết thường.
8. Bất kì tín hiệu nào tích cực thấp phải kết thúc với chữ "I".

Dựa vào những quy định đặt tên này, tên của tín hiệu "IN_enable","IN_data" và "OUT_data" là tín hiệu ngõ vào và ngõ ra ở mức chip. Tên của tín hiệu được liên kết nối giữa module A và module B là "A2Ben","A2Bdata" và B2AshkI".

Những quy tắc đặt tên dùng ở đây chỉ là một ví dụ. Một dự án thiết kế thực sự có thể dùng những nguyên tắc giống như trên và cũng có thể khác.

Ví dụ 4.1 trình bày mã Verilog cho module A và module B và chip cùng những liên kết nối giữa chúng.

```
module module_A (IN_enable, IN_data, B2AshkI, A2Ben, A2Bdata);

input IN_enable, IN_data, B2AshkI;
output A2Ben, A2Bdata;
// your Verilog code for module_A
endmodule

module module_B (A2Ben, A2Bdata, B2AshkI, OUT_data);

input A2Ben, A2Bdata;
output B2AshkI, OUT_data;
// your Verilog code for module_A
endmodule

module fullchip ( IN_enable, IN_data, OUT_data);

input IN_enable, IN_data;

output OUT_data;

wire A2Ben, A2Bdata, A2BshkI;

module_A module_A_instance ( IN_enable, IN_data, B2AshkI, A2Ben, A2Bdata);
```

```
module_B module_B_instance (A2Ben, A2Bdata, B2AshkI, OUT_data);

endmodule
```

4.2 : Phân vùng thiết kế (Design Partition)

Trong thực tiễn thiết kế người thiết kế thường phân chia thiết kế của mình ra thành nhiều module nhỏ hơn. Mỗi module được phân chia theo chức năng và đặc tính riêng của nó. Có một phân chia thiết kế hợp lý, người thiết kế có thể "rã" thiế kế thành những module nhỏ hơn do đó dễ quản lý hơn. Theo cách này người thiết kế có thể xác định chức năng của mỗi module và viết mô tả HDL cho từng module riêng biệt.

Tuy nhiên, người thiết kế cần cẩn thận khi phân vùng thiết kế. Mỗi module không thể quá nhỏ cũng không thể quá lớn. Phân vùng module quá nhỏ sẽ làm cho việc tổng hợp không tối ưu, module quá lớn thì khó viết mã và cũng không mang lại thuận lợi trong quá trình tổng hợp mạch. Kích thước module hợp lý sẽ dễ quản lý và cho phépo tổng hợp tốt hơn, một tiêu chuẩn trong phân chia là khoảng 5000 đến 15000 cổng cho một module riêng lẻ.

Một điểm nữa cần quan tâm khi tiến hành phân vùng thiết kế là tạo ra những khối kết nối. Phân vùng thành nhiều khối có thể xảy ra tình huống trong đó nảy sinh nhu cầu tạo ra nhiều đường kết nối hơn giữa các khối. Những tín hiệu bổ sung thêm này có thể gây ra sự tắc nghẽn trong giai đoạn layout vì có quá nhiều đường đan xen nhau. Do vậy, đòi hỏi người thiết kế phải có hiểu biết sâu sắc và đầy đủ về kiến trúc và vi kiến trúc của thiết kế trước khi cố phân vùng nó. Phân vùng tốt sẽ mang đến ưu điểm về quản lý trong khi phân vùng không tốt sẽ tạo ra sự tắc nghẽn (congestion) trong giai đoạn layout và làm cho việc quản lý thiết kế trở nên khó khăn hơn rất nhiều.

4.3 : Clock

Đa số thiết kế ASIC có ít nhất một xung clock, một số khác có nhiều hơn. Nếu thiết kế là đơn clock (single clock) hay đa clock (multi -clock), người thiết kế cần xem những khối clock này như clock toàn cục (global clock). Toàn cục nghĩa là mỗi clock được định tuyến qua tất cả các module trong thiết kế với tín hiệu clock xuất phát từ một module clock.

Module clock tạo ra clock toàn cục không tổng hợp được và được thiết kế dùng sơ đồ mạch. Những khối clock tuần tự sẽ được tích hợp với những khối logic khác ở mức độ chip.

Lưu ý : những khối tương tự (analogue block) không thể tổng hợp được. Trong luồng thiết kế ASIC những khối tương tự được thiết kế độc lập và được tích hợp với khối logic ở mức độ tích hợp chip. Người đọc nên ghi chú lại những khối này trong quá trình viết mã.

Ta dùng hình vẽ dưới đây để minh họa :

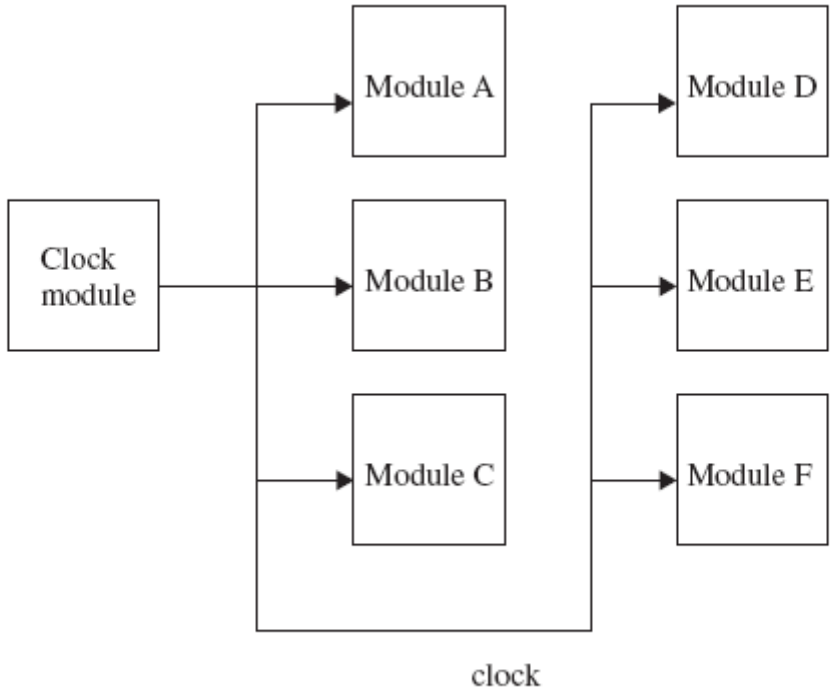


FIGURE 4.2. Diagram showing a fullchip level of global clock interconnect.

Dựa vào hình trên module A tới module F tổng hợp được mỗi module sẽ được mã hóa trong HDL được kiểm tra dùng HDL testbench và tổng hợp. Trong quá trình tích hợp mức độ chip những module clock sẽ được kết nối với những module logic khác.

Khi người thiết kế mã hóa module A tới module F, anh ta/ chị ta giả sử đầu vào clock là đầu vào clock toàn cục có thể thỏa mãn tất cả những đòi hỏi về sai lệch clock (clock skew). Ngõ vào clock toàn cục được giả sử có chu kì được xác định trong đặc tả thiết kế. Với những giả sử này, người thiết kế không thể đệm tín hiệu xung clock từ bên trong. Nói cách khác tín hiệu xung clock được mặc định là bất biến.

Xem tín hiệu xung clock là bất biến là một quan điểm thực tiễn. Bằng cách không đệm clock, người thiết kế giả sử tín hiệu clock có thể thỏa mãn tất cả những đòi hỏi về đặc tả mà thông thường không có được trong thực tế. Sai lệch clock (clock skew) phụ thuộc vào việc sắp đặt các cell và việc định tuyến tín hiệu clock. Do đó, trong quá trình mả hóa và tổng hợp, ta sẽ xem clock là bất biến và tránh đệm clock để loại bỏ hiệu ứng sai lệch clock. Công việc này sẽ được thực hiện trong quá trình tổng hợp "cây clock" là một phần công việc trong bước APR của luồng thiết kế ASIC đã trình bày trước đó trong chương 2. Thêm vào đó, tùy biến chu kì xung clock để có được tín hiệu clock mong muốn chỉ ảnh hưởng đến module tương tự trong hình vẽ ở trên.

Hình vẽ 4.3 là một thiết kế lý tưởng trong đó tín hiệu clock được kết nối trực tiếp với Flip Flop dùng trong thiêt kế mà không có bất kì một bộ đệm hay cổng logic nào trên đường clock. Người thiết kế nên cố gắng đạt đến mô hình lý tưởng này trong thiết kế mỗi khi có thể.

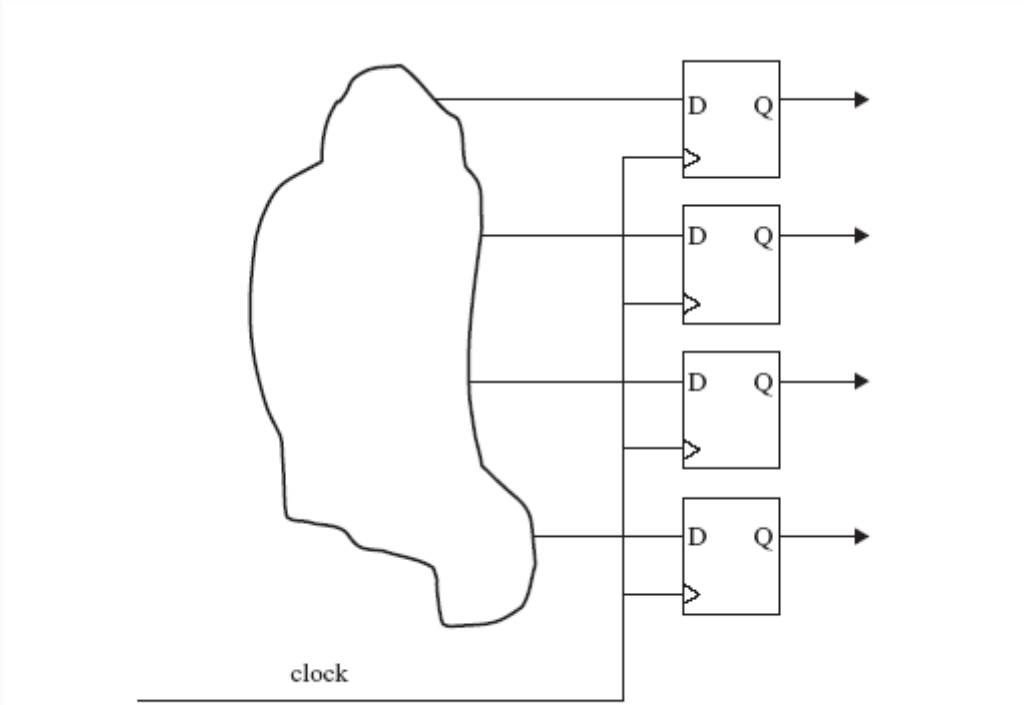


FIGURE 4.3. Diagram showing ideal connectivity of clock signal in a design.

Ưu điểm chính của việc tạo ra những mô hình thiết kế lý tưởng như trên là việc tạo điều kiện cho công cụ APR tổng hợp "cây clock" (clock tree) và đưa vào những bộ đệm clock mỗi khi cần thiết. Làm được như vậy thì những thay đổi trong sai lệch clock (clock skew) có thể được loại bỏ trong giai đoạn viết mã HDL.

4.3.1 : Tín hiệu clock được tạo ra bên trong

Một tín hiệu clock được tạo ra bên trong (internally generated clock) nên dùng càng ít càng tốt. Lý tưởng thì thiết kế tổng hợp không có bất kì một clock nào thuộc loại này.

Những thiết kế có Flip Flop hay chốt (latch) được clock bằng loại clock này sẽ làm phức tạp quá trình phân tích định thì bởi rất khó ràng buộc tín hiệu clock được tạo ra bên trong trong quá trình tổng hợp.

Hình 4.4 trình bày ngõ ra của một Flip Flop đang được dùng làm tín hiệu clock cho một flip flop khác.

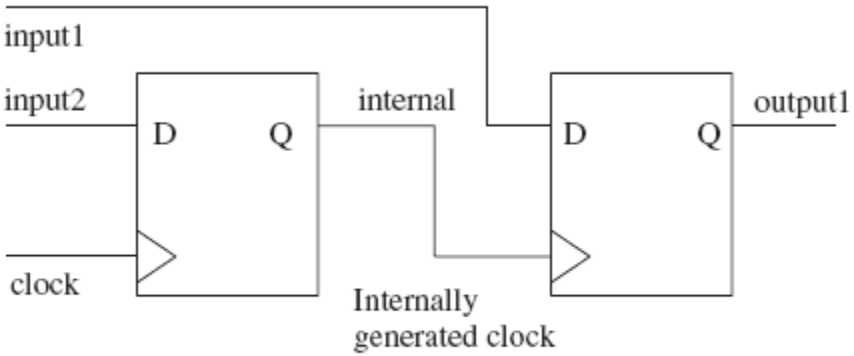


FIGURE 4.4. Diagram showing an output flip-flop driving another flip-flop.

Những thiết kế như thế này sẽ làm phức tạp quá trình ràng buộc định thì. Đa số những công cụ phân tích và tổng hợp gặp khó khăn với kiểu clock được tạo từ bên trong này.

Mã Verilog cho thiết kế ở hình 4.4

```
module internal_clock ( input1, input2, clock, output1);

input input1, input2, clock;

output output1;
```

```
reg internal, output1;

always @ ( posedge clock)

begin

internal <= input2;
end

always @ ( posedge internal)
begin

output1 <= input1;

end
endmodule
```

4.3.2 : Gated Clock

Một thiết kế có một tín hiệu cho phép nhằm "cho phép" một xung clock bên trong dựa vào clock toàn cục được gọi là "gated clock". Cụm từ này chỉ ra rằng tín hiệu clock toàn cục bị gắn với một tín hiệu nào đó để tạo ra tín hiệu clock bên trong.

Thiết kế có gated clock thường được dùng khi người thiết kế muốn tắt tín hiệu clock dưới một điều kiện nào đó do mục đích tiết kiệm năng lượng. Hình 4.5 chỉ ra một ví dụ có flip flop được clock bởi một tín hiệu gated clock được tạo ra từ điều kiện AND của tín hiệu "enable" và clock.

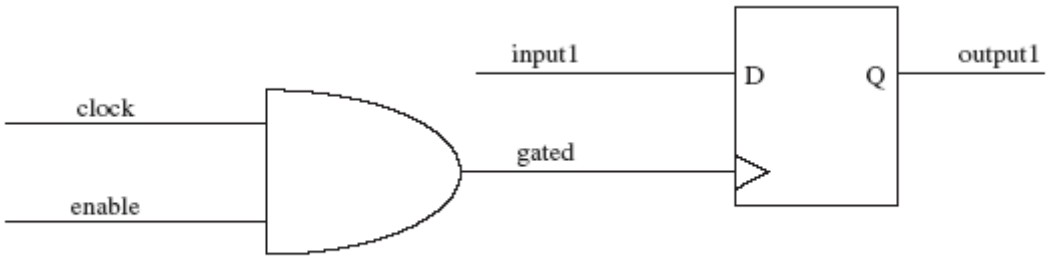


FIGURE 4.5. Diagram showing a gated clock driving a flip-flop.

Dựa vào ví dụ của hình 4.5 ta thấy có rất nhiều cách có thể dùng để tạo ra gated clock trong thiết kế. Phương pháp thông dụng nhất là dùng phép gán boolean và thể hiện cổng. Ví dụ 4.3 trình bày mã Verilog dùng phép gán Boolean còn ví dụ 4.4 dùng phương pháp thể hiện cổng.

Ví dụ 4.3 : Dùng phép gán Boolean

```
module gated_clock ( input1, enable, clock, output1);
input input1, enable, clock;
output output1;
wire gated;
reg output1;

assign gated = clock & enable;
always @ ( posedge gated)
begin
output1 <= input1;
end

endmodule
```

Ví dụ 4.4 : Dùng phương pháp thể hiện cổng

```
module gated_clock ( input1, enable, clock, output1);
input input1, enable, clock;
output output1;
wire gated;
reg output1;

AND_gate AND_instance (.I1(clock), .I2(enable), .O(gated));

always @ ( posedge gated)
begin
output1 <= input1;
end

endmodule
```

[Tất nhiên bạn phải có một module khai báo AND_gate trước và để tiện cho các bạn theo dõi tôi trình bày luôn ở đây.]

```
module AND_gate (I1, I2, O);
input I1, I2;
output O;

assign O = I1&I2;

endmodule
```

Trong hai phương pháp thì cách thể hiện cổng được ưu thích hơn khi làm việc với gated clock bởi khi thể hiện cổng cho phép

người thiết kế quản lý được khả năng **fanout** của tín hiệu bị gated. Ví dụ : ta giả sử rằng tín hiệu gated dùng để lái 32 flip flops như trong hình 4.6 :

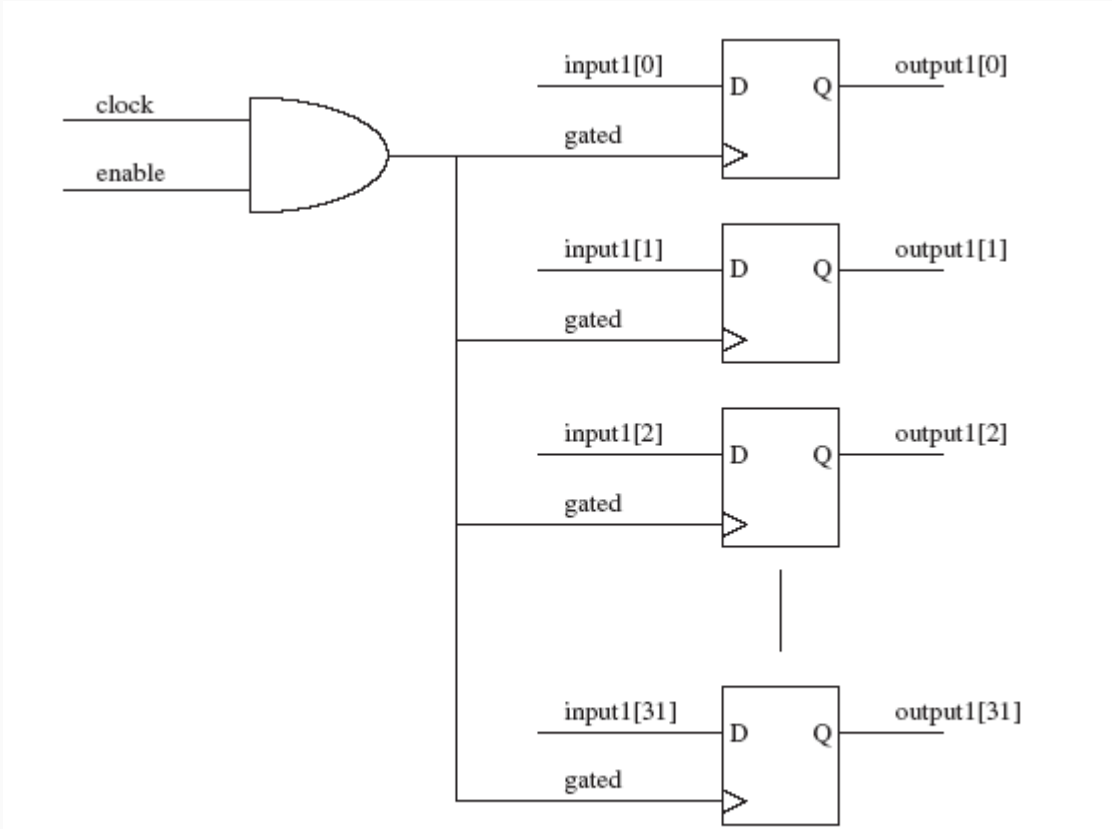


FIGURE 4.6. Diagram showing signal *gated* driving clock of 32 flip-flops.

Dựa vào hình 4.6 ta thấy là mức fanout lên đến 32 của tín hiệu gated có thể là gây quá tải cho cổng logic AND dẫn đến hiện tượng skew (sai lệch) trên tín hiệu gated rất lớn. Tất nhiên người thiết kế có thể đệm tín hiệu gated trong quá trình tổng hợp tuy nhiên đệm tín hiệu gated thường không được khuyến khích do đây là tín hiệu clock. Do vậy bất kì việc đệm tín hiệu gated

nào cũng sẽ được làm trong giai đoạn APR.

Do đó, một phương pháp tốt hơn sẽ là cách thể hiện cổng. Phương pháp này cho phép người thiết kế điều khiển khả năng tải trên cổng AND lái tín hiệu gated. Sử dụng cùng một ví dụ trong hình 4.6 người thiết kế có thể phân tín hiệu gated thành nhiều tín hiệu và mỗi tín hiệu sẽ chỉ lái một số flip flop giới hạn.

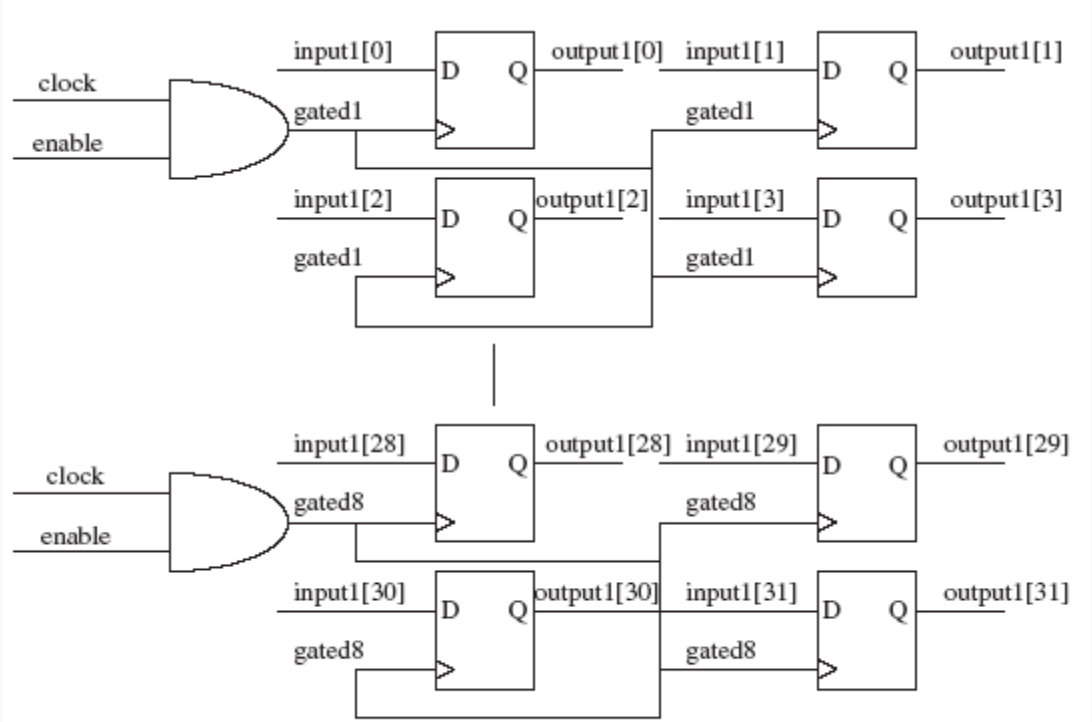


FIGURE 4.7. Diagram showing multiple *gated* signal to drive 32 flip-flops.

Dựa vào hình 4.7 tín hiệu clock và tín hiệu enable được dùng để tạo ra 8 tín hiệu gated phân biệt từ gated1 đến gated8. Mỗi tín hiệu gated chỉ lái 4 flip flop. Để làm được điều này người thiết kế thể hiện 8 cổng AND riêng biệt để tạo ra 8 tín hiệu gated khác nhau. Các này cho phép giảm tải trên mỗi tín hiệu gated và cho phép người thiết kế kiểm soát được hiện tượng skew trên tín hiệu gated. Vì dụ 4.5 trình bày mã Verilog phương pháp thể hiện cổng cho phép kiểm soát skew trên tín hiệu

gated.

Vì dụ 4.5 : Mã Verilog thể hiện cổng trong thiết kế gated clock

```
module gated_clock ( input1, enable, clock, output1);

input [31:0] input1;
input enable, clock;
output [31:0] output1;
wire gated1, gated5, gated2, gated3, gated4, gated6, gated7, gated8;

reg output1;

AND_gated AND_instance1 ( .I1(clock), .I2 (enable), .O(gated4));

AND_gated AND_instance9 ( .I1(clock), .I2 (enable), .O(gated3));
AND_gated AND_instance10 ( .I1(clock), .I2 (enable), .O(gated3));
AND_gated AND_instance11 ( .I1(clock), .I2 (enable), .O(gated3));
AND_gated AND_instance12 ( .I1(clock), .I2 (enable), .O(gated4));
AND_gated AND_instance13 ( .I1(clock), .I2 (enable), .O(gated4));
AND_gated AND_instance14 ( .I1(clock), .I2 (enable), .O(gated4));
AND_gated AND_instance15 ( .I1(clock), .I2 (enable), .O(gated4));
AND_gated AND_instance16 ( .I1(clock), .I2 (enable), .O(gated5));
AND_gated AND_instance17 ( .I1(clock), .I2 (enable), .O(gated5));
AND_gated AND_instance18 ( .I1(clock), .I2 (enable), .O(gated5));
AND_gated AND_instance19 ( .I1(clock), .I2 (enable), .O(gated5));
AND_gated AND_instance20 ( .I1(clock), .I2 (enable), .O(gated6));
AND_gated AND_instance21 ( .I1(clock), .I2 (enable), .O(gated6));
AND_gated AND_instance22 ( .I1(clock), .I2 (enable), .O(gated6));
```

```
AND_gated AND_instance23 ( .I1(clock), .I2 (enable), .O(gated6));
AND_gated AND_instance24 ( .I1(clock), .I2 (enable), .O(gated7));
AND_gated AND_instance25 ( .I1(clock), .I2 (enable), .O(gated7));
AND_gated AND_instance26 ( .I1(clock), .I2 (enable), .O(gated7));
AND_gated AND_instance27 ( .I1(clock), .I2 (enable), .O(gated7));
AND_gated AND_instance28 ( .I1(clock), .I2 (enable), .O(gated8));
AND_gated AND_instance29 ( .I1(clock), .I2 (enable), .O(gated8));
AND_gated AND_instance30 ( .I1(clock), .I2 (enable), .O(gated8));
AND_gated AND_instance31 ( .I1(clock), .I2 (enable), .O(gated8));
AND_gated AND_instance32 ( .I1(clock), .I2 (enable), .O(gated1));
AND_gated AND_instance2 ( .I1(clock), .I2 (enable), .O(gated1));
AND_gated AND_instance3 ( .I1(clock), .I2 (enable), .O(gated1));
AND_gated AND_instance4 ( .I1(clock), .I2 (enable), .O(gated1));
AND_gated AND_instance5 ( .I1(clock), .I2 (enable), .O(gated2));
AND_gated AND_instance6 ( .I1(clock), .I2 (enable), .O(gated2));
AND_gated AND_instance6 ( .I1(clock), .I2 (enable), .O(gated2));
AND_gated AND_instance7 ( .I1(clock), .I2 (enable), .O(gated2));

always @ ( gated1)

begin

output1[3:0] <= input1 [3:0];

end

always @ ( gated2)

begin
```

```
output1[7:4] <= input1 [7:4];

end
always @ ( gated3)

begin

output1[11:8] <= input1 [11:8];

end

always @ ( gated4)

begin

output1[15:12] <= input1 [15:12];

end

always @ ( gated5)

begin

output1[19:16] <= input1 [19:16];

end

always @ ( gated6)
```

```
begin

output1[23:20] <= input1 [23:20];

end

always @ ( gated7)

begin

output1[27:24] <= input1 [27:24];

end

always @ ( gated8)

begin

output1[31:28] <= input1 [31:28];

end
```

4.4: Reset

Mỗi thiết kế có một số tín hiệu reset đó là một đòi hỏi thường thấy vì thiết kế luôn cần phải "reset" về một trạng thái đã biết trong những điều kiện cụ thể nào đó.

Có hai kiểu reset : reset đồng bộ và reset bất đồng bộ. Cả hai đều reset thiết kế nhưng ảnh hưởng của chúng rất khác nhau.

4.4.1 : Reset bất đồng bộ (asynchronous reset)

Reset bất đồng bộ xảy ra bất kì lúc nào mà không có bất kì tham khảo nào về định thì của tín hiệu này với những tín hiệu khác và xả ra độc lập với bất kì điệu kiện hay tín hiệu nào khác trong mạch. Hình 4.8 chỉ ra một ví dụ đơn giản với reset giành cho flip flop. Ngõ ra của Flip Flop ở mức logic 0 mỗi khi có tín hiệu reset ở mức 1 [Người ta gọi tín hiệu reset kiểu này là reset tích cực cao]. Ví dụ 4.6 là mã Verilog cho kiểu reset bất đồng bộ này.

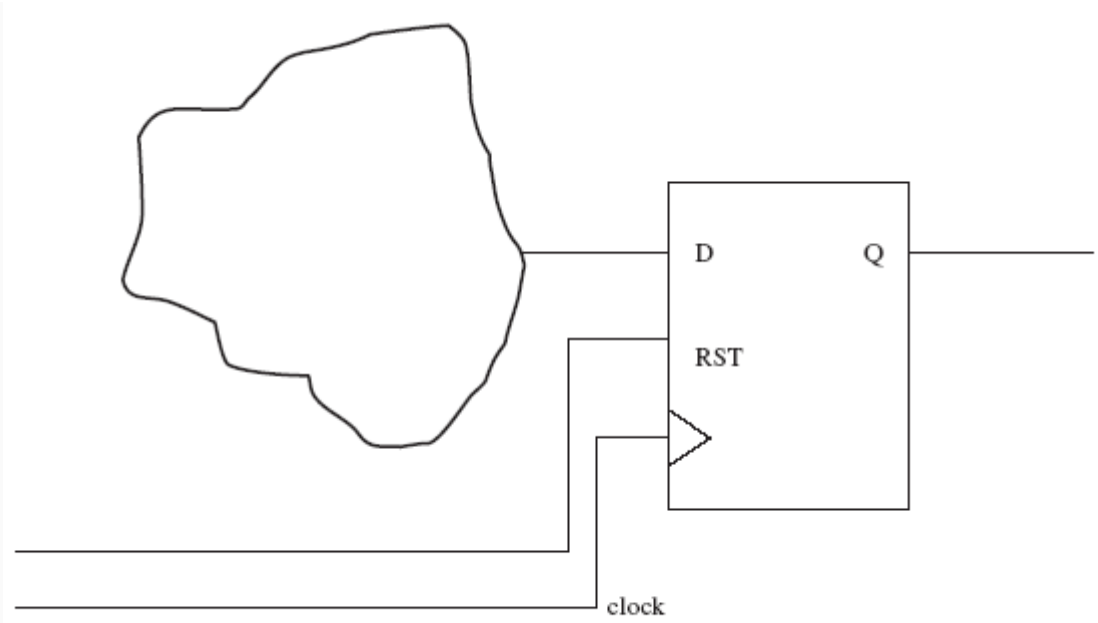


FIGURE 4.8. Diagram showing a design with an asynchronous reset flip-flop.

Ví dụ 4.6 : Mã Verilog cho reset bất đồng bộ

```
module asynchronous_reset ( input1, reset, clock, output1);
```

```
input input1, reset, clock;

output output1;

reg output1;

always @ ( posedge clock or posedge reset)
begin

if (reset)

output1 <= 1'b0;

else

output1 <= input1;

end

endmodule
```

4.4.2 : Reset đồng bộ

Reset đồng bộ là reset xảy ra ở cạnh lên (rising edge) hay cạnh xuống (falling edge) của tín hiệu xung clock. Điều này có nghĩa là reset đồng bộ chỉ được xảy ra ở cạnh của clock hay nói cách khác reset này "tham khảo định thì" của tín hiệu clock và không thể xảy ra độc lập với tín hiệu clock. Hình 4.9 chỉ ra một ví dụ đơn giản của thiết kế có reset đồng bộ. Giá trị ngõ ra của flip flop được cập nhật ở cạnh lên của xung clock. Ngõ ra của flip flop sẽ ở mức logic 0 nếu trong cạnh lên của clock tín hiệu reset có mức logic 1 nếu không thì giá trị ngõ ra của flip flop sẽ là giá trị logic của ngõ vào ở cạnh lên của xung clock.

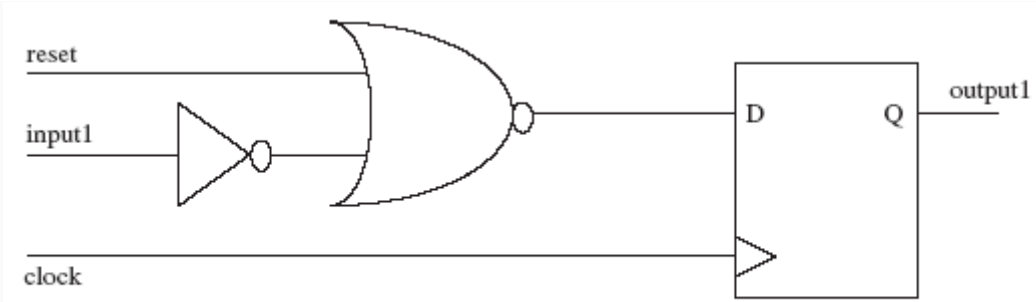


FIGURE 4.9. Diagram showing a design with a synchronous reset flip-flop.

```
module synchronous_reset ( input1, reset, clock, output1);

input input1, reset, clock;

output reg output1;

always @ ( posedge clock)

if ( reset)

output1 <= 1'b0;

else

output1 <= input1;

endmodule
```

4.5 : Vòng lặp định thì (Timing Loop)

Vòng lặp định thì là những vòng lặp trong thiết kế có ngõ ra của mạch logic tổ hợp được hồi tiếp ngược lại thành một phần của ngõ vào tổ hợp.

Đối với những thiết kế đã tổng hợp thì không được có những vòng lặp định thì này. Nếu có những vòng lặp định thì kiểu này sẽ làm phức tạp quá trình phân tích định thì do ngõ ra được đưa ngược trở lại ngõ vào. Hình 4.10 trình bày một ví dụ có vòng lặp định thì lưu ý việc ngõ ra của mạch đảo được đưa ngược về như ngõ vào của cổng AND.

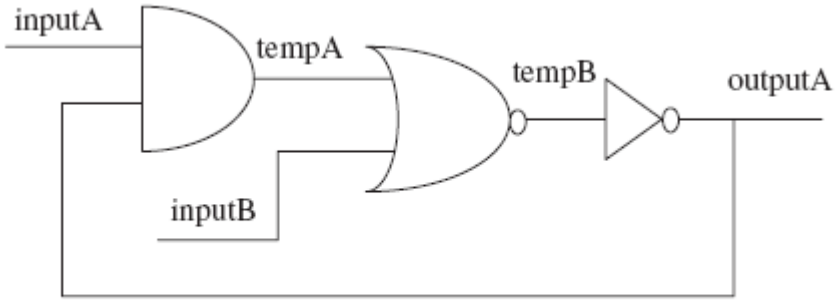


FIGURE 4.10. Diagram showing a design with timing loop.

Khi thiết kế một mạch có vòng lặp định thì người ta khuyên nên chia nó ra bởi một phần tử tuần tự. Điều này bảo đảm vòng lặp định thì mà có thể gây ra glitch có thể được chia thành hai phần : phần trước tuần tự và sau tuần tự.

```
module timing_loop ( inputA, inputB, outputA);

input inputA, inputB;

output outputA;
```

```
wire tempA, tempB;

assign tempA = inputA & outputA;
assign tempB = ~(tempA | inputB);
assign outputA = ~tempB;

endmodule
```

4.6 : Câu lệnh blocking và non blocking

Blocking và Non blocking là những câu lệnh gán thủ tục được dùng trong mã Verilog. Cả hai phép gán này được dùng trong những câu lệnh gán tuần tự. Mỗi câu lệnh gán blocking hay non blocking có những đặc tính và hành vi khác nhau.

Câu lệnh gán blocking được biểu diễn bằng ký hiệu "=". Khi một câu lệnh blocking được dùng câu lệnh sẽ được thực thi trước khi bộ mô phỏng (simulator) đi tới câu lệnh tiếp theo nói cách khác, câu lệnh blocking thực sự là tuần tự.

Câu lệnh non blocking được biểu diễn bằng kí hiệu "<=". Khi một câu lệnh gán non blocking được dùng câu lệnh này được lên kế hoạch (scheduled) và thực thi cùng với những câu lệnh non blocking khác. Điều này có nghĩa là non blocking cho phép nhiều phép gán được lên kế hoạch và thực thi cùng với nhau kết quả là câu lệnh gán non blocking **không phụ thuộc vào thứ tự phép gán được viết trong mô tả Verilog.** Những vì dụ từ 4.8 đến 4.15 sẽ giải thích sự khác biệt giữa câu lệnh gán blocking và non blocking.

Lưu ý : câu lệnh gán blocking và non blocking chỉ có trong mã Verilog trong khi VHDL không cần khái niệm này. [Thực ra khi tổng hợp thì dù viết mã non blocking hay blocking thì mạch cũng sẽ như nhau, chia thành hai loại như thế này nhằm tạo điều kiện cho việc mô phỏng]

Vì dụ 4.8 trình bày mã Verilog của một thiết kế đơn giản dùng câu lệnh gán non blocking. Module này về cơ bản là một thiết kế thanh ghi có reset đồng bộ.

```
module non_blocking ( clock, input1, reset, output1, output2, output3);
input reset, clock;
input [3:0] input1;
output [3:0] output1, output2, output3;

always @ ( posedge clock)
begin

if (reset)
begin

output1 <= 4'b0000;
output2 <= 4'b0000;
output3 <= 4'b0000;
end

else

begin

output1 <= input1;
output2 <= output1;
output3 <= output2;
end

end

endmodule
```

Vì dụ 4.9 : Thay đổi thứ tự gán của những phép gán non blocking

```
module non_blocking ( clock, input1, reset, output1, output2, output3);
input reset, clock;
input [3:0] input1;
output [3:0] output1, output2, output3;

always @ ( posedge clock)
begin

if (reset)
begin

output1 <= 4'b0000;
output2 <= 4'b0000;
output3 <= 4'b0000;
end

else

begin

output1 <= input1;
output3 <= output2; // Rearrange this line
output2 <= output1;

end

end
```

```
endmodule
```

Ví dụ 4.10 :

```
module non_blocking ( clock, input1, reset, output1, output2, output3);
input reset, clock;
input [3:0] input1;
output [3:0] output1, output2, output3;

always @ ( posedge clock)
begin

if (reset)
begin

output1 <= 4'b0000;
output2 <= 4'b0000;
output3 <= 4'b0000;
end

else

begin

output2 <= output1;
output3 <= output2; // Rearrange this block
output1 <= input1;
```

```
end

end

endmodule
```

Ta sẽ thấy là mã Verilog trong cả hai ví dụ trên cho cùng một kết quả mô phỏng ngoại trừ việc thứ tự sắp xếp của chúng có khác nhau. Ta sẽ dùng một testbench đơn giản để kiểm tra những thiết kế này.

Ví dụ 4.11 : Testbench cho những ví dụ trên

```
module nonblocking_tb ();

reg [3:0] input1;
reg clock, reset;
wire [3:0] output1, output2, output3;

// Generating initial stimulus for inputs and global clock

initial

begin

clock = 0;
input1 = 0;

forever #50 clock = ~clock; // for clock
end
```

```
// Test vectors

initial
begin

#10;
reset = 0;
#10;
reset = 1;
#10;
input1 = 1;
#50;
input1 = 2;
#200;
$finish;
end

// This is where the design under test is instantiated

non_blocking nonblocking_instance ( clock, input1, reset, output1, output2, output3);

endmodule
```

Kết quả mô phỏng dưới hình thức giản đồ sóng (waveform diagram)

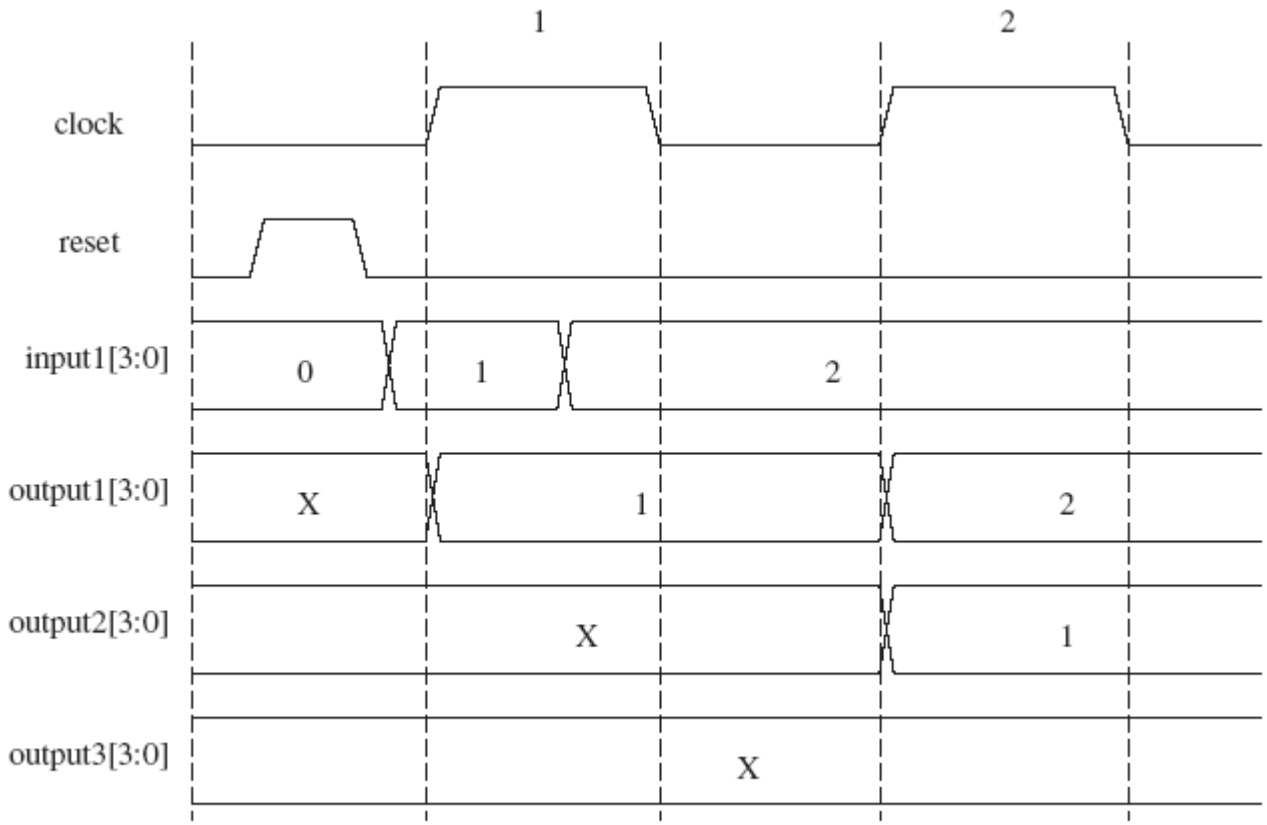


FIGURE 4.11. Diagram showing simulation results of Verilog code in Example 4.8.

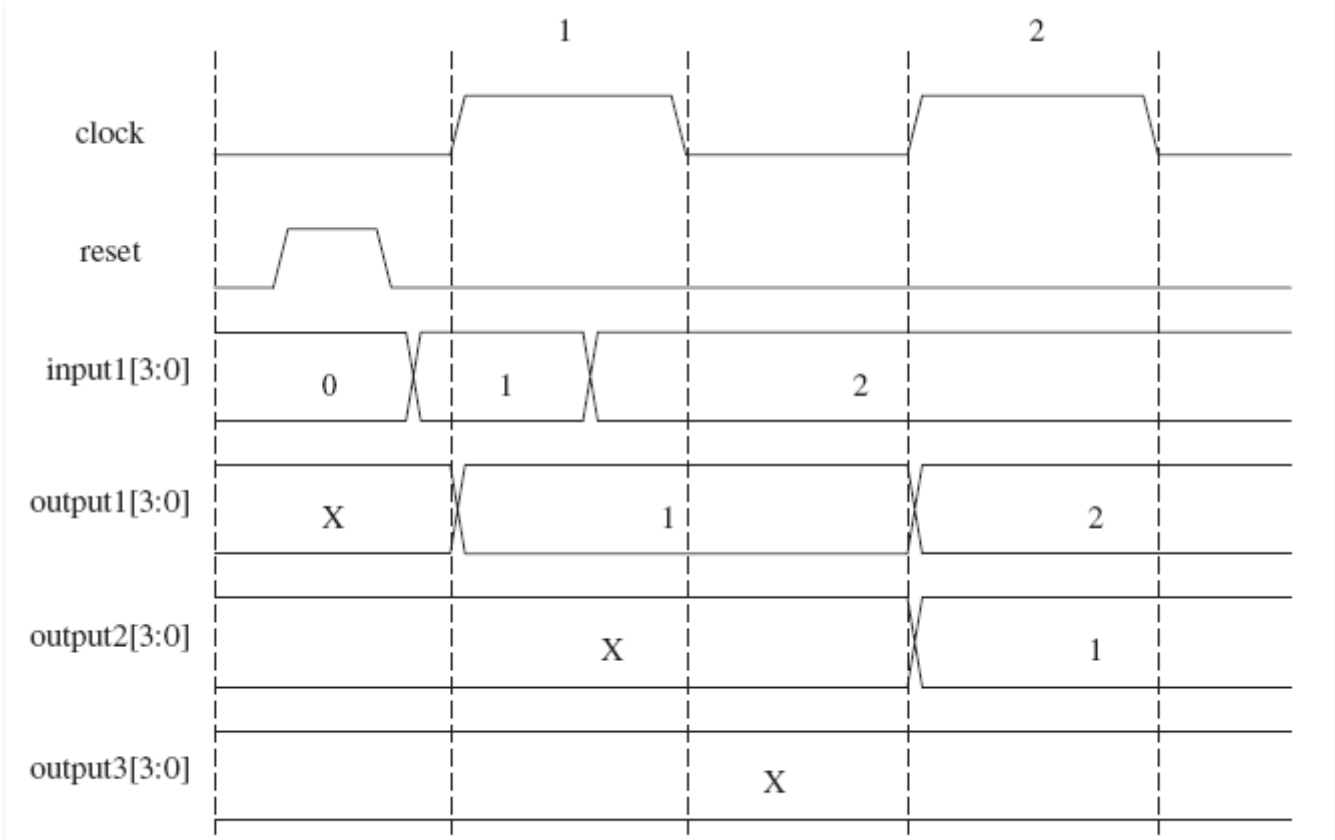


FIGURE 4.12. Diagram showing simulation results of Verilog code in Example 4.9.

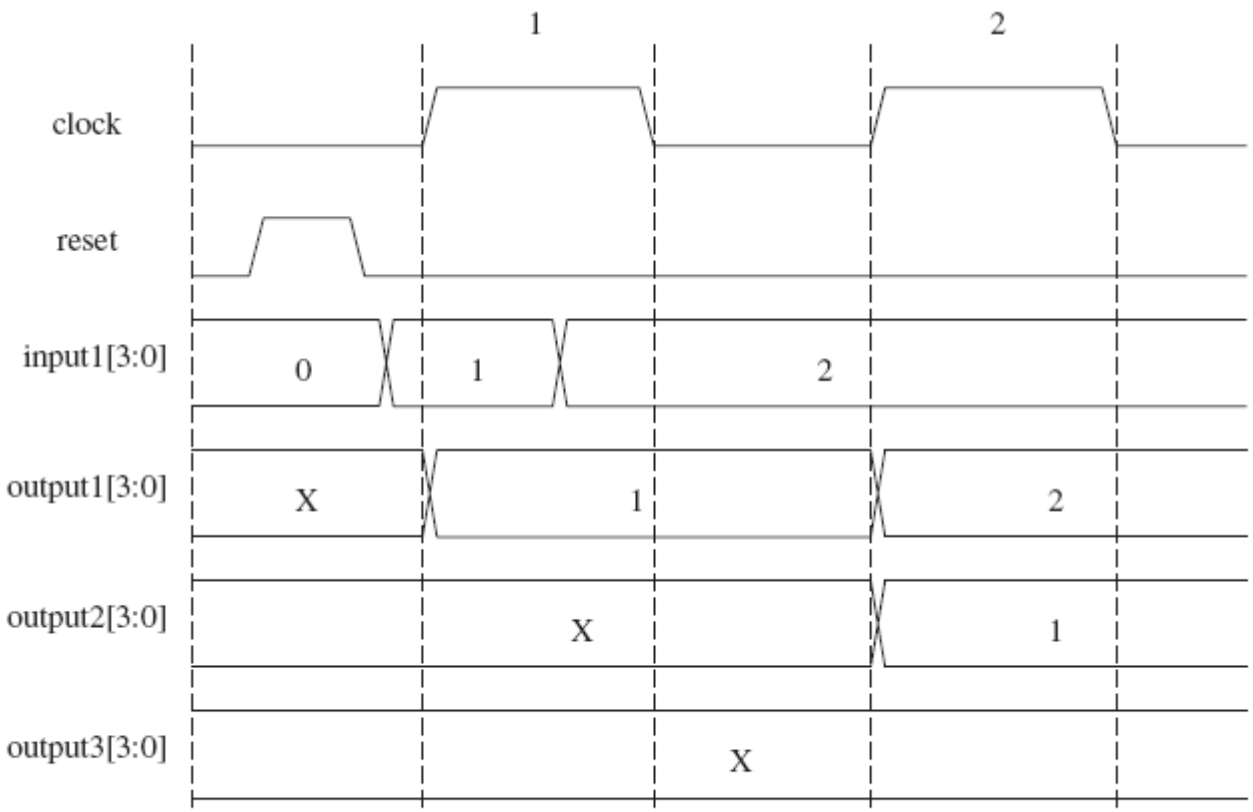


FIGURE 4.13. Diagram showing simulation results of Verilog code in Example 4.10.

[Ta nhận thấy rằng mặc dù thứ tự của những câu lệnh trong phép gán non blocking đã thay đổi nhưng kết quả mô phỏng với cùng một testbench có kết quả giống nhau. Điều này cho chúng ta thấy là phép gán non blocking không phụ thuộc vào thứ tự sắp xếp của các câu lệnh]

Những ví dụ tiếp theo từ 4.12 đến 4.13 là cùng một dạng mã Verilog như những ví dụ trước đó nhưng câu lệnh blocking được dùng thay vì nonblocking

Ví dụ 4.12

```
module non_blocking ( clock, input1, reset, output1, output2, output3);
input reset, clock;
input [3:0] input1;
output [3:0] output1, output2, output3;

always @ ( posedge clock)
begin

if (reset)
begin

output1 = 4'b0000;
output2 = 4'b0000;
output3 = 4'b0000;
end

else

begin

output1 = input1;
output2 = output1;
output3 = output2;
end

end
```

```
endmodule
```

Ví dụ 4.13 : Thay đổi thứ tự

```
module non_blocking ( clock, input1, reset, output1, output2, output3);
input reset, clock;
input [3:0] input1;
output [3:0] output1, output2, output3;

always @ ( posedge clock)
begin

if (reset)
begin

output1 = 4'b0000;
output2 = 4'b0000;
output3 = 4'b0000;
end

else

begin

output1 = input1;
output3 = output2; // This line is rearranged
output2 = output1;
```

```
end

end

endmodule
```

Ví dụ 4.14

```
module non_blocking ( clock, input1, reset, output1, output2, output3);
input reset, clock;
input [3:0] input1;
output [3:0] output1, output2, output3;

always @ ( posedge clock)
begin

if (reset)
begin

output1 = 4'b0000;
output2 = 4'b0000;
output3 = 4'b0000;
end

else

begin
```

```
output2 = output1;
output3 = output2;
output1 = input1;

end

end

endmodule
```

Ta sẽ dùng cùng một testbench như trong ví dụ trước và quan sát các kết quả

```
module nonblocking_tb ();

reg [3:0] input1;
reg clock, reset;
wire [3:0] output1, output2, output3;

// Generating initial stimulus for inputs and global clock

initial

begin

clock = 0;
input1 = 0;

forever #50 clock = ~clock; // for clock
end
```

```
// Test vectors

initial
begin

#10;
reset = 0;
#10;
reset = 1;
#10;
input1 = 1;
#50;
input1 = 2;
#200;
$finish;
end

// This is where the design under test is instantiated

non_blocking nonblocking_instance ( clock, input1, reset, output1, output2, output3);

endmodule
```

Kết quả mô phỏng sẽ rất khác so với những trường hợp dùng câu lệnh nonblocking

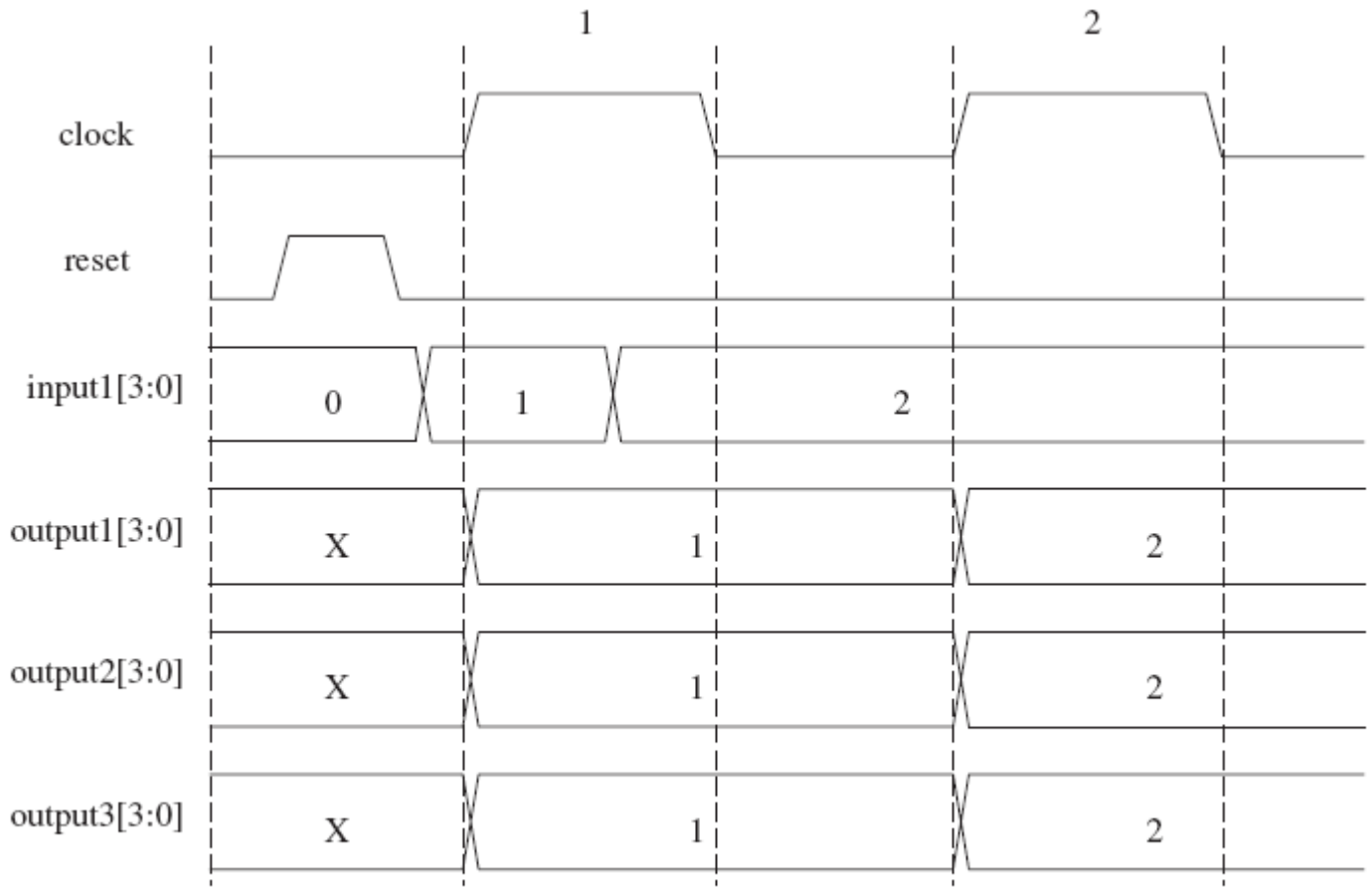


FIGURE 4.14. Diagram showing simulation results of Verilog code in Example 4.12.

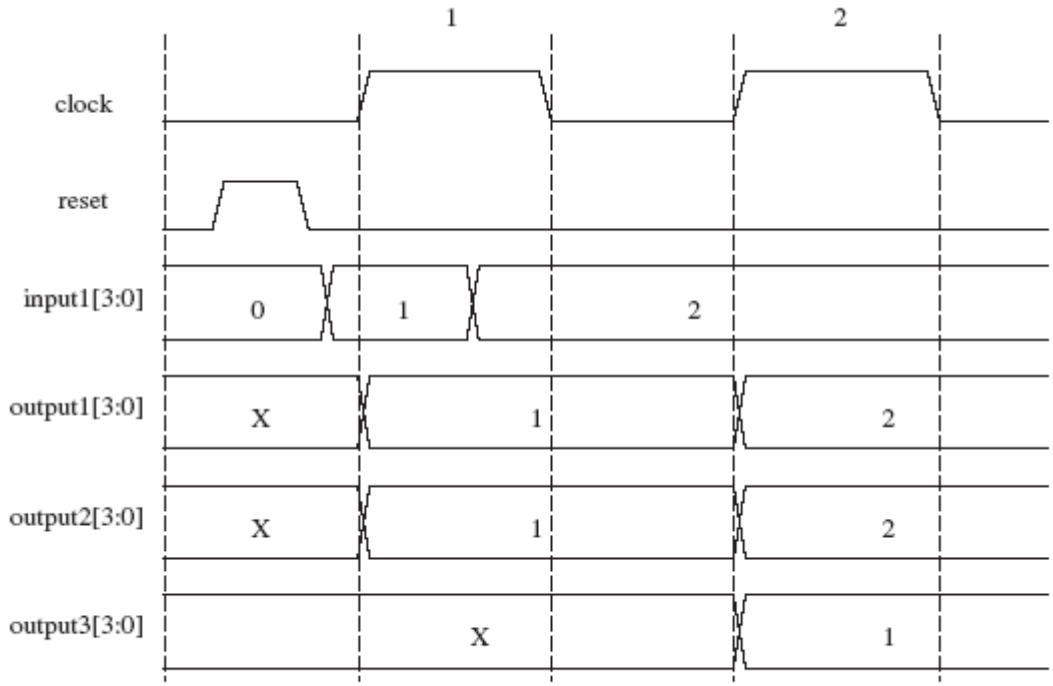


FIGURE 4.15. Diagram showing simulation results of Verilog code in Example 4.13.

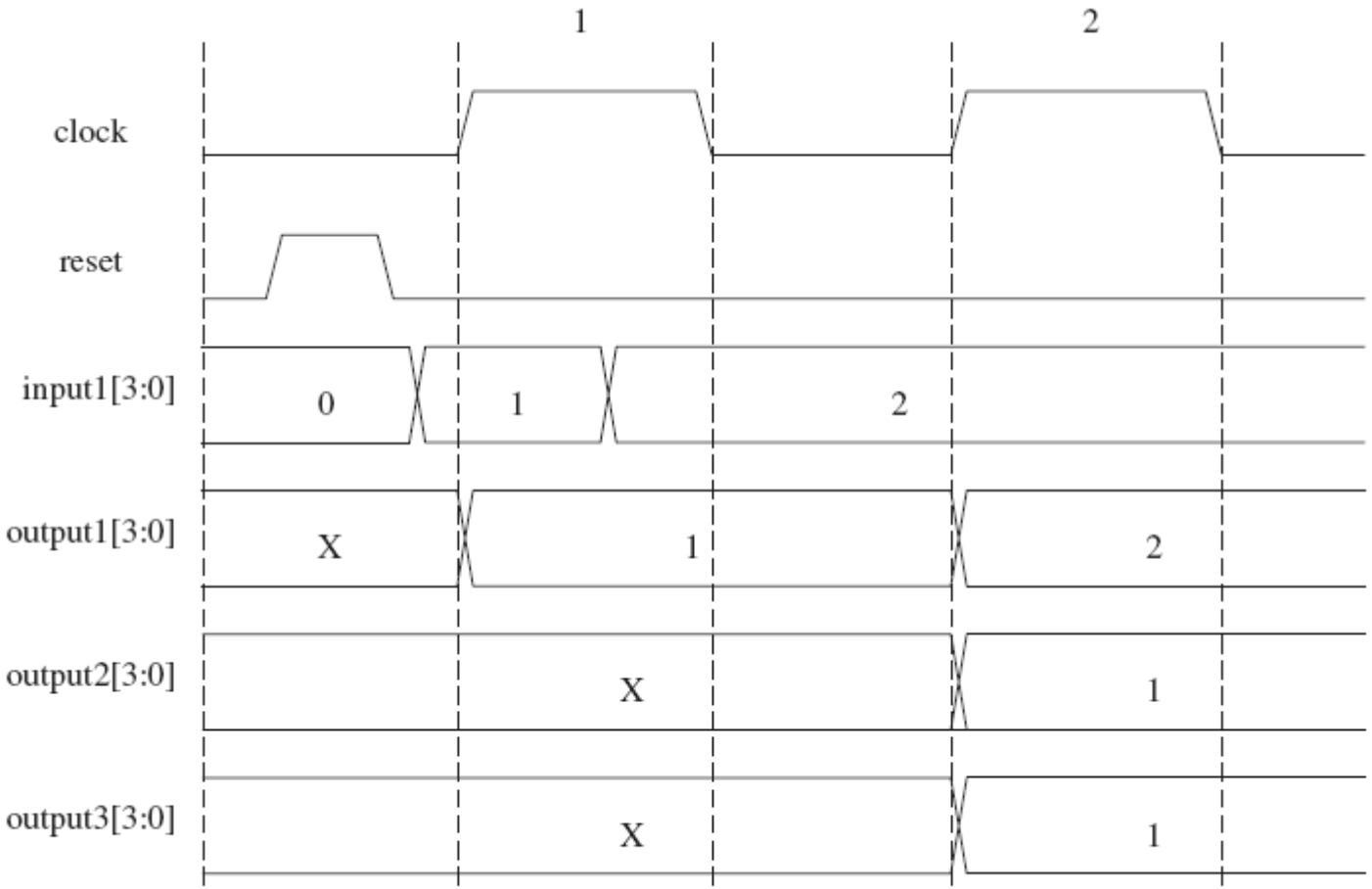


FIGURE 4.16. Diagram showing simulation results of Verilog code in Example 4.14.

Như vậy ta có thể nhận ra một điều là : khi mô phỏng thanh ghi thì trong khối always ta sẽ dùng những câu lệnh gán non blocking trong khi ta sẽ dùng câu lệnh blocking cho trường hợp mô tả mạch tổ hợp.

4.7 : Danh sách nhạy (Sentivity list)

Verilog dùng danh sách nhạy (sentivity list) để quyết định xem một khối lệnh tuần tự nào sẽ được thực hiện trong một chu kì mô phỏng. Đối với Verilog danh sách nhạy đòi hỏi một câu lệnh **always**.

Ví dụ : 4.16 trình bày mã Verilog cho một module thiết kế có một khối **always**. Khối này sẽ được tính toán bởi bộ mô phỏng mỗi khi có một sự thay đổi nào đó của những tín hiệu tương ứng trong danh sách nhạy.

```
module senselist ( X, Y, Z, AB);
input X, Y, Z;
output AB;

always @ ( X, Y, Z)
begin

// design source code

end
endmodule
```

Dựa vào ví dụ trên, ta thấy danh sách nhạy có 3 tín hiệu là X, Y, Z. Khối lệnh tuần tự trong **always** sẽ được tính toán bởi bộ mô phỏng khi có một sự thay đổi của một tín hiệu nào đó trong danh sách nhạy.

Một danh sách tín hiệu không đầy đủ trong danh sách nhạy cho một khối **always** nào đó sẽ gây ra sai trong quá trình mô phỏng, nó cũng có thể tạo ra sự không thống nhất kết quả tổng hợp và mô phỏng. Do đó điều quan trọng là luôn phải đưa tất cả những tín hiệu có liên quan vào trong danh sách nhạy cùa khối **always**.

Sau đây ta sẽ chỉ ra hai ví dụ của việc dùng danh sách nhạy. Trường hợp đầu là danh sách nhạy đầy đủ còn lại là trường hợp không đầy đủ.

Trường hợp đầy đủ :

```
module sense ( inputA, inputB, inputC, outputA);
input inputA, inputB, inputC;
output outputC;
reg outputA;

always @ ( inputA, inputB, inputC)
begin
if(inputA & inputB & inputC)
outputA = 0;
else
outputA = 1;
end

endmodule
```

Trường hợp không đầy đủ :

```
module sense ( inputA, inputB, inputC, outputA);
input inputA, inputB, inputC;
output outputC;
reg outputA;

always @ ( inputA or inputB)
begin
if(inputA & inputB & inputC)
outputA = 0;
```

```
else
outputA = 1;
end

endmodule
```

Testbench dùng để mô phỏng cả hai trường trên :

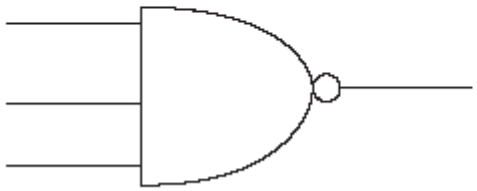
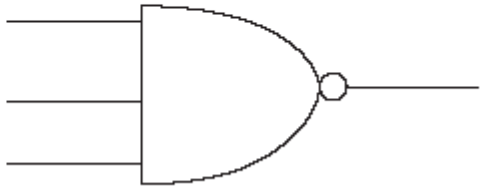
```
module sense_tb();
reg inputA, inputB, inputC;
wire outputA;
integer i;
initial
begin

for ( i = 0; i < 8; i = i +1)
begin
{inputA, inputB, inputC} = i; // Concatenation bus
#100;
end
end

// This is where you instantiated design under test

sense sense_instance ( .inputA(inputA), .inputB(inputB), .inputC(inputC), .outputA(outputA)); // Using explicit port map

initial

begin
```

```
$monitor ( " inputA %b, inputB %b, inputC %b, outputA %b", inputA, inputB, inputC,outputA);
end
endmodule
```

Kết quả mô phỏng và tổng hợp của hai thiết kế này khác nhau

TABLE 4.1. (Continued)

Simulation Results (Complete Sensitivity List)				Simulation Results (Incomplete Sensitivity List)			
inputA	inputB	inputC	outputA	inputA	inputB	inputC	outputA
0	0	0	1	0	0	0	1
0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1
0	1	1	1	0	1	1	1
1	0	0	1	1	0	0	1
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	0	1	1	1	1

Synthesized Logic (Complete Sensitivity List)	Synthesized Logic (Incomplete Sensitivity List)
	

4.8 Toán tử Verilog

Verilog cho phép sử dụng một số lớn những toán tử, là thành phần cơ bản nhất khi viết mã Verilog. Toán tử cho phép người thiết kế đạt được những chức năng khác nhau trong mô tả HDL.

Tất cả những toán tử của Verilog đều có thể tổng hợp được và được phân vào những nhóm khác nhau với mỗi nhóm có những chức năng nhất định.

4.8.1 : Toán tử điều kiện (Conditional Operators)

Toán tử điều kiện thường được dùng để mô tả những mạch tổ hợp và hoạt động giống như những chuyển mạch. Một toán tử điều kiện bao gồm ba toán hạng (operand)

- Biểu thức đầu vào.
- Tín hiệu điều khiển chọn lựa sẽ quyết định tín hiệu ngõ vào nào được truyền tới ngõ ra.
- Biểu thức ngõ ra

Cú pháp của toán tử điều kiện như sau :

assign output_signal = control_signal ? input1 : input2;

trong đó **output_signal** là ngõ ra của câu lệnh điều kiện; **control_signal** là tín hiệu chọn ngõ vào **input1** hày **input2** được truyền tới ngõ ra.

Dưới đây là module dùng để minh hoạ việc sử dụng toán tử điều kiện trong viết mã Verilog.

TABLE 4.2. Truth table showing functionality for module "conditional"

InputA	InputB	ControlC	OutputA
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

```
module conditional ( inputA, inputB, controlC, outputA);
input inputA, inputB, inputC;
output outputA;
wire outputA;
assign outputA = controlC ? inputA : inputB;

endmodule
```

Khi ví dụ trên được tổng hợp sẽ cho ra mạch như hình 4.7 :Đó thực sự là một mạch dồn kênh (Multiplexer). Do đó khi ta cần thiết kế một mạch dồn kênh thì nên sử dụng toán tử điều kiện.

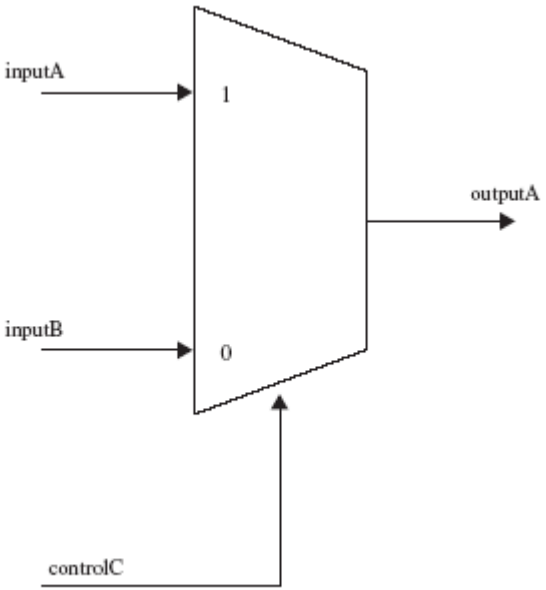


FIGURE 4.17. Diagram showing synthesized logic for module "*conditional*."

4.8.2 : Toán tử ghép bus

Nhiều tín hiệu có thể được ghép tạo thành bus, điều này được thực hiện trong Verilog bằng toán tử ghép (concatenation operator). Cú pháp sử dụng toán tử này như sau :

assign signal_bus = {signal1, signal2, signal3};

trong đó **signal_bus** là tên của bus được ghép có ba bit còn **signal1**, **signal2**, **signal3** là những tín hiệu được ghép với nhau.

Ví dụ 4.18 : Ghép bus 3 và 4 bit

```
module concatenation ( inputA, inputB, inputC, inputD, outputA, outputB);
input inputA, inputB, inputC, inputD;
output [2:0] outputA;
output [3:0] outputB;
wire [2:0] outputA;
wire [2:0] outputB;

assign outputA = {inputA, inputB, inputC};

assign outputB = {inputA, inputB, inputC, inputD};

endmodule
```

4.8.3 : Toán tử dịch

Toán tử dịch có thể được thực hiện trong Verilog dùng hai loại toán tử là dịch trái (shift left) và dịch phải (shift right). Ví dụ 4.19 sẽ trình bày mã Verilog dùng toán tử dịch trái để dịch trái ba bit tín hiệu bus **tempA** bởi một bit sang trái.

Ví dụ 4.19 : Minh họa toán tử dịch trái

```
module shift_left ( inputA, inputB, outputA);

input [2:0] inputA, inputB;
output [2:0] outputA;
wire [2:0] outputA;

wire [2:0] tempA;

assign tempA = inputA & inputB;
```

```
assign outputA = tempA << 1; // This is the shift left operator

endmodule
```

Sau khi tổng hợp ta sẽ được mạch như hình vẽ 4.18

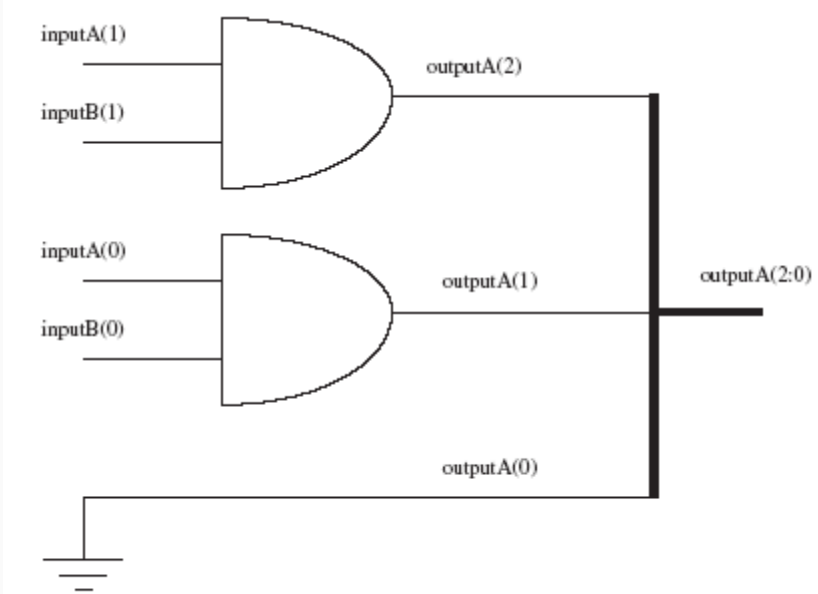


FIGURE 4.18. Diagram showing synthesized logic for module "*shift_left*."

Tiếp theo ta sẽ viết testbench nhằm mô phỏng toán tử dịch trái ở trên

```
module shift_left_tb();
reg [2:0] inputA_reg, inputB_reg;
wire [2:0] outputA_wire;
```

```
integer i,j;

initial
begin

for ( i = 0; i < 8; i = i + 1)
begin
// To force stimulus to inputA
input_regA = i;
// To force stimulus to inputB
for ( j = 0; j < 8; j = j +1 )
begin
inputB_reg = j;
#10;
end
end
end

// This is where you instantiated design under test

shift_left DUT ( .inputA ( inputA_reg), .inputB( inputB_reg), .outputA(outputA_wire));

initial

begin

$monitor ( " inputA %b%b%b, inputB %b%b%b, tempA %b%b%b, outputA %b%b%b", inputA_reg[2], inputA_reg[1],
inputA_reg[0],inputB_reg[2],inputB_reg[1],inputB_reg[0],DUT.tempA[2],DUT.tempA[1],DUT.tempA[0],outputA_wire[2],out
putA_wire[1],outputA_wire[0]);
```

```
end

endmodule
```

Kết quả mô phỏng như sau :

Example 4.21 Simulation Results of Test Bench Module "*shift_left_tb*"

```
inputA 000 inputB 000 tempA 000 outputA 000
inputA 000 inputB 001 tempA 000 outputA 000
inputA 000 inputB 010 tempA 000 outputA 000
inputA 000 inputB 011 tempA 000 outputA 000
inputA 000 inputB 100 tempA 000 outputA 000
inputA 000 inputB 101 tempA 000 outputA 000
inputA 000 inputB 110 tempA 000 outputA 000
inputA 000 inputB 111 tempA 000 outputA 000
inputA 001 inputB 000 tempA 000 outputA 000
inputA 001 inputB 001 tempA 001 outputA 010
inputA 001 inputB 010 tempA 000 outputA 000
inputA 001 inputB 011 tempA 001 outputA 010
inputA 001 inputB 100 tempA 000 outputA 000
inputA 001 inputB 101 tempA 001 outputA 010
inputA 001 inputB 110 tempA 000 outputA 000
inputA 001 inputB 111 tempA 001 outputA 010
inputA 010 inputB 000 tempA 000 outputA 000
inputA 010 inputB 001 tempA 000 outputA 000
inputA 010 inputB 010 tempA 010 outputA 100
inputA 010 inputB 011 tempA 010 outputA 100
inputA 010 inputB 100 tempA 000 outputA 000
inputA 010 inputB 101 tempA 000 outputA 000
inputA 010 inputB 110 tempA 010 outputA 100
inputA 010 inputB 111 tempA 010 outputA 100
```

```
inputA 011 inputB 000 tempA 000 outputA 000
inputA 011 inputB 001 tempA 001 outputA 010
inputA 011 inputB 010 tempA 010 outputA 100
inputA 011 inputB 011 tempA 011 outputA 110
inputA 011 inputB 100 tempA 000 outputA 000
inputA 011 inputB 101 tempA 001 outputA 010
inputA 011 inputB 110 tempA 010 outputA 100
inputA 011 inputB 111 tempA 011 outputA 110
inputA 100 inputB 000 tempA 000 outputA 000
inputA 100 inputB 001 tempA 000 outputA 000
inputA 100 inputB 010 tempA 000 outputA 000
inputA 100 inputB 011 tempA 000 outputA 000
inputA 100 inputB 100 tempA 100 outputA 000
inputA 100 inputB 101 tempA 100 outputA 000
inputA 100 inputB 110 tempA 100 outputA 000
inputA 100 inputB 111 tempA 100 outputA 000
inputA 101 inputB 000 tempA 000 outputA 000
inputA 101 inputB 001 tempA 001 outputA 010
inputA 101 inputB 010 tempA 000 outputA 000
inputA 101 inputB 011 tempA 001 outputA 010
inputA 101 inputB 100 tempA 100 outputA 000
inputA 101 inputB 101 tempA 101 outputA 010
inputA 101 inputB 110 tempA 100 outputA 000
inputA 101 inputB 111 tempA 101 outputA 010
inputA 110 inputB 000 tempA 000 outputA 000
inputA 110 inputB 001 tempA 000 outputA 000
inputA 110 inputB 010 tempA 010 outputA 100
inputA 110 inputB 011 tempA 010 outputA 100
inputA 110 inputB 100 tempA 100 outputA 000
inputA 110 inputB 101 tempA 100 outputA 000
inputA 110 inputB 110 tempA 110 outputA 100
inputA 110 inputB 111 tempA 110 outputA 100
inputA 111 inputB 000 tempA 000 outputA 000
inputA 111 inputB 001 tempA 001 outputA 010
inputA 111 inputB 010 tempA 010 outputA 100
inputA 111 inputB 011 tempA 011 outputA 110
inputA 111 inputB 100 tempA 100 outputA 000
inputA 111 inputB 101 tempA 101 outputA 010
inputA 111 inputB 110 tempA 110 outputA 100
inputA 111 inputB 111 tempA 111 outputA 110
```

Tiếp theo sẽ là module minh họa việc sử dụng toán tử

dịch phải (shift right)

```
module shift_right ( inputA, inputB, outputA);

input [2:0] inputA, inputB;
output [2:0] outputA;
wire [2:0] outputA;

wire [2:0] tempA;

assign tempA = inputA & inputB;
assign outputA = tempA >> 1; // This is the shift left operator

endmodule
```

Kết quả tổng hợp ra mạch logic

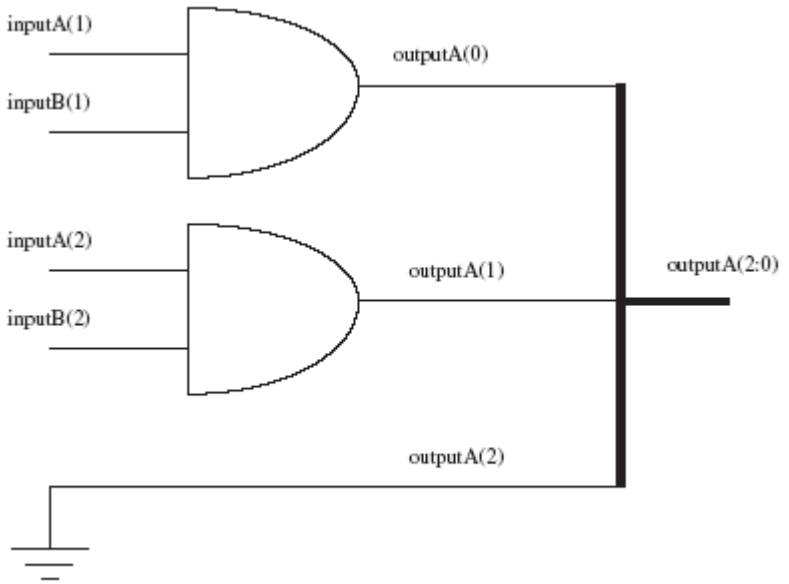


FIGURE 4.19. Diagram showing synthesized logic for module "*shift_right*."

Testbench dùng để mô phỏng toán tử trên

```
module shift_right_tb();
reg [2:0] inputA_reg, inputB_reg;
wire [2:0] outputA_wire;

integer i,j;

initial
begin

for ( i = 0; i < 8; i = i + 1)
```

```
begin
// To force stimulus to inputA
input_regA = i;
// To force stimulus to inputB
for ( j = 0; j < 8; j = j +1 )
begin
inputB_reg = j;
#10;
end
end
end

// This is where you instantiated design under test

shift_right DUT ( .inputA ( inputA_reg), .inputB( inputB_reg), .outputA(outputA_wire));

initial

begin

$monitor ( " inputA %b%b%b, inputB %b%b%b, tempA %b%b%b, outputA %b%b%b", inputA_reg[2], inputA_reg[1],
inputA_reg[0],inputB_reg[2],inputB_reg[1],inputB_reg[0],DUT.tempA[2],DUT.tempA[1],DUT.tempA[0],outputA_wire[2],out
putA_wire[1],outputA_wire[0]);
end

endmodule
```

Kết quả mô phỏng

Example 4.24 Simulation Results of Verilog Test Bench Module "*shift_right_tb*"

```
inputA 000 inputB 000 tempA 000 outputA 000
inputA 000 inputB 001 tempA 000 outputA 000
inputA 000 inputB 010 tempA 000 outputA 000
inputA 000 inputB 011 tempA 000 outputA 000
inputA 000 inputB 100 tempA 000 outputA 000
inputA 000 inputB 101 tempA 000 outputA 000
inputA 000 inputB 110 tempA 000 outputA 000
inputA 000 inputB 111 tempA 000 outputA 000
inputA 001 inputB 000 tempA 000 outputA 000
inputA 001 inputB 001 tempA 001 outputA 000
inputA 001 inputB 010 tempA 000 outputA 000
inputA 001 inputB 011 tempA 001 outputA 000
inputA 001 inputB 100 tempA 000 outputA 000
inputA 001 inputB 101 tempA 001 outputA 000
inputA 001 inputB 110 tempA 000 outputA 000
inputA 001 inputB 111 tempA 001 outputA 000
inputA 010 inputB 000 tempA 000 outputA 000
inputA 010 inputB 001 tempA 000 outputA 000
inputA 010 inputB 010 tempA 010 outputA 001
inputA 010 inputB 011 tempA 010 outputA 001
inputA 010 inputB 100 tempA 000 outputA 000
inputA 010 inputB 101 tempA 000 outputA 000
inputA 010 inputB 110 tempA 010 outputA 001
inputA 010 inputB 111 tempA 010 outputA 001
inputA 011 inputB 000 tempA 000 outputA 000
inputA 011 inputB 001 tempA 001 outputA 000
inputA 011 inputB 010 tempA 010 outputA 001
inputA 011 inputB 011 tempA 011 outputA 001
inputA 011 inputB 100 tempA 000 outputA 000
inputA 011 inputB 101 tempA 001 outputA 000
inputA 011 inputB 110 tempA 010 outputA 001
inputA 011 inputB 111 tempA 011 outputA 001
inputA 100 inputB 000 tempA 000 outputA 000
inputA 100 inputB 001 tempA 000 outputA 000
inputA 100 inputB 010 tempA 000 outputA 000
inputA 100 inputB 011 tempA 000 outputA 000
inputA 100 inputB 100 tempA 100 outputA 010
inputA 100 inputB 101 tempA 100 outputA 010
inputA 100 inputB 110 tempA 100 outputA 010
```

```
inputA 100 inputB 111 tempA 100 outputA 010
inputA 101 inputB 000 tempA 000 outputA 000
inputA 101 inputB 001 tempA 001 outputA 000
inputA 101 inputB 010 tempA 000 outputA 000
inputA 101 inputB 011 tempA 001 outputA 000
inputA 101 inputB 100 tempA 100 outputA 010
inputA 101 inputB 101 tempA 101 outputA 010
inputA 101 inputB 110 tempA 100 outputA 010
inputA 101 inputB 111 tempA 101 outputA 010
inputA 110 inputB 000 tempA 000 outputA 000
inputA 110 inputB 001 tempA 000 outputA 000
inputA 110 inputB 010 tempA 010 outputA 001
inputA 110 inputB 011 tempA 010 outputA 001
inputA 110 inputB 100 tempA 100 outputA 010
inputA 110 inputB 101 tempA 100 outputA 010
inputA 110 inputB 110 tempA 110 outputA 011
inputA 110 inputB 111 tempA 110 outputA 011
inputA 111 inputB 000 tempA 000 outputA 000
inputA 111 inputB 001 tempA 001 outputA 000
inputA 111 inputB 010 tempA 010 outputA 001
inputA 111 inputB 011 tempA 011 outputA 001
inputA 111 inputB 100 tempA 100 outputA 010
inputA 111 inputB 101 tempA 101 outputA 010
inputA 111 inputB 110 tempA 110 outputA 011
inputA 111 inputB 111 tempA 111 outputA 011
```

4.8.4 : Toán tử toán học

Verilog cho phép 5 toán tử toán học khác nhau, bao gồm :

Toán tử cộng (Addition Operator)

Toán tử trừ (Subtraction Operator)
Toán tử nhân (Multiplication Operator)
Toán tử chia (Division Operator)
Toán tử lấy dư (Modulus Operator)

Khi dùng những toán tử này người thiết kế cần cẩn thận về mạch logic được tạo ra trong quá trình tổng hợp có thể khác nếu những ràng buộc về thiết kế được dùng.

4.8.4.1 : Toán tử cộng Như tên chỉ ra, toán tử này cho phép thực hiện phép toán cộng và được mã trong Verilog dùng kí hiệu "+".

```
module addition ( inputA, inputB, outputA);
input inputA, inputB;
output [1:0] outputA;
wire outputA;
assign outputA = inputA + inputB;
endmodule
```

Hình vẽ 4.20 chỉ ra sơ đồ được tổng hợp của module trên:

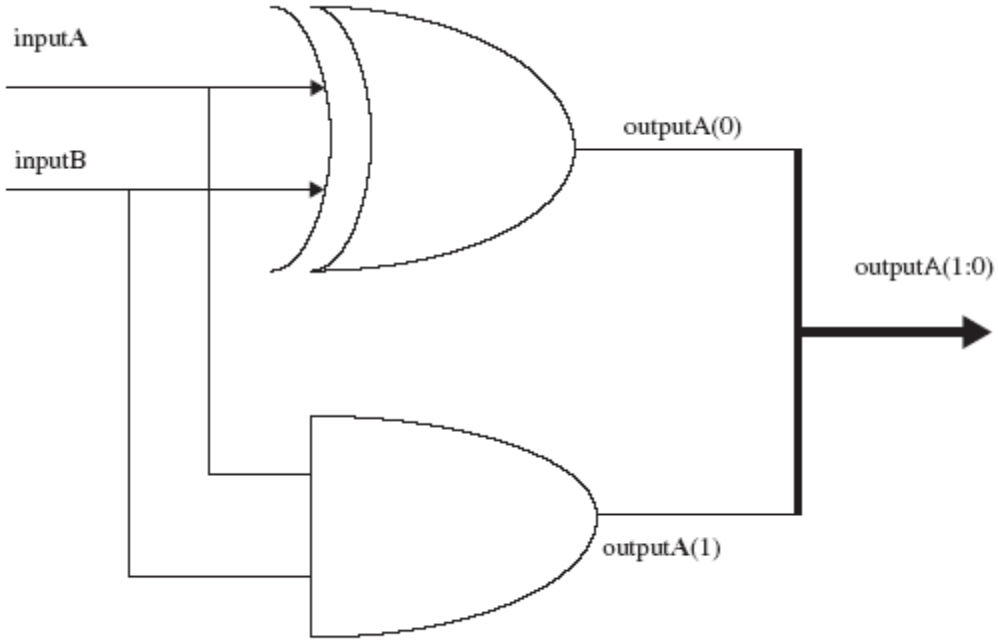


Figure 4.20. Diagram showing synthesized logic for module "addition."

Ví dụ 4.26 là mã Testbench dùng để mô phỏng hoạt động của mạch cộng vừa thiết kế ở trên :

```
module addition_tb();
reg inputA_reg, inputB_reg;
wire [1:0] outputA_wire;
integer i,j;

initial
begin
```

```
for ( i = 0; i < 2; i = i +1)
begin
inputA_reg = i;
for ( j =0; j <2; j = j+1)
begin
inputB_reg = i;
#10; // delay for 10 time units
end
end
end

// This is where you instantiated your design under test

addition DUT (.inputA ( inputA_reg), .inputB(inputB_reg), .outputA(outputA_wire));

initial

begin
$monitor ( "inputA %b inputB %b outputA %b%b", inputA_reg, inputB_reg, outputA_wire[1],outputA_wire[0]);
end

endmodule
```

Kết quả mô phỏng

Example 4.27 Simulation Results for Verilog Test Bench Module "*addition_tb*"

```
inputA 0 inputB 0 outputA 00
inputA 0 inputB 1 outputA 01
inputA 1 inputB 0 outputA 01
inputA 1 inputB 1 outputA 10
```

4.8.4.2 : Toán tử trừ Như tên chỉ ra, toán tử này cho phép thực hiện phép toán trừ và được mã trong Verilog dùng kí hiệu "-".

```
module subtraction ( inputA, inputB, outputA);
input inputA, inputB;
output [1:0] outputA;
wire outputA;
assign outputA = inputA - inputB;
endmodule
```

Hình vẽ 4.29 chỉ ra sơ đồ được tổng hợp của module trên:

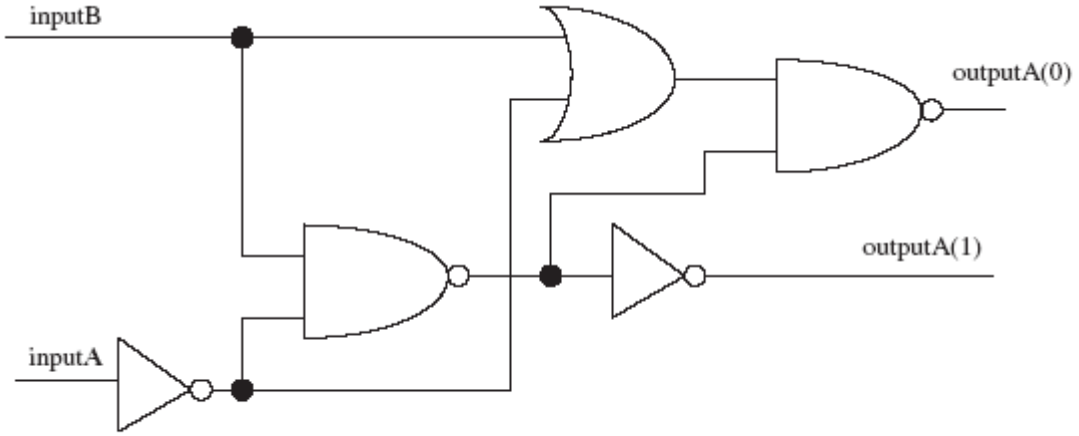


Figure 4.21. Diagram showing synthesized logic for module "subtraction".

Ví dụ 4.26 là mã Testbench dùng để mô phỏng hoạt động của mạch trừ vừa thiết kế ở trên :

```
module subtraction_tb();
reg inputA_reg, inputB_reg;
wire [1:0] outputA_wire;
integer i,j;

initial
begin
for ( i = 0; i < 2; i = i +1)
begin
inputA_reg = i;
for ( j =0; j <2; j = j+1)
begin
inputB_reg = i;
```

```
#10; // delay for 10 time units
end
end
end

// This is where you instantiated your design under test

subtraction DUT (.inputA ( inputA_reg), .inputB(inputB_reg), .outputA(outputA_wire));

initial

begin
$monitor ( "inputA %b inputB %b outputA %b%b", inputA_reg, inputB_reg, outputA_wire[1],outputA_wire[0]);
end

endmodule
```

Kết quả mô phỏng

Example 4.30 Simulation Results for Verilog Test Bench Module "*subtraction_tb*"

```
inputA 0 inputB 0 outputA 00
inputA 0 inputB 1 outputA 11
inputA 1 inputB 0 outputA 01
inputA 1 inputB 1 outputA 00
```

When "***inputA*** = 0" and "***inputB*** = 1", "***inputA*** – ***inputB*** = -1". Two's complement of -1 is "11".

4.8.4.2 : Toán tử nhân

Như tên chỉ ra, toán tử này cho phép thực hiện phép toán nhân và được mã trong Verilog dùng kí hiệu "*".

```
module multiplication ( inputA, inputB, outputA);
input [1:0] inputA, inputB;
output [3:0] outputA;
wire outputA;
assign outputA = inputA * inputB;
endmodule
```

Hình vẽ 4.22 chỉ ra sơ đồ được tổng hợp của module trên:

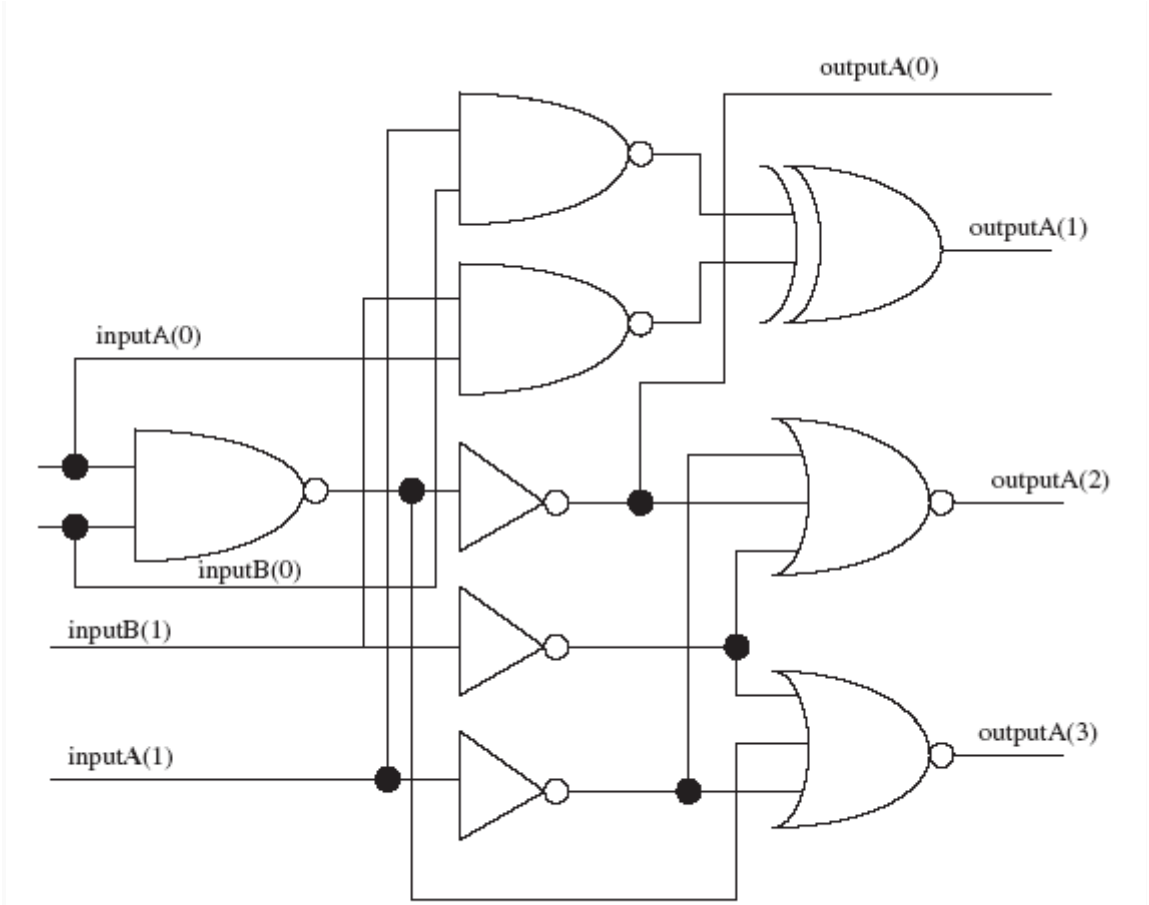


Figure 4.22. Diagram showing synthesized logic for module "multiplication".

Ví dụ 4.32 là mã Testbench dùng để mô phỏng hoạt động của mạch trừ vừa thiết kế ở trên :

```
module multiplication_tb();
reg [1:0] inputA_reg, inputB_reg;
```

```
wire [3:0] outputA_wire;
integer i,j;

initial
begin
for ( i = 0; i < 4; i = i +1)
begin
inputA_reg = i;
for ( j =0; j <4; j = j+1)
begin
inputB_reg = i;
#10; // delay for 10 time units
end
end
end

// This is where you instantiated your design under test

multiplication DUT (.inputA ( inputA_reg), .inputB(inputB_reg), .outputA(outputA_wire));

initial

begin
$monitor ( "inputA %h inputB %h outputA %h", inputA_reg, inputB_reg, outputA_wire);
end

endmodule
```

Kết quả mô phỏng

Example 4.33 Simulation Results for Verilog Test Bench Module "*multiplication_tb*"

```
inputA 0 inputB 0 outputA 0
inputA 0 inputB 1 outputA 0
inputA 0 inputB 2 outputA 0
inputA 0 inputB 3 outputA 0
inputA 1 inputB 0 outputA 0
inputA 1 inputB 1 outputA 1
inputA 1 inputB 2 outputA 2
inputA 1 inputB 3 outputA 3
inputA 2 inputB 0 outputA 0
inputA 2 inputB 1 outputA 2
inputA 2 inputB 2 outputA 4
inputA 2 inputB 3 outputA 6
inputA 3 inputB 0 outputA 0
inputA 3 inputB 1 outputA 3
inputA 3 inputB 2 outputA 6
inputA 3 inputB 3 outputA 9
```

4.8.5 : Toán tử chia

Như tên đã chỉ ra toán tử chia cho phép ta thực hiện phép toán chia đượ mã hóa trong Verilog dùng kí hiệu "/". Phải cẩn thận khi dùng toán tử chia trong mã Verilog tổng hợp được. Toán tử chia chỉ có thể dùng với **hằng số** chứ *không phải trên biến*, nếu toán tử chia được thực hiện trên một giá trị không phải là hằng số thì công cụ sẽ không thể tổng hợp được.

Ví dụ : 4.34 trình bày về toán tử chia trong Verilog

```
module division ( inputA, inputB, outputA, outputB);

input [3:0] inputA;
input [3:0] inputB;
output [3:0] outputA;
output [3:0] outputB;

reg [3:0] outputA, outputB;

always @ ( inputA or inputB)
begin
if
(inputA == 4'b1010)
outputA = 3/3;
else
outputA = 0;
if
(inputB == 4'b0011)
output = 8/5;

else outputB = 0;
end
```

```
endmodule
```

Hình vẽ 4.23 là mạch tổng hợp được của toán tử chia ở trên

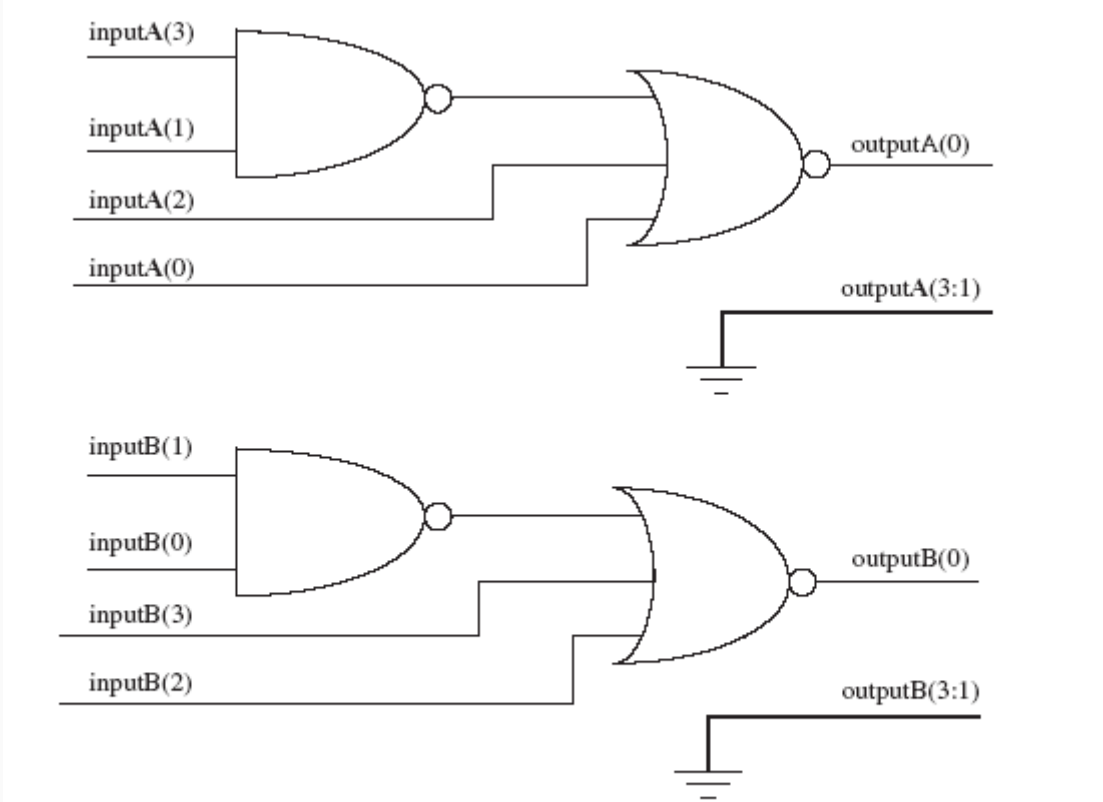


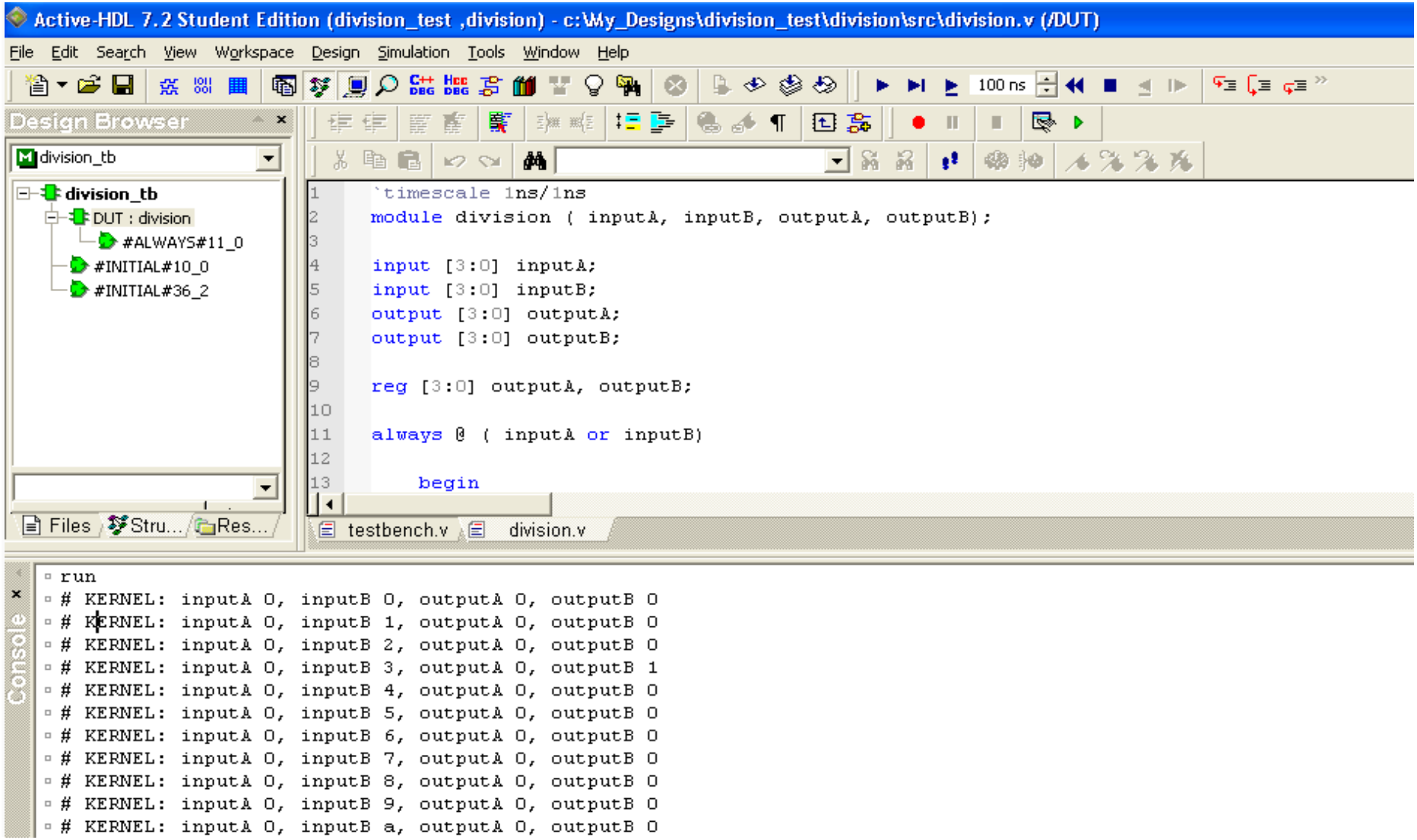
Figure 4.23. Diagram showing synthesized logic for design module "***division***."

Ví dụ 4.35 là testbench dùng để mô phỏng mã Verilog của toán tử chia.

```
module division_tb();
```

```
reg [3:0] inputA_reg, inputB_reg;

wire [3:0] outputA_wire, outputB_wire;

integer i,j;

initial

begin

for ( i = 0; i <16; i = i +1)
begin
inputA_reg = i;

for ( j =0; j <16; j = j +1)
begin
inputB_reg = j;
#10;
end
end
end
division DUT (.inputA(inputA_reg), .inputB(inputB_reg), .outputA(outputA_wire), .outputB(outputB_wire));

initial


begin


$monitor ("inputA %h, inputB %h, outputA %h, outputB %h", inputA_reg, inputB_reg, outputA_wire, outputB_wire);
end
endmodule
```

Kết quả mô phỏng bằng ActiveHDL 7.2 Student Edition



4.8.5 : Toán tử lấy phần dư (modulus)

Như tên đã chỉ ra toán tử lấy phần dư cho phép ta thực hiện phép toán lất dư được mã hóa trong Verilog dùng kí hiệu "%". Phải cẩn thận khi dùng toán tử lấy dư trong mã Verilog tổng hợp được. Toán tử lấy dư chỉ có thể dùng với **hằng số** chứ *không phải trên biến*, nếu toán tử lấy dư được thực hiện trên một giá trị không phải là hằng số thì công cụ sẽ không thể tổng hợp được.

Ví dụ : 4.34 trình bày về toán tử chia trong Verilog

```
module modulus ( inputA, inputB, outputA, outputB);

input [3:0] inputA;
input [3:0] inputB;
output [3:0] outputA;
output [3:0] outputB;

reg [3:0] outputA, outputB;

always @ ( inputA or inputB)
begin
if
(inputA == 4'b1010)
outputA = 8%3;
else
outputA = 0;
if
(inputB == 4'b0011)
output = 20%7;

else outputB = 0;
end
```

```
endmodule
```

Hình vẽ 4.23 là mạch tổng hợp được của toán tử chia ở trên

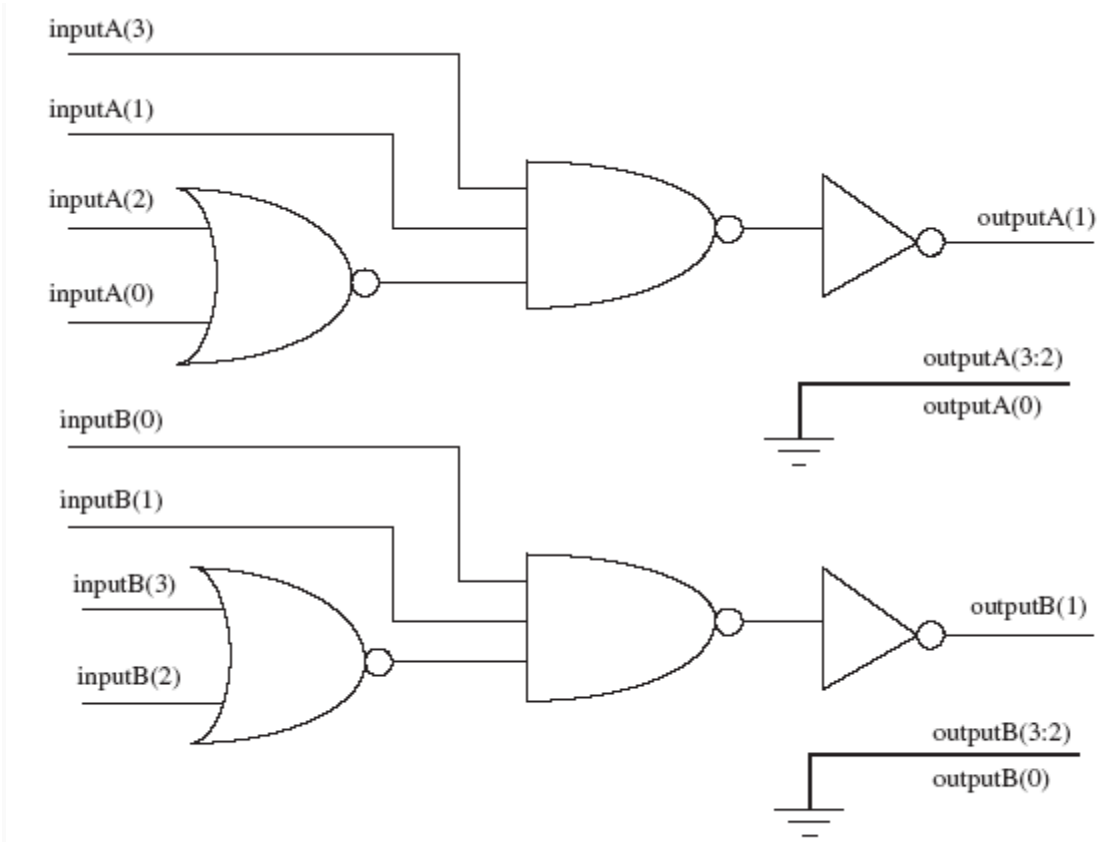


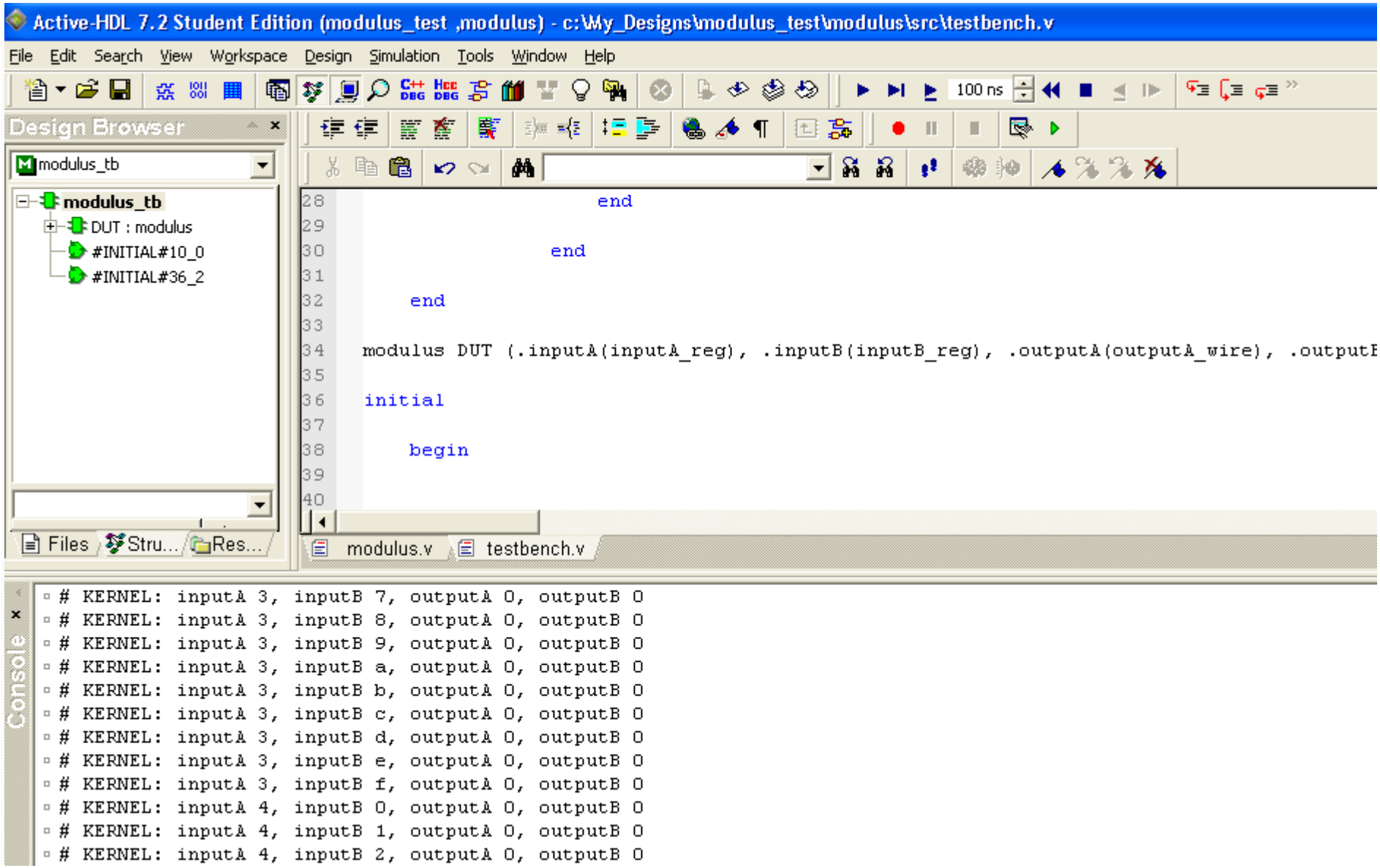
Figure 4.24. Diagram showing synthesized logic for design module "***modulus***."

Ví dụ 4.35 là testbench dùng để mô phỏng mã Verilog của toán tử chia.

```
module modulus_tb();
```

```
reg [3:0] inputA_reg, inputB_reg;

wire [3:0] outputA_wire, outputB_wire;

integer i,j;

initial

begin

for ( i = 0; i <16; i = i +1)
begin
inputA_reg = i;

for ( j =0; j <16; j = j +1)
begin
inputB_reg = j;
#10;
end
end
end
modulus DUT (.inputA(inputA_reg), .inputB(inputB_reg), .outputA(outputA_wire), .outputB(outputB_wire));

initial


begin


$monitor ("inputA %h, inputB %h, outputA %h, outputB %h", inputA_reg, inputB_reg, outputA_wire, outputB_wire);
end
endmodule
```

Kết quả mô phỏng bằng ActiveHDL 7.2 Student Edition


Active-HDL 7.2 Student Edition (modulus_test ,modulus) - c:\My_Designs\modulus_test\modulus\src\testbench.v
File Edit Search View Workspace Design Simulation Tools Window Help
100 ns
Design Browser
modulus_tb
modulus_tb
DUT : modulus
#INITIAL#10_0
#INITIAL#36_2
Files Stru... Res...
28 end
29
30 end
31
32 end
33
34 modulus DUT (.inputA(inputA_reg), .inputB(inputB_reg), .outputA(outputA_wire), .outputB
35
36 initial
37
38 begin
39
40
modulus.v testbench.v
Console
KERNEL: inputA 3, inputB 7, outputA 0, outputB 0
KERNEL: inputA 3, inputB 8, outputA 0, outputB 0
KERNEL: inputA 3, inputB 9, outputA 0, outputB 0
KERNEL: inputA 3, inputB a, outputA 0, outputB 0
KERNEL: inputA 3, inputB b, outputA 0, outputB 0
KERNEL: inputA 3, inputB c, outputA 0, outputB 0
KERNEL: inputA 3, inputB d, outputA 0, outputB 0
KERNEL: inputA 3, inputB e, outputA 0, outputB 0
KERNEL: inputA 3, inputB f, outputA 0, outputB 0
KERNEL: inputA 4, inputB 0, outputA 0, outputB 0
KERNEL: inputA 4, inputB 1, outputA 0, outputB 0
KERNEL: inputA 4, inputB 2, outputA 0, outputB 0

4.8.7 : Toán tử logic

Toán tử logic tác động lên một nhóm toán hạng và kết quả trả về là giá trị nhị phân 0 hay 1. Toán hạng có thể đơn bit hay đa bit nhưng kết quả luôn là đơn bit. Có ba loại toán tử logic khác nhau được dùng trong Verilog.

- && : Đây là toán tử logic AND thực hiện chức năng hàm AND và trả về giá trị dơn bit.
- || : Toán tử OR thực hiện phép toán OR và cũng trả về một giá trị đơn bit.
- ! : Thực hiện toán tử logic NOT, nó thực hiện phép đảo một hàm nào đó và trã về giá trị đơn bit.

Ví dụ : 4.40 là mã Verilog minh họa việc dùng toán tử logic

```
module example (inputA, inputB, inputC, inputD, outputA, outputB, outputC, outputD);

input inputA, inputB, inputC;
input [2:0] inputD;
output outputA, outputB, outputC;
output [2:0] outputD;

// for logical AND

assign outputA = inputA && inputB;

// for logical OR

assign outputB = inputA || inputB;

// for logical NOT

assign outputC = !inputC;

// for vector format
```

```
assign outputD = {inputA, inputB, inputC} && inputD;

endmodule
```

Mạch tổng hợp được

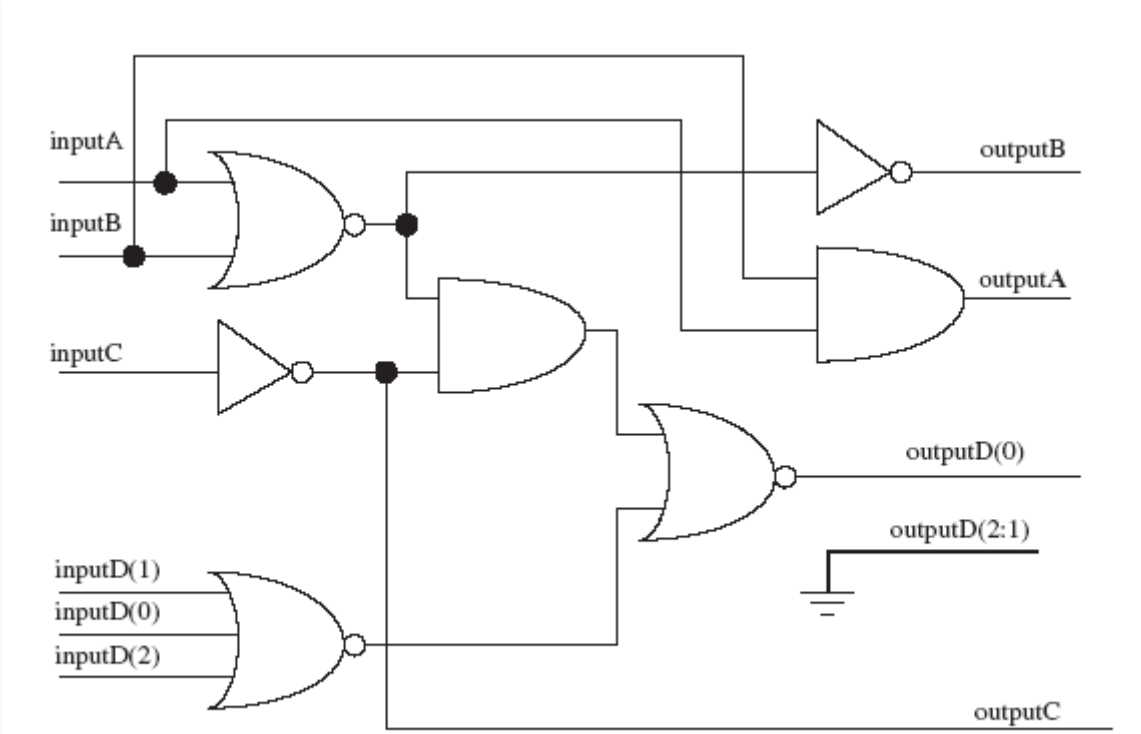


FIGURE 4.25. Diagram showing synthesized logic for verilog code module "logical."

Testbench mô phỏng mạch thiết kế ở trên

```
module logical_tb();
```

```
reg inputA_reg, inputB_reg, inputC_reg;
reg [2:0] inputD_reg;
wire outputA_wire, outputB_wire, outputC_wire;
wire outputD_wire;

integer i,j;

initial

begin
for (i=0;i<8;i=i+1)
begin
{inputA_reg, inputB_reg, inputC_reg} =i;
for(j=0;j<8;j=j+1)
begin
inputD_reg =j;
#10;
end
end
end
// Instantiating your design here

example DUT (.inputA(inputA_reg), .inputB(inputB_reg), .inputC(inputC_reg), .inputD(inputD_reg), .outputA(outputA_wire),
.outputB(outputB_wire), .outputC(outputC_wire),.outputD(outputD_wire));

initial

begin

$monitor ("inputA %b, inputB %b, inputC %b, inputD %h, outputA %b, outputB %b, outputC %b, outputD %h",inputA_reg,
inputB_reg, inputC_reg, inputD_reg, outputA_wire, outputB_wire, outputC_wire, outputD_wire);
```

```
End

endmodule
```

Kết quả mô phỏng bằng Active HDL student Edition

4.8.8 : Toán tử Bitwise

Toán tử bitwise giống như toán tử logic ngoại trừ toán tử này tác động lên bus và trả về giá trị bus. Ví dụ, nếu một toán tử bitwise được dùng trên hai toán hạng 3 bit kết quả của toán hạng cũng sẽ có 3 bit. Có 4 kiểu toán hạng bitwise :

& Đây là toán tử bitwise AND. Nó thực hiện hàm AND và trả về một giá trị tương ứng với động rộng của toán hạng.

| Đây là toán tử bitwise OR. Nó thực hiện hàm OR và trả về một giá trị tương ứng với động rộng của toán hạng.

~ Đây là toán tử bitwise NOT. Nó thực hiện hàm NOT và trả về một giá trị tương ứng với động rộng của toán hạng.

^ Đây là toán tử bitwise XOR. Nó thực hiện hàm XOR và trả về một giá trị tương ứng với động rộng của toán hạng.

Ví dụ 4.43 chỉ ra mã Verilog dùng cho toán tử bitwise. Giản đồ hình 4.26 trình bày mạch logic tổng hợp được từ mã Verilog.

```
module bitwise ( inputA, inputB, inputC, inputD, outputA, outputB, outputC, outputD, outputE);

input inputA, inputB, inputC ;

input [2:0] inputD;
```

```
output outputA, outputB, outputC, outputE;

output [2:0] outputD;

wire outputA, ouputB, outputC, outputE;

wire [2:0] ouputD;

// for bitwise AND
assign outputA = inputA & inputB;

// for bitwise OR
assign outputB = inputA | inputB;

// for bitwise NOT

assign outputC = ~inputC;

// for bitwise XOR

assign outputE = inputA^inputB;

// for vector format

assign ouputD = {inputA, inputB, inputC} & inputD;

endmodule
```

Download bitwise.v

http://www.box.net/shared/00frcb60zv

Mạch tổng hợp được của mã trên

Verilog Testbench cho ví dụ về toán tử bitwise

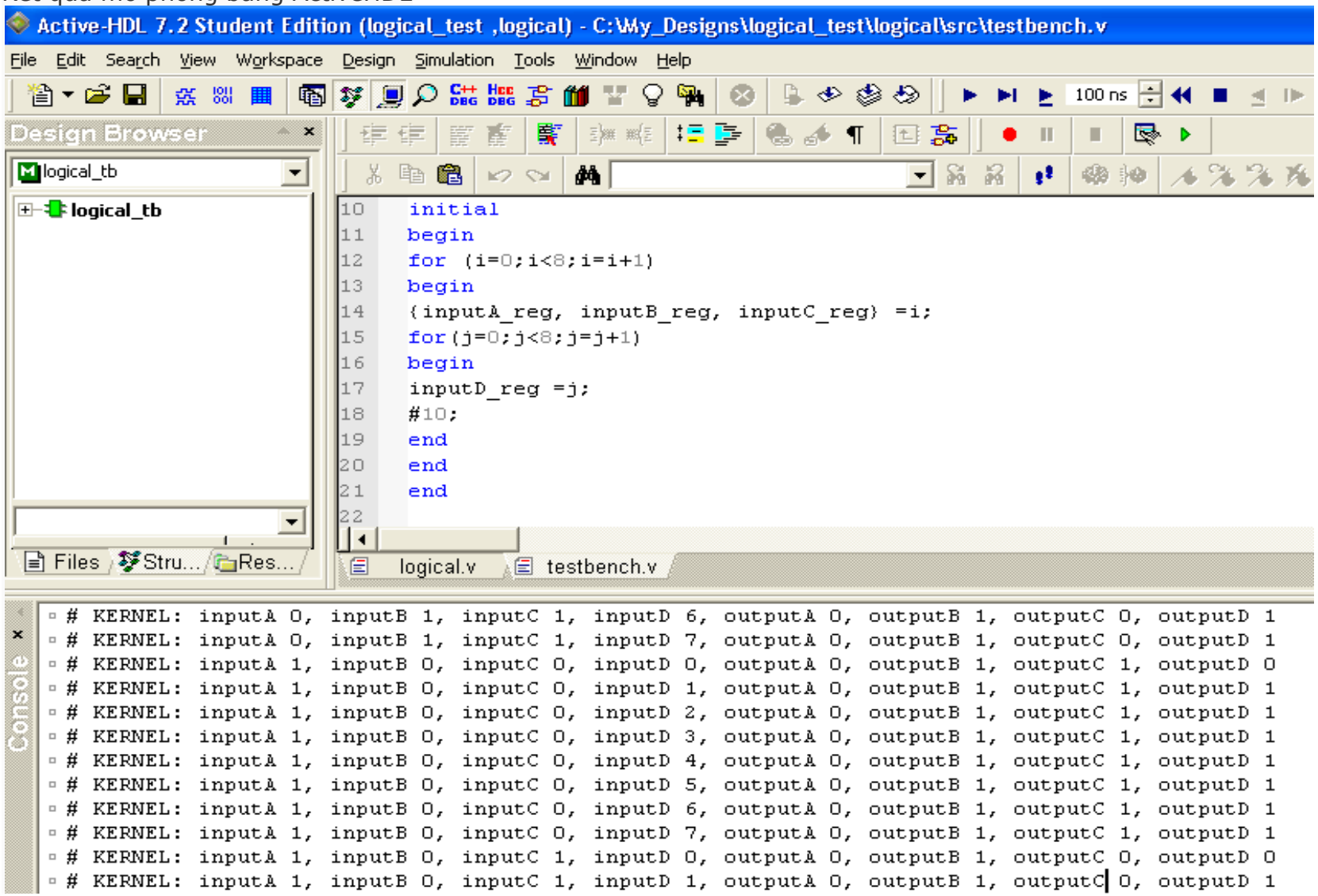
```
module bitwise_tb ();

reg inputA_reg, inputB_reg, inputC_reg;
reg [2:0] inputD;

wire outputA_wire, outputB_wire, outputC_wire, outputD_wire;
wire [2:0] outputD_wire;

integer i,j;

initial
begin
for ( i=0; i <8; i = i+1)
begin
{inputA_reg, inputB_reg, inputC_reg} = i;
for ( j = 0; j <8; j = j +1)
begin
inputD_reg = j;
#10;
end
```

```
end
end
// This is where you instantiate your design under test
bitwise DUT (.inputA(inputA_reg), .inputB(inputB_reg), .inputC(inputC_reg),.inputD(inputD_reg), .outputA(outputA_wire),
.ouputB(outputB_wire), .outputC(outputC_wire), .outputD(outputD_wire),.outputE(outputE_wire));

initial
begin

$monitor ("inputA %b, inputB %b, inputC %b, inputD %h, outputA %b, outputB %b, outputC %b, outputD %h, outputE
%b", inputA_reg, inputB_reg, inputC_reg, inputD_reg, outputA_wire, outputB_wire, outputC_wire, outputD_wire,
outputE_wire);

end

endmodule
```

Download bitwise_tb.v

http://www.box.net/shared/hbm45yk9bz

Kết quả mô phỏng bằng ActiveHDL



4.8.9 : Toán tử bằng (Equality operator)

Toán tử bằng được dùng trong việc so sánh các toán hạng. Có hai kiểu toán tử bằng trong mã Verilog:

1. Toán tử bằng logic (logical equality) được biểu diễn bằng dấu **"=="** cho phép so sánh bằng và **"!="** cho phép toán không bằng. Những ký hiệu này được dùng thường xuyên trong mã Verilog. Toán tử bằng logic có thể tạo ra các giá trị logic 0, 1 hay không xác định (X) (unknown). Phép so sánh cho kết quả là X khi bất kì toán hạng nào trong phép so sánh có giá trị không xác định hay đang ở trạng thái tổng trở cao. (High Z). Nếu toán hạng có giá trị bốn bit "1001" và toán hạng B có giá trị "1010" thì kết quả của phép so sánh "A==B" sẽ cho kết quả logic 0. Mặt khác, nếu phép so sánh là **"A!=B"** thì kết quả là 1.

2. Toán tử bằng trong câu lệnh case được biểu diễn bằng ký hiệu **"==="** cho phép so sánh bằng và **"!=="** cho phép so sánh không bằng. Toán tử so sánh bằng trong câu lệnh case luôn tạo ra kết quả là 0 hay 1 và không thể tạo ra kết quả không xác định (unknown). Toán tử này so sánh cả giá trị X và Z. Nếu toán hạng A có giá trị 4 bit là **"1xz0"** và toán hạng B có giá trị **"1xz0"** thì phép so sánh **"A===B"** sẽ cho kết quả 1 mặt khác phép toán **"A!==B"** sẽ cho kết quả là 0 (false). Người thiết kế cần chú ý là toán tử so sánh bằng trong câu lệnh case không tổng hợp được bởi ta không thể tạo ra mạch logic có thể nhận dạng được trạng thái không xác định hay tổng trở cao.

Mã verilog cho minh họa việc sử dụng toán tử so sánh bằng

```
module logicalequal (inputA, inputB, inputC, inputD, outputA, outputB);

input inputA, inputB, inputC, inputD;
output outputA, outputB;
wire outputA, outputB;

assign outputA = (inputA ==inputB);
assign outputB = (inputC != inputD);

endmodule
```

Download logicalequal.v

http://www.box.net/shared/5xszrlt04y

Mạch tổng hợp được

Ví dụ bên dưới là mã Testbench dùng để mô phỏng hoạt động của module logicalequal ở trên

```
module logical_tb();

reg inputA_reg, inputB_reg, inputC_reg, inputD_reg;
wire outputA_wire, outputB_wire;

integer i;

initial
begin
for(i =0; i<16;i=i+1)
begin
{inputA_reg, inputB_reg, inputC_reg, inputD_reg} = i;
#10;
end
end

// This is where you instantiate your DUT

logicalequal DUT (.inputA(inputA_reg), .inputB(inputB_reg), .inputC(inputC_reg), inputD(inputD_reg),
.outputA(outputA_wire), .outputB(outputB_wire));

initial
begin
$monitor ( "inputA %b, inputB %b, inputC %b, inputD %b, outputA %b, outputB %b", inputA_reg, inputB_reg, inputC_reg,
```

inputD_reg, outputA_wire, outputB_wire);
end

Download logical_tb.v

http://www.box.net/shared/psen86gkvr

Kết quả mô phỏng bằng ActiveHDL

4.8.10 : Toán tử rút gọn (Reduction Operators)

Toán tử rút gọn có cùng chức năng giống như toán tử logic ngoại trừ việc toán tử này tác động lên bit của bản thân toán hạng. Kết quả có được từ toán tử là đơn bit. Những kiểu toán tử suy ra khác nhau trong Verilog bao gồm :

- & : Toán tử suy ra AND
- | : Toán tử suy ra OR
- ^ : Toán tử suy ra XOR
- ~& : Toán tử suy ra NAND
- ~| : Toán tử suy ra NOR
- ~^ : Toán tử suy ra XNOR

Hình dưới đây chỉ ra mạch được tổng hợp cho toán tử suy ra :

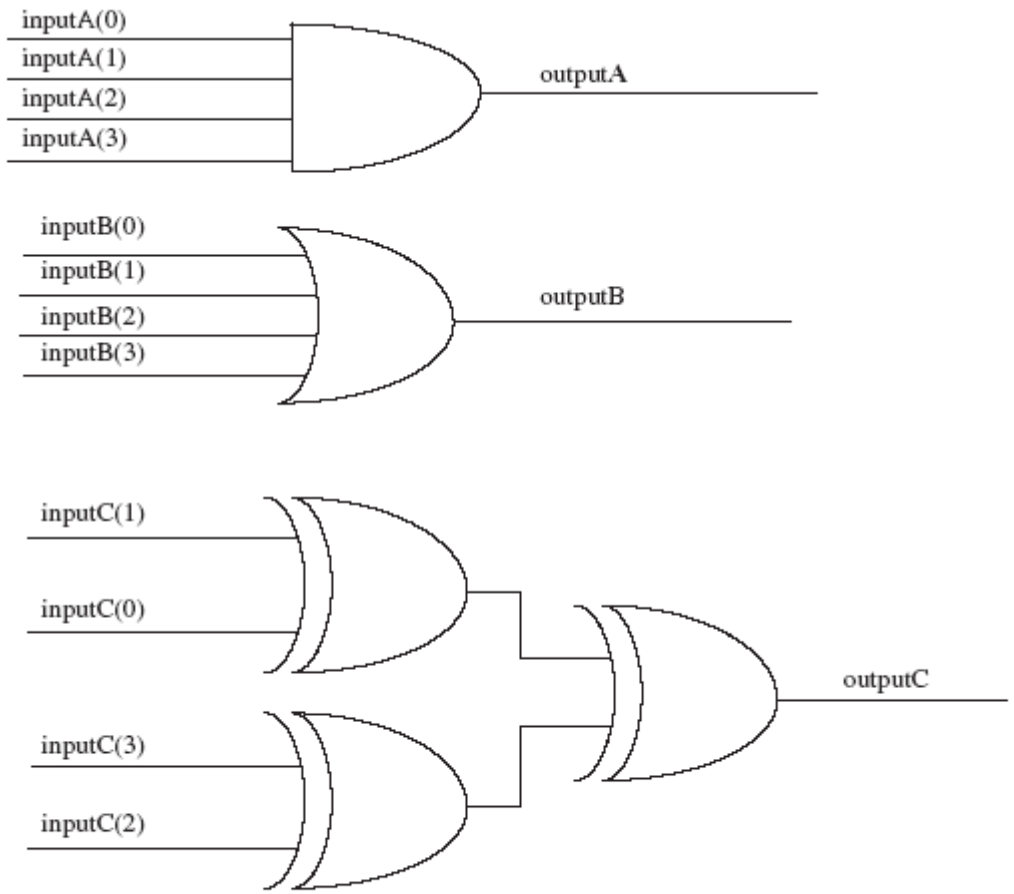


FIGURE 4.28. Diagram showing synthesized logic for verilog code module "*reduction.*"

Ví dụ 4.49 Mã Verilog dùng toán tử suy ra

```
module reduction ( inputA, inputB, inputC, outputA, outputB, outputC);
input [3:0] inputA, inputB, inputC;
```

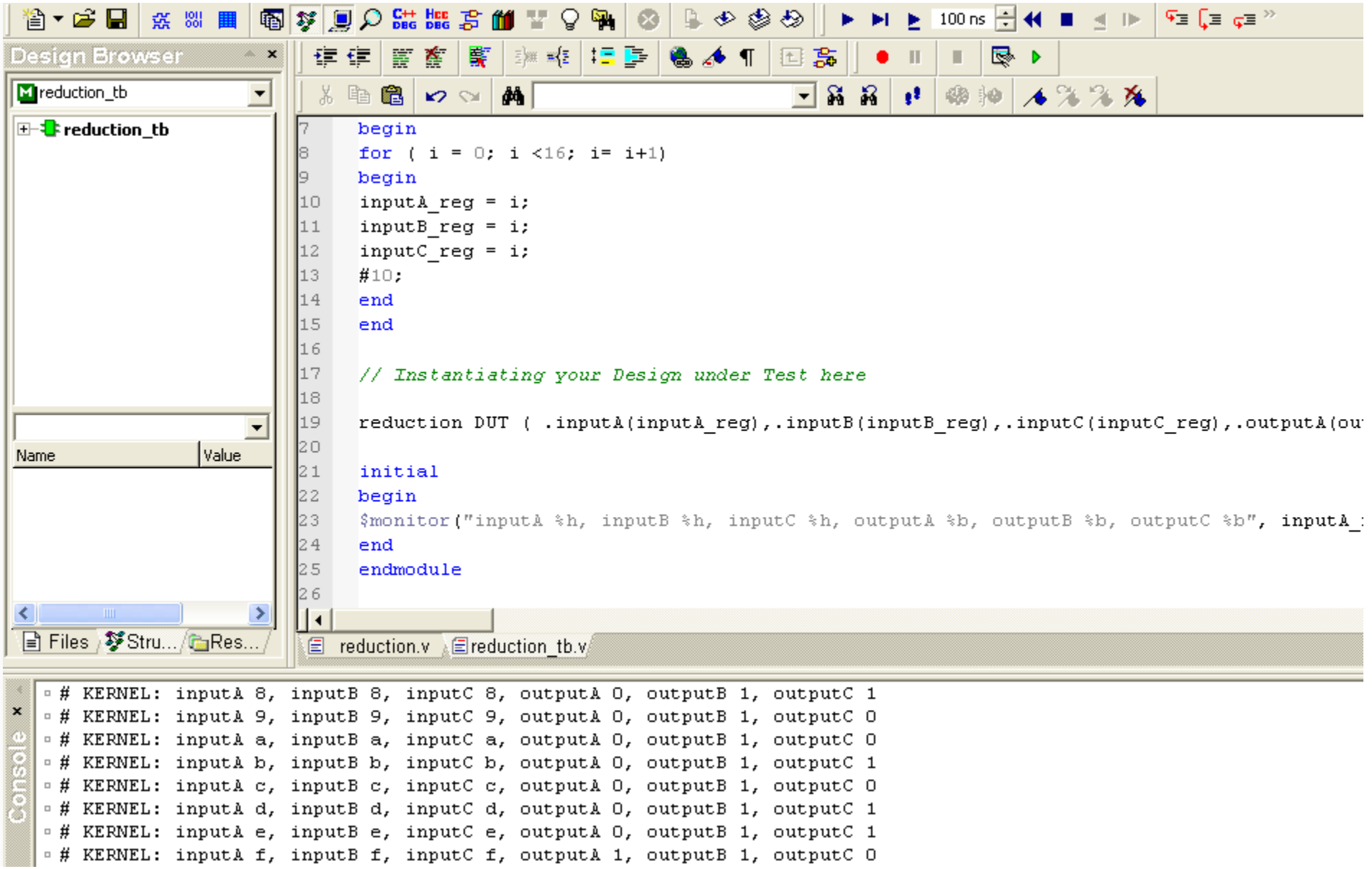
```
output outputA, outputB, outputC;
wire outputA, outputB, outputC;

// for reduction AND

assign outputA = &inputA;

//for reduction OR

assign outputB = |inputB;

//for reduction XOR

assign outputC = ^inputC;

endmodule
```

Ví dụ 4.50 chỉ ra Verilog testbench dùng để mô phỏng module reduction ở trên

```
module reduction_tb();
reg [3:0] inputA_reg, inputB_reg, inputC_reg;
wire outputA_wire, outputB_wire, outputC_wire;
integer i;

initial
begin
for ( i = 0; i <16; i= i+1)
begin
inputA_reg = i;
inputB_reg = i;
inputC_reg = i;
#10;
end
```

```
end

// Instantiating your Design under Test here

reduction DUT ( .inputA(inputA_reg),.inputB(inputB_reg),.inputC(in putC_reg),.outputA(outputA_wire),.outputB(outputB_
wire),.outputC(outputC_wire));

initial
begin
$monitor("inputA %h, inputB %h, inputC %h, outputA %b, outputB %b, outputC %b", inputA_reg, inputB_reg, inputC_reg,
outputA_wire, outputB_wire, outputC_wire);
end
endmodule
```

Dưới đây là kết quả mô phỏng bằng ActiveHDL


Design Browser
reduction_tb
reduction_tb
Name
Value
Files
Stru...
Res...
7 begin
8 for (i = 0; i <16; i= i+1)
9 begin
10 inputA_reg = i;
11 inputB_reg = i;
12 inputC_reg = i;
13 #10;
14 end
15 end
16
17 // Instantiating your Design under Test here
18
19 reduction DUT (.inputA(inputA_reg),.inputB(inputB_reg),.inputC(inputC_reg),.outputA(ou
20
21 initial
22 begin
23 $monitor("inputA %h, inputB %h, inputC %h, outputA %b, outputB %b, outputC %b", inputA_
24 end
25 endmodule
26
reduction.v
reduction_tb.v
Console
KERNEL: inputA 8, inputB 8, inputC 8, outputA 0, outputB 1, outputC 1
KERNEL: inputA 9, inputB 9, inputC 9, outputA 0, outputB 1, outputC 0
KERNEL: inputA a, inputB a, inputC a, outputA 0, outputB 1, outputC 0
KERNEL: inputA b, inputB b, inputC b, outputA 0, outputB 1, outputC 1
KERNEL: inputA c, inputB c, inputC c, outputA 0, outputB 1, outputC 0
KERNEL: inputA d, inputB d, inputC d, outputA 0, outputB 1, outputC 1
KERNEL: inputA e, inputB e, inputC e, outputA 0, outputB 1, outputC 1
KERNEL: inputA f, inputB f, inputC f, outputA 1, outputB 1, outputC 0

4.8.11 : Toán tử quan hệ (Relational Operators)

Toán tử quan hệ giống với toán tử bằng ngoại trừ trả về giá trị so sánh. Kết quả trả về từ toán tử này có giá trị đơn bit. Có 4 loại toán tử bằng khác nhau :

Lớn hơn (Greater than) Được biểu diễn bằng ký hiệu ">", trả về giá trị "1" nếu toán tử đem so sánh lớn hơn một toán tử khác.Nếu toán tử A có giá trị 5; điều kiện "A >3" sẽ cho kết quả là "1" do toán hạng A lớn hơn 3.

Nhỏ hơn (Less than) Được biểu diễn bằng ký hiệu "<" với chú giải giống như đối với toán tử lớn hơn tức là trả vế giá trị "1" nếu toán hạng cần so sánh có giá trị nhỏ hơn một chuẩn hay toán hạng khác.

Lớn hơn hay bằng (Greater than or equal) : Chú giải giống như trên và được kí hiệu là ">="

Nhỏ hơn hay bằng (Less than or equal) : Được ký hiệu là "<=".

Ví dụ 4.52 là mã Verilog sử dụng toán tử quan hệ

```
module relational (inputA, inputB, inputC, inputD, outputA, outputB, outputC, outputD, outputE, outputF, outputG, outputH);

input [1:0] inputA, inputB, inputC, inputD;

output outputA, outputB, outputC, outputD, outputE, outputF, outputG, outputH;

wire outputA, outputB, outputC, outputD, outputE, outputF, outputG, outputH;

assign outputA = (inputA >1);
assign outpuB = (inputB <2);
assign outputC = (inputC >= 1);
assign outputD = (inputD <= 2);
```

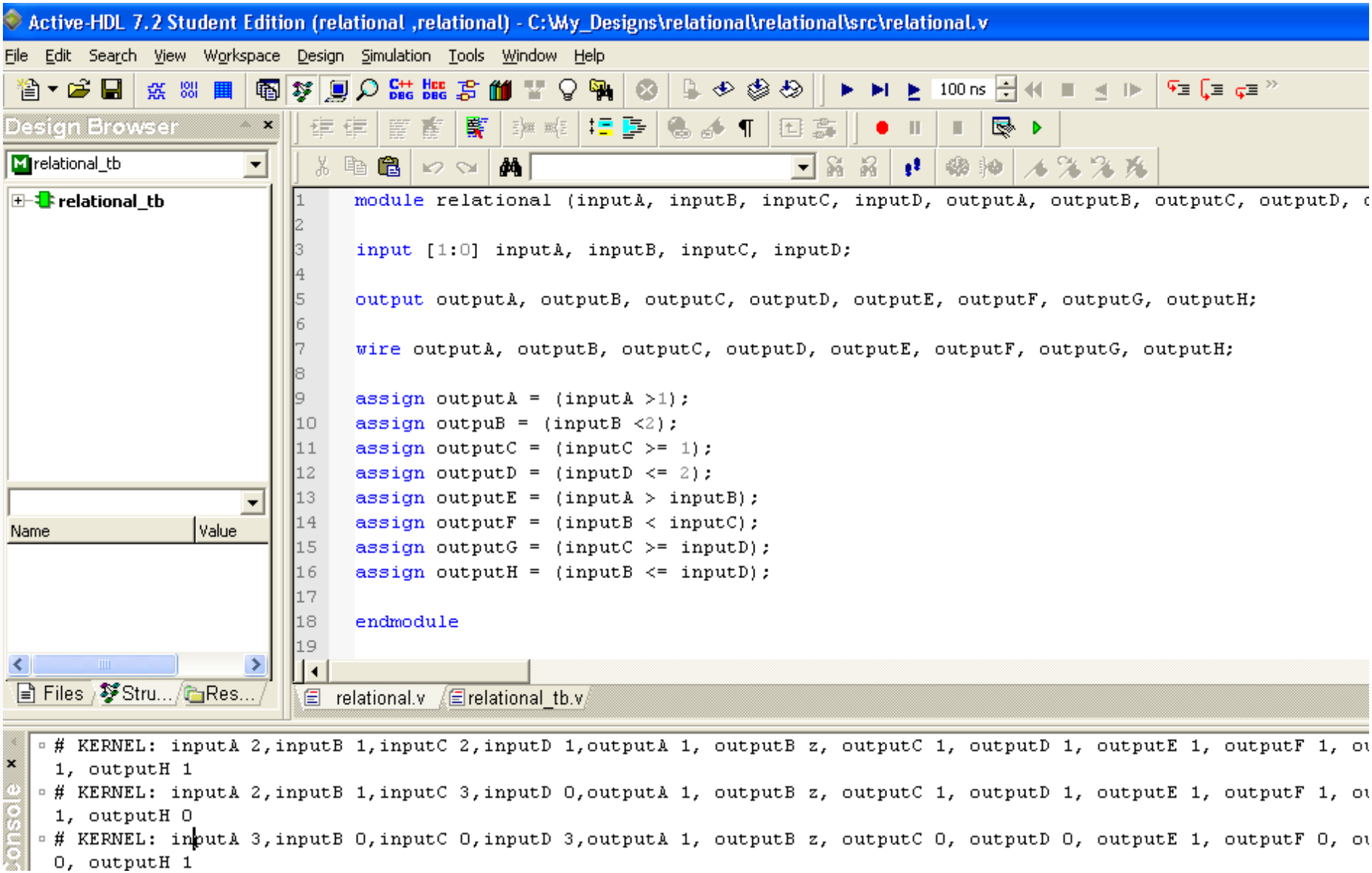
```
assign outputE = (inputA > inputB);
assign outputF = (inputB < inputC);
assign outputG = (inputC >= inputD);
assign outputH = (inputB <= inputD);

endmodule
```

Mã Testbench dùng để mô phỏng hoạt động của mạch trên

```
module relational_tb();

reg [1:0] inputA, inputB, inputC, inputD;

wire outputA, outputB, outputC, outputD, outputE, outputF, outputG, outputH;

integer i,j;

initial

begin

for(i=0;i<4;i=i+1)
begin
inputA = i;
inputB = (3 -i);

for (j=0; j <4; j = j+1)
begin
inputC = j;
```

```
inputD = (3-j);
#10;
end
end
end

// Instantiating your DUT

relational DUT (inputA, inputB, inputC, inputD, outputA, outputB, outputC, outputD, outputE, outputF, outputG, outputH);

initial
begin
$monitor ("inputA %h,inputB %h,inputC %h,inputD %h,outputA %b, outputB %b, outputC %b, outputD %b, outputE %b,
outputF %b, outputG %b, outputH %b",inputA, inputB, inputC, inputD,outputA, outputB, outputC, outputD, outputE,
outputF, outputG, outputH);
end

endmodule
```

Kết quả mô phỏng bằng ActiveHDL


Active-HDL 7.2 Student Edition (relational ,relational) - C:\My_Designs\relational\relational\src\relational.v
File Edit Search View Workspace Design Simulation Tools Window Help
100 ns
Design Browser
relational_tb
relational_tb
Name
Value
Files
Stru...
Res...
1 module relational (inputA, inputB, inputC, inputD, outputA, outputB, outputC, outputD,
2
3 input [1:0] inputA, inputB, inputC, inputD;
4
5 output outputA, outputB, outputC, outputD, outputE, outputF, outputG, outputH;
6
7 wire outputA, outputB, outputC, outputD, outputE, outputF, outputG, outputH;
8
9 assign outputA = (inputA >1);
10 assign outpuB = (inputB <2);
11 assign outputC = (inputC >= 1);
12 assign outputD = (inputD <= 2);
13 assign outputE = (inputA > inputB);
14 assign outputF = (inputB < inputC);
15 assign outputG = (inputC >= inputD);
16 assign outputH = (inputB <= inputD);
17
18 endmodule
19
relational.v
relational_tb.v
Console
KERNEL: inputA 2,inputB 1,inputC 2,inputD 1,outputA 1, outputB z, outputC 1, outputD 1, outputE 1, outputF 1, ou
1, outputH 1
KERNEL: inputA 2,inputB 1,inputC 3,inputD 0,outputA 1, outputB z, outputC 1, outputD 1, outputE 1, outputF 1, ou
1, outputH 0
KERNEL: inputA 3,inputB 0,inputC 0,inputD 3,outputA 1, outputB z, outputC 0, outputD 0, outputE 1, outputF 0, ou
0, outputH 1

Kết quả mô phỏng định thì của mạch trên

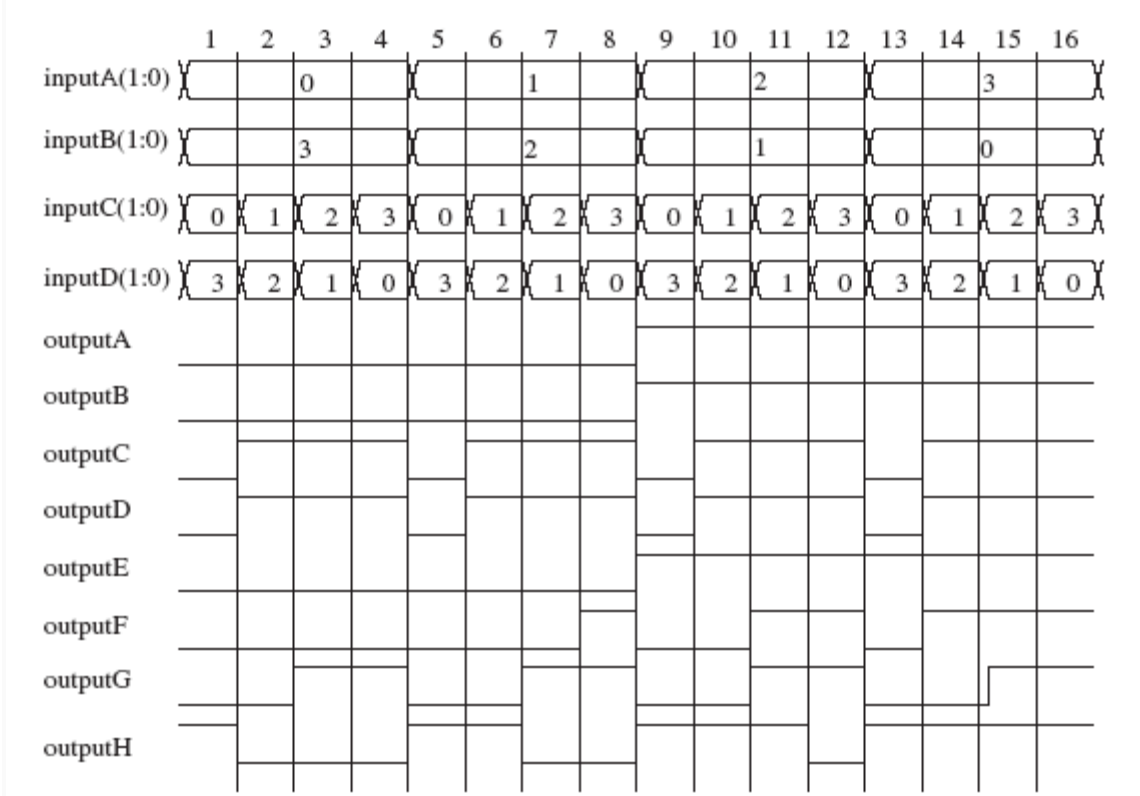


FIGURE 4.29. Diagram showing simulation results of Verilog test Bench module "*relational_tb*."

4.9 : Latch Inference

Khi viết mã Verilog người thiết kế phải hết sức cẩn thận để bảo đảm rằng không có những con chốt không mong muốn xuất hiện. Điều kiện dẫn đến những con chốt không mong muốn thường xảy ra khi người thiết kế dùng "if" hay "case" mà không hoàn chỉnh, tức là chưa liệt kê hết tất cả các trường hợp.

Ví dụ 4.54 là mã Verilog dùng "if" để tạo ra mạch tổ hợp tuy nhiên do không định nghĩa hết tất cả các trường hợp của câu lệnh if nên một con chốt không mong muốn đã bị suy ra (inferred).

```
module latch_infer ( inputA, inputB, inputC, inputD, outputA);

input inputA, inputB, inputC, inputD;
output outputA;

reg outputA;

always @ ( inputA or inputC or inputB or inputD)
begin

if ( inputA & inputB) // This is where inferred latch occurred

begin

if ( inputC | ~inputD)
outputA = 1'b1;
else
outputA = 1'bz;

end

end

endmodule
```

Hình 4.30 chỉ ra mạch được tổng hợp

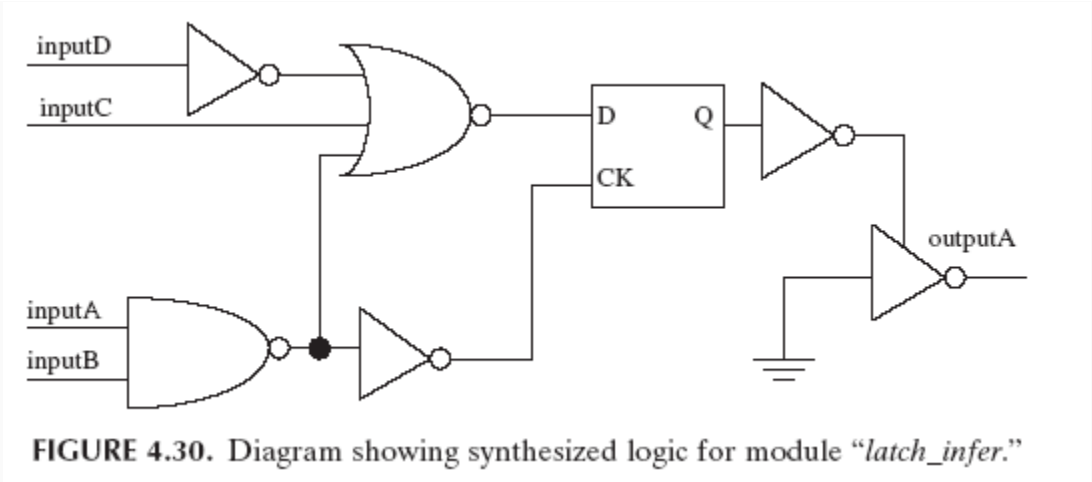


FIGURE 4.30. Diagram showing synthesized logic for module "*latch_infer.*"

Trong hình trên ta thấy rằng vì trong câu lệnh **if (inputA & inputB)**, nếu hai ngõ vào inputA và inputB cùng có giá trị 1 thì ngõ ra sẽ được được lái bằng giá trị 1 hay high-Z tùy vào hai ngõ inputC và inputD. Tuy nhiên điều gì sẽ xảy ra khi có một trong hai giá trị inputA hay inputB có giá trị khác 1. Việc này buộc, giá trị ngõ ra phải giữ nguyên giá trị trước đó nên phải có một con chốt trước đó để bảo toàn giá trị này.

Để sửa lỗi này ta chỉ cần thêm vào thiết kế trên như sau :

```
module latch_infer_modified ( inputA, inputB, inputC, inputD, outputA);

input inputA, inputB, inputC, inputD;
output outputA;

reg outputA;
```

```
always @ ( inputA or inputC or inputB or inputD)
begin

if ( inputA & inputB) // This is where inferred latch occurred

begin

if ( inputC | ~inputD)
outputA = 1'b1;
else
outputA = 1'bz;

end

else outputA = 1'b0; // This is modification

end

endmodule
```

Mạch tổng hợp đã loại bỏ được chốt

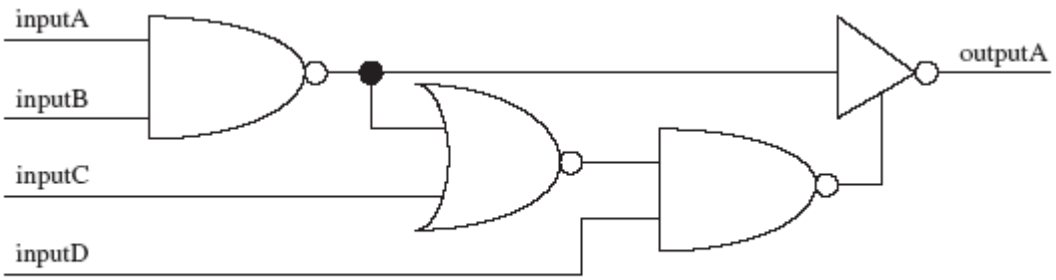


FIGURE 4.31. Diagram showing synthesized logic of module "*latch_noninfer*."

Ngoài câu lệnh if, những câu lệnh case cũng có thể dẫn đến những con chốt không mong muốn. Nếu một câu lệnh case được sử dụng mà không khai báo tất cả các trường hợp thì chốt cũng sẽ tự động được suy ra.

Ví dụ 4.56 chỉ ra mã Verilog trong đó một con chốt bị suy ra khi trong câu lệnh case không khai báo hết các trường hợp.

```
module case_infer ( inputA, inputB, select, outputA);

input inputA, inputB;

input [1:0] select;

output outputA;

reg outputA;

always @ ( inputA or inputB or select)
begin
case ( select)
```

```
2'b00 : outputA = inputA;
2'b01 : outputA = inputB;
endcase
end

endmodule
```

Hình 4.32 chỉ ra sơ đồ cho mạch logic được tổng hợp cho mô tả ở trên. Lưu ý một con chốt đã được tạo ra ở ngõ ra outputA.

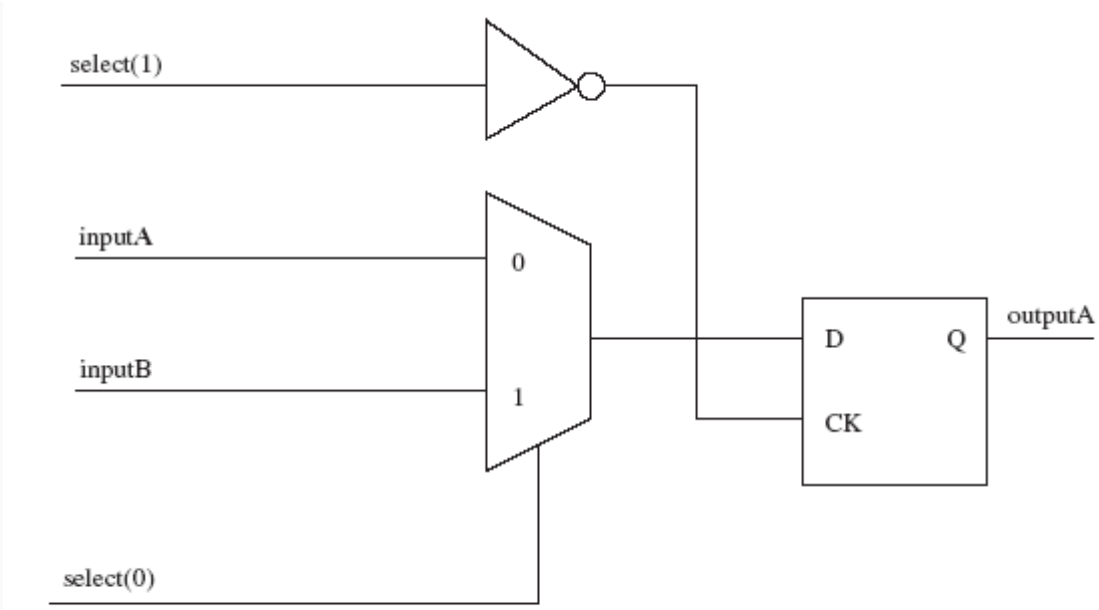


FIGURE 4.32. Diagram showing synthesized logic for module "*case_infer*."

Con chốt này được tạo ra do mã Verilog của câu lệnh case không xác định giá trị của tín hiệu ngõ ra khi select có giá trị khác

"00" hay "01". Điều này làm cho giá trị trước đó của outputA được giữa lại khi select không phải là "00" hay "01".

Một phương cách đơn giản để loại bỏ con chốt này là thêm điều kiện mặc định (default) trong câu lệnh case. Chính điều kiện này làm cho ngõ ra outputA được lái đến mức logic 0 khi select không phải là "00" hay "01".

```
module case_infer ( inputA, inputB, select, outputA);

input inputA, inputB;

input [1:0] select;

output outputA;

reg outputA;

always @ ( inputA or inputB or select)
begin
case ( select)

2'b00 : outputA = inputA;
2'b01 : outputA = inputB;
default : outputA = 1'b0; // This is the modification
endcase
end

endmodule
```

Kết quả tổng hợp sau khi đã chỉnh thêm vào điều kiện mặc định để loại bỏ chốt

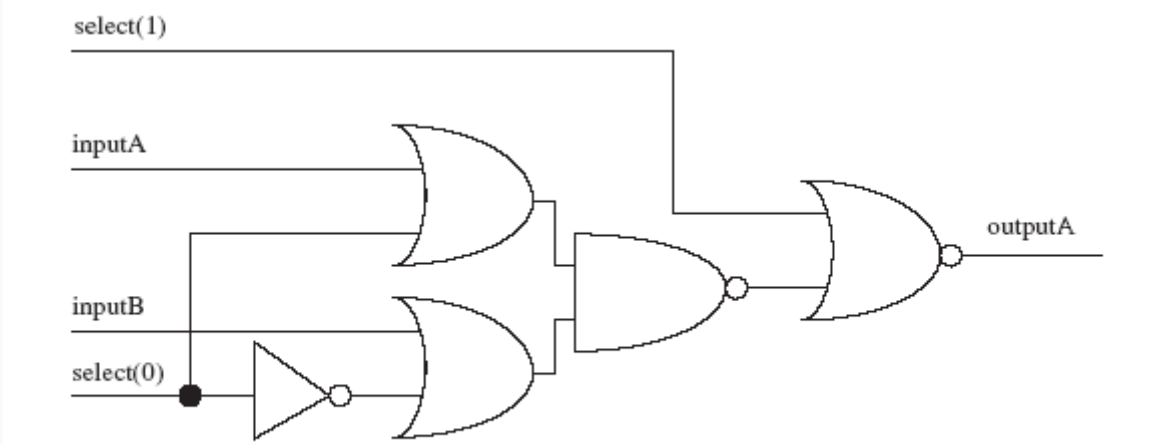


FIGURE 4.33. Diagram showing synthesized logic for module "*case_uninfer_diff*."

Ngoài cách dùng điều kiện mặc định như chỉ ra trong mã Verilog của module trên, sự suy ra chốt cũng có thể tránh được nếu ta xác định tất cả các khả năng trong select như sau :

```
module case_uninfer ( inputA, inputB, select, outputA);

input inputA, inputB;

input [1:0] select;

output outputA;

reg outputA;

always @ ( inputA or inputB or select)
```

```
begin

case ( select)

2'b00 : outputA = inputA;
2'b01 : outputA = inputB;
2'b10 : outputA = 1'b0;
2'b11 : outputA = 1'b0;

endcase

end

endmodule
```

Trong những ví dụ trước đó ta đã dùng câu lệnh case để mô phỏng chức năng của một mạch dồn kênh (Multiplexer) trong khối always. Ngoài cách này ra ta có thể dùng câu lệnh điều kiện để mô tả chức năng này.

```
module case_uninfer_assign ( inputA, inputB, select, outputA);

input inputA, inputB;

input [1:0] select;

output outputA;

wire outputA;
```

```
assign outputA = select[1] ? 1'b0 : select[0] ? inputB : inputA;

endmodule
```

4.10 : Mảng nhớ (Memory Array)

Khi viết mã Verilog tổng hợp đôi khi người thiết kế muốn mô tả một mảng nhớ. Việc mã hóa này thường thấy theo phương pháp viết mô tả hành vi (behavioral) nhưng khi chuyển qua tổng hợp thì có những giới hạn nhất định về kich thước.

Trong tổng hợp, khi một cell nhớ 1 bit được mã hóa sau khi tổng hợp sẽ thành một mạch dồn kênh và một flip flop. Việc biểu diễn này gây tốn tài nguyên silicon.Tuy nhiên, dù kích thước của một cell nhớ là tương đối lớn, việc thiết kế những mảng nhớ vẫn thường thấy trong quá trình thiết kế vi mạch.

Một ví dụ điển hình là người thiết kê cần thiết kế một dãy những thanh ghi để lưu trữ một giá trị nào đó hay người thiết kế có thể mã hóa một mảng ô nhớ dành cho tập thanh ghi trong khi thiết kế vi điều khiển hay vi xử lý.

Hình 4.34 chỉ ra mạch tổng hợp được của 1 cell nhớ. Tổng hợp những mảng nhớ lớn rất tốn kém về diện tích chip tuy nhiên việc làm này lại tương đối dễ dàng trong Verilog.

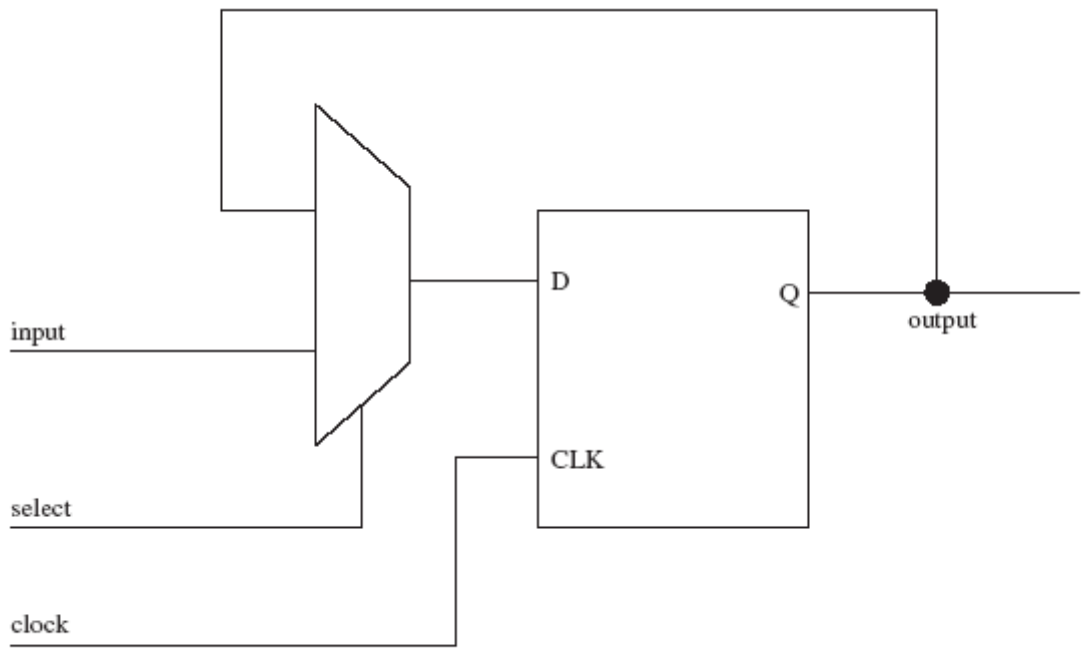


FIGURE 4.34. Diagram showing synthesis representation of a one-bit memory cell.

Ví dụ 4.60 trình bày mã Verilog cho 1 KByte bộ nhớ có thể tổng hợp được.Tuy nhiên người thiết kế nên lưu ý là, việc mô tả một mảng nhớ trong Verilog tương đối đơn giản nhưng công việc tổng hợp những mảng nhớ có kích thước lớn thường mất nhiều hơn vài phút so với việc tổng hợp những mã khác.

```
module memory ( addr, data_in, data_out, write, read, clock, reset);

// 1Kbyte memory module -- 128 address * 8 bits

input [6:0] addr;
input [7:0] data_in;
```

```
input write, read, clock, reset;
output [7:0] data_out;

reg [7:0] data_out;
reg [7:0] memory [127:0];

integer i;

// asynchronous reset

always @ (posedge clock or posedge reset)
begin
if( reset)
begin
data_out = 0;
// to initializing all memory to zero

for(i =0 ; i <128; i=i+1)
memory [i] <= 0;
end
else
begin

if(read)
data_out <= memory [addr];
else if (write)
begin
data_out <= 0;
memory [addr] <= data_in;
```

```
end
end
end

endmodule
```

Vì dụ 4.61 chỉ ra mã Verilog testbench dùng để kiểm tra tính chính xác của module memory trên

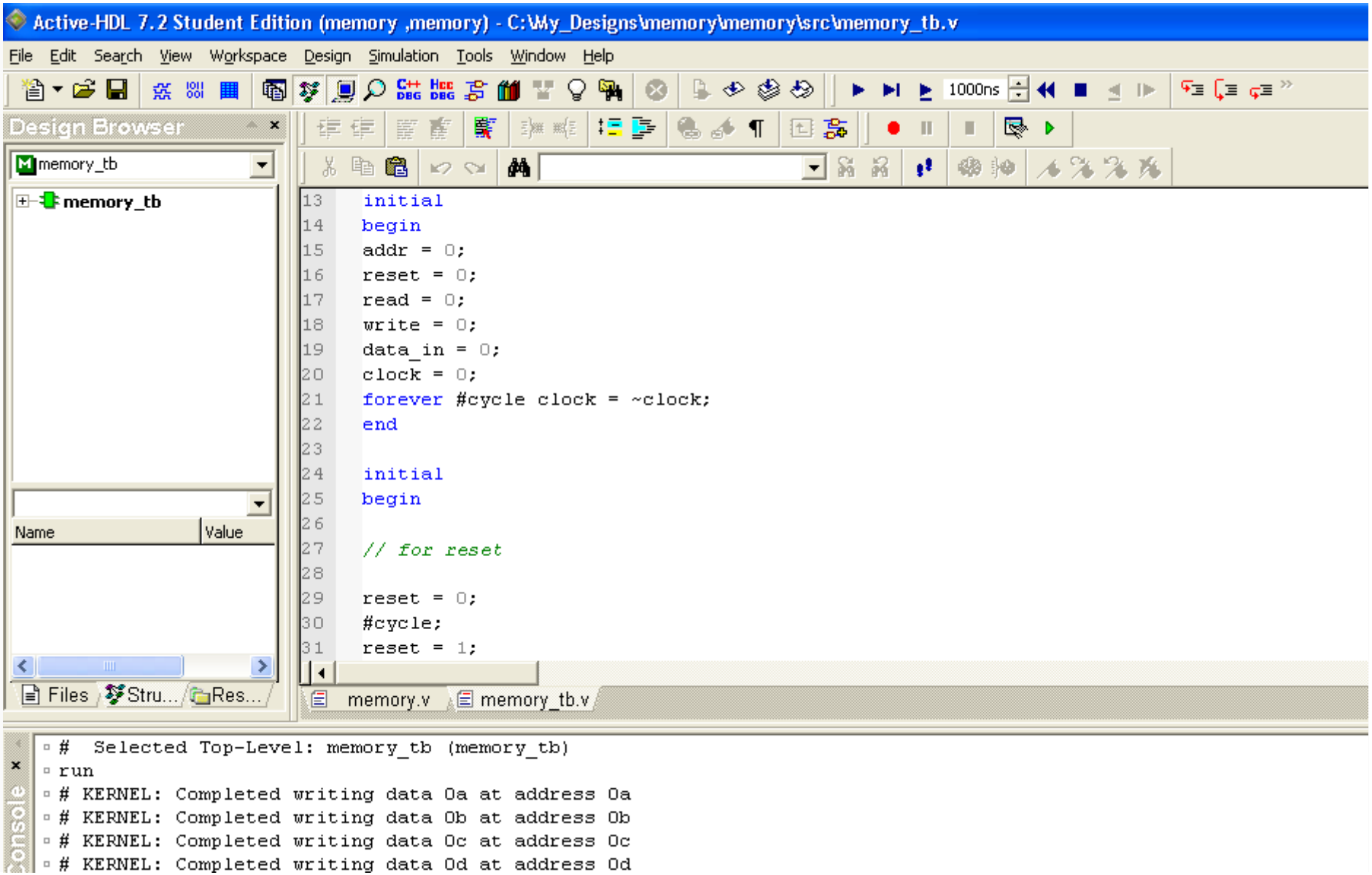
```
module memory_tb();

reg [6:0] address, addr;
reg [7:0] data, data_in;
reg write, read, clock, reset;

parameter cycle = 20;

// initializing and clock generation

initial
begin
addr = 0;
reset = 0;
read = 0;
write = 0;
data_in = 0;
clock = 0;
forever #cycle clock = ~clock;
end
```

```
initial
begin

// for reset

reset = 0;
#cycle;
reset = 1;
#cycle;
reset = 0;
#cycle;

for ( i = 10; i<15; i=i+1)
begin
address = i;
data = i;
memory_write ( address,data);
#cycle;
end

for ( i = 14; i>=10; i = i-1)
begin

address = i;
memory_read (address);
#cycle;

end
$stop;
```

```
end

task memory_write;

input [7:0] data;
input [6:0] address;
begin

addr = address;
data_in = data;
write = 0;
#cycle;
write = 1;

repeat (2) #cycle;
write = 0;
$display ( "Completed writing data %h at address %h", data_in,addr);

end

endtask

task memory_read;

input [6:0] address;

begin

read = 0;
```

```
addr = address;
data_in = 0;
#cycle;
read =1;
repeat (2) #cycle;
read = 0;
#cycle;
$display("Completed reading memory at address %h.Data is %h", addr,data_out);
end
endtask

memory DUT ( addr, data_in,data_out, write, read, clock,reset);

endmodule
```

Kết quả mô phỏng bằng ActiveHDL


Active-HDL 7.2 Student Edition (memory ,memory) - C:\My_Designs\memory\memory\src\memory_tb.v
File Edit Search View Workspace Design Simulation Tools Window Help
1000ns
Design Browser
memory_tb
memory_tb
Name
Value
Files
Stru...
Res...
13 initial
14 begin
15 addr = 0;
16 reset = 0;
17 read = 0;
18 write = 0;
19 data_in = 0;
20 clock = 0;
21 forever #cycle clock = ~clock;
22 end
23
24 initial
25 begin
26
27 // for reset
28
29 reset = 0;
30 #cycle;
31 reset = 1;
memory.v
memory_tb.v
Console
Selected Top-Level: memory_tb (memory_tb)
run
KERNEL: Completed writing data 0a at address 0a
KERNEL: Completed writing data 0b at address 0b
KERNEL: Completed writing data 0c at address 0c
KERNEL: Completed writing data 0d at address 0d

Giản đồ định thì cho quá trình đọc và ghi bộ nhớ ở trên

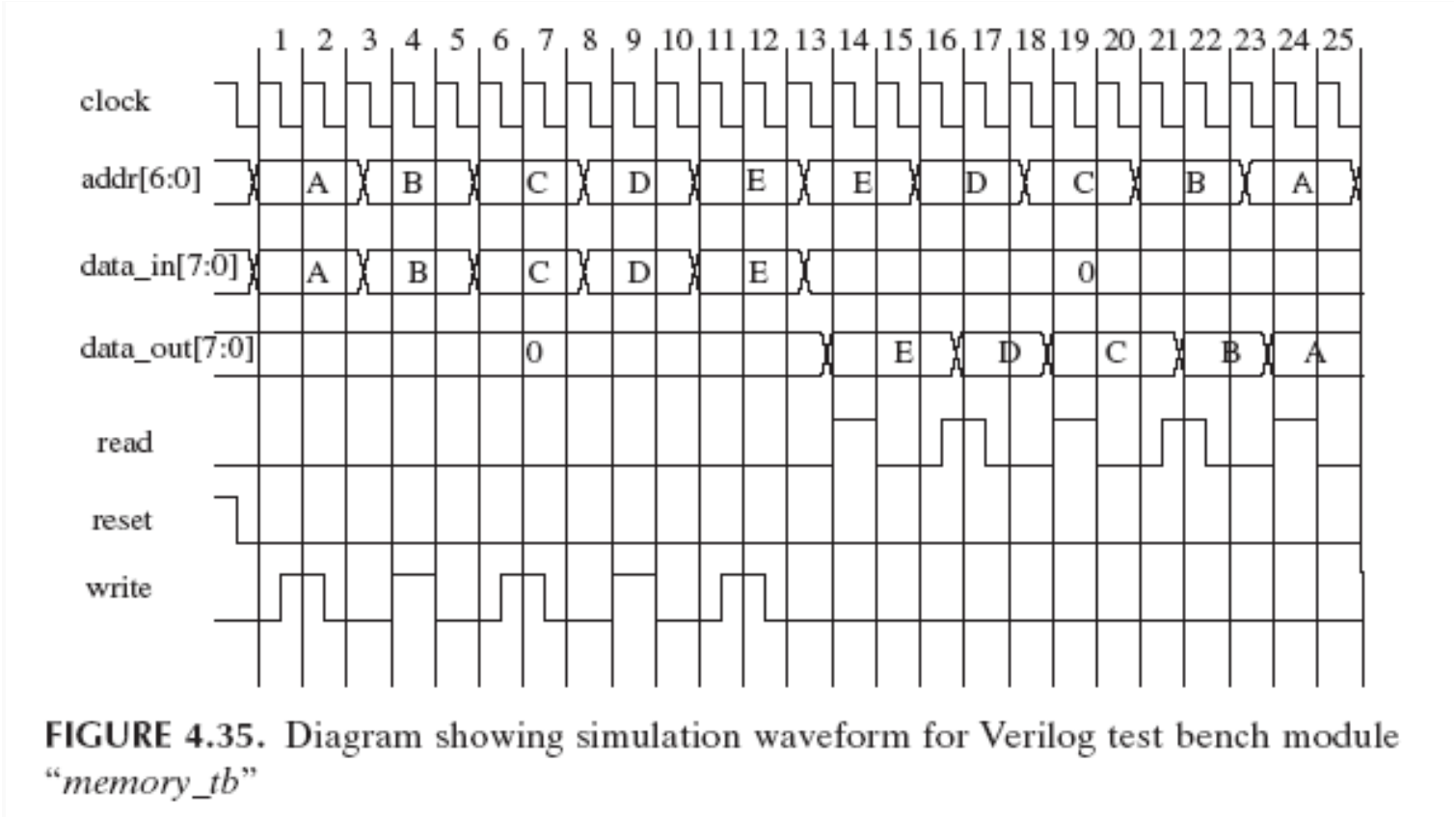


FIGURE 4.35. Diagram showing simulation waveform for Verilog test bench module "*memory_tb*"