

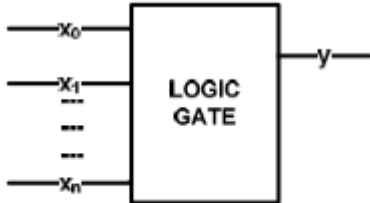
Đề cương ôn tập TKLGS

Chương I, II

Câu 1. Cổng logic cơ bản, tham số thời gian của cổng logic tổ hợp, nêu ví dụ. Khái niệm mạch tổ hợp và cách tính thời gian trễ trên mạch tổ hợp, khái niệm critical paths.

Trả lời:

- Cổng logic hay logic gate là cấu trúc mạch điện (sơ đồ khối hình) được lắp ráp từ các linh kiện điện tử để thực hiện chức năng của các hàm logic cơ bản $y = f(x_n, x_{n-1}, \dots, x_1, x_0)$. Trong đó các tín hiệu vào $x_{n-1}, x_{n-2}, \dots, x_1, x_0$ của mạch tương ứng với các biến logic $x_{n-1}, x_{n-2}, \dots, x_1, x_0$ của hàm. Tín hiệu ra y của mạch tương ứng với hàm logic y . Với các cổng cơ bản thường giá trị $n \leq 4$.



Hình 1.3: Mô hình cổng logic cơ bản

- Thời gian của cổng logic tổ hợp:
 - + Thời gian trễ lan truyền T_{pd} (Propagation delay) là thời gian tối thiểu kể từ thời điểm bắt đầu xảy ra sự thay đổi từ đầu vào X cho tới khi sự thay đổi này tạo ra sự thay đổi xác định tại đầu ra Y , hay nói một cách khác cho tới khi đầu ra Y ổn định giá trị..
 - + T_{cd} (Contamination delay) là khoảng thời gian kể từ thời điểm xuất hiện sự thay đổi của đầu vào X cho tới khi đầu ra Y bắt đầu xảy ra sự mất ổn định. Sau giai đoạn mất ổn định hay còn gọi là giai đoạn chuyển tiếp tín hiệu tại đầu ra sẽ thiết lập trạng thái xác định vững bền..
- ⇒ Như vậy $T_{pd} > T_{cd}$ và khi nhắc đến độ trễ của cổng thì là chỉ tới T_{pd} .
- Mạch logic tổ hợp (Combinational logic circuit) là mạch mà giá trị tổ hợp tín hiệu ra tại một thời điểm chỉ phụ thuộc vào giá trị tổ hợp tín hiệu vào tại thời điểm đó. Hiểu một cách khác mạch tổ hợp không có trạng thái, không chứa các phần tử nhớ mà chỉ chứa các phần tử thực hiện logic chức năng như AND, OR, NOT
- ...
- Cách tính thời gian trễ trên mạch tổ hợp:
 - + T_{delay} : là khoảng thời gian lớn nhất kể từ thời điểm xác định tất cả các giá trị đầu vào cho tới thời điểm tất cả các kết quả ở đầu ra trở nên ổn định. Trên thực tế với vi mạch tích hợp việc thời gian trễ rất nhỏ nên việc tìm tham số độ trễ của mạch được thực hiện bằng cách liệt kê tất cả các đường biến đổi tín hiệu có thể từ tất cả các đầu vào tới tất cả đầu ra sau đó dựa trên thông số về thời gian của các cổng và độ trễ đường truyền có thể tính được độ trễ của các đường dữ liệu này và tìm ra đường có độ trễ lớn nhất
 - + Các đường truyền có độ trễ lớn nhất được gọi là Critical paths (đường tới hạn). Các đường truyền này cần đặc biệt quan tâm trong quá trình tối ưu hóa độ trễ của vi mạch.

Câu 2. Các loại Flip-flop cơ bản, tham số thời gian của Flip-flop. Khái niệm mạch dãy, cách tính thời gian trễ trên mạch dãy. Khái niệm pipelined, các phương pháp tăng hiệu suất mạch dãy.

Trả lời:

các loại flip-flop cơ bản:

D-flip-flop, RS flip-flop, JK flip-flop, T flip-flop: trong đó D flip-flop hay được sử dụng.

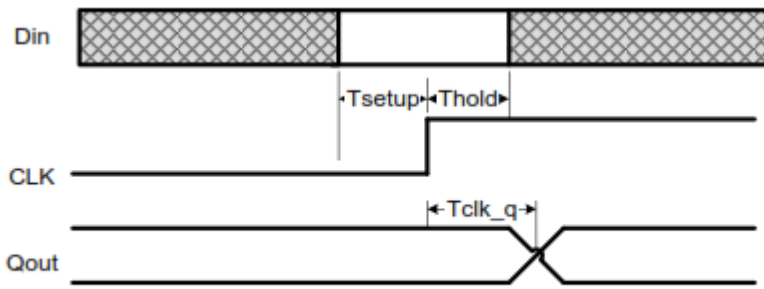
D flip-flop: là phần tử nhớ làm việc theo mức xung, có 2 dạng sườn là sườn âm và sườn dương. (thường dùng sườn dương)

Tham số của flip-flop:

Đối với D-flip-flop và D-latch nhớ thì có hai tham số thời gian hết sức quan trọng là T_{setup} , và T_{hold} .

+ T_{setup} : là khoảng thời gian cần thiết cần giữ ổn định đầu vào trước sườn tích cực của xung nhịp Clock.

+ T_{hold} : Là khoảng thời gian tối thiểu cần giữ ổn định dữ liệu đầu vào sau sườn tích cực của xung nhịp Clock.



Hình 1-7. Tham số thời gian của D-Flip-Flop

Khái niệm mạch dãy:

Mạch logic dãy (Sequential logic circuits) còn được gọi là mạch logic tuần tự là mạch số mà tín hiệu ra tại một thời điểm không những phụ thuộc vào tổ hợp tín hiệu đầu vào tại thời điểm đó mà còn phụ thuộc vào tín hiệu vào tại các thời điểm trước đó. Hiểu một cách khác mạch dãy ngoài các phần tử tổ hợp có chứa các phần tử nhớ và nó lưu trữ lớn hơn một trạng thái của mạch.

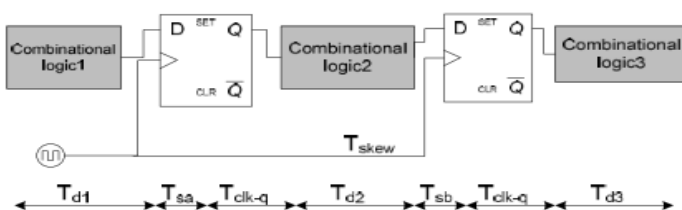
Tham số thời gian của mạch tuần tự:

Có quan hệ mật thiết với đặc điểm của tín hiệu đồng bộ Clock. Ví dụ với một mạch tuần tự điển hình dưới đây. Mạch tạo từ hai lớp thanh ghi sử dụng Flip-flop A và B, trước giữa và sau thanh ghi là ba khối logic tổ hợp Combinational logic 1, 2, 3, các tham số thời gian cụ thể như sau:

+ T_{d1} , T_{d2} , T_{d3} . Là thời gian trễ tương ứng của 3 khối mạch tổ hợp 1, 2, 3. T_{sa} , T_{sb} là thời gian thiết lập (T_{setup}) của hai Flipflop A, B tương ứng

+ T_{clk-q} là khoảng thời gian cần thiết để dữ liệu tại đầu ra Q xác định sau thời điểm kích hoạt của sườn Clock

T_{skew} : Đối với mạch đồng bộ thì sẽ là lý tưởng nếu như điểm kích hoạt (sườn lên hoặc sườn xuống) của xung nhịp Clock tới các Flip-flop cùng một thời điểm. Tuy vậy trên thực tế bao giờ cũng tồn tại độ trễ giữa hai xung Clock đến hai Flip-flop khác nhau. T_{skew} là độ trễ lớn nhất của xung nhịp Clock đến hai Flip-flop khác nhau trong mạch. Thời gian chênh lệch lớn nhất giữa tín hiệu xung nhịp, thời gian trễ này sinh ra do độ trễ trên đường truyền của xung Clock từ A đến B. Trên thực tế T_{skew} giữa hai Flip-flop liên tiếp có giá trị rất bé so với các giá trị độ trễ khác và có thể bỏ qua, nhưng đối với những mạch cỡ lớn khi số lượng Flip-flop nhiều hơn và phân bố xa nhau thì giá trị T_{skew} có giá trị tương đối lớn.



Hình 1.9: Tham số thời gian của mạch tuần tự

Những tham số trên cho phép tính toán các đặc trưng thời gian của mạch tuần tự đó là:

+Thời gian trễ trước xung nhịp Clock tại đầu vào: $T_{input_delay} = T_{d1} + T_{sa}$

+ Thời gian trễ sau xung nhịp Clock tại đầu ra.: $T_{output_delay} = T_{d3} + T_{clk_q}$

+Chu kỳ tối thiểu của xung nhịp Clock, hay là khoảng thời gian tối thiểu đảm bảo cho dữ liệu trong mạch được xử lý và truyền tải giữa hai lớp thanh ghi liên tiếp mà không xảy ra sai sót. Nếu xung nhịp đầu vào có chu kỳ nhỏ hơn T_{clk_min} thì mạch sẽ không thể hoạt động theo thiết kế.

$$T_{clk_min} = T_{clk-q} + T_{d2} + T_{sb} + T_{skew}$$

+ từ đó tính được xung nhịp tối đa của vi mạch là

$$F_{max} = 1 / T_{clk_min} = 1 / (T_{clk-q} + T_{d2} + T_{sb} + T_{skew})$$

Câu 3. Các phương pháp thể hiện thiết kế mạch logic số, nêu và phân tích các ưu điểm của phương pháp sử dụng HDL.

Trả lời:

Các phương pháp thể hiện thiết kế mạch logic số

Có hai phương pháp cơ bản được sử dụng để mô tả vi mạch số là mô tả bằng sơ đồ logic (schematic) và mô tả bằng ngôn ngữ mô tả phần cứng HDL (Hardware Description Language).

+ *Mô tả bằng sơ đồ:* vi mạch được mô tả trực quan bằng cách ghép nối các phần tử logic khác nhau một cách trực tiếp giống như ví dụ ở hình vẽ dưới đây. Thông thường các phần tử không đơn thuần là các đối tượng đồ họa mà còn có các đặc tính vật lý gồm chức năng logic, thông số tải vào ra, thời gian trễ... Những thông tin này được lưu trữ trong thư viện logic thiết kế. Mạch vẽ ra có thể được mô phỏng để kiểm tra chức năng và phát hiện và sửa lỗi một cách trực tiếp

Ưu điểm của phương pháp này là cho ra sơ đồ các khối logic rõ ràng thuận tiện cho việc phân tích mạch, tuy vậy phương pháp này chỉ được sử dụng để thiết kế những mạch cỡ nhỏ, độ phức tạp không cao. Đối với những mạch cỡ lớn hàng trăm ngàn cổng logic thì việc mô tả đồ họa là gần như không thể và nếu có thể cũng tốn rất nhiều thời gian, chưa kể những khó khăn trong công việc kiểm tra lỗi trên mạch sau đó

+ *Mô tả bằng HDL:* là mô tả bằng phần cứng bằng ngôn ngữ lập trình, Có ba ngôn ngữ mô tả phần cứng phổ biến hiện nay là:

Ngôn ngữ Verilog: Ra đời năm 1983, Verilog được IEEE chính thức tiêu chuẩn hóa vào năm 1995 và sau đó là các phiên bản năm 2001, 2005. Đây là một ngôn ngữ mô tả phần cứng có cấu trúc và cú pháp gần giống với ngôn ngữ lập trình C, ngoài khả năng hỗ trợ thiết kế thì Verilog rất mạnh trong việc hỗ trợ cho quá trình kiểm tra thiết kế (Design testing).

Ngôn ngữ VHDL: VHDL viết tắt của Very-high-speed intergrated circuits Hardware Description Language, hay ngôn ngữ mô tả cho các mạch tích hợp tốc độ cao. VHDL lần đầu tiên được phát triển bởi Bộ Quốc Phòng Mỹ nhằm hỗ trợ cho việc thiết kế những vi mạch tích hợp chuyên dụng (ASICs). VHDL cũng được IEEE chuẩn hóa vào các năm 1987, 1991, 2002, và 2006 và mới nhất 2009. VHDL được phát triển dựa trên cấu trúc của ngôn ngữ lập trình Ada. Cấu trúc của mô tả VHDL tuy phức tạp hơn Verilog nhưng mang tính logic chặt chẽ và gần với phần cứng hơn.

Ngôn ngữ AHDL: Altera HDL được phát triển bởi công ty bán dẫn Altera với mục đích dùng thiết kế cho các sản phẩm FPGA và CPLD của Altera. AHDL có cấu trúc hết sức chặt chẽ và là ngôn ngữ rất khó sử dụng nhất so với 2 ngôn ngữ trên. Bù lại AHDL cho phép mô tả thực thể logic chi tiết và chính xác hơn. Ngôn ngữ này ít phổ biến tuy vậy nó cũng được rất nhiều chương trình phần mềm hỗ trợ mô phỏng biên dịch.

Ưu điểm của phương pháp sử dụng VHDL trong thiết kế mạch số:

+ khi thiết kế các mạch cơ lớn, độ phức tạp lớn thì việc mô tả bằng sơ đồ khối gần như là không thể nếu có cũng mất rất nhiều thời gian và công tác kiểm tra rất khó khăn. Chính vì vậy sử dụng pp VHDL để thiết kế các mạch có độ mức độ phức tạp lớn là rất cần thiết. (cấu trúc mô tả của VHDL phức tạp hơn của verilog nhưng mang tính logic chặt chẽ và gần gũi phần cứng hơn)

+ Mô tả bằng HDL tốt hơn cách mô tả bằng sơ đồ, vì nó mô tả thực thể logic chi tiết và chính xác hơn, những mô hình lớn hơn, phức tạp hơn một cách chi tiết và chính xác hơn

Câu 4. Nguyên lý hiện thực hóa các hàm logic trên các IC khả trình dạng PROM, PAL, PLA, GAL, cấu trúc ma trận AND, OR, macrocell.

Trả lời:

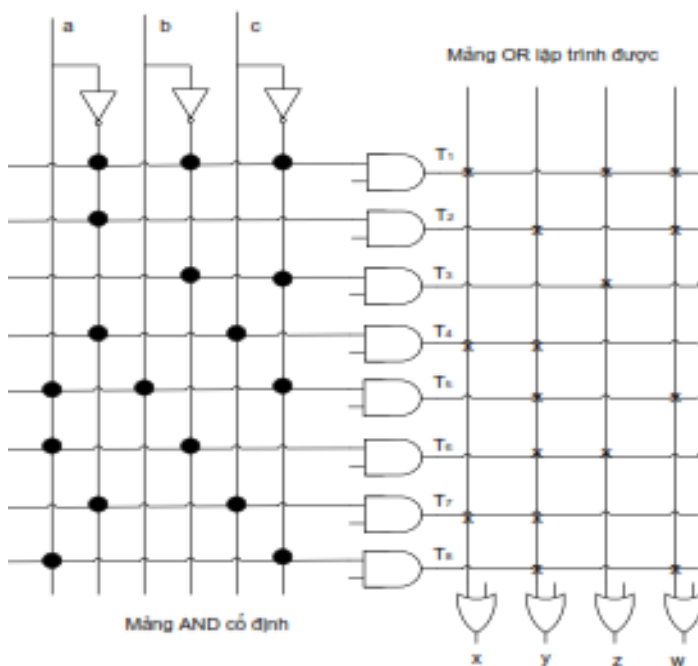
Nguyên lý thực hóa các hàm logic trên các IC khả trình PROM, PAL, PLA, GAL:

+ Trong Kỹ thuật số ta đã chỉ ra mọi hàm logic tổ hợp đều có thể biểu diễn dưới dạng chuẩn tắc tuyển tức là dưới dạng tổng của các tích đầy đủ, hoặc chuẩn tắc hội, tức là dạng tích của các tổng đầy đủ. Hai cách biểu diễn này là hoàn toàn tương đương.

+ Nguyên lý này cho phép hiện thực hóa hệ hàm logic tổ hợp có thể bằng cách ghép hai mảng ma trận nhân (AND) và ma trận cộng (OR). Nếu một trong các mảng này có tính khả trình thì IC sẽ có tính khả trình.

Tính khả trình của PROM được thực hiện thông qua các kết nối antifuse (cầu chì ngược). Antifuse là một dạng vật liệu làm việc với cơ chế như vật liệu ở cầu chì (fuse) nhưng theo chiều ngược lại. Nếu như cầu chì trong điều kiện kích thích (quá tải về dòng điện) thì nóng chảy và ngắt dòng thì antifuse trong điều kiện tương tự như tác động hiệu ứng phù hợp sẽ biến đổi từ vật liệu không dẫn điện thành dẫn điện. Ở trạng thái chưa lập trình thì các điểm nối là antifuse nghĩa là ngắt kết nối, khi lập trình thì chỉ những điểm nối xác định bị “đốt” để tạo kết nối vĩnh viễn. Quá trình này chỉ được thực hiện một lần và theo một chiều vì PROM không thể tái lập trình được.

Cấu trúc Ma trận AND là:



Hình 1-12. Cấu trúc PROM

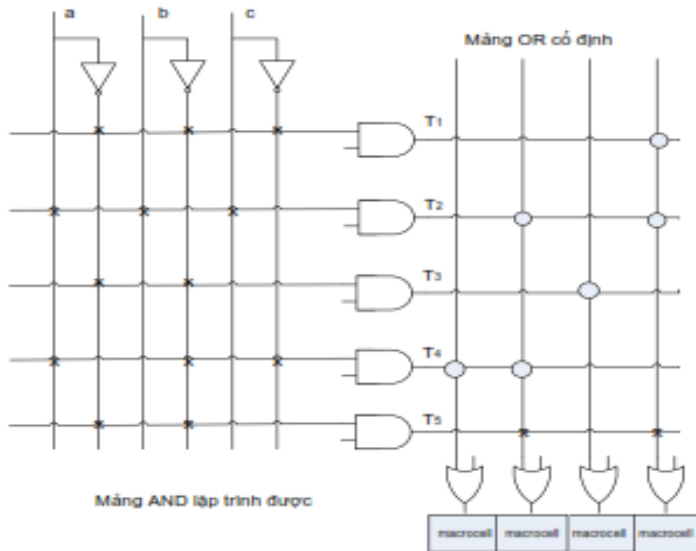
Tại mảng nhân AND, các đầu vào sẽ được tách thành hai pha, ví dụ a thành pha thuận a và nghịch \bar{a} , các chấm (•) trong mảng liên kết thể hiện kết nối cứng, tất cả các kết nối trên mỗi đường ngang sau đó được thực hiện phép logic AND, như vậy đầu ra của mỗi phần tử AND là một nhân tử tương ứng của các đầu vào. Ví dụ như hình trên thu được các nhân tử T1, T3 như sau:

$$T_1 = \bar{a} \cdot \bar{b} \cdot \bar{c} \quad T_3 = \bar{b} \cdot \bar{c}$$

Các nhân tử được gửi tiếp đến mảng cộng OR, ở mảng này “X” dùng để biểu diễn kết nối lập trình được. Ở trạng thái chưa lập trình thì tất cả các điểm nối đều là X tức là không kết nối, tương tự như trên, phép OR thực hiện đối với toàn bộ các kết nối trên đường đứng và gửi ra các đầu ra X, Y, Z,... Tương ứng với mỗi đầu ra như vậy thu được hàm dưới dạng tổng của các nhân tử, ví dụ tương ứng với đầu ra Y:

$$Y = T_1 + T_3 = \bar{a} \cdot \bar{b} \cdot \bar{c} + \bar{b} \cdot \bar{c}$$

Cấu trúc ma trận OR:



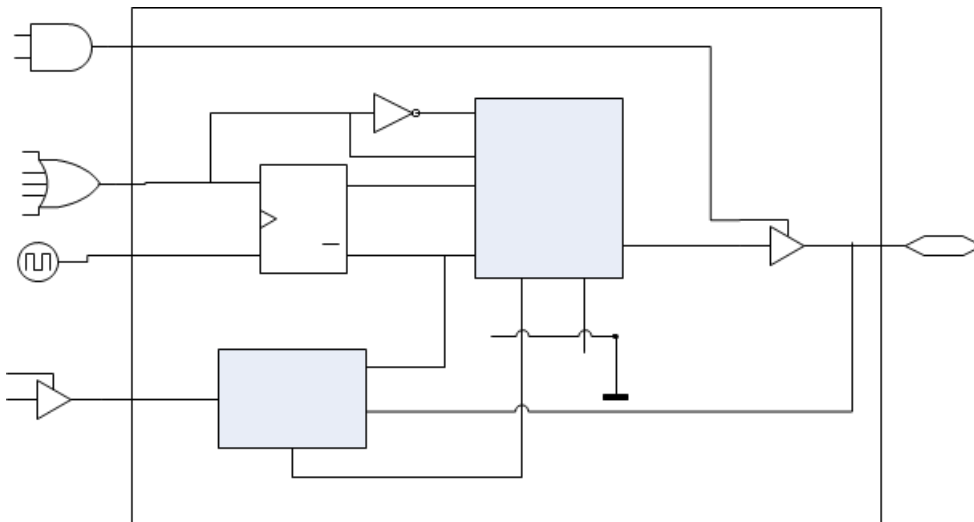
Hình 1-13. Cấu trúc PAL

sử dụng hai mảng logic nhưng nếu như ở PROM mảng OR là mảng lập trình được thì ở PAL mảng AND lập trình được còn mảng OR được gắn cứng, nghĩa là các thành phần tích có thể thay đổi nhưng tổ hợp của chúng sẽ cố định, cải tiến này tạo sự linh hoạt hơn trong việc thực hiện các hàm khác nhau.

Ngoài ra cấu trúc cổng ra của PAL còn phân biệt với PROM ở mỗi đầu ra của mảng OR lập trình được được dẫn bởi khối logic gọi là Macrocell

Cấu trúc của macrocell: Mỗi macrocell chứa 1 Flip-Flop Register, hai bộ dồn kênh (Multiplexers) 2 và 4 đầu vào Mux2, Mux4. Đầu ra của Mux2 thông qua một cổng 3 trạng thái trả lại mảng AND, thiết kế này cho kết quả đầu ra có thể sử dụng như một tham số đầu vào, tất nhiên trong trường hợp đó thì kết quả đầu ra buộc phải đi qua Flip-flop trước.

Đầu ra của macrocell cũng thông qua cổng 3 trạng thái có thể lập trình được để nối với cổng giao tiếp của PAL.



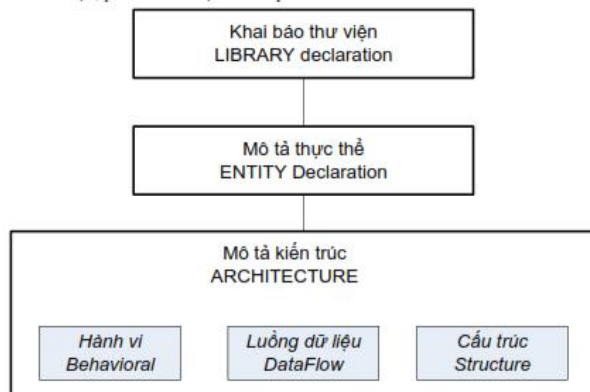
Hình 1-14. Cấu trúc Macrocell

Câu 5. Cấu trúc của thiết kế bằng VHDL, đặc điểm và ứng dụng của các dạng mô tả kiến trúc trong VHDL, ví dụ. Trình bày về dữ liệu kiểu BIT và STD_LOGIC.

Trả lời:

Cấu trúc của thiết kế bằng VHDL:

- Cấu trúc tổng thể của một module VHDL gồm ba phần, phần khai báo thư viện, phần mô tả thực thể và phần mô tả kiến trúc.



Hình 2-1. Cấu trúc của một thiết kế VHDL

- **Khai báo thư viện** phải được đặt đầu tiên trong mỗi thiết kế VHDL, lưu ý rằng nếu ta sử dụng một tệp mã nguồn để chứa nhiều khối thiết kế khác nhau thì mỗi một khối đều phải yêu cầu có khai báo thư viện đầu tiên, nếu không khi biên dịch sẽ phát sinh ra lỗi.
- **Khai báo entity** là khai báo về mặt cấu trúc các cổng vào ra (port), các tham số tĩnh dùng chung (generic) của một khối thiết kế VHDL.
Trong thành phần của khai báo thực thể ngoài khai báo cổng và khai báo generic còn có thể có hai thành phần khác là khai báo kiểu dữ liệu, thư viện người dùng chung, chương trình con... Và phần phát biểu chung chỉ chứa các phát biểu đồng thời. Các thành phần này nếu có sẽ có tác dụng đối với tất cả các kiến trúc của thực thể
- **Mô tả kiến trúc (ARCHITECTURE)** là phần mô tả chính của một khối thiết kế VHDL, nếu như mô tả entity chỉ mang tính chất khai báo về giao diện của thiết kế thì mô tả kiến trúc chứa nội dung về chức năng của đối tượng thiết kế
 - **behavioral**: Mô tả hành vi gần giống như mô tả bằng lời cách thức tính toán kết quả dựa vào các kết quả đầu vào. Toàn bộ mô tả hành vi phải được đặt trong một khối {process (signal list) end process;} ý nghĩa của khối này là nó tạo một quá trình để “theo dõi” sự thay đổi của tất cả các tín hiệu có trong danh sách tín hiệu (sensitive list), khi có bất kỳ một sự thay đổi giá trị nào của tín hiệu trong danh sách thì nó sẽ thực hiện quá trình tính toán ra kết quả tương ứng ở đầu ra. Chính vì vậy trong đó rất hay sử dụng các phát biểu tuần tự như if, case, hay các vòng lặp loop.
 - **dataflow**: dạng mô tả tương đối ngắn gọn và rất hay được sử dụng khi mô tả các khối mạch tổ hợp. Các phát biểu trong khối begin end là các phát biểu đồng thời (concurrent statements) nghĩa là không phụ thuộc thời gian thực hiện của nhau, nói một cách khác không có thứ tự ưu tiên trong việc sắp xếp các phát biểu này đứng trước hay đứng sau trong đoạn mã mô tả.
 - **Structure**: Mô tả cấu trúc là mô tả sử dụng các mô tả có sẵn dưới dạng module con (component). ưu điểm của phương pháp này là khi tổng hợp trên thư viện cổng sẽ cho ra kết quả đúng với ý tưởng thiết kế nhất

Câu 6. Các dạng phát biểu có trong VHDL, phát biểu tuần tự, phát biểu song song, đặc điểm ứng dụng, lấy ví dụ.

Trả lời:

- Trong VHDL có phân loại 3 loại đối tượng dữ liệu: là variable, constant, signal.
+các đối tượng được khai báo: object_tupe data : ture[:=]
- Các kiểu dữ liệu trong VHDL: kiểu dữ liệu tiền định nghĩa, kiểu Dữ liệu vô hướng trong, Dữ liệu phức hợp.
- Dữ liệu tiền định nghĩa là dữ liệu được định nghĩa trong các bộ thư viện chuẩn của VHDL.
- dữ liệu người dùng định nghĩa là các dữ liệu được định nghĩa lại dựa trên cơ sở dữ liệu tiền định nghĩa, phù hợp cho từng trường hợp sử dụng khác nhau.

Phát biểu tuần tự:

Trong ngôn ngữ VHDL phát biểu tuần tự (sequential statement) được sử dụng để diễn tả thuật toán thực hiện trong một chương trình con (subprogram) tức là dạng function hoặc procedure hay trong một quá trình (process). Các lệnh tuần tự sẽ được thực thi một cách lần lượt theo thứ tự xuất hiện của chúng trong chương trình.

Các dạng phát biểu tuần tự bao gồm: phát biểu đợi (wait statement), xác nhận (assert statement), báo cáo (report statement), gán tín hiệu (signal assignment statement), gán biến (variable assignment statement), gọi thủ tục (procedure call statement), các lệnh rẽ nhánh và vòng lặp, lệnh điều khiển if, loop, case, exit, return, next statements), lệnh trống (null statement)

Ứng dụng của phát biểu tuần tự: là sử dụng để thiết kế các mạch logic

Phát biểu đồng thời:

Phát biểu đồng thời (concurrent statements) được sử dụng để mô tả các kết nối giữa các khối thiết kế, mô tả các khối thiết kế thông qua cách thức làm việc của nó (process statement). Hiểu một cách khác các phát biểu đồng thời dùng để mô tả phân cứng về mặt cấu trúc hoặc cách thức làm việc đúng như nó vốn có. Khi mô phỏng thì các phát biểu đồng thời được thực hiện song song độc lập với nhau. Mã VHDL không quy định về thứ tự thực hiện của các phát biểu. Bất kể phát biểu đồng thời nào có quy định thứ tự thực hiện theo thời gian đều gây ra lỗi biên dịch.

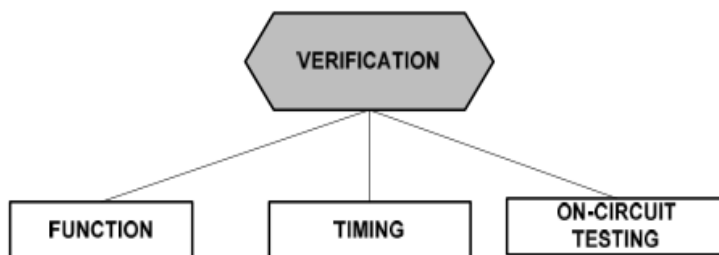
Có tất cả 7 dạng phát biểu đồng thời: khối (block statement), quá trình (process statement), gọi thủ tục (procedure call statement), xác nhận gán tín hiệu (signal assignment statement), khai báo khối con (component declaration statement), và phát biểu sinh (generate statement).

Ứng dụng của phát biểu đồng thời:

Câu 7. Các dạng kiểm tra thiết kế, vai trò và yêu cầu chung đối với kiểm tra thiết kế trên VHDL, sơ đồ các dạng kiểm tra thiết kế trên VHDL và vai trò của chúng.

Trả lời:

Thiết kế trên FPGA có thể được kiểm tra ở nhiều mức khác nhau về cả chức năng lẫn về các yêu cầu khác về mặt tài nguyên hay hiệu suất làm việc.



Hình 3.46. Verification FPGA design

Kiểm tra bằng mô phỏng: Các công cụ mô phỏng có thể dùng để mô phỏng chức năng (Functional Simulation) của mạch thiết kế và mô phỏng về mặt thời gian (Timing simulation). Kiểm tra có thể được thực hiện từ bước đầu tiên của quá trình thiết kế (mô tả VHDL) cho tới bước cuối cùng (PAR).

- Sau khi có mô tả bằng VHDL thiết kế cần được kiểm tra kỹ về mặt chức năng tại thời điểm này trước khi thực hiện các bước ở bên dưới.
- Kiểm tra sau tổng hợp: một mô tả netlist sau tổng hợp (post-synthesis simulation model), thư viện UNISIM được tạo ra phục vụ cho quá trình kiểm tra sau tổng hợp. Để kiểm tra thiết kế này trình mô phỏng cần có mô tả tương ứng của thư viện UNISIM, nếu có phát sinh lỗi về mặt chức năng thì phải quay lại bước mô tả VHDL để sửa lỗi.
- Kiểm tra sau Translate: mô tả netlist sau translate (post-synthesis simulation model) là mô tả trên thư viện SIMPRIM. Chương trình mô phỏng cũng phải cần được tích hợp hoặc hỗ trợ thư viện SIMPRIM.
- Kiểm tra sau Map: mô tả netlist sau translate (post-map simulation model, và post-map static timing) là mô tả trên thư viện SIMPRIM, thư viện này có tham số mô tả về mặt thời gian thực và kể từ bước này ngoài mô phỏng để kiểm tra về mặt chức năng thì thiết kế có thể được kiểm tra các tham số chính xác về mặt thời gian.
- Kiểm tra sau Place and Route: Tương tự như kiểm tra sau Mapping, điểm khác là mô phỏng để kiểm tra các tham số thời gian tĩnh ở đây có độ chính xác gần với mạch thật trên FPGA nhất.

Phân tích tham số thời gian tĩnh: Phân tích thời gian tĩnh (Static timing analysis) cho phép nhanh chóng xác định các tham số về mạch thời gian sau quá trình Place & Routing, kết quả của bước kiểm tra này cho phép xác định có hay không các đường truyền vi phạm các điều kiện ràng buộc về mặt thời gian, chỉ ra các đường gây trễ vi phạm để người thiết kế tiến hành những thay đổi để tối ưu mạch nếu cần thiết

Kiểm tra trực tiếp trên mạch : (On-circuit Testing) là quá trình thực hiện sau khi cấu hình FPGA đã được nạp vào IC, đối với những thiết kế đơn giản thì mạch được nạp có thể được kiểm tra một cách trực quan bằng các đối tượng như màn hình, LED, switch, cổng COM.

Với những thiết kế phức tạp Xilinx cung cấp các công cụ phần mềm kiểm tra riêng. ChipScope là một phần mềm cho phép kiểm tra trực tiếp thiết kế bằng cách nhúng thêm vào trong khối thiết kế những khối đặc biệt có khả năng theo dõi giá trị các tín hiệu vào ra hoặc bên trong khi cấu hình được nạp và làm việc. ChipScope sử dụng chính giao thức JTAG để giao tiếp với FPGA. Việc thêm các khối gỡ rối vào trong thiết kế làm tăng kích thước và thời gian tổng hợp thiết kế lên đáng kể. Chi tiết hơn về cách sử dụng Chipscope Pro có thể xem trong tài liệu hướng dẫn của Xilinx.

Chương III

Câu 8. Trình bày thuật toán cộng Carry look ahead adder, so sánh với thuật toán cộng nối tiếp về các tiêu chí tài nguyên và tốc độ.

Trả lời:

Ý tưởng của phương pháp này là sử dụng sơ đồ có khả năng phát huy tối đa các phép toán song song để tính các đại lượng trung gian độc lập với nhau nhằm giảm thời gian đợi khi tính các bit nhớ.

Giả sử các đầu vào là $a(31:0)$, $b(31:0)$ và đầu vào Cin. Khi đó định nghĩa:

$g_i = a_i \text{ and } b_i$ – nhớ phát sinh (generate carry) Nếu a_i, b_i bằng 1 thì g_i bằng 1 khi đó sẽ có bit nhớ sinh ra ở vị trí thứ i của chuỗi.

$p_i = a_i \text{ or } b_i$ – nhớ lan truyền (propagation carry). Nếu hoặc a_i, b_i bằng 1 thì ở vị trí thứ i bit nhớ sẽ được chuyển tiếp sang vị trí $i+1$, nếu cả hai a_i, b_i bằng 0 thì chuỗi nhớ trước sẽ “dừng” lại ở vị trí i .

Các giá trị p, g có thể được tính song song sau một lớp trễ. Từ ý nghĩa của p_i và g_i có thể xây dựng công thức cho chuỗi nhớ như sau, gọi c_i là bit nhớ sinh ra ở vị trí thứ i . Khi đó $c_i = 1$ nếu hoặc g_i bằng 1 nghĩa là có sinh nhớ tại vị trí này, hoặc có một bit nhớ sinh ra tại vị trí thứ $-1 \leq j < i$ $g_j = 1$ (với quy ước $g_{-1} = \text{Cin}$) và bit nhớ này lan truyền qua các bit tương ứng từ $j+1, j+2, \dots, i$. nghĩa là tích $g_j \cdot p_{j+1} \cdot p_{j+2} \cdot \dots \cdot p_i = 1$.

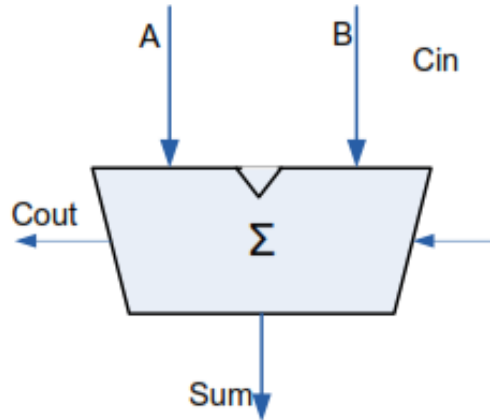
Trên thực tế bộ cộng Carry Look Ahead Adder thường được xây dựng từ các bộ 4 bit CLA, mỗi bộ này có nhiệm vụ tính toán các giá trị trung gian.

so sánh với thuật toán cộng nối tiếp về các tiêu chí tài nguyên và tốc độ: Từ bảng tính độ trễ của một CLA có thể suy ra để tính được c_3 cần 6 lớp trễ logic, tính được c_7 cần 7 lớp trễ logic, c_{11} cần 8 lớp trễ logic, c_{15} cần 9 lớp trễ logic. Nếu so sánh với bộ cộng 16 bit cần $16 \times 2 = 32$ lớp trễ logic thì đó đã là một sự cải thiện đáng kể về tốc độ, bù lại ta sẽ mất nhiều tài nguyên hơn do việc sử dụng để tính các giá trị trung gian trên.

Câu 9. Trình bày thuật toán cộng dùng 1 full_adder, ưu nhược điểm của thuật toán này.

Trả lời:

Khối cộng đơn giản: thực hiện phép cộng giữa hai số được biểu diễn dưới dạng `std_logic_vector` hay `bit_vector`. Các cổng vào gồm hạng tử A, B, bit nhớ Cin, các cổng ra bao gồm tổng Sum, và bit nhớ ra Cout:



Hình 3-1. Sơ đồ khối bộ cộng

Hàm cộng có thể được mô tả trực tiếp bằng toán tử “+” mặc dù với kết quả này thì mạch cộng tổng hợp ra sẽ không đạt được tối ưu về tốc độ cũng như tài nguyên, mô tả VHDL của bộ cộng như sau:

----- Bo cong don gian -----

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_ARITH.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adder32 **is**

port(

Cin : in *std_logic*;

A : in *std_logic_vector*(31 downto 0);

B : in *std_logic_vector*(31 downto 0);

SUM : out *std_logic_vector*(31 downto 0);

Cout: out *std_logic*

);

end adder32;

architecture behavioral **of** adder32 **is**

signal A_temp : *std_logic_vector*(32 downto 0);

signal B_temp : *std_logic_vector*(32 downto 0);

signal Sum_temp : *std_logic_vector*(32 downto 0);

begin

A_temp <= '0' & A;

```

B_temp <= '0' & B;
sum_temp <= a_temp + b_temp + Cin;
SUM <= sum_temp(31 downto 0);
Cout <= sum_temp(32);

```

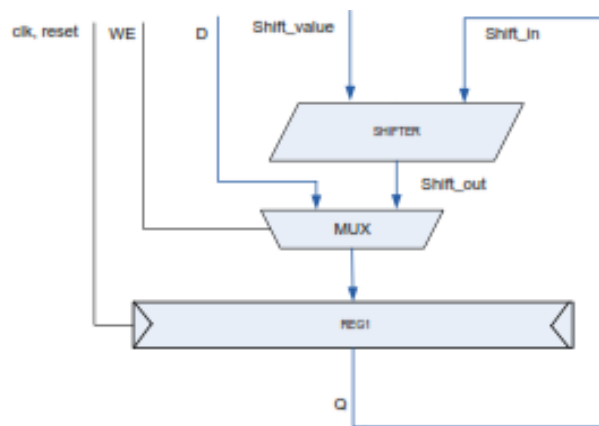
end behavioral;

Kết quả mô phỏng cho thấy giá trị đầu ra Sum và Cout, thay đổi tức thì mỗi khi có sự thay đổi các giá trị đầu vào A, B hoặc Cin.

Câu 10. Trình bày cấu trúc thanh ghi dịch, thuật toán dịch không dùng toán tử dịch, ví dụ ứng dụng thanh ghi dịch.

Trả lời:

Tương tự như trường hợp của khối cộng tích lũy, kết hợp khối dịch và thanh ghi ta được cấu trúc của thanh ghi dịch như ở hình sau:



Hình 3-13. Sơ đồ thanh ghi dịch

Thanh ghi có thể làm việc ở hai chế độ, chế độ thứ nhất dữ liệu đầu vào được lấy từ đầu vào D, chế độ thứ hai là chế độ dịch, khi đó dữ liệu đầu vào của thanh ghi lấy từ khối dịch, đầu ra của thanh ghi được gán bằng đầu vào của khối dịch. Ở chế độ này dữ liệu sẽ bị dịch mỗi xung nhịp một lần.

----- SHIFTER_REG module-----

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-----

entity shift_reg_32 is
port(
    shift_value    : in  std_logic_vector(4 downto 0);
    D              : in  std_logic_vector(31 downto 0);
    Q              : buffer std_logic_vector(31 downto 0);
    CLK           : in  std_logic;

```

```

        WE      : in std_logic;
        RESET   : in std_logic

    );
end shift_reg_32;

-----

architecture structure of shift_reg_32 is
    signal shift_temp : std_logic_vector(31 downto 0);
    signal D_temp     : std_logic_vector(31 downto 0);
    ----COMPONENT SHIFTER----
    component shifter_32 is
    port (
        shift_in   : in std_logic_vector(31 downto 0);
        shift_value : in std_logic_vector(4 downto 0);
        shift_out  : out std_logic_vector(31 downto 0)
    );
    end component;
    ----COMPONENT REG_32----
    component reg_32 is
    port (
        D : in std_logic_vector(31 downto 0);
        Q : out std_logic_vector(31 downto 0);
        CLK : in std_logic; RESET: in std_logic
    );
    end component;

begin
    process (WE, shift_temp)
    begin
        if WE = '1' then
            D_temp <= D;
        else
            D_temp <= shift_temp;
        end if;
    end process;

    sh32: component shifter_32
        port map (Q, shift_value, shift_temp);

    reg32: component reg_32

```

```
port map (D_temp, Q, CLK, RESET);
end structure;
```

Câu 11. Trình bày thuật toán và cấu trúc khối nhân cộng dịch trái cho số nguyên không dấu. Lấy ví dụ.

Trả lời:

Thuật toán: Khối nhân dùng thuật toán cộng dịch (shift-add multiplier), xét phép nhân hai số 4 bit không dấu như sau: $x.a = x_0.a + 2.x_1.a + 2^2.x_2.a + 2^3.x_3.a$ với $x = x_3x_2x_1x_0, b = b_3b_2b_1b_0$

```

0101  - số bị nhân   multiplicand
0111  - số nhân     multiplier
-----
0101  - tích riêng   partial products
0101
0101
0000
-----
0100011 - kết quả nhân product
```

Theo sơ đồ trên thì số bị nhân (multiplicand) sẽ được nhân lần lượt với các bit từ thấp đến cao của số nhân (multiplier), kết quả của phép nhân này bằng số bị nhân “0101” nếu bit nhân tương ứng bằng ‘1’, và bằng “0000” nếu như bit nhân bằng ‘0’, như vậy bản chất là thực hiện hàm AND của bit nhân với 4 bit của số bị nhân.

Để thu được các tích riêng (partial products) ta phải dịch các kết quả nhân lần lượt sang trái với bit nhân thứ 0 là 0 bit, thứ 1 là 1 bit, thứ 2 là hai bit và thứ 3 là 3 bit. Kết quả nhân (product) thu được sau khi thực hiện phép cộng cho 4 tích riêng.

Với sơ đồ cộng dịch trái phép nhân được thực hiện theo công thức sau:

$$x.a = x_0.a + 2.x_1.a + 2^2.x_2.a + 2^3.x_3.a = ((x_3.a.2 + x_2.a).2 + x_1.a).2 + x_0.a$$

a		0	1	0	1
x		0	1	1	1

P(0)		0	0	0	0
2 p(0)	0	0	0	0	0
+x3.a		0	0	0	0

p(1)		0	0	0	0
2P(1)	0	0	0	0	0
+x2.a		0	1	0	1

2p(2)	0	0	0	1	0
P(2)	0	0	0	1	0
+x2.a		0	1	0	1

2p(3)	0	0	0	1	1
P(2)	0	0	0	1	1
+x3.a		0	1	0	1

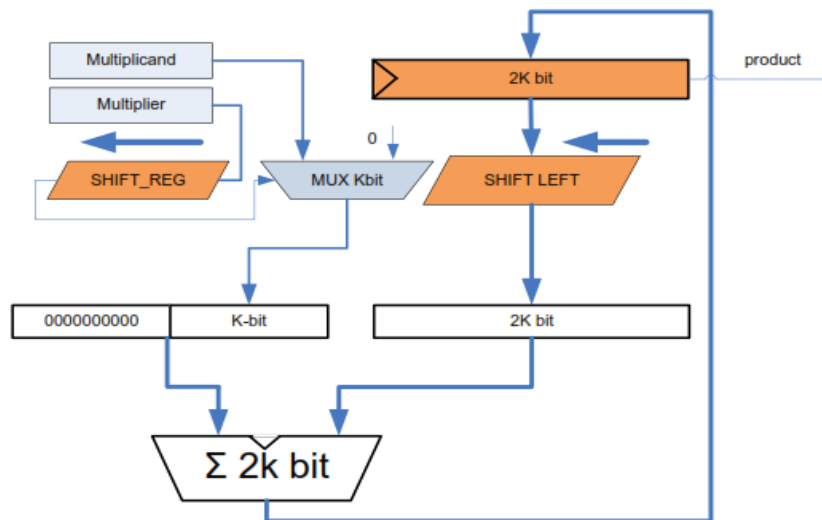
P	0	0	1	0	0

Sơ đồ cộng dịch trái khác sơ đồ cộng dịch phải ở chỗ số nhân được dịch dần sang trái, các tích riêng được tính lần lượt từ trái qua phải, tức là từ bit cao đến bit thấp, kết quả tích lũy khi đó dịch sang trái trước khi cộng với kết quả nhân của bit kế tiếp.

Kết quả tích lũy ban đầu được khởi tạo bằng $p(0) = 0$ và thực hiện dịch sang trái 1 bit trước khi cộng với $x_3.a$ để thu được $p(1)$ là một số 5 bit. $P(1)$ tiếp tục được dịch trái 1 bit và cộng với $x_2.a$ để thu được $p(2)$ là một số 6 bit. Làm tương tự như vậy cho tới cuối ta thu được p là một số 8 bit.

Như vậy cũng giống như sơ đồ cộng dịch phải ở sơ đồ cộng dịch trái sẽ chỉ cần dùng một khối dịch cố định cho kết quả trung gian, điểm khác biệt là phép cộng ở sơ đồ này luôn cộng các bit thấp với nhau, bit nhớ của phép cộng này sẽ ảnh hưởng tới giá trị các bit cao nên buộc phải sử dụng một khối cộng 2K- bit để thực hiện cộng.

Sơ đồ hiện thực thuật toán cộng dịch trái trên phần cứng như sau:



Hình 3-23. Sơ đồ hiện thực hóa thuật toán nhân cộng dịch trái cho hai số K-bit

Đối với sơ đồ dùng thuật toán cộng dịch trái, khối dịch cho số nhân phải là khối dịch phải mỗi xung nhịp một bit do các tích riêng được tính lần lượt từ trái qua phải. Khối cộng sẽ luôn là khối cộng 2-K bit, mặc dù hạng tử thứ nhất (opa) có K bit cao bằng K-bit từ đầu ra MUX, K bit thấp của opa luôn bằng 0. Thanh ghi kết quả trung gian là thanh ghi 2K bit, Giá trị trong thanh ghi được dịch sang trái 1 bít bằng khối dịch trước khi đi vào khối cộng qua cổng opb. Giá trị cộng SUM được lưu trữ lại vào thanh ghi trong xung nhịp kế tiếp

Câu 12. Trình bày thuật toán và cấu trúc khối nhân cộng dịch phải cho số nguyên không dấu, so sánh với khối nhân cộng dịch trái. Lấy ví dụ.

Trả lời:

Thuật toán: Khối nhân dùng thuật toán cộng dịch (shift-add multiplier), xét phép nhân hai số 4 bit không dấu như sau: $x.a = x_0.a + 2.x_1.a + 2^2.x_2.a + 2^3.x_3.a$ với $x = x_3x_2x_1x_0, b = b_3b_2b_1b_0$

```

    0101  - số bị nhân      multiplicand
    0111  - số nhân        multiplier
-----
    0101  - tích riêng     partial products
    0101
    0101
    0000
-----
0100011  - kết quả nhân   product

```

Theo sơ đồ trên thì số bị nhân (multiplicand) sẽ được nhân lần lượt với các bit từ thấp đến cao của số nhân (multiplier), kết quả của phép nhân này bằng số bị nhân “0101” nếu bit nhân tương ứng bằng ‘1’, và bằng “0000” nếu như bit nhân bằng ‘0’, như vậy bản chất là thực hiện hàm AND của bit nhân với 4 bit của số bị nhân.

Để thu được các tích riêng (partial products) ta phải dịch các kết quả nhân lần lượt sang trái với bit nhân thứ 0 là 0 bit, thứ 1 là 1 bit, thứ 2 là hai bit và thứ 3 là 3 bit. Kết quả nhân (product) thu được sau khi thực hiện phép cộng cho 4 tích riêng.

Với sơ đồ công dịch phải phép nhân được thực hiện theo công thức sau:

$$x . a = x_0 . a + 2 . x_1 . a + 2^2 . x_2 . a + 2^3 . x_3 . a = x_0 . a + 2 . (x_1 . a + 2 (x_2 . a + 2 (x_3 . a)))$$

Ví dụ số cho thuật toán cộng dịch phải:

a	0	1	0	1						
x	0	1	1	1						

P(0)	0	0	0	0						
2P(0)	0	0	0	0	0					
+x0.a	0	1	0	1						

2p(1)	0	0	1	0	1					
P(1)	0	0	1	0	1					
+x1.a	0	1	0	1						

2p(2)	0	0	1	1	1	1				
P(2)	0	0	1	1	1	1				
+x2.a	0	1	0	1						

2p(3)	0	1	0	0	0	1	1			
P(3)	0	1	0	0	0	1	1			
+x3.a	0	0	0	0						

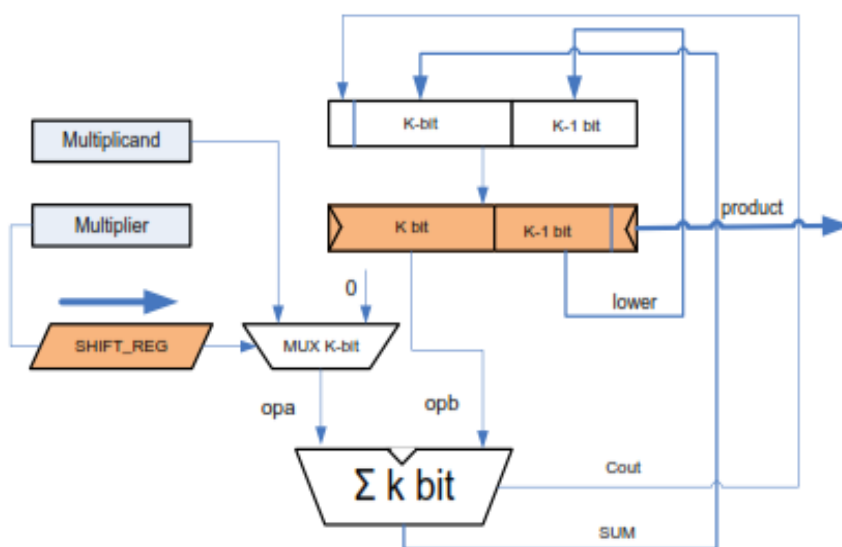
P(4)	0	0	1	0	0	0	1	1		
P	0	0	1	0	0	0	1	1		

Với sơ đồ này thì số nhân được dịch từ trái qua phải, tức là số bị nhân sẽ được nhân lần lượt với các bit từ thấp đến cao x_0, x_1, x_2, x_3 . Các giá trị $p(i)$ là giá trị tích lũy của các tích riêng. Giá trị $p(0)$ được khởi tạo bằng 0. $P(1) = p(0) + x_0.a$, đây là phép cộng 4 bit và $p(1)$ cho ra kết quả 5 bit trong đó bit thứ 5 là bit nhớ, riêng trường hợp $p(1)$ thì bit nhớ này chắc chắn bằng 0 do $p(0) = 0$.

Kết quả $p(2)$ có 6 bit sẽ bằng kết quả phép cộng của $x_1.a$ đã dịch sang phải 1 bit và cộng với giá trị $p(1)$. Nhận xét rằng bit cuối cùng của $x_1.a$ khi dịch sang trái luôn bằng 0 do vậy hay vì phải dịch $x_1.a$ sang phải 1 bit ta xem như $p(1)$ đã bị dịch sang trái 1 bit, nghĩa là phải lấy 4 bit cao của $p(1)$ cộng với $x_1.a$. Kết quả thu được của phép cộng này là một số 5 bit đem hợp với bit cuối cùng của $p(1)$ sẽ thu được $p(2)$ là một số 6 bit.

Tiếp tục như vậy thay vì dịch $x_2.a$ ta lại xem như $p(2)$ dịch sang trái 1 bit và cộng 4 bit cao của $p(2)$ với $x_2.a$, kết quả phép cộng hợp với 2 bit sau của $p(2)$ thu được $p(3)$ là một số 7 bit. Làm như vậy với tích riêng cuối cùng thu được kết quả (product) là một số 8 bit.

Như vậy trên sơ đồ trên ta luôn cộng 4 bit cao của kết quả tích lũy với kết quả nhân.



Hình 3-22. Sơ đồ hiện thực hóa thuật toán nhân cộng dịch phải cho hai số K-bit

Khối cộng có một hạng từ cố định K-bit là đầu vào tích riêng (opb), để tính các tích riêng sử dụng một khối chọn kênh MUX k-bit, khối này chọn giữa giá trị số bị nhân (multiplicand) và 0 phụ thuộc vào bit tương ứng của

số nhân (multiplier) là 1 hay 0. Để đưa lần lượt các bit của số nhân vào cổng điều khiển cho khối chọn này thì giá trị của số nhân được lưu trong một thanh ghi dịch sang phải mỗi xung nhịp 1 bit.

Đầu vào thứ hai của bộ cộng lấy từ k bit cao của thanh ghi 2k-bit. Thanh ghi này trong quá trình làm việc lưu trữ kết quả tích lũy của các tích riêng. Đầu vào của thanh ghi này bao gồm K+1 bit cao, ghép bởi bit nhớ (Cout) và K-bit (Sum) từ bộ cộng, còn K-1 bit thấp (lower) lấy từ bit thứ K-1 đến 1 lưu trong thanh ghi ở xung nhịp trước ở xung nhịp trước đó, nói một cách khác, đây là thanh ghi có phần K-1 bit thấp hoạt động trong chế độ dịch sang phải 1 bit, còn K+1 bit cao làm việc ở chế độ nạp song song.

Phép nhân được thực hiện sau K xung nhịp, kết quả nhân lưu trong thanh ghi cộng dịch 2k-bit.

Câu 13. Trình bày thuật toán và cấu trúc khối nhân số có dấu dùng mã hóa BOOTH cơ số 2. Lấy ví dụ.

Trả lời:

$$x_{n-1}x_{n-2} \dots x_1x_0 = -2^{n-1}x_{n-1} + 2^{n-2}x_n - 2 + \dots + 2x_1 + x_0 \quad (3.7)$$

Ở đây bước đầu ta sẽ xét mã hóa Booth ở cơ số 2 (Radix 2 Booth encoding). Để hiểu về mã hóa Booth quay trở lại với công thức (3.7) tính giá trị cho số biểu diễn dạng bù 2. Công thức này có thể được biến đổi như sau:

$$\begin{aligned} x_{n-1}x_{n-2} \dots x_1x_0 &= -2^{n-1}x_{n-1} + 2^{n-2}x_{n-2} + \dots + 2x_1 + x_0 \\ &= -2^{n-1}x_{n-1} + 2^{n-1}x_{n-2} - 2^{n-2}x_{n-2} + \dots + 2^2x_1 - 2x_1 + 2x_0 - x_0 + 0 \\ &= 2^{n-1}(-x_{n-1} + x_{n-2}) + 2^{n-2}(-x_{n-2} + x_{n-3}) + \dots + 2(-x_1 + x_0) + (-x_0 + 0) \quad (3.8) \end{aligned}$$

Từ đó xây dựng bảng mã như sau, cặp hai bit liên tiếp x_{i+1}, x_i sẽ được thay bằng giá trị.

$b_i = (-x_{i+1} + x_i)$ với $i = -1, n-2$, và $x_{-1} = 0$. Công thức trên được viết thành:

$$x_{n-1}x_{n-2} \dots x_1x_0 = -2^{n-1}x_{n-1} + 2^{n-2}x_{n-2} + \dots + 2x_1 + x_0 = 2^{n-1}b_{n-1} + 2^{n-2}b_{n-2} + \dots + 2b_1 + b_0 \quad (3.9)$$

Công thức này có dạng tương tự công thức cho số nguyên không dấu. Điểm khác biệt duy nhất là b_i được mã hóa theo bảng sau: **Mã hóa Booth cơ số 2**

x_{i+1}	x_i	Radix-2 booth encoding b_i
0	0	0
0	1	1
1	0	-1
1	1	0

Để thực hiện phép nhân số có dấu đầu tiên sẽ mã hóa số nhân dưới dạng mã Booth, khi thực hiện phép nhân nếu bit nhân là 0, 1 ta vẫn làm bình thường, nếu bit nhân là -1 thì kết quả nhân bằng số bù hai của số bị nhân. Có thể áp dụng mã hóa Booth cho cả sơ đồ cộng dịch trái và cộng dịch phải và hỗ trợ thao tác mở rộng có dấu.

2's complement (0) 1 1 1 0 0 1 0 1 1 0 1 0 0 1(0)

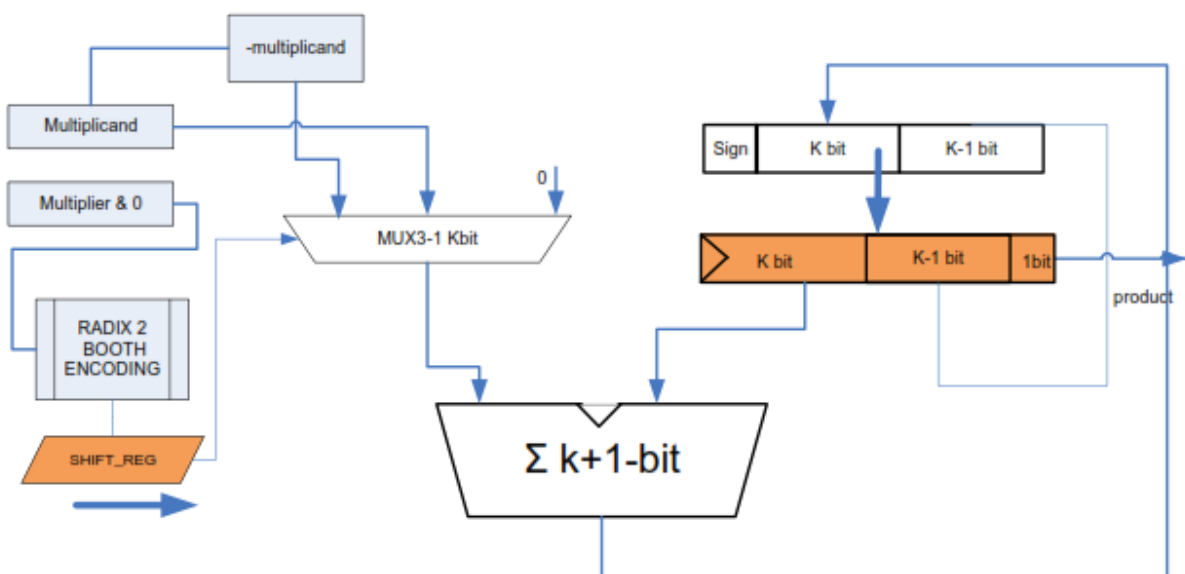
Radix-2 Booth (1) 0 0-1 0 1-1 1 0-1 1-1 0 1-1

Ví dụ đầy đủ về phép nhân có dấu sử dụng mã hóa Booth cơ số 2 như sau:

a	1 1 0 1	Bù 2 a = 0 0 1 1 (a = -3)
x	0 1 1 1	x = + 7
b	1 0 0-1	Radix 2 booth encoding of x
<hr/>		
P(0)	0 0 0 0	
+b0.a	0 0 1 1	
<hr/>		
2p(1)	0 0 0 1 1	mở rộng với bit dấu 0
P(1)	0 0 0 1 1	
+b1.a	0 0 0 0	
<hr/>		
2p(2)	0 0 0 0 1 1	mở rộng với bit dấu 0
P(2)	0 0 0 0 1 1	
+x2.a	0 0 0 0	
<hr/>		
2p(2)	0 0 0 0 0 1 1	mở rộng với bit dấu 0
P(3)	0 0 0 0 0 1 1	
+x3.a	1 1 0 1	
<hr/>		
2p(3)	1 1 0 1 0 1 1	mở rộng với bit dấu 1
P	1 1 1 0 1 0 1 1	= -21

Sơ đồ trên có thể sửa đổi một chút để nó vẫn đúng với cả phép nhân với số không dấu, khi đó ta bổ xung thêm bit 0 vào bên trái của số bị nhân và chuỗi Booth sẽ dài hơn một bit.

Sơ đồ của khối nhân có dấu dùng mã hóa Booth cơ số 2:



Hình 3-25. Sơ đồ khối nhân dùng thuật toán nhân dùng mã hóa Booth cơ số 2

Số nhân được chèn thêm một bit 0 vào tận cùng bên phải và được đưa dần vào khối mã hóa BOOTH, khối này sẽ đưa ra kết quả mã hóa và đẩy kết quả này ra khỏi chọn kênh MUX3_1, khối này chọn giữa các giá trị multiplicand, - multiplicand và 0 tương ứng phép nhân với 1, -1, 0 và đưa vào khối cộng. Hạng tử thứ hai của khối cộng lấy từ K bit cao thành ghi dịch $2 \cdot K$ bit. Thanh ghi này hoạt động không khác gì thanh ghi trong sơ đồ cộng dịch phải, điểm khác cũng cần lưu ý là các bit mở rộng trong các kết quả tích lũy không phải là bit nhớ như trong trường hợp số có dấu mà phải là bit dấu, nói một cách khác phép dịch sang phải ở đây là phép dịch số học chứ không phải phép dịch logic.

Câu 14. Trình bày thuật toán và cấu trúc khối nhân số có dấu dùng mã hóa BOOTH cơ số 4, so sánh với các thuật toán nhân thông thường. Lấy ví dụ.

trả lời:

Khối nhân theo cơ số 4 (Radix-4 multiplier) sử dụng sơ đồ nhân với cặp 2 bit, với cặp 2 bit thì có thể có 4 giá trị 0, 1, 2, 3. Dễ dàng nhận thấy các phép nhân một số với 0 bằng 0 với 1 bằng chính nó, nhân với 2 là phép dịch sang phải 1 bit, tuy vậy nhân với 3 là một phép toán mà thực hiện không dễ dàng nếu so với phép nhân với 0, 1, 2 do phải dùng tới bộ cộng. Tuy vậy khối nhân vẫn có thể tăng tốc theo sơ đồ này bằng cách tính trước số bị nhân với 3.

Nhằm tránh việc nhân với 3, sơ đồ nhân theo cơ số 4 trên được cải tiến bằng mã hóa Booth (radix-4 Booth encoding), mã hóa này giúp việc tính các tích riêng trở nên dễ dàng hơn, cụ thể công thức (3.7) biểu diễn giá trị của số bù 2 có thể biến đổi về dạng sau:

$$\begin{aligned} x_{2n-1}x_{2n-2}\dots x_1x_0 &= -2^{2n-1}x_{2n-1} + 2^{2n-2}x_{2n-2} + \dots + 2x_1 + x_0 \\ &= -2^{2n-2}2x_{2n-1} + 2^{2n-2}x_{2n-2} - 2^{2n-2}x_{2n-3} + \dots - 2x_1 + 2x_0 + 2.0 \\ &= 2^{2n-2}(-2x_{2n-1} + x_{2n-2} + x_{2n-3}) + 2^{2n-4}(-2x_{2n-3} + x_{2n-4} + x_{2n-5}) + \dots + (-2x_1 + x_0 + 0) \quad (3.10) \end{aligned}$$

Trong trường hợp tổng số bit biểu diễn không phải là chẵn ($2n$) thì để thu được biểu diễn như trên rất đơn giản là bổ xung thêm một bit dấu tận cùng bên trái do việc bổ xung thêm bit dấu không làm thay đổi giá trị của số biểu diễn.

Như vậy nếu ta xây dựng bảng mã hóa theo công thức $b_i = (-2x_{2i+1} + x_{2i} + x_{2i-1})$ với $i = 0, 1, 2, \dots, n-1$ (3.11)

ta sẽ mã hóa $2n$ bit của số nhân bằng $[n]$ giá trị theo bảng mã sau:

Bảng mã hóa Booth cơ số 4

x_{i+1}	x_i	x_{i-1}	Radix-4 Booth encoding
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

Ví dụ đầy đủ về phép nhân có dấu sử dụng mã hóa Booth cơ số 4 như sau:

```

-----
a      1 1 0 1 0 1      Bù 2 a = 0 0 0 1 0 1 1 (a = -
11)
x      0 1 1 0 1 0 (0)      x = + 26
b      2  -1  -2      Radix 4 booth encoding of x
-----
4P(0) 0 0 0 0 0 0 0
P(0)   0 0 0 0 0 0 0
+b0.a  0 0 1 0 1 1 0
-----

4p(1) 0 0 0 0 1 0 1 1 0      mở rộng với bit dấu 0 0
P(1)   0 0 0 0 1 0 1 1 0
+b1.a  0 0 0 1 0 1 1
-----

4p(2) 0 0 0 0 1 0 0 0 0 1 0      mở rộng với bit dấu 0 0
P(2)   0 0 0 0 1 0 0 0 0 1 0
+b2.a  1 1 0 1 0 1 0
-----

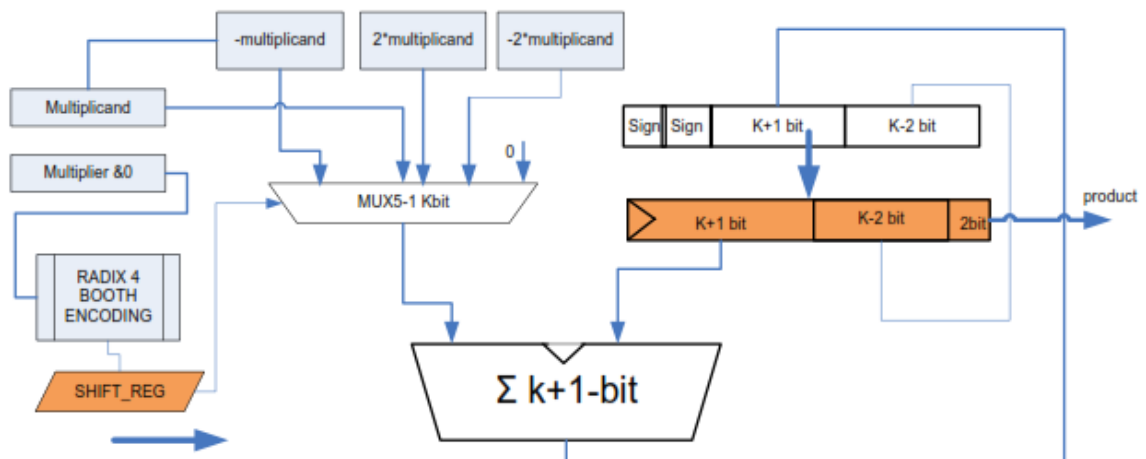
4p(3) 1 1 1 1 0 1 1 1 0 0 0 1 0      mở rộng với bit dấu 1 1
P      1 1 1 1 0 1 1 1 0 0 0 1 0 = -286

```

Sơ đồ hiện thực trên phần cứng của thuật toán nhân dùng mã hóa Booth cơ số 4 giống hệt sơ đồ của phép nhân số có dấu dùng mã hóa Booth cơ số 2. Đầu ra của khối mã hóa cơ số 4 là các chuỗi 3 bit được dịch từ giá trị của Multiplier sau khi thêm 1 bit 0 vào bên phải, mỗi lần dịch hai bit. Từ 3 bit này ta sẽ chọn một trong 5 giá trị gồm 0, multiplicand, - multiplicand, -2 multiplicand và +2 multiplicand bởi khối chọn kênh MUX5_1, đầu ra của khối này được gửi vào khối cộng K+1 bit. Lý do khối cộng ở đây là khối cộng K+1 bit là do giá trị

2.multiplicand phải biểu diễn bằng K+1 bit. Phần còn lại không khác gì sơ đồ phép nhân dùng thuật toán cộng dịch phải. Điểm khác là thanh ghi chứa kết quả sẽ dịch sang phải không phải 1 bit mà dịch 2 bit đồng thời và phép dịch ở đây là phép dịch số học, nghĩa là 2 vị trí bit bị trống đi sẽ điền giá trị dấu hiện tại của số bị dịch. Bộ cộng ở đây buộc phải dùng bộ cộng K+1 bit vì đầu vào có thể nhận giá trị 2.multiplicand biểu diễn dưới dạng K+1 bit.

Kết quả cuối cùng cũng chứa trong thanh ghi tích lũy 2K+1 bit tuy vậy chúng ta chỉ lấy 2K bit thấp của thanh ghi này là đủ, lý do là hai bit có trọng số cao nhất của thanh ghi này giống nhau nên việc lược bỏ bớt 1 bit tận cùng bên trái đi không làm thay đổi giá trị của chuỗi số biểu diễn.



Hình 3-26. Sơ đồ khối nhân dùng thuật toán nhân dùng mã hóa Booth cơ số 4

Câu 15. Trình bày thuật toán và cấu trúc khối chia số nguyên không dấu có phức hồi phần dư.

Lấy ví dụ

trả lời:

Ký hiệu

z : số bị chia (dividend)

d : số chia (divisor)

q : thương số (quotient)

s : số dư (remainder)

Khối chia trên phần cứng được xây dựng trên cơ sở nguyên lý như trên, điểm khác biệt là phép trừ được thay tương đương bằng phép cộng với số bù hai, và cần thực hiện phép trừ trước để xác định giá trị của q_4, q_3, q_2, q_1, q_0 .

Sơ đồ sau được gọi là sơ đồ chia có khôi phục phần dư vì sau mỗi phép trừ nếu kết quả là âm thì phần dư sẽ được khôi phục lại thành giá trị cũ và dịch thêm một bit trước khi thực hiện trừ tiếp.

Ví dụ một phép chia thực hiện theo sơ đồ đó như sau:

z	1 1 1 0 0 1 0 1	$z = 197$
$2^4 d$	1 1 1 0	$d = 14$

$s(0)$	0 1 1 0 0 1 0 1	
$2s(0)$	0 1 1 0 0 0 1 0 1	
$+(-2^4d)$	1 0 0 1 0	

$s(0)$	(0) 1 1 1 1 0 1 0 1	kết quả âm $q_4 = 0$
$2s(0)$	1 1 0 0 0 1 0 1	phục hồi.
$+(-2^4d)$	1 0 0 1 0	

$s(1)$	(0) 1 0 1 0 1 0 1	kết quả dương $q_3 = 1$
$2s(1)$	1 0 1 0 1 0 1	
$+(-2^4d)$	1 0 0 1 0	

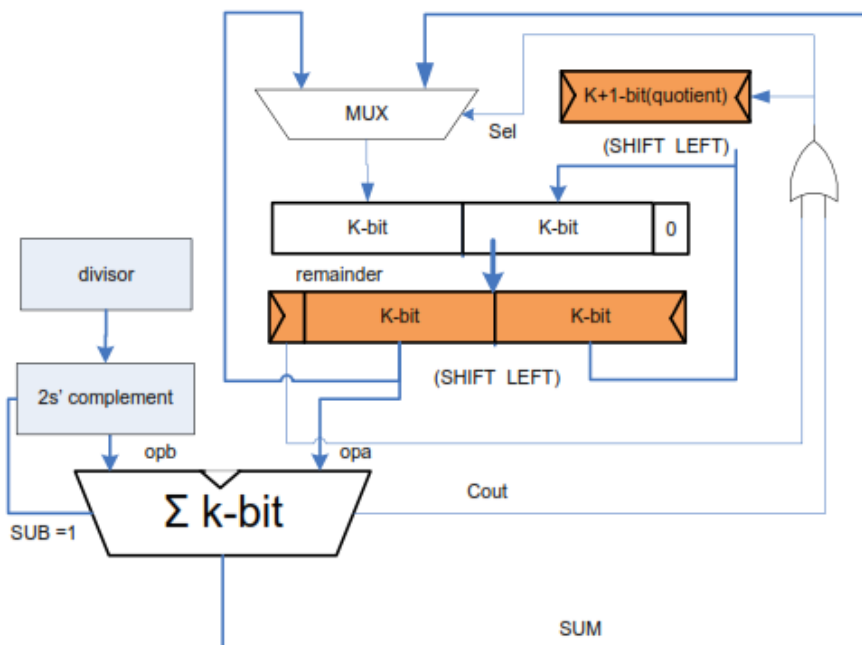
$s(2)$	(0) 0 1 1 1 0 1	kết quả dương $q_2 = 1$
$2s(2)$	0 1 1 1 0 1	
$+(-2^4d)$	1 0 0 1 0	

$s(3)$	(0) 0 0 0 0 1	kết quả dương $q_1 = 1$
$2s(3)$	0 0 0 0 1	
$+(-2^4d)$	1 0 0 1 0	

$s(4) =$	(0) 0 0 1 1 = 1	kết quả âm $q_0 = 0$
$2s(4) =$	0 0 0 1	trả về giá trị dư cũ
$s = 2s(4) =$	0 0 0 1 = 1	$q = 0 1 1 1 0 = 14$

(*) Trong ví dụ trên số trong ngoặc là bit nhớ của phép cộng các số 4 bit bên tay phải chứ không phải là kết quả cộng của cột các bit có trọng số cao nhất.

Sơ đồ phân chi tiết phần cứng được thiết kế như sau:



Hình 3-27. Sơ đồ khối chia có khôi phục phần dư

Gọi k là số bit của số chia và thương số. Thanh ghi dịch Remainder $2k+1$ -bit trong sơ đồ trên được sử dụng để lưu trữ giá trị dư trung gian. Thanh ghi này được khởi tạo giá trị bằng giá trị của số bị chia hợp với một bit 0 ở tận cùng bên trái và mỗi xung nhịp được dịch sang bên trái một bit. Bit nhớ đặc biệt thứ $2k+1$ luôn lưu trữ bit có trọng số cao nhất của phần dư.

Bộ cộng K -bit được sử dụng để thực hiện phép trừ K bit bằng cách cộng phần K bit cao của thanh ghi số dư kể từ bit thứ 2 từ bên trái với bù hai của số chia. Vì d là số không dấu nên khi đổi sang số bù hai phải dùng $K+1$ bit để biểu diễn. Cũng vì d là số nguyên không âm nên khi đổi sang số bù 2 bit cao nhất thu được luôn bằng 1. (ví dụ $d = 1011$, bù 1 $d = 10100$, bù 2 $d = 10101$). Lợi dụng tính chất đó việc cộng K bit cao của số bị chia với số bù 2 của d

đáng lẽ phải sử dụng bộ cộng K+1 bit nhưng thực tế chỉ cần dùng bộ cộng K bit. Bit đầu của kết quả không cần tính mà có thể thu được thông qua giá trị của bit nhớ Cout và giá trị bit có trọng số cao nhất của phần dư trong thanh ghi. Nếu giá trị bit có trọng số cao nhất của phần dư này bằng 1 thì dĩ nhiên phần dư lớn hơn số chia do thực tế ta đang trừ một số 5 bit thực sự cho 1 số 4 bit, còn nếu bit này bằng 0, nhưng phép trừ đưa ra bit nhớ Cout = 1, tương ứng kết quả là số dương thì phần dư cũng lớn hơn hoặc bằng số chia. Trong cả hai trường hợp này $q_i = 1$, trong các trường hợp khác $q_i = 0$, đó là lý do tại sao trước thanh ghi quotient đặt một cổng logic OR hai đầu vào.

Thanh ghi dịch quotient k+1-bit được sử dụng lưu lần lượt các bit q_k, q_{k-1}, \dots, q_0 của thương số q, thanh ghi này cũng được dịch sang phải mỗi lần một bit. Khi hiện thực sơ đồ này trên VHDL có thể bỏ qua, thay vào đó K+1 bit thương số có thể được đẩy vào K+1 bit thấp của thanh ghi phần dư 2K+1 bit mà không ảnh hưởng tới chức năng làm việc của cả khối.

Câu 16. Trình bày thuật toán và cấu trúc khối chia số không dấu không phục hồi phần dư. Lấy ví dụ.

Trả lời:

Khi phân tích về sơ đồ thuật toán chia có phục hồi số dư, có một nhận xét là không nhất thiết phải thực hiện thao tác phục hồi giá trị phần dư, và trên cơ sở đó có thể xây dựng sơ đồ khối chia không khôi phục hồi phần dư.

Về mặt toán học, giả sử giá trị tại thanh ghi chứa phần dư là s, ở bước tiếp theo ta sẽ thực hiện $s - 24d$, phép trừ này cho ra kết quả âm, tức là kết quả không mong muốn, nếu như với sơ đồ khôi phục phần dư ở bước tiếp theo ta sẽ trả lại giá trị u, dịch sang phải 1 bit và thực hiện phép trừ $2.s - 24d$. Tuy vậy nếu lưu ý rằng $2.s - 24d = 2(s - 24d) + 24d$. Ta có thể thay đổi sơ đồ đi một chút mà chức năng của mạch vẫn không thay đổi, ở bước trên ta vẫn lưu giá trị sai vào thanh ghi, ở bước sau ta sẽ vẫn dịch giá trị trong thanh ghi sang phải 1 bit nhưng thay vì cộng với số bù 2 của 24d ta sẽ cộng trực tiếp với 24d, việc đó đảm bảo kết quả của bước tiếp theo vẫn đúng.

Quy luật chung là:

Nếu giá trị trong thanh ghi là âm \rightarrow thực hiện cộng với 24d. Nếu giá trị trong thanh ghi là âm \rightarrow thực hiện trừ với 24d.

Ví dụ một phép chia theo sơ đồ không phục hồi phần dư như sau:

z	1 0 0 0 0 1 0 1	z = 133
2^4d	1 1 1 0	d = +14 = 01110
		-d = -14 = 10010
s(0)	0 0 0 1 0 0 0 0 1 0 1	
2s(0)	0 0 1 0 0 0 0 1 0 1	
$+(-2^4d)$	1 1 0 0 1 0	
s(1)	(0) 1 1 0 1 0 1 0 1	kết quả âm q4 = 0
2s(1)	1 1 0 1 0 0 1 0 1	
$+(+2^4d)$	0 0 1 1 1 0	
s(2)	(1) 0 0 0 1 0 1 0 1	kết quả dương q3 = 1
2s(2)	0 0 0 1 0 1 0 1	
$+(-2^4d)$	1 1 0 0 1 0	
s(3)	(0) 1 0 1 1 1 0 1	kết quả âm q2 = 0
2s(3)	1 0 1 1 1 0 1	
$+(+2^4d)$	0 0 1 1 1 0	
s(4)	(0) 1 1 1 0 0 1	kết quả âm q1 = 0
2s(4)	1 1 1 0 0 1	
$+(+2^4d)$	0 0 1 1 1 0	
s(5)	(1) 0 0 1 1 1	kết quả dương q0 = 1
s = 2s(5)	= 0 1 1 1 = 7	q = 0 1 0 0 1 = 9

- bit nằm trong dấu ngoặc là bit nhớ của khối cộng 4 bit.

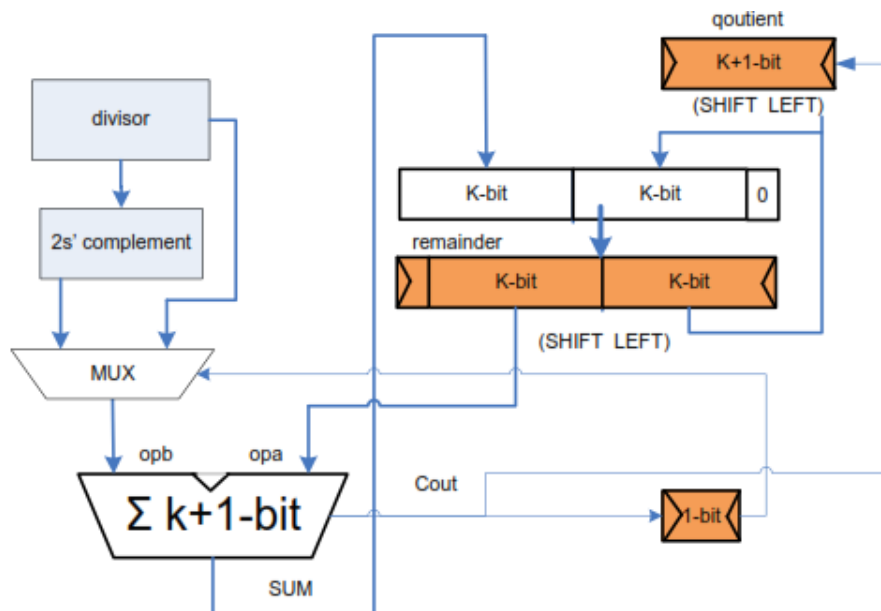
- Các bit tận cùng bên trái dấu | là bit đầu của các toán hạng, liệt kê chỉ để làm rõ giá trị cộng, các bit này luôn là 1,0 hoặc 0,1.

Với quy luật như trên có thể dễ dàng suy ra rằng hai toán tử của phép cộng ở trên luôn ngược dấu, do vậy bit đầu của các hạng tử này luôn trái ngược nhau. Lợi dụng đặc điểm đó ta có thể lược bớt giá trị bit đầu mà vẫn thu được đúng dấu hiệu lớn hơn hay nhỏ hơn 0 của kết quả cộng. Nếu phép cộng có nhớ ($Cout = 1$) thì kết quả là dương còn phép cộng không có nhớ ($Cout = 0$) kết quả là âm. Cũng vì thế mặc dù giá trị phần dư biểu diễn là số có dấu nhưng khi dịch sang trái để thu được 2.s thì phải sử dụng thêm một bit, bit này thực chất ẩn đi nhưng vẫn đảm bảo xác định được dấu của kết quả theo quy luật trên.

Khối chia sử dụng sơ đồ khôi phục phần dư để hiểu, tuy vậy nếu nghiên cứu kỹ về độ trễ logic ta sẽ thấy tín đường dữ liệu dài nhất (trong 1 xung nhịp đồng bộ) phải trải qua ba thao tác: thao tác dịch ở thanh ghi, thao tác cộng ở bộ cộng, và thao tác lưu giá trị vào thanh ghi Quotient và thanh ghi Remainder. Thao tác cuối được thực hiện sau khi bit nhớ Cout của chuỗi cộng xác định.

Nếu chấp nhận luôn lưu trữ giá trị kết quả phép trừ vào thanh ghi, không quan tâm dù đó là giá trị đúng hay sai, bỏ qua thao tác phục hồi giá trị phần dư thì đồng thời sẽ bỏ sự lệ thuộc vào Cout và tốc độ của bộ chia có thể được cải thiện thêm một lớp trễ. Về mặt tài nguyên thì không có sự thay đổi nhiều vì thay cho thao tác phục hồi mà thực chất là khối chọn kênh K bit thì phải thực hiện chọn hạng tử cho khối cộng giữa d và bù 2 của d cũng là một khối chọn kênh K bit.

Sơ đồ phân chi tiết phân cứng được thiết kế như sau:



Hình 3-28. Sơ đồ khối chia không phục hồi phần dư

Sơ đồ về cơ bản giống như sơ đồ khối chia thực hiện bằng thuật toán không phục hồi phần dư, điểm khác biệt duy nhất là bộ chọn kênh được chuyển xuống vị trí trước bộ cộng để chọn giá trị đầu vào tương ứng cho khối này, việc chọn giá trị cộng cũng như thực hiện phép toán cộng hay trừ trong bộ cộng được quyết định bởi bit Cout được làm trễ 1 xung nhịp, nghĩa là Cout của lần cộng trước.

Câu 16. Trình bày thuật toán và cấu trúc khối chia số nguyên có dấu. Lấy ví dụ.

Trả lời:

Hai sơ đồ trên chỉ đúng cho phép chia đối với số không dấu, để thực hiện phép chia đối với số có dấu cũng giống như trường hợp của khối nhân, phép chia được thực hiện bằng cách chuyển toàn bộ số chia và số bị chia thành hai phần giá trị tuyệt đối và dấu, phần giá trị tuyệt đối có thể tiến hành chia theo một trong hai sơ đồ trên, phần dấu được tính toán đơn giản bằng một cổng XOR, kết quả thu được lại chuyển ngược lại thành biểu diễn dưới dạng số bù 2.

Dưới đây ta sẽ tìm hiểu một thuật toán khác nhanh hơn và tiết kiệm hơn để thực hiện phép chia có dấu. Ý tưởng của thuật toán này xuất phát từ sơ đồ chia không phục hồi phần dư.

Mục đích của phép chia là đi tìm biểu diễn sau: $z = q_{k-1} \cdot 2^{k-1} \cdot d + q_{k-2} \cdot 2^{k-2} \cdot d + \dots + q_0 \cdot d + s$.

Trong đó z là số bị chia, d là số chia, s là phần dư còn thương số chính là giá trị nhị phân của $q_{k-1}q_{k-2}\dots q_0$.
 Qì nhận giá trị 0 hay 1 tương ứng ở bước thứ i phần dư được cộng thêm hoặc trừ đi tương ứng giá trị d tương ứng với giá trị phần dư là dương hay âm.

Một cách tổng quát cho phép chia: mục đích của các sơ đồ trên là làm giảm dần trị tuyệt đối của phần dư bằng cách trừ nếu nó cùng dấu với số chia và cộng nếu nó khác dấu với số chia, quá trình này kết thúc khi cả hai điều kiện sau được thỏa mãn:

1. Phần dư s cùng dấu với z
2. Trị tuyệt đối của s nhỏ hơn trị tuyệt đối của d .

Cũng tổng quát hóa từ sơ đồ chia không phục hồi phần dư, nếu ta mã hóa q_i khác đi như sau:

$p_i = 1$ nếu $s(i)$ và d cùng dấu

$p_i = -1$ nếu $s(i)$ và d khác dấu.

Khi đó đẳng thức biểu diễn ở trên $z = p_{k-1} \cdot 2^{k-1} \cdot d + p_{k-2} \cdot 2^{k-2} \cdot d + \dots + p_0 \cdot d + s$, vẫn đúng, chỉ khác ở tập giá trị của q_i là $\{-1, 1\}$. Trong trường hợp triển khai z như trên nhưng cuối cùng thu được s khác dấu với z thì phải tiến hành điều chỉnh s bằng cách cộng thêm hay trừ đi d , đồng thời cộng thêm hoặc bớt đi một vào giá trị của q .

p có thể thu được thông qua sơ đồ chia không phục hồi phần dư, vấn đề còn lại là phải chuyển q về dạng biểu diễn thông thường của số bù 2. Có thể chứng minh rằng nếu ta làm lần lượt các bước sau thì có thể đưa $q = q_{k-1}q_{k-2}\dots q_0$ về dạng biểu diễn nhị phân $p = p_{k-1}p_{k-2}\dots p_0$ với $p_i \in \{0, 1\}$ và giá trị $p = q$.

- Chuyển tất cả các p_i giá trị -1 thành 0 . Gọi giá trị này là $r = r_{k-1}r_{k-2}\dots r_0$. Có thể dễ dàng chứng minh $q_i = 2r_i - 1$.

- Lấy đảo của r_{k-1} , thêm 1 vào cuối r , giá trị thu được dưới dạng bù 2 chính là thương số $q = (k-1r_{k-2}\dots r_01)2$'s complement

Có thể chứng minh như sau: $q = (k-1r_{k-2}\dots r_01)2$'s complement

$$\begin{aligned} &= -(1-p_{k-1}) \cdot 2^k + 1 + \sum_{i=0}^{k-2} r_i \cdot 2^{i+1} \\ &= -(2^k - 1) + 2 \cdot \sum_{i=0}^{k-1} r_i \cdot 2^i \\ &= \sum_{i=0}^{k-1} (2r_i - 1) \cdot 2^i = p \end{aligned}$$

Ví dụ một phép chia hai số có dấu, kết quả cuối cùng không phải điều chỉnh:

z	1 1 0 1 0 1 0 1	z = -43
2 ⁴ d	0 1 1 0	d = 6 = 0 1 1 0
		-d = -6 = 1 0 1 0
s(0)	1 1 1 0 1 0 1 0 1	
2s(0)	1 1 1 0 1 0 1 0 1	d, s khác dấu q4 = -1
+(2 ⁴ d)	0 0 1 1 0	thực hiện cộng
s(1)	(1) 0 0 1 1 0 1 0 1	
2s(1)	0 0 1 1 0 1 0 1	d, s cùng dấu q3 = +1
+(-2 ⁴ d)	1 1 0 1 0	thực hiện trừ
s(2)	(1) 0 0 0 0 1 0 1	
2s(2)	0 0 0 0 1 0 1	d, s cùng dấu q2 = +1
+(+2 ⁴ d)	1 1 0 1 0	thực hiện trừ
s(3)	(0) 1 0 1 1 0 1	
2s(3)	1 0 1 1 0 1	d, s khác dấu q1 = -1
+(+2 ⁴ d)	0 0 1 1 0	thực hiện cộng
S(4) =	(0) 1 1 0 0 1	
2S(4)	1 1 0 0 1	s, q khác dấu q0 = -1
+(-2 ⁴ d)	0 0 1 1 0	thực hiện cộng
S =	1 1 1 1 1 = -1	s, z cùng dấu không cần

điều chỉnh số dư:

$$p = -1 + 1 + 1 - 1 - 1 \quad r = 0 \ 1 \ 1 \ 0 \ 0$$

$$q = (1 \ 1 \ 1 \ 0 \ 0 \ 1)2\text{'s complement} = -7$$

- bit nằm trong dấu ngoặc là bit nhớ của khối cộng 4 bit.

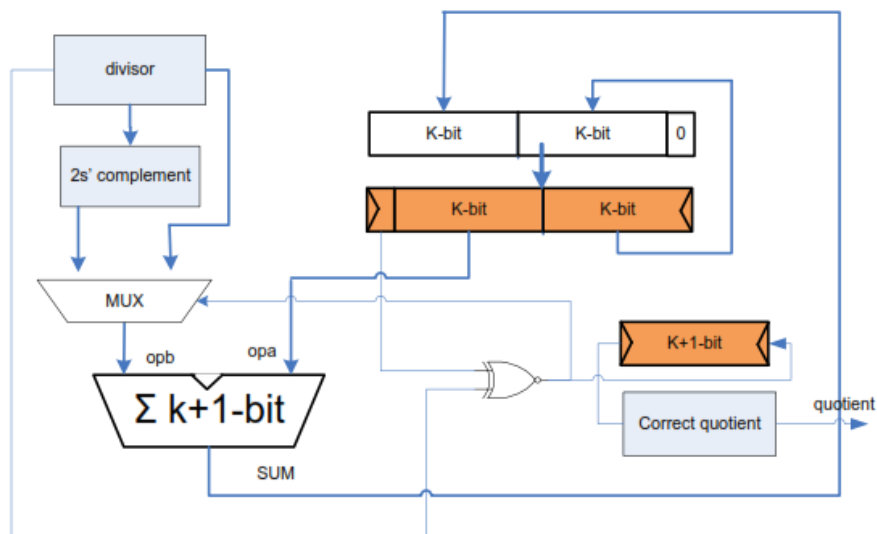
- Các bit tận cùng bên trái là bit dấu của các toán hạng, liệt kê chỉ để làm rõ giá trị cộng, các bit này luôn là 1,0 hoặc 0,1. $-43 = +6((-1)2^4 + (+1)2^3 + (+1)2^2 + (-1)2^1 + (-1)2^0) + (-1)$

Ví dụ sau thể hiện trường hợp cần phải điều chỉnh thương số và số dư:

z	1 1 0 1 1 0 0 1	z = -39
2 ⁴ d	0 1 1 0	d = 6 = 0 1 1 0
		-d = -6 = 1 0 1 0

s(0)	1 1 1 0 1 1 0 0 1		
2s(0)	1 1 1 0 1 1 0 0 1	d,s khác dấu	q4 = -1
+(2 ⁴ d)	0 0 1 1 0	thực hiện cộng	
<hr/>			
s(1)	(1) 0 0 1 1 1 0 0 1	d,s cùng dấu	q3 = +1
2s(1)	0 0 1 1 1 0 0 1	thực hiện trừ	
+(-2 ⁴ d)	1 1 0 1 0		
<hr/>			
s(2)	(1) 0 0 0 1 0 0 1	d,s cùng dấu	q2 = +1
2s(2)	0 0 0 1 0 0 1	thực hiện trừ	
+(+2 ⁴ d)	1 1 0 1 0		
<hr/>			
s(3)	(0) 1 1 0 0 0 1	d,s khác dấu	q1 = -1
2s(3)	1 1 0 0 0 1	thực hiện cộng	
+(+2 ⁴ d)	0 0 1 1 0		
<hr/>			
S(4) =	(0) 1 1 1 0 1	s,d khác dấu	q0 = -1
2S(4)	1 1 1 0 1	thực hiện cộng	
+(-2 ⁴ d)	0 0 1 1 0		
<hr/>			
S =	(1) 0 0 1 1 = 3	s, z khác dấu, điều chỉnh -c	
	1 1 0 1 0		
	(0) 1 1 0 1 = -3		
<hr/>			
p = -1 +1 +1 -1 -1 r = 0 1 1 0 0			
q' = (1 1 1 0 0 1)2's complement = -7, do ở trên có điều chỉnh số dư bằng cách trừ d nên phải tăng q thêm 1			
q = q' + 1 = -6.			
- bit nằm trong dấu ngoặc là bit nhớ của khối cộng 4 bit.			
- Các bit tận cùng bên trái dấu là bit dấu của các toán hạng, liệt kê chỉ để làm rõ giá trị cộng, các bit này luôn là 1,0 hoặc 0,1.			
-39 = +6((-1)2 ⁴ + (+1)2 ³ + (+1)2 ² + (-1)2 ¹ + (-1)2 ⁰ + 1) + (-3)			

Sơ đồ phân chi tiết phân cứng được thiết kế như sau:



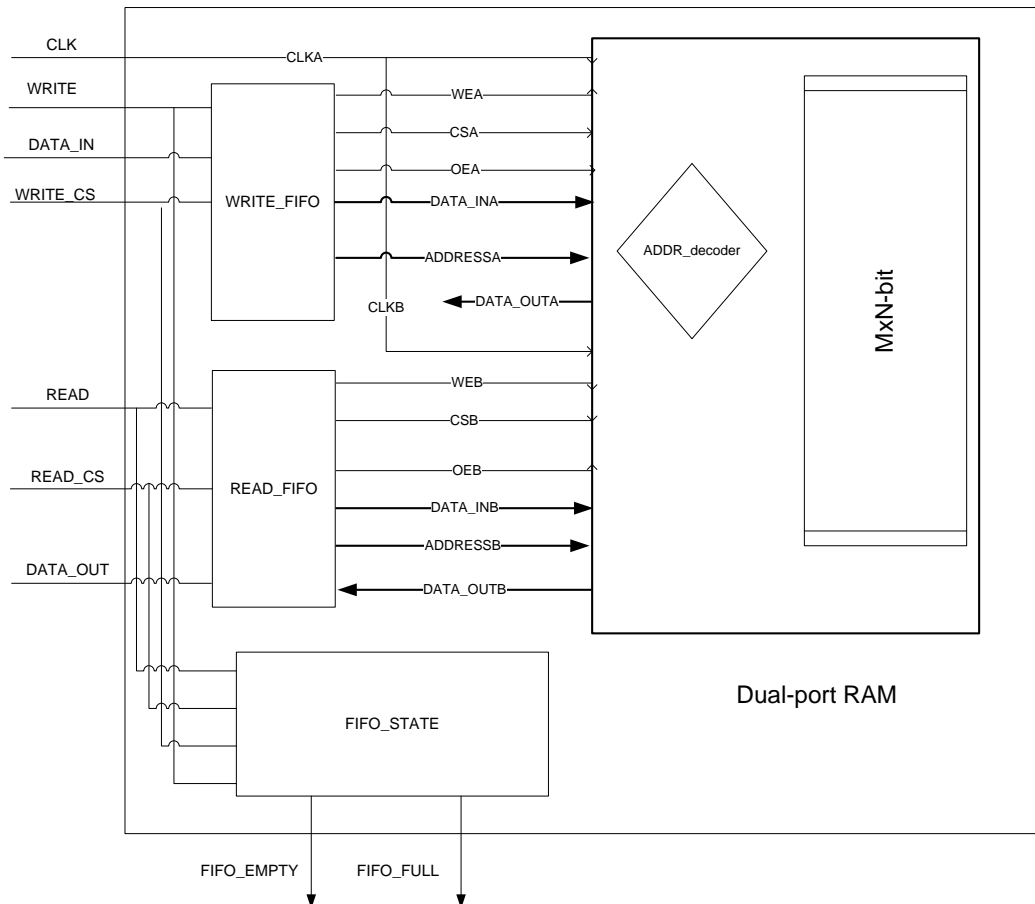
Hình 3.29. Sơ đồ khối chia có dấu

Sơ đồ trên được phát triển từ sơ đồ chia không phục hồi phần dư, điểm khác biệt là tín hiệu lựa chọn việc thực hiện phép cộng hay phép trừ tương ứng được thực hiện bằng một cổng XNOR với hai đầu vào là giá trị dấu của phần dư hiện tại và dấu của số chia, nếu hai giá trị này khác nhau thì giá trị pi tương ứng bằng -1, trên thực tế ta không biểu diễn giá trị -1 mà quy ước biểu diễn là số 0, nếu dấu hai số như nhau thì giá trị pi bằng 1. Các giá trị này được đẩy dần vào thanh ghi k-bit, giá trị $r = rk-1rk-2 \dots r_0$ thu được tại thanh ghi này sau k xung nhịp chưa phải là giá trị thương cần tìm, để tìm đúng giá trị thương ta phải có một khối thực hiện việc chuyển đổi r (Correct quotient) về giá trị q theo quy tắc trình bày ở trên. Khối này ngoài nhiệm vụ biến đổi r về q còn có thể thực hiện điều chỉnh giá trị cuối cùng bằng cách cộng hoặc trừ đi 1 đơn vị trong trường hợp phần dư ở thao tác cuối cùng thu được không cùng dấu với số bị chia.

Lưu ý rằng với biểu diễn bằng chuỗi $\{-1, 1\}$ thì giá trị của q luôn là số lẻ, vì vậy trong trường hợp kết quả thương số là chẵn thì việc điều chỉnh lại thương số ở bước cuối cùng là không thể tránh khỏi.

Câu 17. Trình bày thuật toán xây dựng FIFO và LIFO trên cơ sở Dual-port RAM.**Trả lời:****Bộ nhớ FIFO**

FIFO (First-In-First-Out) là một khối nhớ đặc biệt, rất hay ứng dụng trong các hệ thống truyền dẫn số, dùng làm các khối đệm trong các thiết bị lưu trữ... Đặc điểm duy nhất cũng là yêu cầu khi thiết kế khối này là dữ liệu nào vào trước thì khi đọc sẽ ra trước. Đối với FIFO không còn khái niệm địa chỉ mà chỉ còn các cổng điều khiển đọc và ghi dữ liệu. Tùy theo yêu cầu cụ thể mà FIFO có thể được thiết kế bằng các cách khác nhau. Sơ đồ đơn giản và tổng quát nhất của FIFO là sơ đồ sử dụng khối RAM đồng bộ hai cổng đọc ghi độc lập.



Hình 3-19. Sơ đồ khối FIFO sử dụng Dual-port RAM

Để FIFO làm việc đúng như yêu cầu cần thêm 3 khối điều khiển sau:

- Khối xác định trạng thái FIFO (FIFO_STATE) tạo ra hai tín hiệu FIFO_FULL, và FIFO_EMPTY để thông báo về tình trạng đầy hoặc rỗng tương ứng của bộ nhớ. Để tạo ra các tín hiệu này sử dụng một bộ đếm, ban đầu bộ đếm được khởi tạo bằng 0. Bộ đếm này thay đổi như sau:

- Nếu có thao tác đọc mà không ghi thì giá trị bộ đếm giảm xuống 1.
- Nếu có thao tác ghi mà không đọc thì giá trị bộ đếm tăng lên 1.
- Nếu có thao tác đọc và ghi thì giá trị bộ đếm không thay đổi.

Bằng cách đó

- FIFO_EMPTY = 1 nếu giá trị bộ đếm bằng 0
- FIFO_FULL = 1 nếu giá trị bộ đếm bằng giá trị tổng số ô nhớ của RAM là $2addr_width - 1$

- Khối điều khiển ghi vào FIFO (WRITE_FIFO) thực hiện tiền xử lý cho thao tác ghi dữ liệu trên một kênh cố định trong khối RAM. Địa chỉ của kênh này được khởi tạo bằng 0 và được tăng thêm 1 sau mỗi lần ghi dữ liệu. Khi giá trị địa chỉ đạt tới giá trị cao nhất bằng $2addr_width - 1$ thì được trả lại bằng 0. Thao tác ghi dữ liệu được thực hiện chỉ khi FIFO chưa đầy (FIFO_FULL = 0)

- Khối điều khiển đọc vào FIFO (READ_FIFO) thực hiện tiền xử lý cho thao tác đọc dữ liệu theo kênh còn lại của khối RAM. Địa chỉ của kênh này được khởi tạo bằng 0 và được tăng thêm 1 sau mỗi lần đọc dữ liệu. Khi giá trị địa chỉ đạt tới giá trị cao nhất bằng $2\text{addr_width} - 1$ thì được trả lại bằng 0. Thao tác đọc dữ liệu được thực hiện chỉ khi FIFO chưa đầy ($\text{FIFO_EMPTY} = 0$).

Bộ nhớ LIFO

LIFO (Last-In-First-Out) hay còn được gọi là STACK cũng là một khối nhớ tương tự như FIFO nhưng yêu cầu là dữ liệu nào vào sau cùng thì khi đọc sẽ ra trước. LIFO cũng có thể được thiết kế sử dụng khối RAM đồng bộ hai cổng đọc ghi độc lập. Điều khiển của LIFO khá đơn giản vì nếu có N ô nhớ được lưu trong LIFO thì các ô này sẽ lần lượt được lưu ở các ô từ 0 đến N-1, do đó khi thiết kế ta chỉ cần một bộ đếm duy nhất trở tới vị trí giá trị N.

Nếu có thao tác ghi dữ liệu mà không đọc thì dữ liệu được lưu vào ô có địa chỉ N và N tăng thêm 1

Nếu có thao tác đọc dữ liệu mà không ghi thì dữ liệu được đọc ở ô có địa chỉ N-1 và N giảm đi 1.

Nếu có thao tác đọc, ghi dữ liệu đồng thời thì N không thay đổi và dữ liệu đọc chính là dữ liệu ghi.

Trạng thái của LIFO cũng được xác định thông qua giá trị của N, nếu $N=0$ thì $\text{LIFO_EMPTY} = 1$ và nếu $N = 2\text{addr_width} - 1$ thì $\text{LIFO_FULL} = 1$.

Câu 18. Các dạng biểu diễn số thực, chuẩn số thực dấu phẩy động IEEE/ANSI 754. Các phương pháp làm tròn số thực dấu phẩy động.

Trả lời:

Chuẩn số thực dấu phẩy động IEEE/ANSI 754:

Định dạng chung của số thực dấu phẩy động thể hiện ở hình sau:



Hình 3-30. Định dạng số thực dấu phẩy động

Giá trị của số biểu diễn tính bằng công thức:

$$A = -1^S (FRACTION) 2^{EXPONENT - BIAS} \quad (3.14)$$

Trong đó:

Bit trọng số cao nhất S biểu diễn dấu, nếu $S = 1$ thì dấu âm, $S=0$ thì dấu dương.

K bit tiếp theo $E_k E_{k-1} \dots E_1$ biểu diễn số mũ (exponent), Các bit này biểu diễn các giá trị không dấu từ 0 đến $2^k - 1$. Chuẩn số thực quy định giá trị thực biểu diễn thực chất bằng $e = \text{exponent} - (2^k - 1)$, giá trị $(2^k - 1)$ gọi là độ dịch của số mũ (exponent bias) có nghĩa là miền giá trị của số mũ từ $-(2^k - 1)$ đến $+(2^k - 1)$.

N bit cuối cùng dùng để biểu diễn phần thập phân FRACTION với giá trị tương ứng là $m = 1, m m m m - 1 \dots m_1$, Với số thực chuẩn thì số 1 trong công thức này cũng không được biểu diễn trực tiếp mà ngầm định luôn luôn có nên còn gọi là bit ẩn.

Các số biểu diễn theo quy tắc trên được gọi là các số chuẩn (Normalized), ngoài ra còn quy định các số thuộc dạng không chuẩn và các giá trị đặc biệt bao gồm:

Denormalized : $EXPONENT + BIAS = 0, FRACTION \neq 0$

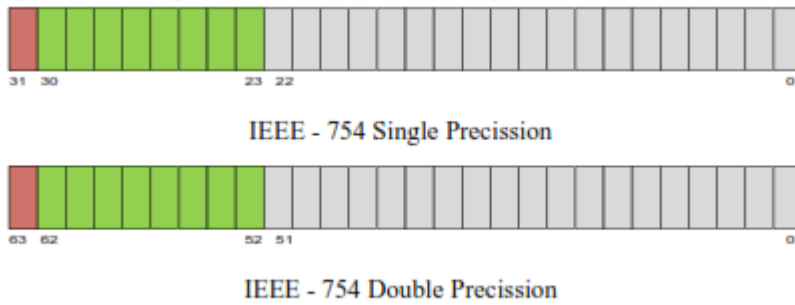
Zero ± 0 : $EXPONENT + BIAS = 0, FRACTION = 0$

Infinity $\pm \infty$: $EXPONENT + BIAS = 2^k - 1, FRACTION = 0$

Not a Number NaN : $EXPONENT + BIAS = 2^k - 1, FRACTION \neq 0$

Số không chuẩn (Denormalized) là những số có số mũ thực tế bằng 0 ($EXPONENT + BIAS = 0$) và số thập phân biểu diễn dưới dạng $m = 0, m m m m - 1 \dots m_1$, nghĩa là số 1 ẩn được thay bằng số 0, số không chuẩn được sử dụng để biểu diễn các số có trị tuyệt đối nhỏ hơn trị tuyệt đối số bé nhất trên miền số chuẩn là Chuẩn số thực 32bit (Float)

và 64-bit (Double) là hai định dạng được sử dụng phổ biến trong các IC tính toán được định nghĩa bởi IEEE-754 1985.



Hình 3-31. Chuẩn số thực ANSI/ IEEE - 754

Các phương pháp làm tròn số thực dấu phẩy động.

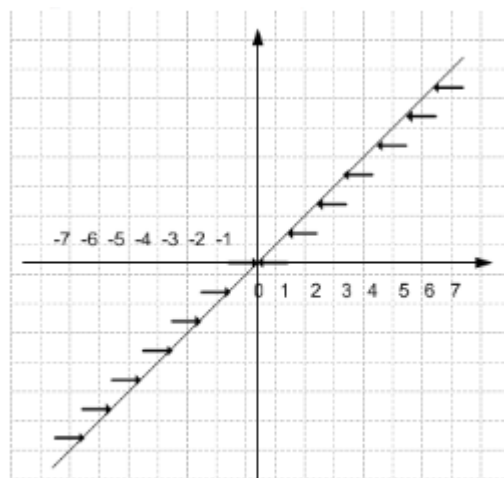
Làm tròn (Rounding) là chuyển giá trị biểu diễn ở dạng chính xác hơn về dạng không chính xác hơn nhưng có thể biểu diễn ở đúng định dạng quy định của số đang xét. Giá trị muốn biểu diễn ở dạng chính xác hơn cần phải sử dụng số lượng bit lớn hơn so với bình thường. Làm tròn thường là động tác cuối cùng trong chuỗi các động tác thực hiện phép tính với số thực dấu phẩy động. Có nhiều phương pháp để làm tròn số, không mất tính tổng quát có thể xem chuỗi số đầu vào là một số thập phân có dạng:

$$X, Y = x_{n-1}x_{n-2} \dots x_0.y_0y_1 \dots y_m$$

Chuỗi số đầu ra là một số nguyên có dạng:

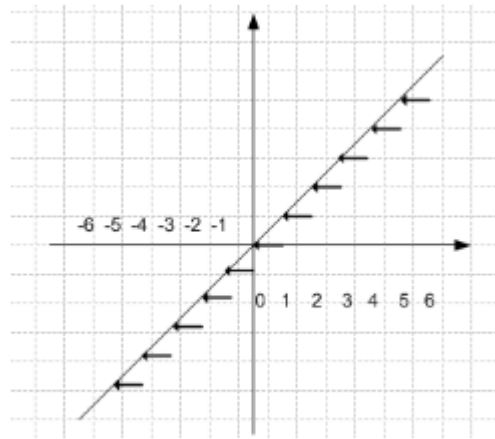
$$Z = z_{n-1}z_{n-2} \dots z_0$$

Phương pháp đơn giản nhất là bỏ toàn bộ phần giá trị sau dấu phẩy. Nếu với số biểu diễn dạng dấu-giá trị thì việc cắt bỏ phần thừa số mới có giá trị tuyệt đối nhỏ hơn hoặc bằng số cũ. Với số giá trị dương thì số sau làm tròn nhỏ hơn số trước làm tròn, còn với số âm thì ngược lại, chính vì vậy phương pháp này được gọi là làm tròn hướng tới 0 (Round toward 0).

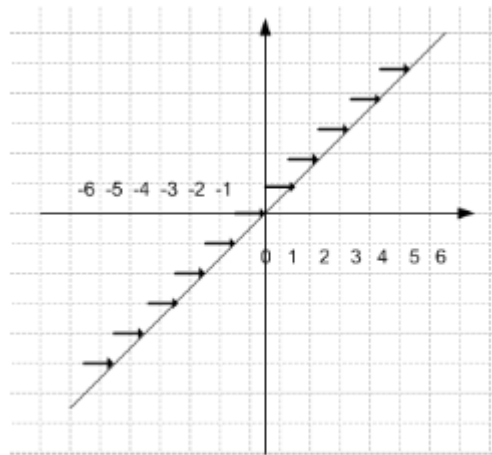


Hình 3-33. Làm tròn hướng tới 0

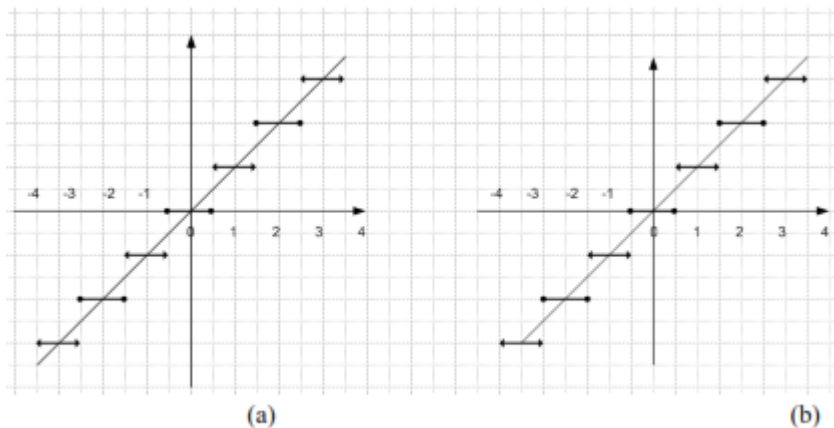
Với số biểu diễn dạng bù 2 thì có thể chỉ ra rằng việc cắt bỏ phần thừa luôn làm cho giá trị số sau làm tròn nhỏ hơn giá trị số trước làm tròn, chính vì vậy phương pháp làm tròn này được gọi là làm tròn hướng tới $-\infty$ (round toward $-\infty$). Hình vẽ bên thể hiện cơ chế làm tròn này

Hình 3-34. Làm tròn hướng tới $-\infty$

Nếu như số làm tròn được làm tròn lên giá trị cao hơn, thay vì làm tròn với giá trị thấp hơn thì ta có kiểu làm tròn tới $+\infty$ (round toward $+\infty$). Với kiểu này thì giá trị sau làm tròn luôn lớn hơn giá trị trước làm tròn.

Hình 3-35. Làm tròn hướng tới $+\infty$

Ba kiểu làm tròn trên hoặc làm tròn lên cận trên hoặc làm tròn xuống cận dưới nên không mang lại hiệu quả tối ưu về mặt sai số. Kiểu làm tròn được hỗ trợ ngầm định trong chuẩn ANSI/IEEE 754 là làm tròn tới giá trị gần nhất, với cách đặt vấn đề như trên thì giá trị làm tròn sẽ được lấy là số nguyên cận trên hay dưới tùy thuộc vào khoảng cách giữa số cần làm tròn tới các cận này. Cận nào gần hơn thì sẽ được làm tròn tới. Trong trường hợp giá trị cần làm tròn nằm chính giữa thì sẽ phải quy định thêm là sẽ làm tròn lên hoặc xuống.



Hình 3-36. Làm tròn tới số gần nhất chẵn (a) và Làm tròn tới số gần nhất lẻ (b)

Nếu quy định rằng trong trường hợp giá trị cần làm tròn nằm chính giữa ví dụ 1,50 ta luôn làm tròn lên 2,0 khi đó giá trị sai số trung bình có chiều hướng luôn dương. Chính vì vậy trên thực tế có thêm một quy định nữa là việc làm tròn lên hay xuống của số chính giữa phụ thuộc vào việc phân nguyên hiện tại là chẵn hay lẻ, nếu phần

nguyên hiện tại là chẵn thì ta luôn làm tròn xuống cận dưới (tới giá trị chẵn). Cách làm tròn này gọi là làm tròn tới số gần nhất chẵn (Round to nearest even), với cách này thì xác suất làm tròn tới cận trên và dưới là bằng nhau và vì vậy trung bình sai số bằng 0. Tương tự ta cũng có kiểu làm tròn tới số gần nhất lẻ (Round to nearest odd). Về lý thuyết thì hai phương pháp này có hiệu quả tương đương nhau nhưng trên thực tế kiểu làm tròn ngầm định trong chuẩn ANSI/IEEE-754 là làm tròn tới số gần nhất chẵn.

Câu 19. Trình bày về thuật toán và cấu trúc khối cộng số thực dấu phẩy động theo chuẩn IEEE/ANSI 754.

Trả lời:

Để thực hiện phép cộng của hai số thực A, B với các giá trị như sau.

$$A = -1^{signa}(1, a_{n-1}..a_1a_0)2^{ea-BIAS}$$

$$B = -1^{signb}(1, b_{n-1}..b_1b_0)2^{eb-BIAS}$$

Trong đó ea, eb là các giá trị biểu diễn còn ea-BIAS, eb-BIAS là các giá trị thực tế của số mũ.

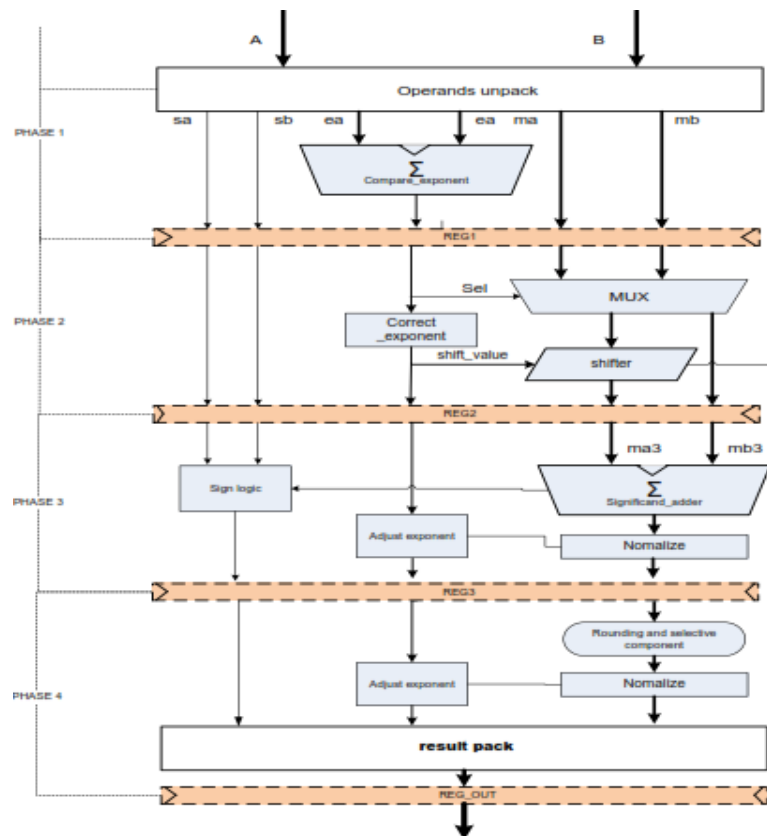
Một trong những bước đầu tiên là quy đổi hai số về cùng một số mũ chung, nếu xét về mặt toán học có thể thực hiện quy đổi về số mũ chung ea hoặc eb đều cho kết quả như nhau nếu như khối cộng đảm bảo sự chính xác tuyệt đối. Tuy vậy trên thực tế phép cộng số thực theo chuẩn xét trên đa phần không có kết quả tuyệt đối chính xác do những giới hạn về thực tế phần cứng. Lý thuyết khi đó chỉ ra rằng quy đổi hai số về biểu diễn của số mũ của số lớn hơn bao giờ cũng mang lại độ chính xác cao hơn cho phép toán (do giảm thiểu ảnh hưởng của thao tác làm tròn lên độ chính xác của kết quả). Giả sử như ea > eb khi đó B được viết lại thành:

$$B = -1^{signb}(1, b_{n-1}..b_1b_0)2^{-(ea-eb)}2^{ea-BIAS}$$

Tiếp theo tiến hành cộng phần thập phân của hai số

$$M = (1, a_{n-1}..a_1a_0) + (1, b_{n-1}..b_1b_0)2^{-(ea-eb)} = mA + mB. 2^{-(ea-eb)}$$

Với lý thuyết cơ bản trên khối cộng đã được hiện thực hóa bằng nhiều sơ đồ khác nhau tùy theo mục đích của thiết kế nhưng về cơ bản chúng đều có các khối chính được biểu diễn ở hình sau



Hình 3-37. Sơ đồ khối cộng số thực dấu phẩy động

Để dễ theo dõi ta xem các số đang xét có kiểu số thực 32 bit, phần thập phân $n=23$ bit, ea, eb sẽ biểu diễn bằng 8 bit, $\text{BIAS} = 127$. Các hạng tử (operands) được phân tách thành các thành phần gồm dấu (sa, sb), số mũ (ea, eb) và phần thập phân (ma, mb). Ở sơ đồ trình bày trên các thanh ghi pipelined là không bắt buộc và không ảnh hưởng tới chức năng của mạch.

Khối cộng đầu tiên có chức năng tìm giá trị chênh lệch giữa giá trị số mũ, đồng thời xác định toán tử nào có giá trị bé hơn sẽ bị dịch sang phải ở bước tiếp theo. Khối này luôn thực hiện phép trừ hai số không dấu 8 bit cho nhau. Kết quả thu được ea-eb nếu là số âm thì $a < b$ và phải làm một bước nữa là trả lại đúng giá trị tuyệt đối cho kết quả, thao tác này được thực hiện bởi khối Correct exponent, (thực chất là lấy bù 2 của ea-eb). Trong trường hợp ea-eb ≥ 0 thì $a \geq b$, kết quả ea-eb được giữ nguyên.

Khối mux và shifter có nhiệm vụ chọn đúng hạng tử bé hơn và dịch về bên phải một số lượng bit bằng |ea-eb|. Đó chính là phép nhân với $2^{-(ea-eb)}$ như trình bày ở trên. Ở đây chỉ lưu ý một điểm là mặc dù kết quả ea-eb là một số 8 bit tuy vậy khối dịch chỉ thực hiện dịch nếu giá trị tuyệt đối ea-eb ≤ 26 , lý do là vì phần thập phân được biểu diễn bằng 23 bit, nếu tính thêm các bit cần thiết cho quá trình làm tròn là 3 bit nữa là 26, trong trường hợp giá trị này lớn hơn 26 thì kết quả dịch luôn bằng 0.

Sigfinicand_adder là khối cộng cho phần thập phân, đầu vào của khối cộng là các hạng tử đã được quy chỉnh về cùng một số mũ, số lượng bit phụ thuộc vào giá trị tuyệt đối |ea-eb| tuy vậy cũng như đối với trường hợp khối dịch ở trên, do giới hạn của định dạng số thực hiện tại chỉ hỗ trợ 23 bit cho phần thập phân nên thực tế chỉ cần khối cộng 25 bit nếu tính cả bit ẩn và bit dấu cho phép cộng số biểu diễn dạng bù 2. Các bit “thừa” ra phía bên phải sẽ được điều chỉnh phù hợp cho các kiểu làm tròn tương ứng.



Hình 3-38. Dạng kết quả cần làm tròn

Ở đây ta xét kiểu làm tròn phổ biến là làm tròn tới số gần nhất chẵn với cách làm tròn này ta sẽ phải quan tâm tới các bit đặc biệt gồm bit cuối cùng M0 của phần biểu diễn để xác định tính chẵn lẻ, bit đầu tiên ngoài phần biểu diễn là bit để làm tròn Round, nếu bit này bằng 0 số đang xét được làm tròn xuống cận dưới (phần thừa bị cắt bỏ), nếu bit này bằng 1 thì phải quan tâm xem phần còn lại có bằng 0 hay không. Phần còn lại được đại diện bằng 1 bit duy nhất gọi là Sticky bit này thu được bằng phép OR logic đối với tất cả các bit thừa thứ 2 trở đi. Nếu giá trị này khác 0 số đang xét được làm tròn lên cận trên (cộng thêm 1). Nếu như phần này bằng 0 thì số đang xét được làm tròn tới giá trị chẵn gần nhất, nghĩa là phụ thuộc giá trị M0, nếu bằng 1 thì được làm tròn lên, nếu là 0 thì làm tròn xuống.

Quay trở lại với khối cộng số thực đang xét yêu cầu đối với kết quả phép cộng sẽ phải có các bit như sau:



Hình 3-39. Dạng kết quả cộng của phần thập phân trước làm tròn

Ngoài phần kết quả của phép cộng từ bit Cout đến M0 gồm 25 bit, để thực hiện làm tròn trong thành phần kết quả cần phải thêm 3 bit với tên gọi là Guard bit (G), Round bit (R) và Sticky bit (S).

Trước khi làm tròn kết quả (Rounding) cần phải thực hiện chuẩn hóa giá trị thu được trong khối Normalize, khối này đưa giá trị về dạng biểu diễn cần thiết 1, xx...xxx chính vì vậy cần phải xác định xem số 1 đầu tiên xuất hiện trong chuỗi trên từ bên trái sang nằm ở vị trí nào. Vị trí bit này có thể dễ dàng xác định bằng một khối mã hóa ưu tiên.

Trường hợp thực hiện cộng hai giá trị ma, mb khi đó số 1 đầu tiên bên trái có thể nằm ở vị trí của Cout hoặc M23. Nếu kết quả thu được không có bit nhớ Cout, và bit $M23 = '1'$, khi đó theo quy tắc làm tròn ta sẽ quan tâm tới giá trị của M0 để xác định tính chẵn lẻ, G có vai trò như bit để làm tròn, còn R, S có vai trò như nhau cung cấp thông tin về phần còn lại có khác 0 hay không. Trên thực tế S thu được từ phép OR của tất cả các bit thừa ra từ vị trí S về bên phải.

Nếu kết quả thu được có nhớ thì khi làm tròn sẽ quan tâm tới giá trị của M1 để xét tính chẵn lẻ, M0 để làm tròn, các bit G, R, S cung cấp thông tin cho phần còn lại có khác 0 hay không. Đối với trường hợp phép trừ ma cho mb thì có 3 khả năng cho vị trí của bit 1 đầu tiên từ trái sang.

Trường hợp $|ea-eb| > 1$ thì có thể xảy ra trường hợp thứ nhất là bit này rơi vào vị trí M23 như đã xét ở trên. Trường hợp thứ hai là số 1 rơi vào vị trí M22 (có thể dễ dàng chỉ ra là số 1 không thể nằm ở vị trí thấp hơn). Khi đó

tính chẵn lẻ của số làm tròn quy định bởi G (chính vì thế G được gọi là Guard bit), bit làm tròn là R còn S có vai trò xác định cho giá trị phần còn lại. Còn trường hợp khi số 1 đầu tiên nằm ở vị trí nhỏ hơn 22 thì chỉ xảy ra khi $|ea - eb| \leq 1$ khi đó thì các bit R, S đồng thời bằng 0, việc làm tròn là không cần thiết.

Sau khi thực hiện làm tròn mà thực chất là quyết định xem có cộng thêm 1 hay không vào kết quả cộng phân thập phân. Việc cộng thêm dù 1 đơn vị có thể gây ra tràn kết quả và làm hỏng định dạng chuẩn của số (số 1 đầu tiên dịch sang trái một bit). Đó là khi cộng thêm 1 vào số có dạng $1, 11 \dots 111 + 0, 00 \dots 001 = 10, 00 \dots 000$. Do đó sau khối làm tròn buộc phải có thêm một khối chuẩn hóa thứ hai để điều chỉnh kết quả thu được khi cần thiết. Rõ ràng trường hợp tràn này là duy nhất và với lần điều chỉnh này thì thao tác làm tròn không cần thực hiện lại.

Như vậy thao tác chuẩn hóa và làm tròn tuy không phức tạp nhưng nhiều trường hợp đặc biệt mà người thiết kế phải tính đến. Sau mỗi thao tác chuẩn hóa thì cần phải thực hiện điều chỉnh số mũ tăng hay giảm phụ thuộc vào vị trí đầu tiên của số 1 từ bên trái kết quả cộng. Việc điều chỉnh số mũ tuy đơn giản nhưng khi thiết kế một cách đầy đủ cần tính đến các khả năng tràn số, nghĩa là số mũ sau điều chỉnh nằm ngoài miền biểu diễn, người thiết kế có thể tự tìm hiểu thêm các trường hợp này khi làm hiện thực hóa sơ đồ.

Câu 20. Trình bày về thuật toán và cấu trúc khối nhân số thực dấu phẩy động theo chuẩn IEEE/ANSI 754.

Trả lời:

Ngược lại với các khối tính toán trên số nguyên, khối nhân trên số thực dấu phẩy động đơn giản nếu so sánh với khối cộng. Xét hai số sau:

$$A = -1^{signa}(1, a_{n-1} \dots a_1 a_0) 2^{ea - BIAS}$$

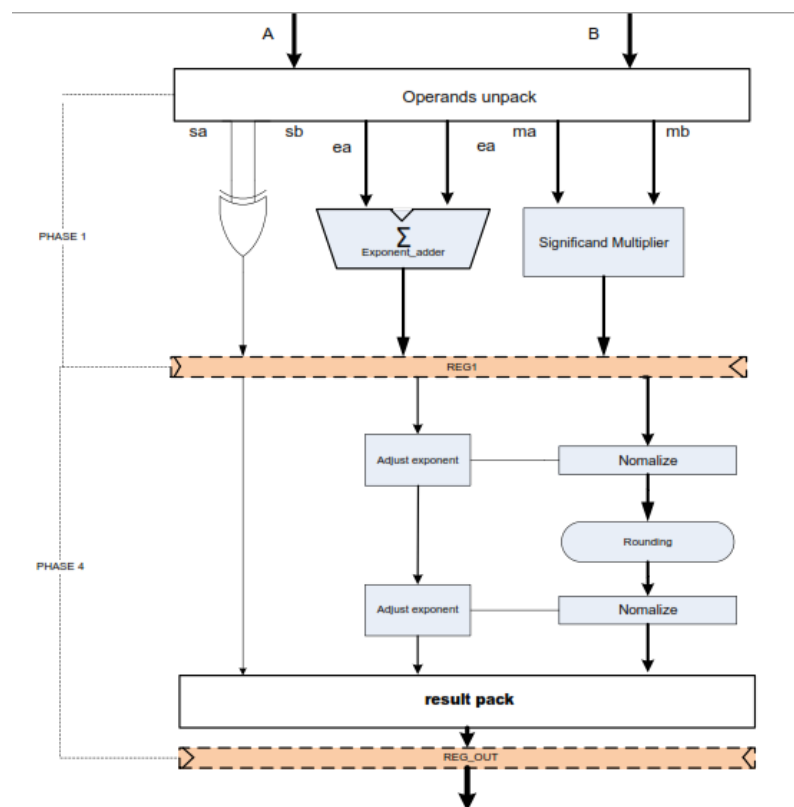
$$B = -1^{signb}(1, b_{n-1} \dots b_1 b_0) 2^{eb - BIAS}$$

Trong đó ea, eb là các giá trị biểu diễn còn ea-BIAS, eb-BIAS là các giá trị thực tế của số mũ.

Khi đó kết quả phép nhân.

$$A.B = -1^{signa+signb}(1, a_{n-1} \dots a_1 a_0) \cdot (1, b_{n-1} \dots b_1 b_0) \cdot 2^{ea+eb-2 \cdot BIAS}$$

Sơ đồ khối nhân sử dụng số thực dấu phẩy động được trình bày ở hình dưới đây:

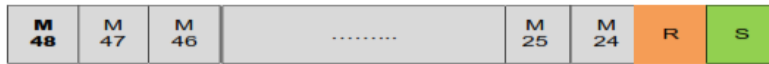


Hình 3-40. Sơ đồ khối nhân số thực dấu phẩy động

Cũng để cho mục đích tiện theo dõi ta xét trường hợp phép nhân với số thực 32-bit theo định dạng ANSI/IEEE 754. Tại khối cộng số mũ, sau khi cộng các giá trị ea, eb ta thu được giá trị ea+eb. Còn giá trị thực tế là ea+eb - 2 * BIAS như công thức trên, nghĩa là giá trị cần biểu diễn là ea+eb - 2*BIAS + BIAS = ea+eb - BIAS. Chính vì vậy để thu được giá trị biểu diễn của số mũ ta cần trừ kết quả cộng đi BIAS.

Để trừ 127 thì lưu ý BIAS = 127 = 11111111 = 28-1, do vậy ta sẽ cho 1 vào bit Cin của bộ cộng sau đó trừ đi 28 = 100000000, phép trừ này thực tế rất

đơn giản vì các bit thấp đều bằng 0, do vậy ta tiết kiệm được một bộ cộng dùng để điều chỉnh BIAS.



Hình 3-41. Dạng kết quả nhân của phần thập phân

Khối nhân phần thập phân (Significand multiplier) thực chất là khối nhân số nguyên 24-bit x 24-bit trong trường hợp này. Tối đa ta thu được số 48 bit, và lưu ý rằng cuối cùng kết quả này được “thu gọn” chỉ còn 24 bit chính vì vậy chúng ta có thể thiết kế một khối nhân không hoàn chỉnh để thu được giá trị 24 bit cần thiết. Do bit 1 đầu tiên bên trái sang của kết quả nhân luôn rơi vào vị trí thứ M48 hoặc M47 nên trong trường hợp này không cần thiết phải có bit Guard khi làm tròn, như vậy chỉ cần 2 bit thêm vào phần biểu diễn là bit R (Round) và bit S (Sticky). Người thiết kế cần chú ý các đặc điểm trên để có một khối nhân tối ưu về mặt tài nguyên logic cũng như tốc độ thực thi.

Phần dưới của khối nhân số thực không khác gì khối cộng, điểm khác biệt là khối điều chỉnh số mũ đơn giản hơn vì chỉ phải điều chỉnh nhiều nhất 1 đơn vị trong cả hai trường hợp trước và sau khi làm tròn.

Câu 21. Trình bày về thuật toán và cấu trúc khối chia số thực dấu phẩy động theo chuẩn IEEE/ANSI 754

Trả lời:

Xét hai số sau:

$$A = -1^{signa}(1, a_{n-1}..a_1a_0)2^{ea-BIAS}$$

$$B = -1^{signb}(1, b_{n-1}..b_1b_0)2^{eb-BIAS}$$

Trong đó ea, eb là các giá trị biểu diễn còn ea-BIAS, eb-BIAS là các giá trị thực tế của số mũ.

Khi đó kết quả phép chia:

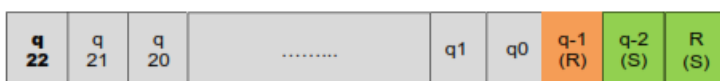
$$A.B = -1^{signa+signb}(1, a_{n-1}..a_1a_0).(1, b_{n-1}..b_1b_0).2^{ea+eb-2*BIAS}$$

Một điểm lưu ý duy nhất là khi thực hiện chia phần giá trị $(1.a_{n-1}..a_1a_0)/(1.b_{n-1}..b_1b_0)$ kết quả thu được cũng phải có dạng chuẩn 1. $c_{n-1}..c_1c_0$. Để có kết quả như vậy phải mở rộng thêm phần giá trị thập phân của A thêm tối thiểu n+1 bit:

Trên thực tế ta sẽ phải thực hiện phép chia số nguyên không dấu như sau:

$$\frac{(1, a_{n-1}..a_1a_0)}{1, b_{n-1}..b_1b_0} = \frac{(1a_{n-1}..a_1a_0).2^{-n}}{(1b_{n-1}..b_1b_0).2^{-n}} = \frac{(1a_{n-1}..a_1a_000..0).2^{-n}}{(1b_{n-1}..b_1b_0)}$$

Kết quả phép chia thu được thương số là một số q = r. Trong đó các bit q_i là kết quả chia, còn r thêm vào đại diện cho phần số dư. r = 0 nếu số dư bằng 0 còn r = 1 nếu số dư khác 0, bit này chỉ có ý nghĩa khi thực hiện làm tròn.



Hình 3-42. Dạng kết quả chia của phần thập phân

Tiếp theo sẽ thực hiện làm tròn q để thu được Mc.

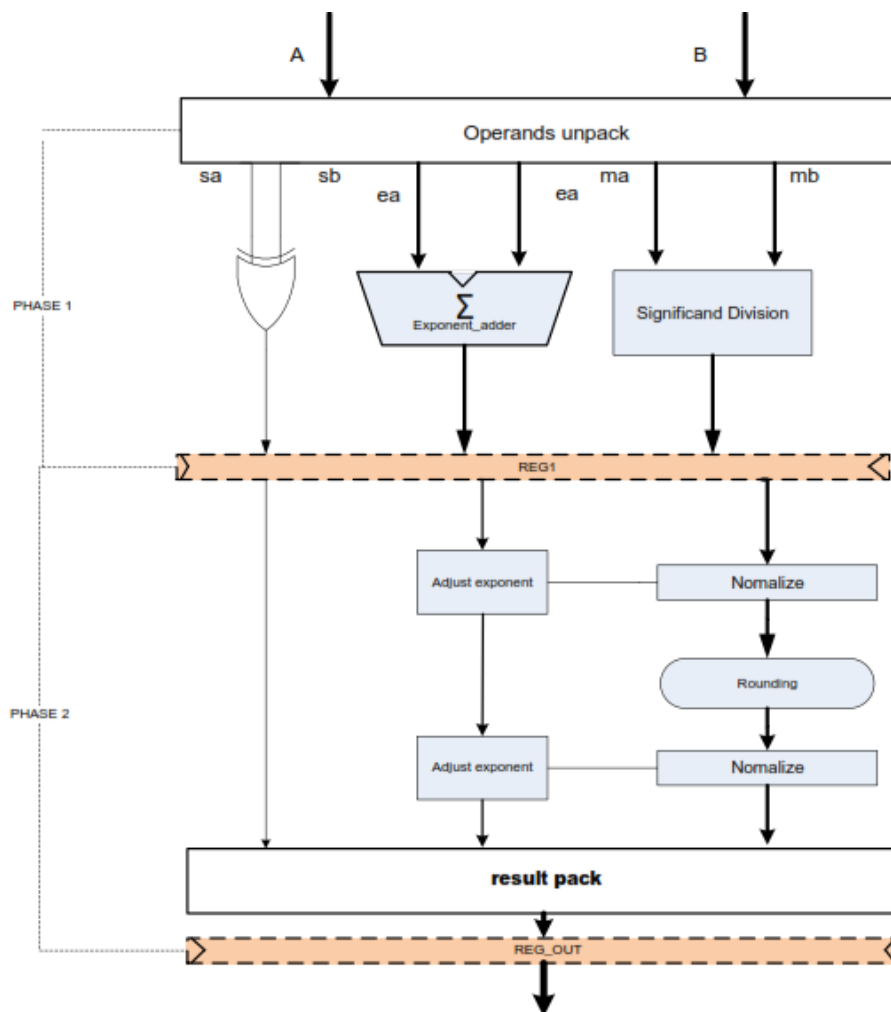
$$Mc = \text{Round } q_{n-1} \dots q_1 q_0 q_{-1} q_{-2}$$

Không mất tổng quát và để dễ theo dõi xét trường hợp số thực dấu phẩy động theo chuẩn IEEE/ANSI 754 với $n = 23$, kết quả chia thể hiện ở hình 3.42. Có thể dễ dàng chỉ ra rằng do đặc điểm của số chia và số bị chia đều là các số 24 bit và 48 bit với bit đầu tiên bằng 1 nên trong kết quả thương q số 1 xuất hiện đầu tiên từ bên trái số sẽ rơi vào hoặc q_{22} hoặc q_{21} . Vì vậy để phục vụ cho việc làm tròn thương số cần thiết phải tính thêm 2 bit sau bit cuối cùng q_0 là q_{-1} và q_{-2} . Trường hợp bit 1 rơi vào bit q_{24} thì bit q_0 quyết định chẵn lẻ, q_{-1} là bit làm tròn, q_{-2} , r đại diện cho phần còn lại. Trường hợp bit 1 rơi vào vị trí q_{23} thì bit quyết định chẵn lẻ là q_{-1} bit làm tròn là q_{-2} còn r đại diện cho phần còn lại.

Khối chia phần thập phân Significand division là khối phức tạp và chiếm nhiều tài nguyên logic nhất. Tuy vậy kết quả của phép chia này là 24 bit và việc hiệu chỉnh để thu được dạng biểu diễn chuẩn khá đơn giản. Phần dưới của khối chia cũng bao gồm hai khối chuẩn hóa và làm tròn như trong trường hợp khối nhân.

Tại khối trừ số mũ, sau khi trừ các giá trị ea , eb ta thu được giá trị $ea - eb$, giá trị này trùng với giá trị thực tế. Để thu được giá trị biểu diễn của số mũ ta cần cộng kết quả với BIAS. Cũng như đối với trường hợp phép nhân, ta sẽ thêm bit $Cin = 1$ vào khối trừ sau đó thực hiện cộng với giá trị 128 để tối ưu về mặt tài nguyên.

Sơ đồ khối chia sử dụng số thực dấu phẩy động được trình bày ở hình dưới đây:



Hình 3-43. Sơ đồ khối chia số thực dấu phẩy động

Chương IV

Câu 22. Khái niệm FPGA, Các ưu điểm của FPGA so sánh với các IC khả trình trước đó, kiến trúc tổng quan của FPGA và kiến trúc của FPGA SPARTAN 3E.

Trả lời:

Khái niệm FPGA:

FPGA là công nghệ IC lập trình mới nhất và tiên tiến nhất hiện nay. Thuật ngữ Field-Programmable chỉ quá trình tái cấu trúc IC có thể được thực hiện bởi người dùng cuối, trong điều kiện bình thường, hay nói một cách khác là người kỹ sư lập trình IC có thể dễ dàng hiện thực hóa thiết kế của mình sử dụng FPGA mà không lệ thuộc vào một quy trình sản xuất hay cấu trúc phần cứng phức tạp nào trong nhà máy bán dẫn. Đây chính là một đặc điểm làm FPGA trở thành một công nghệ PLD được phát triển và nghiên cứu nhiều nhất hiện nay.

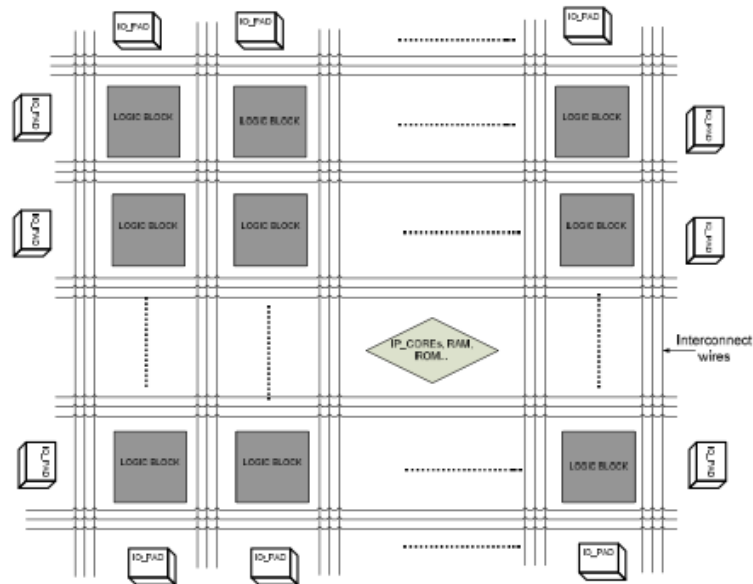
Các ưu điểm của FPGA so sánh với các IC khả trình:

+ Cơ chế tái cấu trúc FPGA, toàn bộ cấu hình của FPGA thường được lưu trong một bộ nhớ động (thông thường SRAM), quá trình tái cấu trúc được thực hiện bằng cách đọc thông tin từ RAM để lập trình lại các kết nối logic trong IC. Nói một cách khác cơ chế đó giống như phần mềm máy tính cũng được lưu trữ trong RAM và khi thực thi sẽ được đọc lần lượt nạp vào vi xử lý. Cũng như vậy việc lập trình lại cho FPGA cũng dễ dàng như lập trình lại phần mềm trên máy tính. Như vậy về mặt nguyên tắc thì quá trình khởi động của FPGA không diễn ra tức thì mà cấu hình từ SRAM phải được đọc sau đó mới diễn ra tái cấu trúc theo thông tin chứa trong SRAM. SRAM chỉ lưu trữ được trong trạng thái làm việc chính vì vậy để lưu giữ cấu hình cho FPGA thường phải dùng thêm một ROM ngoại vi. Đến những dòng sản phẩm FPGA gần đây thì các ROM ngoại vi này dần được thay thế bằng các ROM tích hợp sẵn, việc tích hợp này làm FPGA nạp cấu hình nhanh hơn nhưng cơ chế thực hiện vẫn phải thông qua một bộ nhớ SRAM như cũ.

+ FPGA có khả năng tích hợp logic với mật độ cao hơn hẳn, với số cổng logic tương đương lên tới hàng trăm nghìn, hàng triệu cổng. Khả năng đó nhờ sự đột phá trong kiến trúc của FPGA. Nếu hướng mở rộng của CPLD tích hợp nhiều mảng PAL, PLA lên một chip đơn, trong khi bản thân các mảng này có kích thước lớn và cấu trúc không đơn giản nên số lượng mảng tích hợp nhanh chóng bị hạn chế, dung lượng của CPLD nhiều nhất cũng chỉ đạt được con số trăm nghìn cổng tương đương. Đối với FPGA thì phần tử logic cơ bản không còn là mảng PAL, PLA mà thường là các khối logic lập trình được cho 4 bit đầu vào và 1 đầu ra (thường được gọi là LUT). Việc chia nhỏ đơn vị logic cho phép tạo một cấu trúc khả trình linh hoạt hơn và tích hợp được nhiều hơn số lượng cổng logic trên một chip bán dẫn. Bên cạnh đó hiệu quả làm việc và tốc độ làm việc của FPGA cũng vượt trội so với các IC khả trình trước đó

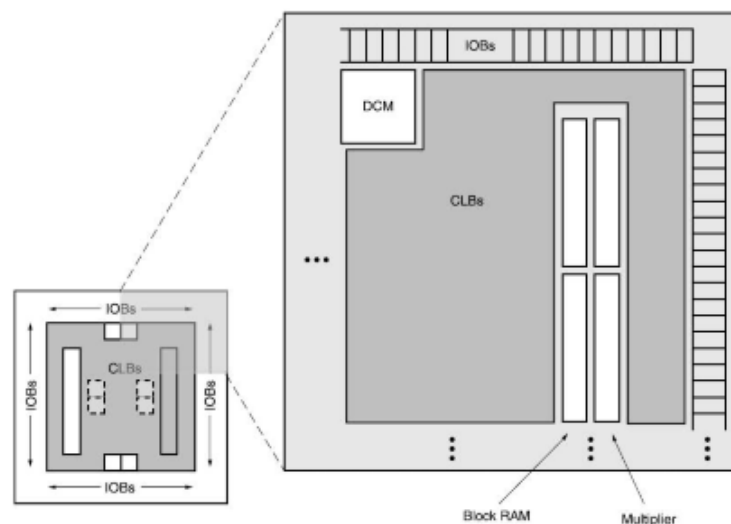
Kiến trúc tổng quan của FPGA :

Cấu trúc chi tiết và tên gọi của các thành phần có thể thay đổi tùy theo các hãng sản xuất khác nhau nhưng về cơ bản FPGA được cấu thành từ các Khối Logic (Logic Block) số lượng của các khối khối này thay đổi từ vài trăm (Xilinx Spartan) đến vài chục nghìn (Xilinx Virtex6) được bố trí dưới dạng ma trận, chúng được nối với nhau thông qua hệ thống các kênh kết nối khả trình. Hệ thống này còn có nhiệm vụ kết nối với các cổng giao tiếp vào ra (IO_PAD) của FPGA. Số lượng các chân vào ra thay đổi từ vài trăm đến cỡ hơn một nghìn. Bên cạnh các thành phần chính đó, những FPGA cỡ lớn còn được tích hợp những khối thiết kế sẵn mà thuật ngữ gọi là IP cores, các IP cores này có thể là các bộ nhớ RAM, ROM, khối nhân, DSP...



Hình 3.2. Kiến trúc tổng quan của FPGA

Kiến trúc của FPGA SPARTAN 3E:



Hình 3.3. Kiến trúc tổng quan của Spartan 3E FPGA

FPGA Spartan 3E được cấu trúc từ các thành phần sau:

+ CLBs (Configurable Logic Blocks) Là các khối logic lập trình được chứa các LUTs và các phần tử nhớ flip-flop có thể được cấu trúc thực hiện các hàm khác nhau.

+ IOBs (Input/Output Blocks) là các khối điều khiển giao tiếp giữa các chân vào của FPGA với các khối logic bên trong, hỗ trợ được nhiều dạng tín hiệu khác nhau. Các khối IO được phân bố xung quanh mảng các CLB.

+ Block RAM các khối RAM 18Kbit hỗ trợ các cổng đọc ghi độc lập, với các FPGA họ Spartan 3 block RAM thường phân bố ở hai cột, mỗi cột chứa một vài module RAM 18Kbit, mỗi khối RAM được nối trực tiếp với một khối nhân 18 bit.

+ Dedicated Multiplier: Các khối thực hiện phép nhân với đầu vào là các số 18 bit.

+ DCM (Digital Clock Manager) Các khối làm nhiệm vụ điều chỉnh, phân phối tín hiệu đồng bộ tới tất cả các khối khác. DCM thường được phân bố ở giữa, với hai khối ở trên và hai khối ở dưới.

+ Interconnect: Các kết nối khả trình và ma trận chuyển dùng để liên kết các phần tử chức năng của FPGA với nhau

Câu 23. Trình bày về các yếu tố tạo nên khả năng tái cấu trúc của FPGA. Khái niệm CLB, SLICE, LUT, Wide Multiplexer và cách thức thực hiện hàm logic 4 và nhiều đầu vào trên FPGA.

Trả lời:

Các yếu tố tạo nên khả năng tái cấu trúc của FPGA:

Ngôn ngữ VHDL được xem là một ngôn ngữ chặt chẽ và phức tạp, VHDL hỗ trợ việc mô tả thiết kế từ mức cao cho đến mức thấp và trên thực tế không thể xếp ngôn ngữ này thuộc nhóm bậc cao, bậc thấp hay bậc chung như các ngôn ngữ lập trình khác.

Về phân loại mã nguồn VHDL có thể chia làm ba dạng chính như sau:

- Mã nguồn chỉ dành cho tổng hợp (HDL for Synthesis): Là những mã nguồn nhằm mô tả thực của cấu trúc mạch. Ngoài việc tuân thủ chặt chẽ các cấu trúc của ngôn ngữ thì mã nguồn dạng này cần phải tuân thủ những tính chất, đặc điểm vật lý của một mạch tích hợp nhằm đảm bảo mã nguồn có thể được biên dịch trên một công nghệ phần cứng cụ thể nào đó.

- Mã nguồn mô phỏng được (HDL for Simulation): Bao gồm toàn bộ mã tổng hợp được và những mã mà chỉ chương trình mô phỏng có thể biên dịch và thể hiện trên môi trường phần mềm, ví dụ các mã sử dụng các lệnh tuần tự dùng để gán tín hiệu theo thời gian, các vòng lặp cố định.

- Mã nguồn dành cho mô tả đặc tính (HDL for Specification): Bao gồm toàn bộ mã mô phỏng được và những cấu trúc dùng để mô tả các đặc tính khác như độ trễ (delay time), điện dung (capacitance)... thường gặp trong các mô tả thư viện cổng hay thư viện đối tượng công nghệ. Trong khuôn khổ của chương trình này ta không tìm hiểu sâu về dạng mô tả này.

Khái niệm CLB: CLB (*Configurable Logic Blocks*) Là các khối logic lập trình được chứa các LUTs và các phần tử nhớ flip-flop có thể được cấu trúc thực hiện các hàm khác nhau.

Mỗi CLB được cấu thành từ 4 Slices, mỗi Slice lại được cấu thành từ 2 LUT (Look Up Tables). Phân bố của các CLB thể hiện ở hình 3.4:

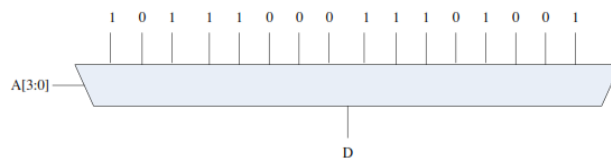
Các CLB được phân bố theo hàng và theo cột, mỗi một CLB được xác định bằng một tọa độ X và Y trong ma trận

SLICE

Mỗi CLB được cấu tạo thành từ 4 slices và các slices này chia làm hai nhóm trái và phải. Nhóm 2 slices bên trái có khả năng thực hiện các chức năng logic và làm việc như phần tử nhớ nên được gọi là SLICEM (SLICE Memory). Nhóm 2 slices bên phải chỉ thực hiện được các chức năng logic nên được gọi là SLICEL (SLICE Logic). Thiết kế như vậy xuất phát từ thực tế là nhu cầu thực hiện chức năng logic thường lớn hơn so với nhu cầu lưu trữ dữ liệu, do đó việc hỗ trợ chỉ một nửa làm việc như phần tử nhớ làm giảm kích thước và chi phí cho FPGA, mặt khác làm tăng tốc độ làm việc cho toàn khối

LUT

Bảng tham chiếu (Look-Up Table) gọi tắt là các LUT được phân bố ở góc trên trái và góc dưới phải của Slice và được gọi tên tương ứng là F-LUT và G-LUT. Phần tử nhớ đóng vai trò là đầu ra của các LUT được gọi tương ứng là Flip-Flop X (FFX) và Flip-Flop Y (FFY). LUT là đơn vị logic và là tài nguyên logic cơ bản của FPGA, LUT có khả năng được cấu trúc để thực hiện một hàm logic bất kỳ với 4 đầu vào. Cấu trúc của LUT được thể hiện ở hình sau:



Hình 4-9. Cấu trúc của LUT

LUT bản chất là một bộ chọn kênh 16 đầu vào, các đầu vào của LUT A[3:0] đóng vai trò tín hiệu chọn kênh, đầu ra của LUT là đầu ra của bộ chọn kênh. Khi cần thực hiện một hàm logic bất kỳ nào đó, một bảng nhớ SRAM 16 bit được tạo để lưu trữ kết quả bảng chân lý của hàm, tổ hợp 16 giá trị của hàm tương ứng sẽ là các kênh chọn của khối chọn kênh. Khi làm việc tùy vào giá trị của A[3:0] đầu ra D sẽ nhận một trong số 16 giá trị lưu trữ tương ứng trong SRAM. Bằng cách đó một hàm logic bất kỳ với 4 đầu vào 1 đầu ra có thể thực hiện được trên LUT. Trong cấu trúc của Slice có chứa hai bộ chọn kênh đặc biệt gọi là Bộ chọn kênh mở rộng - Wide-multiplexer F5MUX và FiMUX.

Hình 4-10. FiMUX và F5MUX

Mỗi một LUT được thiết kế để có thể thực hiện được mọi hàm logic 4 đầu vào. Mục đích của các bộ chọn kênh này là tăng tính linh động của FPGA bằng cách kết hợp các phần tử logic chức năng như LUT, chuỗi bit nhớ, Thanh ghi dịch, RAM phân tán ở các Slices, CLB khác nhau để tạo ra các hàm tùy biến với nhiều đầu vào hơn. Ví dụ ở bảng sau thể hiện cách sử dụng 2 LUT 4 đầu vào và 1 F5MUX để tạo ra một hàm logic tùy biến 5 đầu vào.

Hình 4-11. Nguyên lý làm việc của F5MUX

Đầu tiên đối với hàm 5 biến $OUT = F(X1, X2, X3, X4, X5)$ bất kỳ ta thành lập bảng chân lý tương ứng, bảng này được chia làm hai phần, phần trên với tất cả các giá trị của $X5$ bằng 0, ta gọi hàm này có tên là: $OUT0 = F(X1, X2, X3, X4, 0) = F0(X1, X2, X3, X4)$ phần dưới với tất cả các giá trị của $X5$ bằng 1, ta gọi hàm này có tên là: $OUT1 = F(X1, X2, X3, X4, 1) = F1(X1, X2, X3, X4)$

Hai hàm $F1, F2$ là các hàm 4 đầu vào được thực hiện ở tương ứng bởi LUT1, LUT2. Tín hiệu $X5$ được sử dụng làm tín hiệu chọn kênh cho F5MUX chọn 1 trong hai giá trị đầu ra của LUT1, LUT2, đầu ra của F5MUX chính là kết quả của hàm 5 biến cần thực hiện. $OUT = F0(X1, X2, X3, X4)$ nếu $X5 = 0 = F1(X1, X2, X3, X4)$ nếu $X5 = 1$

Hình 4-12. Cấu tạo của F5MUX

F5MUX được thiết kế dựa trên nguyên lý trên nhưng trên FPGA thực tế ngoài cổng ra thông thường O theo đó kết quả gửi ra phần tử nhớ của CLB, thì kết quả còn được gửi ra tín hiệu trả về LO (Local Output) theo đó kết quả có thể được gửi ngược lại các FiMUX để tiếp tục thực hiện các hàm logic có nhiều cổng vào hơn. Tương tự như vậy có thể thành lập các hàm với số lượng đầu vào lớn hơn bằng 6, 7, 8 ... tương ứng FiMUX sẽ được gọi là F6MUX, F7MUX, F8MUX... Ví dụ 1 hàm 6 biến thì phải thực hiện bằng cách ghép nối 2 CLB liên tiếp thông qua F6MUX.

Ngoài thực hiện các hàm đầy đủ với sự kết hợp hai LUT để tạo ra hàm logic tùy biến 5 đầu vào thì có thể kết hợp để tạo ra các hàm logic không đầy đủ với 6, 7, 8, 9 đầu vào.

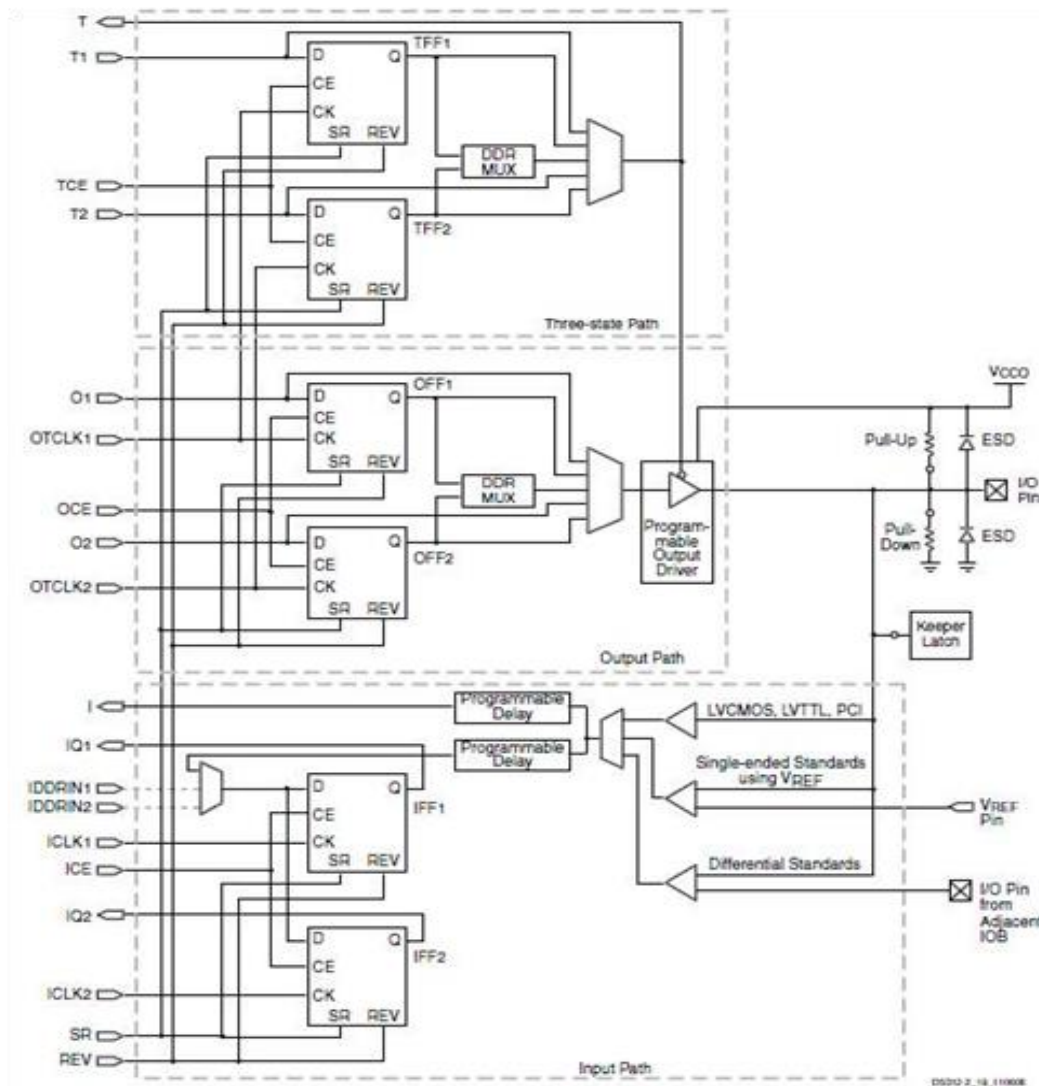
Câu 24. Trình bày về thiết kế chuỗi bit nhớ (Carry Chain), chuỗi số học (Arithmetic Chain) trong FPGA. Trình bày cấu trúc của IOB trong FPGA, khối làm trễ khả trình và ứng dụng, khái niệm DDR.

Trả lời:

Carry chain là chuỗi bit nhớ thường gặp trong phép toán cộng, với mỗi slice chuỗi bit nhớ được bắt đầu từ tín hiệu CIN và kết thúc ở COUT. Các chuỗi đơn lẻ trong có thể được nối trực tiếp giữa các CLB với nhau để tạo thành các chuỗi dài hơn theo yêu cầu. Mỗi một chuỗi bit nhớ này có thể được bắt đầu tại bất kỳ một đầu vào BY hoặc BY nào của các Slices

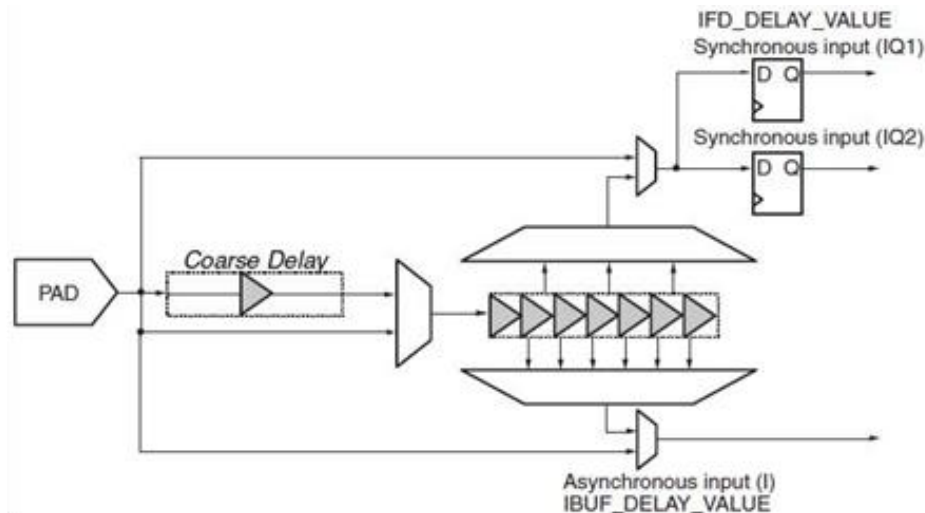
Các chuỗi số học logic (Arithmetic Chain) bao gồm chuỗi thực hiện hàm XOR với các cổng XORG, XORF phân bố ở phần trên và phần dưới của Slice, chuỗi AND với các cổng GAND, FAND. Các chuỗi này kết hợp với các LUT để thực hiện phép nhân hoặc tạo thành các bộ đếm nhị phân

cấu trúc của IOB trong FPGA: Các khối Input/Output Blocks (IOB) trong FPGA cung cấp các cổng vào ra lập trình được một chiều hoặc hai chiều giữa các chân vào ra của FPGA (package pin) với các khối logic bên trong. Các khối một chiều là các khối Input- only nghĩa là chỉ đóng vai trò cổng vào, số lượng của các cổng này thường chiếm không nhiều khoảng 25% trên tổng số tài nguyên IOB của FPGA.



Khái niệm DDR: chỉ dạng đường truyền dữ liệu đồng bộ ở tốc độ gấp 2 lần tốc độ cho phép của xung nhịp đồng hồ bằng cách kích hoạt tại cả thời điểm sườn lên và sườn xuống của xung nhịp. Với cả 3 đường dữ liệu có trong IOB, mỗi đường đều có một cặp phân tử nhớ cho phép thực hiện truyền dữ liệu theo phương thức DDR.

Programmable input delay block: Mỗi một Input path chứa các khối làm trễ lập trình được gọi là programmable input delay block. Các khối này bao gồm một phần tử làm trễ thô (Coarse delay) có thể được bỏ qua, khối này làm trễ tín hiệu ở mức độ chính xác vừa phải. Tiếp theo là chuỗi 6 phần tử làm trễ được điều khiển bởi các bộ chọn kênh. Đối với đường vào đồng bộ thông qua các phân tử nhớ tới IQ1, IQ2 thì có thể chọn 3 mức làm trễ. Còn đối với đường vào không đồng bộ tới cổng I thì có thể thay đổi ở 6 mức làm trễ. Tất cả khối làm trễ có thể được bỏ qua, khi đó tín hiệu được gửi đồng thời tới các chân ra đồng bộ và không đồng bộ.



Hình 3.25. Programmable input delay block

ứng dụng : của khối làm trễ là đảm bảo cho thời gian hold time, là thời gian tối thiểu cần giữ ổn định dữ liệu sau thời điểm kích hoạt của xung nhịp đồng bộ để đảm bảo cho phần tử nhớ hoạt động đúng.

Câu 25. Các dạng tài nguyên kết nối có trong FPGA. Các thành phần Block RAM, Dedicated Multiplier, DCM trong FPGA đặc điểm và ứng dụng.

Trả lời:

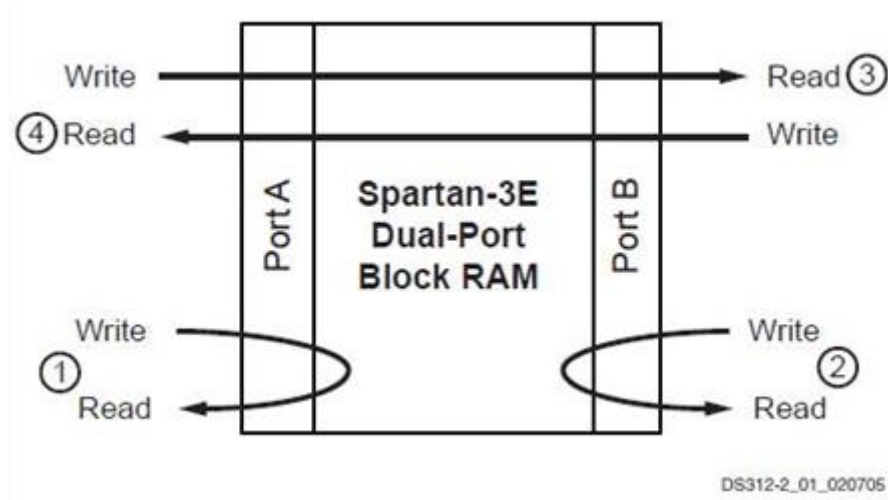
Các dạng tài nguyên kết nối có trong FPGA: Hệ thống kết nối của FPGA dùng để liên kết các phần tử chức năng khác nhau bao gồm IOB, CLB, Block RAM, Dedicated Multipliers, DCM với nhau. Hệ thống kết nối của FPGA được thiết kế cân bằng giữa yếu tố linh động và tốc độ làm việc (giảm thiểu trễ do đường truyền gây ra). Đối với các FPGA họ Spartan 3E có 4 loại kết nối sau:

- kết nối xa (long lines),
- kết nối kép (double lines),
- kết nối hex (hex lines),
- kết nối trực tiếp (direct line).

Các dạng kết nối này liên hệ với nhau thông qua cấu trúc ma trận chuyển (switch matrix).

Các thành phần Block RAM, Dedicated, Multiplier, DCM trong FPGA đặc điểm và ứng dụng.

Bên cạnh nguồn tài nguyên lưu trữ dữ liệu như trình bày ở trên là Distributed RAM với bản chất là một hình thức sử dụng của LUT thì trong Xilinx FPGA còn được tích hợp các Block RAM riêng biệt được cấu hình như một khối RAM hai cổng, số lượng này trong Spartan 3E thay đổi từ 4 đến 36 tùy theo từng IC cụ thể. Tất cả Block RAM hoạt động đồng bộ và có khả năng lưu trữ tập trung một khối lượng lớn thông tin. Cấu trúc bên trong của một Block RAM như sau:



Khối RAM có hai cổng A và B vào ra cho phép thực hiện các thao tác đọc ghi độc lập với nhau, mỗi một cổng có các tín hiệu xung nhịp đồng bộ, bus dữ liệu và các tín hiệu điều khiển riêng. Có 4 đường dữ liệu cơ bản như sau:

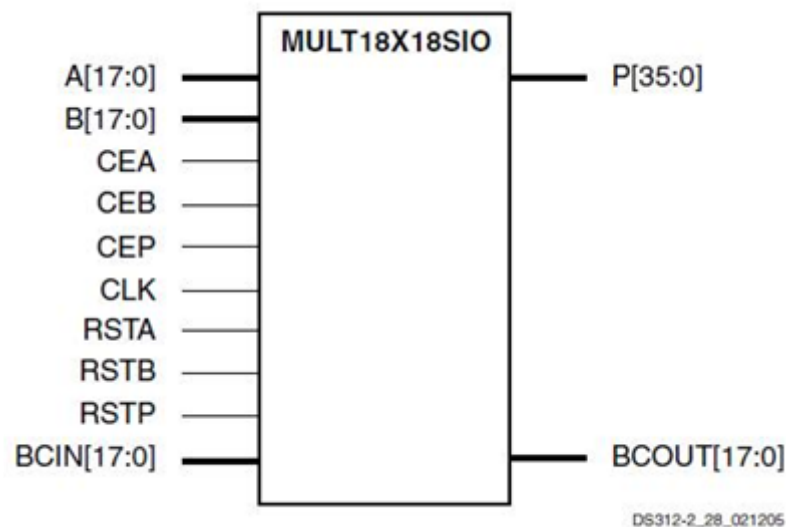
1. Đọc ghi cổng A
2. Đọc ghi cổng B
3. Truyền dữ liệu từ A sang B
4. Truyền dữ liệu từ B sang A.

Các thành phần Dedicated Multiplier, DCM trong FPGA đặc điểm và ứng dụng

Dedicated Multiplier là các khối nhân 18bit x 18bit được thiết kế riêng đặc biệt, thường được ứng dụng trong các bài toán xử lý tín hiệu số, ký hiệu là MULT18X18SIO trong thư viện chuẩn của Xilinx.

Dedicated Multiplier được đặt tại các vị trí sát với các Block RAM nhằm kết hợp hai khối này cho những tính toán lớn với tốc độ cao. Số lượng của Dedicated Multipliers bằng với số lượng của Block RAMs trong FPGA, ngoài ra hai thành phần này còn chia sẻ với nhau các cổng A, B 16 bit dùng chung..

Dedicated Multiplier thực hiện phép nhân hai số 18 bit có dấu, kết quả là một số 36 bit có dấu. Phép nhân không dấu được thực hiện bằng cách giới hạn miền của số nhân và số bị nhân. Mô tả các cổng vào ra của phần tử nhân MULT18X18SIO thể hiện ở hình sau:



Hình 3.36 Interface of Dedicated Multiplier primitive

Các thành phần DCM trong FPGA đặc điểm và ứng dụng

DCM (Digital Clock Manager) Các khối làm nhiệm vụ điều chỉnh, phân phối tín hiệu đồng bộ tới tất cả các khối khác. DCM thường được phân bố ở giữa, với hai khối ở trên và hai khối ở dưới

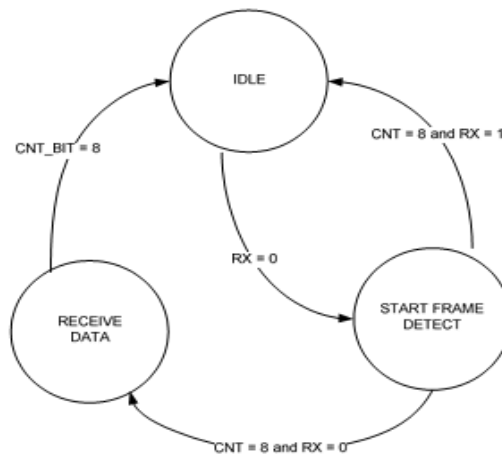
Để tạo ra hai xung nhịp lệch pha nhau có thể dùng khối DCM (Digital Clock Manager) từ một tín hiệu xung nhịp chuẩn sinh ra tín hiệu xung nhịp thứ hai bằng cách dịch pha 180o(hình bên trái). Phương pháp này đạt được độ trễ

xung nhịp (Clock scew) thấp nhất. Bên cạnh đó phương pháp thứ hai như mô tả ở hình bên phải là dùng cổng đảo có trong IOB để tạo lệch pha 180o

Câu 26. Trình bày sơ thuật toán và sơ đồ cấu trúc khối truyền nhận thông tin nối tiếp (UART)

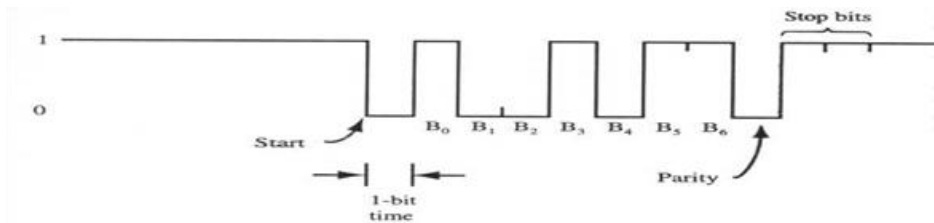
FSM (Finish-state machine) máy trạng thái hữu hạn là mạch có hữu hạn các trạng thái và hoạt động của mạch là sự chuyển đổi có điều kiện giữa các trạng thái đó.

FSM rất hay được sử dụng trong các bài toán số phức tạp, để thực hiện mô tả này đầu tiên người thiết kế phải phân tích thống kê các trạng thái cần có thể của mạch, tối giản và tìm điều kiện chuyển đổi giữa các trạng thái, vẽ giản đồ chuyển trạng thái có dạng như sau.



Hình 2.19: Sơ đồ trạng thái bộ thu UART đơn giản

Sơ đồ thể hiện cách chuyển trạng thái của một khối nhận dữ liệu từ đường truyền UART một cách đơn giản nhất.



Hình 2.20: Tín hiệu vào Rx của khối thu UART

Ví dụ dữ liệu đầu vào của một đường truyền UART như hình vẽ, khi ở trạng thái không truyền dữ liệu thì khối tín hiệu RX ở mức cao và khối nhận nằm ở trạng thái chờ IDLE. Khi RX chuyển từ mức cao xuống mức thấp thì khối nhận chuyển sang trạng thái thứ hai gọi là trạng thái dò tín hiệu start, START_FRAME_DETEC. Bit START được xem là hợp lệ nếu như mức 0 của RX được giữ trong một khoảng thời gian đủ lâu xác định, khối nhận sẽ dùng một bộ đếm để xác nhận sự kiện này, nếu bộ đếm đếm CNT đến 8 mà RX bằng 0 thì sẽ chuyển sang trạng thái tiếp theo là trạng thái nhận dữ liệu RECEIVE DATA. Nếu CNT = 8 mà RX = 1 thì đây không phải bit START, khối nhận sẽ quay về trạng thái chờ IDLE. Ở trạng thái nhận dữ liệu thì khối này nhận liên tiếp một số lượng bit nhất định, thường là 8 bit, khi đó CNT_BIT = 8, sau đó sẽ trở về trạng thái chờ. IDLE.

Mô tả VHDL của khối FSM trên như sau:

----- Simply UART FSM -----

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

entity fsm_receiver is generic (n: positive := 8);

```

port(
    cnt_bit      : in std_logic_vector (3 downto 0);
    cnt8         : in std_logic_vector (2 downto 0);
    Rx           : in std_logic;
    CLK          : in std_logic;
    reset        : in std_logic;
    receiver_state : inout std_logic_vector (1 downto 0)
);
end fsm_receiver;

```

architecture behavioral of fsm_receiver is

```

    constant Idle          : std_logic_vector (1 downto 0) := "00";
    constant start_frame_detect : std_logic_vector (1 downto 0) := "01";
    constant receive_data      : std_logic_vector (1 downto 0) := "11";
begin
    receiving: process (CLK, RESET)
    begin
        if RESET = '1' then
            receiver_state <= Idle;
        elsif clk = '1' and clk'event then
            case receiver_state is
            when Idle =>
                if Rx = '0' then
                    receiver_state <= start_frame_detect;
                end if;
            when start_frame_detect =>
                if cnt8 = "111" then
                    if Rx = '0' then
                        receiver_state <= receive_data;
                    else
                        receiver_state <= Idle;
                    end if;
                end if;
            when receive_data =>
                if cnt_bit = "0111" then
                    receiver_state <= Idle;
                end if;
            end case;
        end if;
    end process;
end fsm_receiver;

```

```

when others =>
    receiver_state <= Idle;
end case;
end if;
end process receiving;
end behavioral;

```

Với mô tả như trên trạng thái của mạch được lưu bằng hai بیت của thanh ghi receiver_state, trạng thái của mạch sẽ thay đổi đồng bộ với xung nhịp hệ thống CLK.

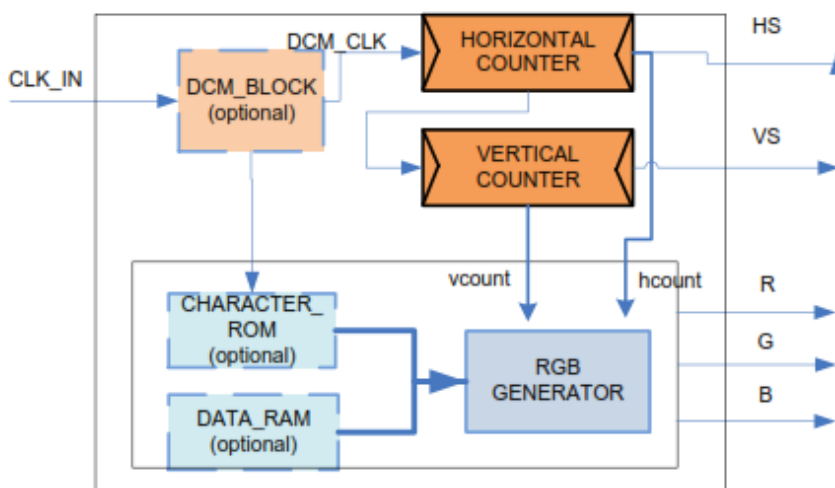
(*) Code trên dùng để minh họa cách viết FSM, trên thực tế khối điều khiển trạng thái bộ nhận UART phức tạp hơn do còn phần điều khiển trạng thái các bộ đếm, thanh ghi.

Câu 27. Trình bày sơ thuật toán và sơ đồ cấu trúc khối giao tiếp VGA .

Trả lời:

Sơ đồ khối điều khiển VGA

Theo như lý thuyết ở trên thì việc điều khiển VGA tương ứng với việc tạo ra xung các xung quét VS và HS theo đúng các yêu cầu về mặt thời gian. Cách đơn giản nhất là sử dụng hai bộ đếm được ghép nối tiếp. Sơ đồ khối thiết kế như sau:

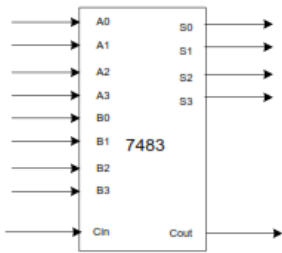


Hình 4-76 Sơ đồ khối điều khiển VGA

Phần bài tập TK - LGS

Yêu cầu: Phần bài tập học sinh thực hiện trên máy tính, mục đích kiểm tra kỹ năng về thực hành, bài làm hoàn chỉnh là bài có giản đồ sóng thể hiện mạch làm việc đúng với một tổ hợp giá trị đầu vào bất kỳ nào đó theo yêu cầu của giáo viên.

Câu 1. Thiết kế full_adder trên VHDL, trên cơ sở đó thiết kế bộ cộng 4 bit tương tự IC 7483.



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity full_adder is
port (
    a : in std_logic;
    b : in std_logic;
    cin : in std_logic;
    s : out std_logic;
    cout : out std_logic
);
end full_adder ;
architecture dataflow of full_adder is
begin
    s <= a xor b xor cin;
    cout <=(a and b) or (cin and (a or b));
end dataflow;
```

```
-----
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity adder4 is
port(
    a: in std_logic_vector(3 downto 0);
    b: in std_logic_vector(3 downto 0);
    ci: in std_logic;
    sum: out std_logic_vector(3 downto 0);
    co: out std_logic
```

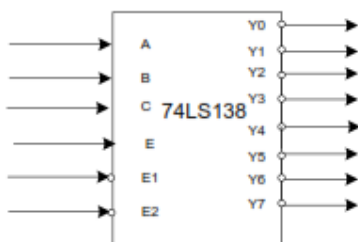
```

    );
end adder4;

architecture structure of adder4 is
    signal c: std_logic_vector(2 downto 0);
    component full_adder
    port(
        a,b,cin : in std_logic;
        s,cout : out std_logic
    );
end component;
begin
    u0:component full_adder
    port map(a => a(0), b => b(0), cin => ci, s=> sum(0), cout => c(0));
    u1:component full_adder
    port map(a => a(1), b => b(1), cin => c(0), s=> sum(1), cout => c(1));
    u2:component full_adder
    port map(a => a(2), b => b(2), cin => c(1), s=> sum(2), cout => c(2));
    u3:component full_adder
    port map(a => a(3), b => b(3), cin => c(2), s=> sum(3), cout => co);
end structure;

```

Câu 2. Thiết kế bộ giải mã nhị phân 3_to_8 có đầu ra thuận, nghịch tương tự IC 74LS138.



Inputs					Outputs							
Enable		Select										
G1	G2 (Note 1)	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-----

entity gm38 is
    port(
        x : in std_logic_vector(2 downto 0);
        y : out std_logic_vector(7 downto 0);
    );
end entity gm38;

```

```

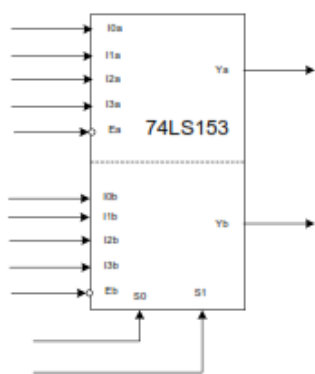
        e : in std_logic          );
end entity;

-----

architecture behavioral of gm38 is
begin
process(x,e)
begin
    if e = '0' then
        y <= "11111111";
    else
        case x is
            when "000" => y <= "11111110";
            when "001" => y <= "11111101";
            when "010" => y <= "11111011";
            when "011" => y <= "11110111";
            when "100" => y <= "11101111";
            when "101" => y <= "11011111";
            when "110" => y <= "10111111";
            when others => y <= "01111111";
        end case;
    end if;
end process;
end behavioral;

```

Câu 3. Thiết bộ chọn kênh 4 đầu vào 1 đầu ra MUX4_1 tương tự IC 74153 nhưng chỉ hỗ trợ một kênh chọn (IC này có hai kênh chọn riêng biệt như hình vẽ)



Ngõ chọn		Cho phép	Ngõ vào dẫn kênh				Ngõ ra Z
S1	S0		I0	I1	I2	I3	
X	X	1	X	X	X	X	0
0	0	0	0	X	X	X	0
0	0	0	1	X	X	X	1
0	1	0	X	0	X	X	0
0	1	0	X	1	X	X	1
1	0	0	X	X	0	X	0
1	0	0	X	X	1	X	1
1	1	0	X	X	X	0	0
1	1	0	X	X	X	1	1

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

```

```

entity mux is
port(
    x0:in std_logic;
        x1:in std_logic;
        x2:in std_logic;
        x3:in std_logic;
        e:in std_logic;
        s:in std_logic_vector(1 downto 0);
        y:out std_logic
);
end entity;

```

architecture trang of mux is

```

begin
    process(x0,x1,x2,x3,e,s)
    begin
        if e = '1' then
            y <='0';
        else
            case s is
                when "00" => y <= x0;
                when "01" => y <= x1;
                when "10" => y <= x2;
                when others => y <= x3;
            end case;
        end if;
    end process;
end trang;

```

Câu 4. Thiết bộ phân kênh 1 đầu vào 4 đầu ra DEMUX1_4.

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity DEMUX is
    port (
        s: in std_logic_vector(1 downto 0);
        x : in std_logic_vector(3 downto 0);
        y1: out std_logic_vector(3 downto 0);

```

```

        y2: out std_logic_vector(3 downto 0);
        y3: out std_logic_vector(3 downto 0);
        y4: out std_logic_vector(3 downto 0)
    );

end DEMUX;

architecture behavioral of DEMUX is
begin
    process(s,x)
    begin
        case s is
            when "00" => y1 <= x;
            when "01" => y2 <= x;
            when "10" => y3 <= x;
            when others => y4 <= x;
        end case;
    end process;
end behavioral;

```

Câu 5. Thiết kế bộ cộng/ trừ 4 bit sử dụng toán tử cộng trên VHDL.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
-----
entity congtru4 is
port(
    a,b : in std_logic_vector(3 downto 0);
    cin,sub :in std_logic;
    cout : out std_logic;
    sum : out std_logic_vector(3 downto 0)
);
end entity;
-----
architecture behavioral of congtru4 is
signal a_temp : std_logic_vector(4 downto 0);
signal b_temp : std_logic_vector(4 downto 0);
signal s_temp : std_logic_vector(4 downto 0);
begin
    process(a,b,cin,sub)

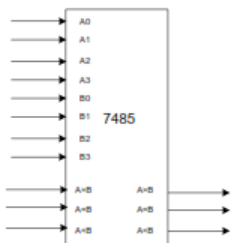
```

```

begin
    a_temp <= '0' & a;
    b_temp <= '0' & b;
    if sub = '0' then
        s_temp <= a_temp + b_temp + cin;
        sum <= s_temp(3 downto 0);
        cout <= s_temp(4);
    else
        s_temp <= a_temp + not(b_temp) + 1;
        sum <= s_temp(3 downto 0);
        cout <= s_temp(4);
    end if;
end process;
end behavioral;

```

Câu 6. Thiết kế bộ so sánh hai số không dấu 4 bit tương tự IC 7485.



Các ngõ vào so sánh				Ngõ vào nối chổng			Ngõ ra		
A3, B3	A2, B2	A1, B1	A0, B0	A>B	A<B	A=B	A>B	A<B	A=B
A ₃ >B ₃	X	X	X	X	X	X	1	0	0
A ₃ <B ₃	X	X	X	X	X	X	0	1	0
A ₃ =B ₃	A ₂ >B ₂	X	X	X	X	X	1	0	0
A ₃ =B ₃	A ₂ <B ₂	X	X	X	X	X	0	1	0
A ₃ =B ₃	A ₂ =B ₂	A ₁ >B ₁	X	X	X	X	1	0	0
A ₃ =B ₃	A ₂ =B ₂	A ₁ <B ₁	X	X	X	X	0	1	0
A ₃ =B ₃	A ₂ =B ₂	A ₁ =B ₁	A ₀ >B ₀	X	X	X	1	0	0
A ₃ =B ₃	A ₂ =B ₂	A ₁ =B ₁	A ₀ <B ₀	X	X	X	0	1	0
A ₃ =B ₃	A ₂ =B ₂	A ₁ =B ₁	A ₀ =B ₀	1	0	0	1	0	0
A ₃ =B ₃	A ₂ =B ₂	A ₁ =B ₁	A ₀ =B ₀	0	1	0	0	1	0
A ₃ =B ₃	A ₂ =B ₂	A ₁ =B ₁	A ₀ =B ₀	X	X	1	0	0	1
A ₃ =B ₃	A ₂ =B ₂	A ₁ =B ₁	A ₀ =B ₀	1	1	0	0	0	0
A ₃ =B ₃	A ₂ =B ₂	A ₁ =B ₁	A ₀ =B ₀	0	0	0	1	1	0

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity ss is
    port(a:in std_logic_vector(3 downto 0);
         b:in std_logic_vector(3 downto 0);
         --g1:in std_logic;
         --g2:in std_logic;
         --g3:in std_logic;
         lon :out std_logic;
         nho :out std_logic;
         bang :out std_logic);

```

end entity;

architecture behavioral of ss is

begin

process (a,b)

begin

if ((a(3)=b(3)) and (a(2)=b(2)) and (a(1)=b(1)) and (a(0)=b(0))) then bang<='1';

elsif ((a(3)=b(3)) and (a(2)=b(2)) and (a(1)=b(1))) then

if a(0)>b(0) then lon <='1';

else nho <='1';

end if;

elsif ((a(3)=b(3)) and (a(2)=b(2))) then

if a(1)>b(1) then lon<='1';

else nho<='1';

end if;

elsif (a(3)=b(3)) then

if(a(2)>b(2)) then lon<='1';

else nho<='1';

end if;

elsif (a(3)>b(3)) then lon<='1';

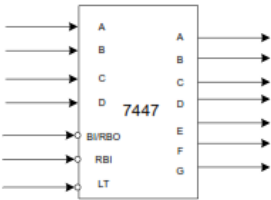
elsif (a(3)<b(3)) then nho<='1';

end if;

end process;

end behavioral;

Câu 7. Thiết kế các bộ chuyển đổi mã từ NBCD – 7-SEG(LED 7 đoạn) tương tự IC 7447, hỗ trợ cổng LamTest, khi cổng này có giá trị bằng 1, tất cả đèn phải sáng không phụ thuộc mã đầu NBCD đầu vào. Để đơn giản, các chân RBI, RBO không cần thiết kế.



Decimal or Function	Inputs							Outputs							Note
	\overline{LT}	\overline{RBI}	A3	A2	A1	A0	$\overline{BI/RBO}$	\overline{a}	\overline{b}	\overline{c}	\overline{d}	\overline{e}	\overline{f}	\overline{g}	
0	H	H	L	L	L	L	H	L	L	L	L	L	L	H	(Note 2)
1	H	X	L	L	L	H	H	H	L	L	H	H	H	H	(Note 2)
2	H	X	L	L	H	L	H	L	L	H	L	L	H	L	
3	H	X	L	L	H	H	H	L	L	L	L	H	H	L	
4	H	X	L	H	L	L	H	H	L	L	H	H	L	L	
5	H	X	L	H	L	H	H	L	H	L	L	H	L	L	
6	H	X	L	H	H	L	H	H	H	L	L	L	L	L	
7	H	X	L	H	H	H	H	L	L	L	H	H	H	H	
8	H	X	H	L	L	L	H	L	L	L	L	L	L	L	
9	H	X	H	L	L	H	H	L	L	L	H	H	L	L	
10	H	X	H	L	H	L	H	H	H	H	L	L	H	L	
11	H	X	H	L	H	H	H	H	H	L	L	H	H	L	
12	H	X	H	H	L	L	H	H	L	H	H	H	L	L	
13	H	X	H	H	L	H	H	L	H	H	L	H	L	L	
14	H	X	H	H	H	L	H	H	H	H	L	L	L	L	
15	H	X	H	H	H	H	H	H	H	H	H	H	H	H	
\overline{BI}	X	X	X	X	X	X	L	H	H	H	H	H	H	H	(Note 3)
\overline{RBI}	H	L	L	L	L	L	L	H	H	H	H	H	H	H	(Note 4)
\overline{LT}	L	X	X	X	X	X	H	L	L	L	L	L	L	L	(Note 5)


```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

-----

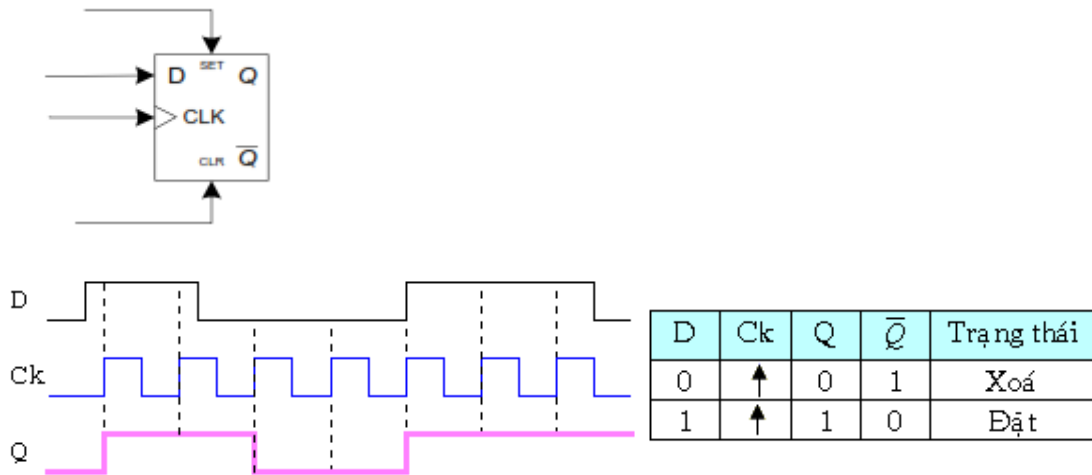
entity NBCD is
port(
    x: in std_logic_vector(3 downto 0);
    lt : in std_logic;
    y: out std_logic_vector(6 downto 0)

);
end entity;

-----

architecture behavioral of NBCD is
begin
process(x,lt)
begin
    if lt = '1' then
        y <= "1111111";
    else
        case x is
            when "0000" => y <= "1111110";
            when "0001" => y <= "0110000";
            when "0010" => y <= "1101101";
            when "0011" => y <= "1111001";
            when "0100" => y <= "0110011";
            when "0101" => y <= "1011011";
            when "0110" => y <= "0011111";
            when "0111" => y <= "1110000";
            when "1000" => y <= "1111111";
            when "1001" => y <= "1110011";
            when others => y <= "0000000";
        end case;
    end if;
end process;
end behavior;
```

Câu 8. Thiết kế các flip-flop đồng bộ D.



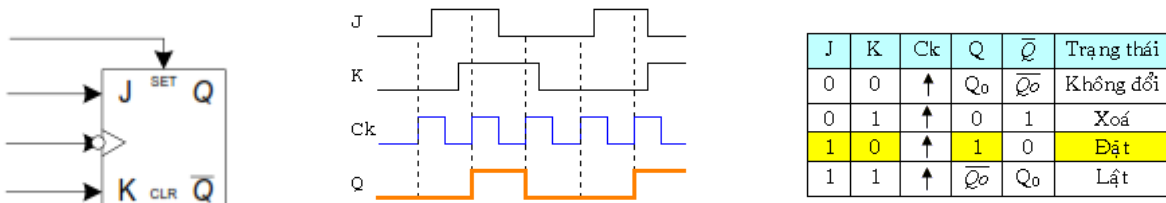
```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity dff is
port(
    d      : in std_logic;
    clk    : in std_logic;
    clr : in std_logic;
    set    : in std_logic;
    q      : inout std_logic;
    notq: inout std_logic
);
end dff;
architecture behavioral of dff is
begin
    notq<=not q;
    process(d,clk,clr,set,q,notq)
    begin
        if clr ='1' then
            q<='0';
        elsif set ='1' then
            q<='1';
        elsif clk ='1' and clk'event then
            q<=d;
        end if;
    end process;
end process;

```

end behavioral;

Câu 9. Thiết kế các flip-flop đồng bộ JK.



Hình 3.1.15 Dạng sóng minh họa cho FF JK

-----JK-FF example -----

library ieee;

use ieee.std_logic_1164.all;

entity jk_ff is

port (

J, K :in std_logic; -- Data input

clk :in std_logic; -- Clock input

q :buffer std_logic -- Q output

);

end entity jk_ff;

architecture behavioral of jk_ff is

begin

process (clk, J, K) begin

if (clk'event and clk = '1') then

if (J = '0' and K = '0') then

q <= q;

elsif (J = '1' and K = '0') then

q <= '1';

elsif (J = '0' and K = '1') then

q <= '0';

elsif (J = '1' and K = '1') then

q <= not(q);

end if;

end if;

end process;

end architecture behavioral;

Câu 10. Thiết kế trên VHDL khối dịch trái qua phải 32-bit, số lượng bit dịch là một số nguyên từ 1-31 trên VHDL (sử dụng toán tử dịch).

```
library ieee;

use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

-----

entity dichphai32 is
port(
    shift_in  : in std_logic_vector( 31 downto 0);
    shift_out  : out std_logic_vector( 31 downto 0);
    shift_value : in std_logic_vector( 4 downto 0)
);
end entity;

-----

architecture dataflow of dichphai32 is
signal shi      : bit_vector( 31 downto 0);
signal sho      : bit_vector( 31 downto 0);
signal sa : integer;
begin
    shi      <= to_bitvector( shift_in);
    sa      <= conv_integer( '0' & shift_value);
    sho      <= shi srl sa;
    shift_out<= to_stdlogicvector( sho);
end dataflow;
```

Câu 11. Thiết kế thanh ghi dịch đồng bộ nối tiếp 4 bit sang bên trái, với đầu vào nối tiếp SL, hỗ trợ tín hiệu Reset không đồng bộ và tín hiệu Enable.

```
library ieee;

use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

-----

entity dichtrai4 is
port(
    sl,reset,e,clk : in std_logic;
    q_out : inout std_logic_vector(3 downto 0)
);
```

```
end entity;
```

```
-----
```

```
architecture behavioral of dichtrai4 is
```

```
signal q_temp :std_logic_vector(3 downto 0);
```

```
begin
```

```
process(sl,e,reset,clk)
```

```
begin
```

```
    if reset = '0' then q_temp <= "0000";
```

```
    elsif e = '1' then
```

```
        if rising_edge(clk) then
```

```
            q_temp <= sl & q_temp(2 downto 0);
```

```
        end if;
```

```
    end if;
```

```
end process;
```

```
    q_out <= q_temp;
```

```
end behavioral;
```

Câu 12. Thiết kế thanh ghi dịch đồng bộ nối tiếp 4 bit sang bên phải, với đầu vào nối tiếp SR, hỗ trợ tín hiệu Reset không đồng bộ và tín hiệu Enable.

```
library ieee;
```

```
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_arith.all;
```

```
use ieee.std_logic_unsigned.all;
```

```
use ieee.numeric_std.all;
```

```
-----
```

```
entity dichphai4 is
```

```
port(
```

```
    sl,reset,e,clk : in std_logic;
```

```
    q_out : out std_logic_vector(3 downto 0)
```

```
);
```

```
end entity;
```

```
-----
```

```
architecture behavioral of dichphai4 is
```

```
signal q_temp :std_logic_vector(3 downto 0);
```

```
begin
```

```
process(sl,e,reset,clk)
```

```
begin
```

```
    if reset = '0' then q_temp <= "0000";
```

```
    elsif e = '1' then
```

```

    if rising_edge(clk) then
        q_temp <= s1 & q_temp(3 downto 1);
    end if;
end if;

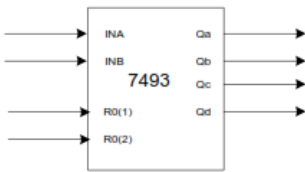
end process;

q_out <= q_temp;

end behavioral;

```

Câu 13. Thiết kế IC đếm nhị phân theo cấu trúc của IC 7493, IC được cấu thành từ một bộ đếm 2 và 1 bộ đếm 8 có thể làm việc độc lập hoặc kết hợp với nhau.



```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-----

entity ic7493 is
port(
    r          :in std_logic_vector(1 downto 0);
    clka,clkb   :in std_logic;
    qa         :out std_logic;
    qb         :out std_logic_vector(2 downto 0)
);
end entity;

-----

architecture behavioral of ic7493 is
    signal q_temp:std_logic_vector(3 downto 0):="0000";
begin
    process(clka,r)
    begin
        if r = "00" then q_temp(0) <= '0';
        elsif rising_edge(clka) then
            q_temp(0) <= not q_temp(0);
        end if;
    end process;

    process(clkb,r)

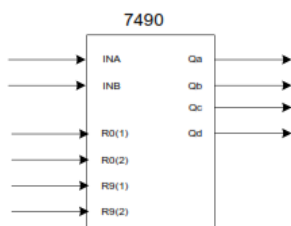
```

```

begin
    if r = "00" then q_temp( 3 downto 1) <= "000";
    elsif    rising_edge(clkb) then
        q_temp( 3 downto 1)    <= q_temp( 3 downto 1) + '1';
    end if;
end process;
qa <= q_temp(0);
qb <= q_temp( 3 downto 1);
end behavioral;

```

Câu 14. Thiết kế IC đếm theo cấu trúc của IC 7490, IC được cấu thành từ một bộ đếm 2 và 1 bộ đếm 5 có thể làm việc độc lập hoặc kết hợp với nhau để tạo thành bộ đếm thập phân.



```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
-----
entity ic7490 is
port(
    clka,clkb    : in std_logic;
    r0,r9        : in std_logic_vector(1 downto 0);
    qa          : inout std_logic;
    qb          : inout std_logic_vector(2 downto 0)
);
end entity;

```

```

-----
architecture behavioral of ic7490 is
signal q_temp : std_logic_vector(3 downto 0);
begin
process(clka,r0,r9)
begin

```

```

    if          r0 = "00" then q_temp(0) <= '0';
    elsif      r9 = "00" then q_temp(0) <= '1';
    elsif      rising_edge(clka) then

```

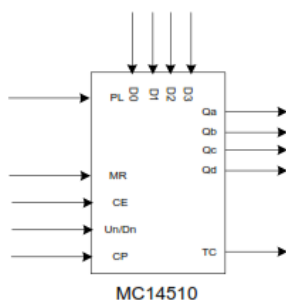


```

        q_temp(0) <= not q_temp(0);
    end if;
end process;
process(clkb,r0,r9)
begin
    if          r0 = "00" then q_temp(3 downto 1) <= "000";
    elsif      r9 = "00" then q_temp(3 downto 1) <= "100";
    elsif      rising_edge(clkb) then
        q_temp(3 downto 1) <= q_temp(3 downto 1) + '1';
        if q_temp(3 downto 1) = "100" then q_temp(3 downto 1) <= "000";
    end if;
    end if;
end process;
qa <= q_temp(0);
qb <= q_temp( 3 downto 1);
end behavioral;

```

Câu 15. Thiết kế IC đếm theo cấu trúc của IC MC14510, có khả năng đếm ngược, xuôi (up/Dn), đặt lại trạng thái (PL và D[3:0]), cho phép đếm (CE), Reset không đồng bộ (MR) như hình vẽ sau:



Câu 16. Thiết kế bộ đếm thập phân đồng bộ, RESET không đồng bộ, có tín hiệu ENABLE.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

```

```
entity dem10 is
port(
    cnt : out std_logic_vector(3 downto 0);
    r,e,clk : in std_logic
    );
end entity;

-----

architecture behavioral of dem10 is
signal cnt1 : std_logic_vector(3 downto 0):="0000";
begin
process(r,e,clk)
begin
    if r='0' then cnt1 <= "0000";
    elsif e='1' then
        if rising_edge(clk) then
            cnt1 <= cnt1 + 1;
            if cnt1 = "1001" then
                cnt1 <= "0000";
            end if;
        end if;
    end if;
end if;
end process;
cnt <= cnt1;
end behavioral;
```

Câu 17. Sử dụng bộ đếm đến 25 để thiết kế bộ chia tần từ tần số 50Hz thành 1Hz, tín hiệu tần số đưa ra có dạng đối xứng.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-----

entity chiatan is
port(
    clk      :in std_logic;
    clk1hz   :inout std_logic;
    r        :in std_logic
    );
end entity;

-----
```

```

architecture behavioral of chiatan is
signal count      :std_logic_vector( 4 downto 0) := "00000";
signal clk1       :std_logic:= '0';
begin
process(clk,r,count,clk1)
begin
    if (r = '1') then
        count <= "00000";
    elsif (rising_edge(clk)) then
        count <= count + 1;
        if count = "11001" then
            count <= "00000";
            clk1 <= not(clk1);
        end if;
    end if;
end process;
clk1hz <= clk1;
end behavioral;

```

Câu 18. Thiết kế khối mã hóa ưu tiên, đầu vào là chuỗi 4 bit đầu ra là mã nhị phân 2 bit thể hiện vị trí đầu tiên từ trái qua phải xuất hiện bit '1'. Trường hợp không có bit '1', thì đầu ra nhận giá trị không xác định. ("XX"). Các bit đánh số thứ tự từ 0 đến 3 từ phải qua trái.

```

library ieee;
use ieee.std_logic_1164.all;

-----

entity mahoal is
port(
    vao : in std_logic_vector(3 downto 0);
    ra  : out std_logic_vector(1 downto 0));
end entity;

-----

architecture behavioral of mahoal is
begin
process(vao)
begin
    if vao(3)= '1' then ra <= "11";
    elsif vao(2) = '1' then ra <= "10";
    elsif vao(1) = '1' then ra <= "01";
    elsif vao(0) = '1' then ra <= "00";
    else    ra<="XX";

```

```

        end if;

    end process;

end behavioral;

```

Câu 19. Thiết kế khối mã hóa ưu tiên, đầu vào là chuỗi 4 bit đầu ra là mã nhị phân 2 bit thể hiện vị trí đầu tiên từ trái qua phải xuất hiện bit '0'. Trường hợp không có bit '0', thì đầu ra nhận giá trị không xác định. ("XX"). Các bit đánh số thứ tự từ 0 đến 3 từ phải qua trái.

```

library ieee;

use ieee.std_logic_1164.all;

-----

entity mahoa0 is

port(

    vao : in std_logic_vector(3 downto 0);

    ra : out std_logic_vector(1 downto 0)

);

end entity;

```

```

-----

architecture behavioral of mahoa0 is

begin

process(vao)

begin

    if vao(3)= '0' then ra <= "11";
    elsif vao(2) = '0' then ra <= "10";
    elsif vao(1) = '0' then ra <= "01";
    elsif vao(0) = '0' then ra <= "00";
    else ra<="XX";

    end if;

end process;

end behavioral;

```

Câu 20. Thiết kế bộ mã hóa thập phân tương tự IC 74147 với 9 đầu vào và 4 đầu ra. Tại một thời điểm chỉ có 1 trong số 9 đầu vào tích cực. Giá trị 4 bit đầu ra là số thứ tự của đầu vào tích cực tương ứng. Nếu không đầu vào nào tích cực thì đầu ra bằng 0.

```

library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_arith.all;

use ieee.std_logic_unsigned.all;

-----

entity ic74147 is

port(

    vao : in std_logic_vector(8 downto 0);

    ra : out std_logic_vector(3 downto 0)

```

```

    );
end entity;

-----

architecture behavioral of ic74147 is
begin
process(vao)
begin
    case vao is
        when "100000000" => ra <= "1001";
        when "010000000" => ra <= "1000";
        when "001000000" => ra <= "0111";
        when "000100000" => ra <= "0110";
        when "000010000" => ra <= "0101";
        when "000001000" => ra <= "0100";
        when "000000100" => ra <= "0011";
        when "000000010" => ra <= "0010";
        when "000000001" => ra <= "0001";
        when "000000000" => ra <= "0000";
        when others => ra <= "XXXX";
    end case;
end process;
end behavioral;

```

Câu 21. Thiết kế khối PARITY có chức năng phát hiện tính chẵn lẻ của số lượng các bit bằng 0 trong một chuỗi 8 bit. Nếu số lượng bit 0 là chẵn thì kết quả đầu ra bằng 1, ngược lại bằng 0.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
---chan la chan 0 le la le 1
--parity chan tong chan =0 thi parity=0 le=1
--parity le tong chan thi parity=1 chan =0
entity parity is
port(
    a          : in std_logic_vector(7 downto 0);
    parity : out std_logic
);
end entity;

architecture behavioral of parity is

```

```

signal s : std_logic;

begin

process(a)

begin

    s<= (a(0) xor a(1) xor a(2) xor a(3) xor a(4) xor a(5) xor a(6) xor a(7));

end process;

    parity <= not s;

end behavioral;

```

Câu 22. Thiết kế khối PARITY có chức năng phát hiện tính chẵn lẻ của số lượng các bit bằng 1 trong một chuỗi 8 bit. Nếu số lượng bit 1 là chẵn thì kết quả đầu ra bằng 1, ngược lại bằng 0.

```

library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_unsigned.all;

use ieee.std_logic_arith.all;

---chan la chan 0 le la le 1

--parity chan tong chan =0 thi parity=0 le=1

--parity le tong chan thi parity=1 chan =0

entity parity is

port(

    a          : in std_logic_vector(7 downto 0);

    parity : out std_logic

);

end entity;

architecture behavioral of parity is

signal s : std_logic;

begin

process(a)

begin

    s<= (a(0) or a(1) or a(2) or a(3) or a(4) or a(5) or a(6) or a(7));

end process;

    parity <= not s;

end behavioral;

```

Câu 23. Thiết kế bộ cộng hai số NBCD 4 bit, kết quả là một số có hai chữ số thể hiện bằng mã NBCD.

```

library ieee;

use ieee.std_logic_1164.all;

use ieee.std_logic_arith.all;

use ieee.std_logic_unsigned.all;

```

```
-----  
entity NBCD is  
port( A: in std_logic_vector(3 downto 0);  
      B: in std_logic_vector(3 downto 0);  
      C: inout std_logic_vector(7 downto 0)  
      );  
end NBCD;
```

```
-----  
architecture cong of NBCD is  
signal A1: std_logic_vector(4 downto 0);  
signal B1: std_logic_vector(4 downto 0);  
signal Sum: std_logic_vector(4 downto 0);  
-----
```

```
begin  
process(A,B,sum)  
begin  
A1 <='0' & A;  
B1 <='0' & B;  
Sum <= A1 + B1;  
if( Sum <= 9) then  
    C(7 downto 4) <= "0000" ;  
    C(3 downto 0) <= Sum(3 downto 0);  
elsif ( sum > 9) then  
    C(7 downto 4) <= "0001";  
    C(3 downto 0) <= Sum(3 downto 0) + "0110";  
end if;  
end process;  
end cong;
```

(tài liệu chỉ mang tính chất tham khảo)

Chú ý: sự khác biệt giữa ‘x’ và “xx”