

HỌC VIỆN KỸ THUẬT QUÂN SỰ  
BỘ MÔN KỸ THUẬT XUNG SỐ, VI XỬ LÝ – KHOA VÔ TUYẾN ĐIỆN  
TỬ

# THIẾT KẾ LOGIC SỐ

(Dùng cho đối tượng đào tạo chính quy hệ quân sự và dân sự)

LUU HANH NOI BO

HÀ NỘI -2011



## LỜI GIỚI THIỆU

*Thiết kế logic số* là môn học kế tiếp của chương trình Điện tử số. Nội dung chính của chương trình môn học tập trung vào hai vấn đề kiến thức chính. Thứ nhất là bài toán *thiết kế về mặt chức năng* cho các khối số có mật độ tích hợp lớn cỡ LSI, VLSI và lớn hơn. Vấn đề thứ hai là giới thiệu căn bản về các *công nghệ* giúp hiện thực hóa thiết kế chức năng thành sản phẩm ứng dụng, trong đó tập trung chính vào *công nghệ FPGA*, một nền tảng công nghệ mới đã và đang phát triển rất mạnh hiện nay. Khác với bài toán tổng hợp và phân tích trong Điện tử số chủ yếu là bài toán cho các mạch cỡ SSI, MSI, các bài toán ở đây có hướng tới các ứng dụng cụ thể thực tiễn với quy mô lớn hơn và buộc phải sử dụng các công cụ trợ giúp thiết kế trên máy tính và ngôn ngữ thiết kế VHDL.

Chương trình *Thiết kế logic số* nhắm vào trang bị kiến thức cơ sở ngành cho tất cả các đối tượng sinh viên thuộc chuyên ngành kỹ thuật Điện tử viễn thông, Điều khiển tự động. Trước khi học môn này các sinh viên này phải học qua các môn cơ sở ngành gồm Cấu kiện điện tử, Điện tử số, Kỹ thuật Vi xử lý trong đó hai môn đầu là bắt buộc.

*Thiết kế logic số* là một môn học mang tính thực hành cao nên trong cấu trúc chương trình sẽ dành nhiều thời gian hơn cho thực hành thí nghiệm cũng như bắt buộc sinh viên khi kết thúc môn học phải thực hiện các đồ án bài tập thiết kế cỡ vừa và lớn theo nhóm dưới dạng Bài tập lớn hoặc Đồ án môn học.

Kiến thức và kỹ năng của sinh viên sẽ giúp ích rất lớn cho các bài toán chuyên ngành và Đồ án tốt nghiệp sau này bởi trong các ứng dụng xử lý số đang dần chiếm vai trò quan trọng trong các hệ thống kỹ thuật. Bên cạnh những công cụ truyền thống là Vi xử lý, máy tính thì thiết kế phần cứng trên FPGA hoặc trên nền các công nghệ tương tự đang là một hướng phát triển mang lại hiệu năng vượt trội và khả năng ứng dụng thích tốt hơn.

Giáo trình chính thức cho môn học được hoàn thiện sau hơn 2 khóa đào tạo cho sinh viên hệ đào tạo dân sự, quân sự tại Học viện Kỹ thuật quân sự. Nhóm tác giả xin chân thành cảm ơn sự ủng hộ nhiệt tình của lãnh đạo Khoa Vô tuyến điện tử, lãnh đạo bộ môn Kỹ thuật xung số, vi xử lý, các đồng nghiệp trong khoa và bộ môn đã có nhiều ý kiến đóng góp quý báu góp phần hoàn thiện nội dung cho giáo trình, cảm ơn anh chị em nhân viên của bộ môn đã góp nhiều công sức cho công việc chế bản cho giáo trình. Nhóm tác giả cũng gửi lời cảm ơn tới

toàn bộ các sinh viên các khóa đào tạo bằng quá trình học tập, nghiên cứu thực tế đã có những ý kiến đóng góp giúp tác giả điều chỉnh về khung chương trình và nội dung ngày hợp lý và hiệu quả hơn.

Vì thời gian hạn chế và là một môn học mới do vậy chắc chắn sẽ còn nhiều những khiếm khuyết trong giáo trình. Nhóm tác giả rất mong tiếp tục nhận được những ý kiến đóng góp của người sử dụng, mọi ý kiến có thể gửi về Bộ môn Kỹ thuật Xung số, Vi xử lý – Học viện KTQS hoặc vào hộp thư điện tử quangkien82@gmail.com.

Hà nội 12-2011

## Mục lục

LỜI GIỚI THIỆU .....	3
DANH SÁCH CÁC KÝ HIỆU VIẾT TẮT .....	11
Chương 1: CÁC KIẾN THÚC CƠ SỞ .....	15
1. Các khái niệm chung.....	16
1.1. Transistor.....	16
1.2. Vị mạch số tích hợp .....	17
1.3. Cỗng logic.....	18
1.4. Phần tử nhớ .....	20
1.5 Mạch logic tổ hợp .....	23
1.6. Mạch logic tuần tự .....	24
1.7 Các phương pháp thể hiện thiết kế .....	25
2. Yêu cầu đối với một thiết kế logic .....	27
3. Các công nghệ thiết kế mạch logic số.....	28
4. Kiến trúc của các IC khả trình .....	31
4.1. Kiến trúc PROM, PAL, PLA, GAL.....	31
4.2. Kiến trúc CPLD, FPGA .....	36
Câu hỏi ôn tập chương 1 .....	39
Chương 2: NGÔN NGỮ MÔ TẢ PHẦN CỨNG VHDL .....	41
1. Giới thiệu về VHDL.....	42
2. Cấu trúc của chương trình mô tả bằng VHDL.....	43
2.1. Khai báo thư viện.....	44
2.2. Mô tả thực thể .....	45
2.3. Mô tả kiến trúc.....	48
2.4. Khai báo cấu hình .....	53
3. Chương trình con và gói .....	56

3.1. Thủ tục .....	56
3.2. Hàm.....	58
3.3. Gói .....	59
4. Đổi tượng dữ liệu, kiểu dữ liệu .....	62
4.1. Đổi tượng dữ liệu.....	62
4.2. Kiểu dữ liệu .....	63
5. Toán tử và biểu thức .....	70
5.1. Toán tử logic.....	70
5.2. Các phép toán quan hệ.....	71
5.3. Các phép toán dịch .....	72
5.4. Các phép toán cộng trừ và hợp .....	74
5.5. Các phép dấu.....	74
5.6. Các phép toán nhân chia, lấy dư .....	75
5.7. Các phép toán khác .....	76
6. Phát biểu tuần tự .....	76
6.1. Phát biểu đợi .....	76
6.2. Phát biểu xác nhận và báo cáo.....	79
6.3. Phát biểu gán biến.....	80
6.4. Phát biểu gán tín hiệu .....	81
6.5. Lệnh rẽ nhánh và lệnh lặp.....	83
7. Phát biểu đồng thời .....	87
7.1. Phát biểu khối .....	88
7.2. Phát biểu quá trình .....	89
7.3. Phát biểu gán tín hiệu đồng thời .....	92
7.4. Phát biểu generate.....	95
7.5. Phát biểu cài đặt khối con.....	97
8. Phân loại mã nguồn VHDL.....	99
9. Kiểm tra thiết kế bằng VHDL.....	101

9.1. Kiểm tra nhanh .....	102
9.1. Kiểm tra tự động nhiều tổ hợp đầu vào .....	104
Bài tập chương 2 .....	111
Bài tập .....	111
Câu hỏi ôn tập lý thuyết .....	116
<b>Chương 3: THIẾT KẾ CÁC KHỐI MẠCH DÃY VÀ TỔ HỢP THÔNG DỤNG.....</b>	<b>117</b>
1. Các khối cơ bản.....	118
1.1. Khối cộng đơn giản .....	118
1.2. Khối trù.....	119
1.3. Khối cộng thấy nhớ trước.....	121
1.4. Thanh ghi .....	125
1.5. Bộ cộng tích lũy.....	127
1.6. Bộ đếm.....	129
1.7. Bộ dịch.....	131
1.8. Thanh ghi dịch .....	133
2. Các khối nhớ .....	136
2.1. Bộ nhớ RAM .....	136
2.2. Bộ nhớ ROM .....	139
2.3. Bộ nhớ FIFO .....	141
2.4. Bộ nhớ LIFO.....	142
3. Máy trạng thái hữu hạn .....	143
4. Khối nhân số nguyên.....	145
4.1. Khối nhân số nguyên không dấu dùng phương pháp cộng dịch	146
4.2. Khối nhân số nguyên có dấu.....	150
4.3. Khối nhân dùng mã hóa Booth cơ số 4 .....	155
5. Khối chia số nguyên.....	158
5.1. Khối chia dùng sơ đồ khôi phục phần dư .....	159

5.2. Khối chia dùng sơ đồ không khôi phục phần dư .....	162
5.3. Khối chia số nguyên có dấu.....	164
6. Các khối làm việc với số thực .....	169
6.1. Số thực dấu phẩy tĩnh .....	169
6.2. Số thực dấu phẩy động .....	170
6.3. Chế độ làm tròn trong số thực dấu phẩy động.....	173
6.4. Phép cộng số thực dấu phẩy động .....	176
6.5. Phép nhân số thực dấu phẩy động .....	181
6.6. Phép chia số thực dấu phẩy động .....	183
Bài tập chương 3 .....	186
Bài tập .....	186
Câu hỏi ôn tập lý thuyết.....	194
Chương 4: THIẾT KẾ MẠCH SỐ TRÊN FPGA .....	195
1. Tổng quan về kiến trúc FPGA .....	196
1.2. Khái niệm FPGA .....	196
1.3. Ứng dụng của FPGA trong xử lý tín hiệu số.....	198
1.4. Công nghệ tái cấu trúc FPGA.....	199
1.5. Kiến trúc tổng quan .....	200
2. Kiến trúc chi tiết Xilinx FPGA Spartan-3E. ....	201
2.1. Khối logic khả trình.....	204
2.2. Khối điều khiển vào ra.....	221
2.3. Hệ thống kết nối khả trình .....	224
2.4. Các phần tử khác của FPGA.....	227
3. Quy trình thiết kế FPGA bằng ISE .....	237
3.1. Mô tả thiết kế .....	238
3.2. Tổng hợp thiết kế.....	239
3.3. Hiện thực hóa thiết kế.....	244
3.4. Cấu hình FPGA .....	250

3.5. Kiểm tra thiết kế trên FPGA .....	250
4. Một số ví dụ thiết kế trên FPGA bằng ISE .....	251
4.1. Thiết kế khôi nhận thông tin UART .....	253
4.2. Thiết kế khôi điều khiển PS/2 cho Keyboard, Mouse .....	267
4.3. Thiết kế khôi tổng hợp dao động số NCO .....	270
4.4. Thiết kế khôi điều khiển LCD1602A .....	282
4.5. Thiết kế điều khiển VGA trên FPGA .....	294
Bài tập chương 4 .....	308
1. Bài tập cơ sở .....	308
2. Bài tập nâng cao.....	309
3. Câu hỏi ôn tập lý thuyết.....	312
PHỤ LỤC .....	313
Phụ lục 1: THỐNG KÊ CÁC HÀM, THỦ TỤC, KIỂU DỮ LIỆU CỦA VHDL TRONG CÁC THƯ VIỆN CHUẨN IEEE.....	314
1. Các kiểu dữ liệu hỗ trợ trong các thư viện chuẩn IEEE .....	314
2. Các hàm thông dụng hỗ trợ trong các thư viện chuẩn IEEE .....	315
3. Các hàm phục vụ cho quá trình mô phỏng kiểm tra thiết kế .....	319
4. Các hàm biến đổi kiểu dữ liệu dùng trong VHDL .....	322
Phụ lục 2: THỰC HÀNH THIẾT KẾ VHDL.....	325
Bài 1: Mô phỏng VHDL trên ModelSim.....	326
Bài 2: Xây dựng bộ cộng trừ trên cơ sở khôi cộng bằng toán tử.....	338
Bài 3: Khối dịch và thanh ghi dịch .....	344
Bài 4: Bộ cộng bit nối tiếp dùng 1 FA (serial-bit adder).....	353
Phụ lục 3: MẠCH PHÁT TRIỀN ỦNG DUNG FPGA .....	364
1. Giới thiệu tổng quan .....	364
2. Các khôi giao tiếp có trên mạch FPGA .....	366
2.4. Khôi giao tiếp Keypad .....	367
2.5. Khôi 8x2 Led-Diod .....	367

2.6. Khối Switch.....	367
2.7. Khối giao tiếp 4x7-seg Digits .....	367
2.9. Khối giao tiếp USB .....	368
2.10. Khối giao tiếp PS/2 .....	368
Phụ lục 4: THỰC HÀNH THIẾT KẾ MẠCH SỐ TRÊN FPGA .....	371
Bài 1: Hướng dẫn thực hành FPGA bằng Xilinx ISE và Kit SPARTAN 3E.....	372
Bài 2: Thiết kế khối giao tiếp với 4x7Seg -digits .....	397
Phụ lục 5: CÁC BẢNG MÃ THÔNG DỤNG.....	407
1. Mã ASCII điều khiển.....	408
2. Mã ASCII hiển thị .....	410
3. Bảng mã ký tự cho LCD 1602A .....	414
TÀI LIỆU THAM KHẢO .....	415

## DANH SÁCH CÁC KÝ HIỆU VIẾT TẮT

<b>AES</b>	: Advance Encryption Standard	Thuật toán mã hóa AES
<b>ALU</b>	: Arithmetic Logic Unit	Khối thực thi số học logic
<b>ASIC</b>	: Application Specific Integrated Circuit	Vi mạch tích hợp với chức năng chuyên dụng.
<b>BJT</b>	: Bipolar Junction Transistor	Transistor lưỡng cực
<b>BRAM</b>	: Block RAM	Khối nhớ truy cập ngẫu nhiên trong FPGA
<b>CLA</b>	: Carry Look-Ahead Adder	Khối cộng thấy nhớ trước
<b>CLB</b>	: Configurable Logic Block	Khối Logic khả trình trong FPGA
<b>CMOS</b>	: CMOS (Complementary-Symmetry Metal-Oxide Semiconductor)	Công nghệ bán dẫn dùng trên cặp bù PN transistor trường.
<b>CPLD</b>	: Complex Programmable Logic Device	Vi mạch khả trình phức tạp (cỡ lớn)
<b>DCM</b>	: Digital Clock Manager	Khối quản lý và điều chỉnh xung nhịp hệ thống trong FPGA
<b>DDR</b>	: Double Data Rate	Truyền dữ liệu với tốc độ gấp đôi tốc độ cung nhịp hệ thống
<b>DES</b>	: Data Encryption Standard	Thuật toán mã hóa DES
<b>DFS</b>	: Digital Frequency Synthesis	Khối tổng hợp tần số
<b>DLL</b>	: Delay Locked Loop	Khối lặp khóa trễ
<b>DRAM</b>	: Dynamic RAM	RAM động
<b>DRC</b>	: Design Rule Check	Kiểm tra các vi phạm trong thiết kế
<b>DUT</b>	: Device Under Test	Đối tượng được kiểm tra
<b>E<sup>2</sup>PROM</b>	: Electric-Erasable Programmable ROM	PROM có thể xóa bằng điện
<b>EDIF</b>	: Electronic Design Interchange Format	Chuẩn công nghiệp để mô tả các khối điện tử.
<b>EDK</b>	: Embedded Development Kit	Tổ hợp phần mềm thiết kế hệ nhúng trên FPGA

<b>EPROM</b>	: Eraseable Programmable ROM	PROM có thể xóa được
<b>F5MUX</b>	: Wide-Multiplexer	Khối chọn kênh mở rộng trong FPGA
<b>FET</b>	: Field Effect Transistor	Transistor dùng hiệu ứng trường
<b>FIFO</b>	: First In First Out	Bộ nhớ có dữ liệu vào trước sẽ được đọc ra trước.
<b>FiMUX</b>	: Wide-Multiplexer	Khối chọn kênh mở rộng trong FPGA
<b>FPGA</b>	: Field-Programmable Gate Array	IC khả trìn cấp độ người dùng cuối
<b>FPU</b>	: Floating Point Unit	Khối xử lý số thực dấu phẩy động
<b>GAL</b>	: Generic Array Logic	IC khả trìn trên công nghệ CMOS
<b>HDL</b>	: Hardware Description Language	Ngôn ngữ mô tả phần cứng
<b>I2C</b>	: Inter-Integrated Circuit	Giao tiếp I2C truyền dữ liệu giữa các IC
<b>IC</b>	: Integrated Circuit	Vi mạch tích hợp
<b>IEEE</b>	: Institute of Electrical and Electronics Engineers	Viện kỹ thuật Điện và Điện tử
<b>IOB</b>	: Input/Output Buffer	Khối đệm vào ra trong FPGA
<b>IP Core</b>	: Intellectual Property core	Thiết kế được đăng ký sở hữu trí tuệ
<b>ISE</b>	: Integrated Software Enviroment	Tổ hợp phần mềm thiết kế FPGA của Xilinx
<b>LIFO</b>	: Last In First Out	Khối nhớ LIFO, dữ liệu vào sau cùng sẽ ra trước nhất
<b>LSI</b>	: Large scale integration	Vi mạch tích hợp cỡ lớn
<b>LUT</b>	: Look-Up Table	Bảng tham chiếu trong FPGA
<b>MOSFET</b>	: Metal-oxide-semiconductor Field-Effect-Transistor	Transistor trường dùng tiếp gián kim loại – bán dẫn
<b>MSI</b>	: Medium scale integration	Vi mạch tích hợp cỡ trung
<b>MULT18</b>	: Dedicated Multiplier 18 x18	Khối nhân chuyên dụng trong FPGA
<b>NCD</b>	: Native Circuit Database	Định dạng sau quá trình Ánh xạ

<b>NCF</b>	: Native Constraint File	công và Sắp đặt kết nối của Xilinx ISE.
<b>NGD</b>	: Native Generic Database	Tệp cài đặt điều kiện ràng buộc cơ bản của thiết kế.
<b>PAL</b>	: Programmable Array Logic	Định dạng sau quá trình Translate của Xilinx ISE
<b>PAR</b>	: Place and Route	Mảng logic khả trình Sắp đặt và kết nối (trong quá trình hiện thực hóa FPGA)
<b>PCF</b>	: Physical Constraint File	Tệp quy định các ràng buộc vật lý của thiết kế trên ISE
<b>PLA</b>	: Programmable Logic Array	Mảng các khối logic khả trình
<b>PLD</b>	: Programmable Logic Device	Vi mạch khả trình
<b>PROM</b>	: Programmable Read-Only Memory	Bộ nhớ ROM khả trình
<b>PS/2</b>	: IBM Personal System 2	Chuẩn giao tiếp cho các ngoại vi như chuột, bàn phím trên máy tính của IBM
<b>RAM</b>	: Read Only Memory	Bộ nhớ truy cập ngẫu nhiên
<b>RSA</b>	: Ronald Rivest, Adi Shamir & Leonard Adleman Cryption Schema	Thuật toán mã hóa RSA
<b>RTL</b>	: Register Tranfer Level	Mô tả lớp thanh ghi truyền tải
<b>SDK</b>	: Software Development Kit	Tổ hợp các chương trình hỗ trợ thiết kế phần mềm nhúng của Xilinx
<b>SHL16</b>	: Shift-Register 16 bit	Thanh ghi dịch 16 bit
<b>SLICEL</b>	: SLICE Logic	Phần tử Logic trong FPGA
<b>SLICEM</b>	: SLICE Memory	Phần tử Logic có khả năng thực hiện chức năng nhớ trong FPGA
<b>SoC</b>	: System On a Chip	Hệ thống tích hợp trên một chíp đơn.
<b>SPI</b>	: Serial Peripheral Interface	Chuẩn kết nối ngoại vi nối tiếp
<b>SPLD</b>	: Simple Programmable Logic	

	Device	
<b>SRAM</b>	: Static Random Access Memory	RAM tĩnh
<b>SSI</b>	: Small scale integration	Vi mạch tích hợp cỡ nhỏ
<b>UART</b>	: Universal Asynchronous Receiver Transceiver	Chuẩn truyền tin dị bộ nối tiếp
<b>UCF</b>	: User Constraint File	Tệp quy định các điều kiện ràng buộc cho thiết kế bởi người dùng.
<b>ULSI</b>	: Ultra large scale intergration	
<b>VGA</b>	: Video Graphic Array	Chuẩn kết nối với màn hình máy tính
<b>VHDL</b>	: Very Hi-speed Integrated Circuit Hardware Description Language	Ngôn ngữ mô tả vi mạch số tích hợp
<b>VLSI</b>	: Very large scale integration	Vi mạch tích hợp cỡ rất lớn
<b>WSI</b>	: Wafer scale intergration	
<b>XPS</b>	: Xilinx Platform Studio	Chương trình phần mềm hỗ trợ xây dựng hệ nhúng trên FPGA
<b>XST</b>	: Xilinx Synthesis Technology	Chương trình tổng hợp thiết kế của Xilinx

## **Chương 1**

### **CÁC KIẾN THỨC CƠ SỞ**

Chương mở đầu có nhiệm vụ cung cấp cho người học những kiến thức, khái niệm cơ bản về thiết kế các khối số, trong đó có những kiến thức được nhắc lại với những bối cảnh phù hợp với mục đích môn học. Người học được giới thiệu qua về cách thức thiết kế làm việc với tín hiệu số được thiết kế chế tạo, phân loại các dạng vi mạch số và các tham số cơ bản cần quan tâm khi thiết kế hay làm việc với vi mạch số.

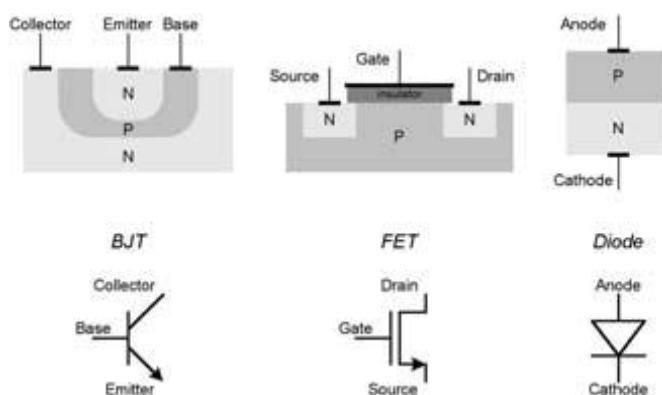
Chương này cũng giới thiệu qua về sự phát triển của một lớp các IC khả trình phần cứng từ PROM cho tới FPGA. Mục đích của phần này giúp cho người học có một cái nhìn tổng quan về lịch sử của thiết kế logic số trước khi tập trung vào các vấn đề kiến thức chính ở các chương sau là ngôn ngữ mô tả phần cứng VHDL và công nghệ FPGA.

# 1. Các khái niệm chung

## 1.1. Transistor

Là linh kiện bán dẫn có khả năng làm việc như một công tắc bật tắt hoặc dùng để khuếch đại tín hiệu. Transistor là phần tử cơ bản của mọi vi mạch số tích hợp, từ các cổng logic đơn giản AND, OR, NOT... đến các loại phức tạp như các mạch điều khiển ngoại vi, vi điều khiển, vi xử lý...

Transistor được làm từ vật liệu bán dẫn (*semiconductor*), là vật liệu vừa có khả năng dẫn điện vừa có khả năng làm việc như những vật liệu cách điện, khả năng này thay đổi tùy theo kích thích từ bên ngoài như nhiệt độ, ánh sáng, trường điện từ, dòng điện... Chất bán dẫn dùng để cấu tạo transistor thường là Germanium (Ge) hoặc Silic (Si) được kích tạp một lượng nhỏ Photpho(P) hoặc Boron (B) với mục đích tăng mật độ electron (kiểu N) tự do hoặc tăng mật độ lỗ trống (kiểu P) tương ứng trong tinh thể bán dẫn. Cấu trúc nguyên lý của các dạng transistor được trình bày ở hình dưới đây:



Hình 1-1. Cấu trúc transistor lưỡng cực BJTs, đơn cực FETs, diode

Transistor lưỡng cực *BJT* (*Bipolar Junction Transistor*) sử dụng nhiều trong thập kỷ 80s, đặc điểm của BJT là tốc độ chuyển mạch nhanh nhưng nhược điểm là mức tiêu thụ năng lượng lớn ngay cả trong trạng thái nghỉ và chiếm nhiều diện tích.

Sau đó BJTs dần được thay thế bằng transistor đơn cực *FETs* (*Field Effect Transistors*) làm việc trên hiệu ứng trường và kênh dẫn chỉ dùng một loại bán dẫn loại p hoặc n. *MOSFETs* (*Metal-oxide-semiconductor Field-Effect-Transistors*) là transistor FETs nhưng dùng cực Cổng metal (về sau lớp metal được thay bằng polysilicon) phủ trên một lớp oxide cách điện và lớp này phủ trên vật liệu bán

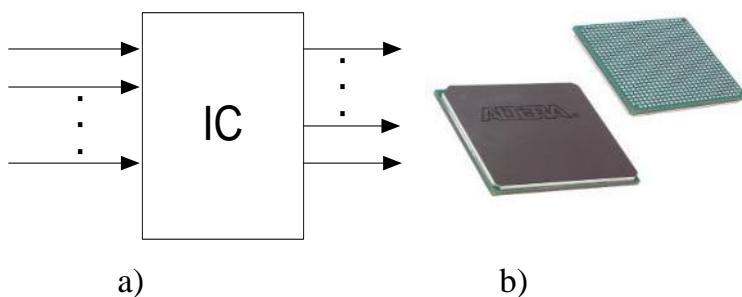
dẫn, tùy theo loại vật liệu bán dẫn mà transistor này có tên gọi là NMOS (kênh dẫn n) và PMOS (kênh dẫn p).

*CMOS* (*Complementary-Symmetry Metal-Oxide Semiconductor*) là transistor tạo thành từ việc ghép cặp bù PMOS và NMOS, có nhiều ưu điểm so với các dòng transistor cũ như hiệu điện thế làm việc thấp, độ chống nhiễu cao, tiêu tốn ít năng lượng và cho phép tích hợp trong IC số với mật độ cao. CMOS là công nghệ chế tạo bán dẫn được sử dụng rộng rãi nhất hiện nay.

## 1.2. Vi mạch số tích hợp

Còn được gọi là IC – Intergrated Circuits, chip, là cấu trúc mạch điện được thu nhỏ bằng cách tích hợp chủ yếu từ các transistor với mật độ cao, ngoài ra còn có thể có các linh kiện điện thụ động khác trên một khối bán dẫn mỏng.

Các vi mạch tích hợp đều có một số lượng tín hiệu đầu vào và đầu ra để thực hiện một chức năng cụ thể nào đó. Trong khuôn khổ giáo trình này chủ yếu nghiên cứu về vi IC số, tức là dạng IC chỉ làm việc với các tín hiệu số.



Hình 1-2: a) Mô hình Vi mạch số tích hợp

b) Vi mạch tích hợp thực tế

Vi mạch tích hợp ra đời từ những năm 1960s và được ứng dụng rộng rãi trong thực tế, đã và đang tạo ra cuộc cách mạng trong lĩnh vực điện tử. Ví dụ về vi mạch tích hợp như các IC đa dụng (*general purposes IC*) họ 7400, 4000, các dòng vi xử lý 80x86 dùng trong máy vi tính, chíp xử lý dùng cho điện thoại di động, máy ảnh kỹ thuật số, các vi điều khiển dùng trong các thiết bị dân dụng, ti vi, máy giặt, lò vi sóng... Các vi mạch này có mật độ tích hợp từ hàng vài chục đến hàng trăm triệu, và hiện nay đã đến hàng tỷ transistor trong một miếng bán dẫn có kích thước xấp xỉ kích thước đồng xu. *Mật độ tích hợp* được định nghĩa là tổng số những phần tử tích cực (transistor hoặc cổng logic) chứa trên một đơn vị

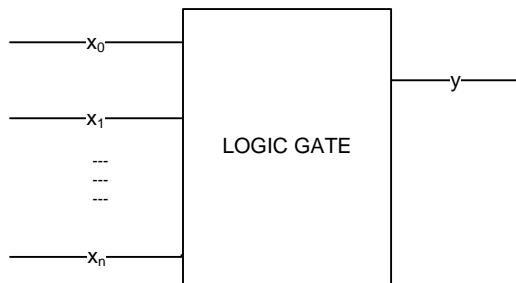
diện tích của khối tinh thể bán dẫn. Theo mật độ tích hợp chia ra các loại vi mạch sau:

- Vi mạch cỡ nhỏ SSI (*Small scale integration*), có hàng chục transistor trong một vi mạch.
- Vi mạch cỡ vừa MSI (*Medium scale integration*), có hàng trăm transistor trong một vi mạch.
- Vi mạch cỡ lớn LSI (*Large scale integration*), có hàng ngàn đến hàng chục ngàn transistor trong một vi mạch.
- Vi mạch cực lớn VLSI (*Very large scale integration*), có hàng vạn, hàng triệu, hàng chục triệu transistor và lớn hơn trong một vi mạch, thời điểm hiện nay đã xuất hiện nhưng vi mạch có độ tích hợp đến hàng tỷ transistor.
- Vi mạch siêu lớn ULSI (*Ultra large scale intergration*), vi mạch có độ tích hợp với mức độ hàng triệu transistor trở lên.
- WSI (*Wafer-scale-Intergration*) là giải pháp tích hợp nhiều vi mạch chức năng trên một tấm silicon (wafer) để tăng hiệu suất cũng như giảm giá thành sản phẩm, ví dụ hệ vi xử lý nhiều nhân được tích hợp bằng WSI.
- SoC (*System-on-a-Chip*) Khái niệm chỉ một hệ tính toán, xử lý mà tất cả các khối chức năng số và cả tương tự được thiết kế để tích hợp vào trong một chip đơn.

Trong khuôn khổ chương trình này sẽ dành thời lượng chính cho việc nghiên cứu cơ bản về công nghệ, phương pháp, quá trình thiết kế các vi mạch cỡ LSI, VLSI.

### 1.3. Cổng logic

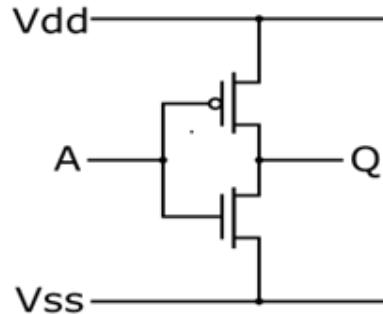
Cổng logic hay *logic gate* là cấu trúc mạch điện (sơ đồ khôi hình) được lắp ráp từ các linh kiện điện tử để thực hiện chức năng của các hàm logic cơ bản  $y = f(x_n, x_{n-1}, \dots, x_1, x_0)$ . Trong đó các tín hiệu vào  $x_{n-1}, x_{n-2}, \dots, x_1, x_0$  của mạch tương ứng với các biến logic  $x_{n-1},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

Giá trị của các tín hiệu vào và ra chỉ có hai mức là mức thấp (Low - L) và mức cao (High - H) tương ứng với hai giá trị 0 và 1 của các biến logic và hàm logic.

Ví dụ: Một cỗng NOT loại CMOS (hình 1.4) tương ứng hàm NOT hai biến  $Q = \text{not } A$ .

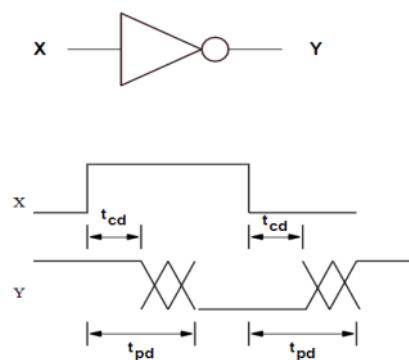


Hình 1-4. Mạch điện cỗng NOT

Trên sơ đồ dễ nhận thấy rằng, chỉ khi A có mức tích cực cao thì transistor trên đóng còn transistor dưới thông, Q có mức tích cực thấp, khi A có mức tích cực thấp thì transistor trên mở và dưới đóng nên Q có mức tích cực cao, như vậy mạch điện với sơ đồ trên thực hiện vai trò của cỗng NOT.

Các mạch logic đều được biểu diễn bằng các hệ hàm logic và do đó có thể phát biểu là: Mọi mạch logic đều có thể xây dựng từ các cỗng logic cơ bản.

Đối với các cỗng logic cơ bản đó thì có hai tham số thời gian cơ bản:



Hình 1.5. Tham số thời gian của cỗng NOT

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 cỗng thì là chỉ tới giá trị  $T_{pd}$ .

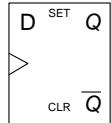
## 1.4. Phần tử nhớ

### 1.4.1. D-Latch và D flip-flop

Latch và Flip-Flop là các phần tử nhớ quan trọng trong thiết kế VLSI, sơ đồ cấu tạo chi tiết và mô tả đã được trình bày kỹ trong phần *Kỹ thuật số*. Ở phần này chỉ nhắc lại những tính chất cơ bản nhất của các Flip-Flop và bổ sung thêm các tham số thời gian thực của các phần tử này.

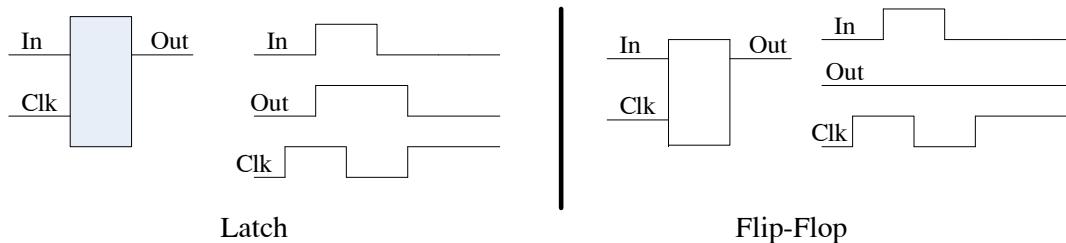
Bảng 1-1

D-Flip flop và D-latch

D-flip flop				D-latch			
Clock	D	Q	$Q_{prev}$	Clock	D	Q	
Rising edge	1	1	x	0	X	$Q_{prev}$	
Rising edge	0	0	x	1		D	
Non-rising	x	$Q_{prev}$					

D-Latch là phần tử nhớ làm việc theo mức xung, cụ thể khi tín hiệu Clock bằng 1 thì giá trị Q đầu ra bằng giá trị đầu vào, khi tín hiệu Clock = 0 thì giá trị đầu ra không đổi. Nói một cách khác D-latch làm việc như một cửa đóng mở giữa tín hiệu Q và D tương ứng với mức điện áp của xung Clock.

D-flip-flop là phần tử nhớ làm việc theo sườn xung, có hai dạng sườn là sườn lên (rising edge) khi xung thay đổi từ 0->1 và sườn xuống (falling edge) khi xung thay đổi từ 1->0. Khi không có yêu cầu gì đặc biệt thì Flip-flop làm việc với sườn xung lên thường được sử dụng. Khác với D-latch giá trị đầu ra của Flip-Flop chỉ thay vào thời điểm sườn xung . Với cách làm việc như vậy giá trị đầu ra sẽ không thay đổi trong suốt thời gian một chu kỳ xung nhịp dù cho tín hiệu đầu vào thay đổi. D Flip-flop rất hay được dùng trong mạch có nhớ vì vậy đôi khi nói đến phần tử nhớ thường ngầm hiểu là D Flip-flop.

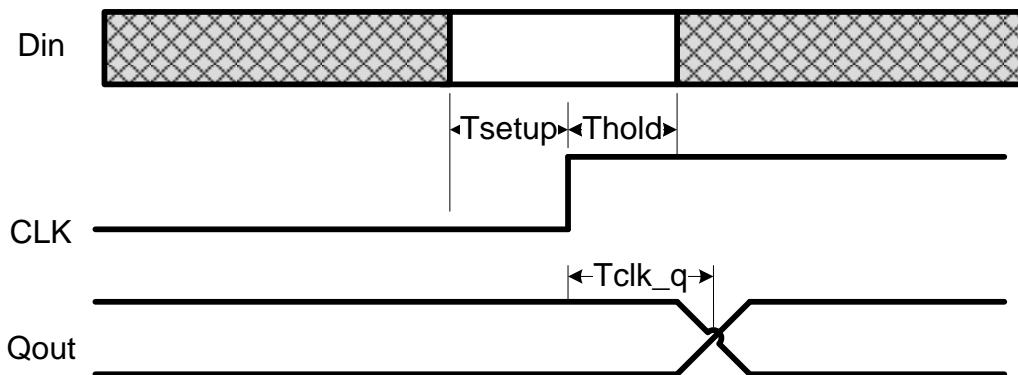


Hình 1-6. Đồ thị thời gian của D Flip-flop và D Latch

Đối với D-flip-flop và D-latch nhớ thì có hai tham số thời gian hết sức quan trọng là  $T_{\text{setup}}$ , và  $T_{\text{hold}}$ . Đây là tham số thời gian đối với dữ liệu đầu vào cổng Din để đảm bảo việc truyền dữ liệu sang công ra  $Q_{\text{out}}$  là chính xác, cụ thể đối với Flip-flop.

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

$T_{\text{hold}}$ : 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

#### 1.4.2 Các flip-flop khác

- RS Flip-flop:

Bảng 1-2

RS Flip-flop

R	S	$Q_{\text{next}}$	$\overline{Q}$
0	0	$Q_{\text{prev}}$	
0	1	1	0
1	0	0	1
1	1	Chạy đua	

RS Flip-flop có đầu vào là hai tín hiệu Reset và Set. Set =1 thì tín hiệu đầu ra nhận giá trị 1 không phụ thuộc giá trị hiện tại Q, Reset =1 thì đầu ra Q = 0 không phụ thuộc giá trị hiện tại Q. Đối với RS-flipflop không đồng bộ thì giá trị Q thay đổi phụ thuộc R/S ngay tức thì, còn đối với RS flip-flop đồng bộ thì tín hiệu Q chỉ thay đổi tại thời điểm sườn xung Clock.

Trạng thái khi R= 1, S= 1 là trạng thái cấm vì khí đó đầu ra nhận giá trị không xác định, thực chất sẽ xảy ra sự thay quá trình “chạy đua” hay tự dao động giá trị Q từ 0 đến 1 và ngược lại với chu kỳ bằng độ trễ chuyển mạch của flip-flop.

- JK-flip-flop

Bảng 1-3

### JK Flip-flop

J	K	$Q_{next}$	<p>J <sup>SET</sup> Q K <sub>CLR</sub> <math>\bar{Q}</math></p>
0	0	$Q_{prev}$	
0	1	0	
1	0	1	
1	1	NOT $Q_{prev}$	

Theo bảng chân lý JK-flip flop hoạt động khá linh hoạt thực hiện chức năng giống như D-flip flop hoặc RS flip-flop, trạng thái khí J=0, K=1 là Reset, J=1, K=0 là Set. Tuy không có đầu vào dữ liệu D nhưng để JK flip-flop làm việc như một D-flip flop thì tín hiệu D nối với J còn K cho nhận giá trị đối của J.

- T- flip-flop

Bảng 1-4

### T Flip-flop

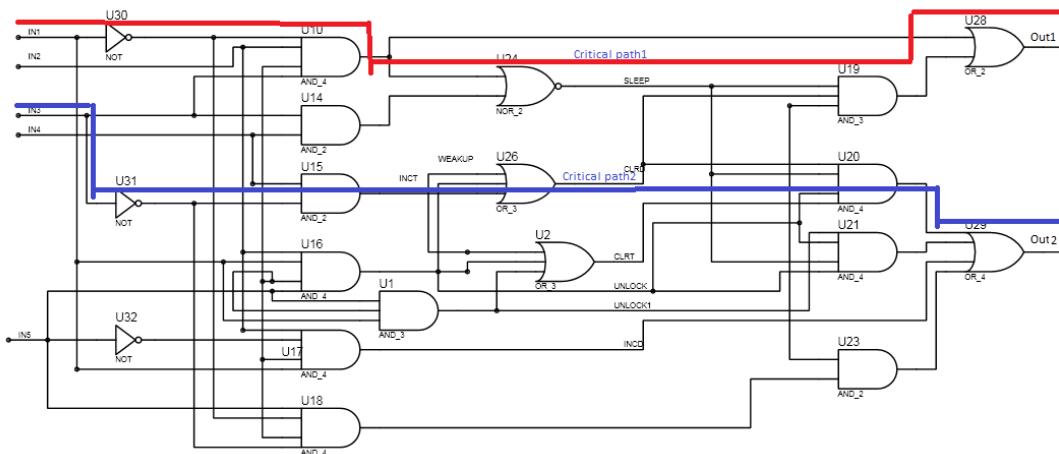
T	Q	$Q_{next}$	<p>T Q <math>\bar{Q}</math></p>
0	0	0	
0	1	1	
1	0	1	
1	1	0	

Khi T bằng 1 thì giá trị  $Q_{next}$  bằng đảo của giá trị trước  $Q_{prev}$  khi T = 0 thì giá trị đầu ra không thay đổi

## 1.5 Mạch logic tổ hợp

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 ...

Đối với mạch tổ hợp tham số thời gian trễ  $T_{delay}$  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, giá trị đó chính là  $T_{delay}$ .



Hình 1-8. Độ trễ của mạch tổ hợp

Minh họa cho độ trễ trong mạch tổ hợp như ở hình 1-8. Về lý thuyết để xác định độ trễ của mạch cần liệt kê tất cả các đường tín hiệu từ 4 đầu vào In1, In2, In3, In4 đến 2 đầu ra Out1, Out2. Đối với mỗi cặp đầu ra đầu vào tồn tại nhiều đường truyền khác nhau vì vậy tổng số lượng các đường truyền này thường rất lớn. Chính vì thế đối với những mạch tổ hợp lớn thì việc xác định độ trễ đều phải thực hiện bằng sự hỗ trợ của máy tính.

Ví dụ để xác định độ trễ của hai đường truyền 1 và 2 trên hình vẽ: đường 1 lùn lượt đi qua các cổng NOT, AND\_4, NOR, AND\_3, OR. Đường 2 lùn lượt đi qua cổng NOT, AND, OR\_4, AND\_4, OR\_4. Độ trễ của các đường truyền này tính bằng độ trễ của các cổng nó đi qua cộng với độ trễ dây dẫn ( $T_{write}$ ).

$$T_1 = T_{\text{NOT}} + T_{\text{AND\_4}} + T_{\text{NOR}} + T_{\text{AND\_3}} + T_{\text{AND\_3}} + T_{\text{Wire1}} \quad (1.1)$$

$$T_2 = T_{\text{NOT}} + T_{\text{AND}} + T_{\text{OR\_4}} + T_{\text{AND\_4}} + T_{\text{OR\_4}} + T_{\text{Wire2}} \quad (1.2)$$

Do độ trễ của cổng nhiều đầu vào lớn hơn độ trễ của cổng ít đầu vào nên mặc dù số cổng đi qua trên đường truyền như nhau nhưng đường truyền 2 sẽ có độ trễ lớn hơn đường 1. Các đường truyền có độ trễ lớn nhất được gọi là *Critical paths*. Các đường truyền này cần đặc biệt quan tâm trong quá trình tối ưu hóa độ trễ của mạch.

### 1.6. Mạch logic tuần tự

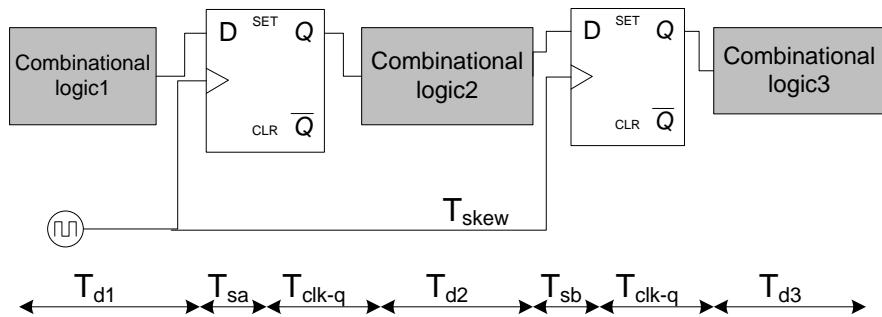
*Mạch logic dãy (Sequential logic circuits)* còn được gọi là mạch logic tuần tự là mạch số mà tín hiệu ra tại một thời điểm không những phụ thuộc vào tổ hợp tín hiệu đầu vào tại thời điểm đó mà còn phụ thuộc vào tín hiệu vào tại các thời điểm trước đó. Hiểu một cách khác mạch dãy ngoài các phần tử tổ 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 được tính khác với mạch tổ hợp, sự khác biệt đó 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 ( $T_{\text{setup}}$ ) của hai Flipflop A, B tương ứng,

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



Hình 1-9. Tham số thời gian của mạch tuần tự

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.  $T_{\text{skew}}$  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.

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

$$T_{input\_delay} = T_{d1} + T_{sa} \quad (1.3)$$

- Thời gian trễ sau xung nhịp Clock tại đầu ra.

$$T_{output\_delay} = T_{d3} + T_{clk\_q} \quad (1.4)$$

- 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.

$$T_{clk\_min} = T_{clk-q} + T_{d2} + T_{sb} + T_{skew} \quad (1.5)$$

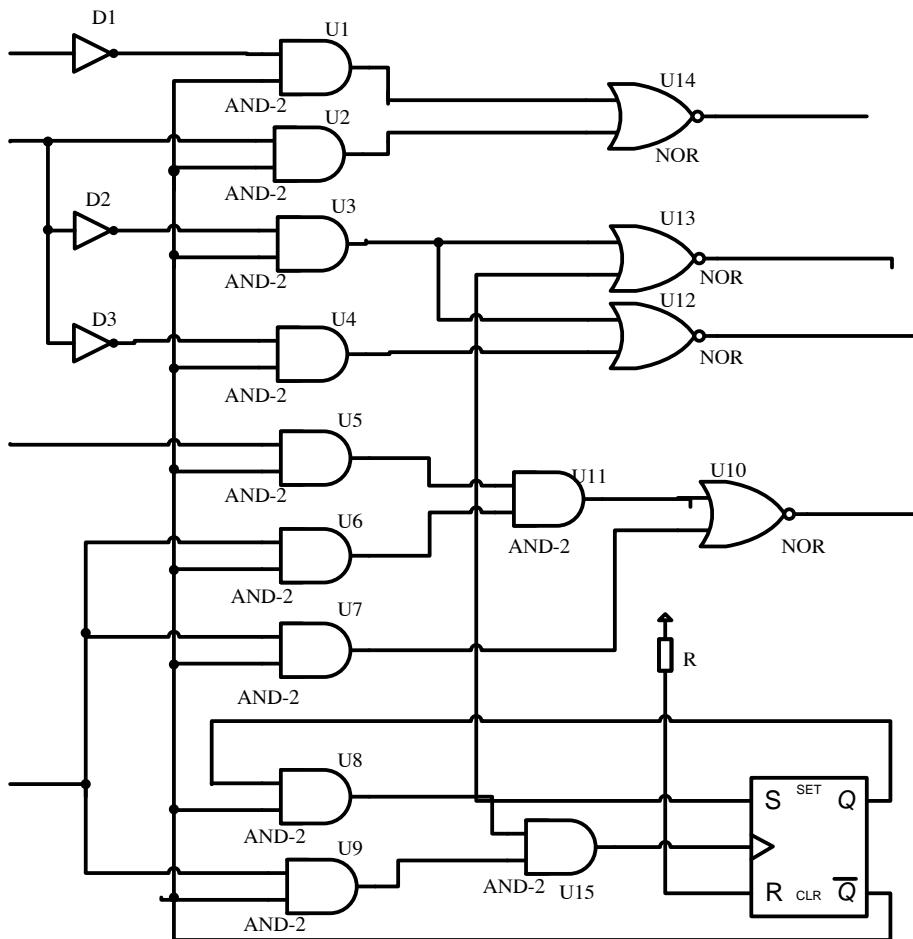
- Từ đó tính được xung nhịp tối đa của vi mạch là

$$F_{max} = 1 / T_{clk\_min} = 1 / (T_{clk-q} + T_{d2} + T_{sb} + T_{skew}) \quad (1.6)$$

## 1.7 Các phương pháp thể hiện thiết kế.

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.



Hình 1-10. Mô tả mạch số bằng sơ đồ

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 công 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:* HDL cho phép mô tả vi mạch bằng các cú pháp tương tự như cú pháp của ngôn ngữ lập trình. Có ba ngôn ngữ mô tả phần cứng phổ biến hiện nay là:

*Verilog:* Ra đời năm 1983, do hai kỹ sư Phil Moorby và Prabhu Goel làm việc tại Automated Integrated Design Systems (sau này thuộc sở hữu của Cadence). 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 logic thì Verilog rất mạnh trong việc hỗ trợ cho quá trình kiểm tra thiết kế.

**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.

**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 mở rộng biên dịch.

Bên cạnh các ngôn ngữ trên thì một loạt các ngôn ngữ khác đã và đang phát triển cũng hỗ trợ khả năng mô tả phần cứng, đáng chú ý là System Verilog là phiên bản mở rộng của Verilog hướng của C++ như hỗ trợ các kiểu dữ liệu khác nhau, sử dụng Class và nhiều hàm hệ thống bậc cao.

**SystemC** không hoàn toàn phải là một HDL mà là một dạng mở rộng của C++ cho phép hỗ trợ kiểm tra các thiết kế bằng VHDL hay Verilog.

## 2. Yêu cầu đối với một thiết kế logic

Yêu cầu đối với một thiết kế IC bao gồm:

- Yêu cầu chức năng: mạch gồm có các đầu vào đầu ra như thế nào, thực hiện nhiệm vụ gì...
- Yêu cầu về mặt công nghệ: Mạch thiết kế sử dụng nền công nghệ bán dẫn nào PLD, ASIC, FPGA...
- Yêu cầu về mặt tài nguyên: Giới hạn về số lượng cổng, số lượng transistors, về diện tích quy đổi chuẩn, về kích thước của IC thiết kế.
- Yêu cầu về khả năng làm việc (*performance*): là yêu cầu về các tham số thời gian của mạch bao gồm độ trễ cổng vào, độ trễ cổng ra, độ trễ logic với mạch tổ hợp, các xung nhịp làm việc, số lượng xung nhịp cho một chu trình xử lý dữ liệu, số lượng dữ liệu xử lý trên một đơn vị thời gian.

- Yêu cầu về mức tiêu hao năng lượng (*power consumption*).
- Yêu cầu về chi phí cho quá trình thiết kế và chế tạo (*design cost*).

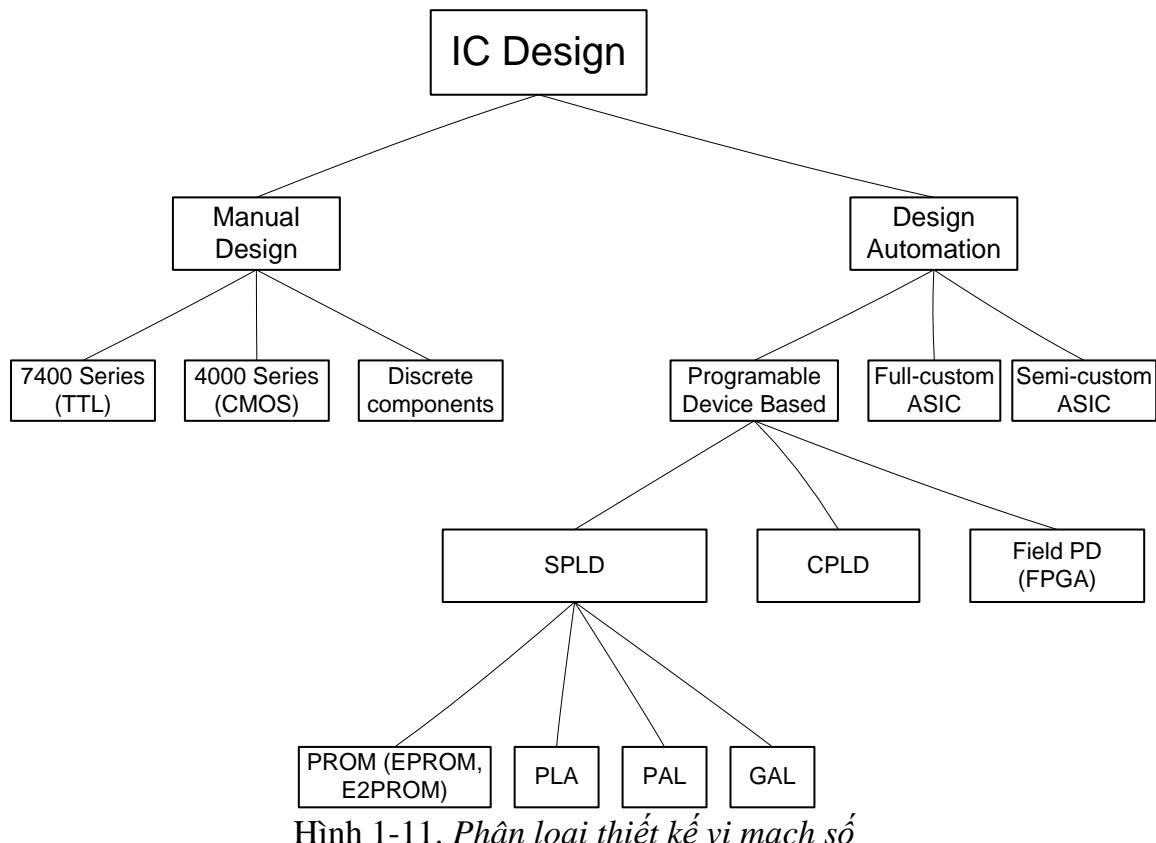
Các yêu cầu kể trên có quan hệ mật thiết với nhau và thông thường chúng không thể đồng thời đạt được tối ưu. Ví dụ năng lượng tiêu thụ của mạch muôn nhỏ thì số lượng công sử dụng hạn chế và sẽ hạn chế tốc độ làm việc, hoặc việc sử dụng các công nghệ rẻ tiền hơn hoặc dùng các công nghệ xuất thấp cũng là nhân tố giảm hiệu năng làm việc của mạch.

Trong thực tế Các IC phục vụ các mục đích khác nhau thì có yêu cầu khác nhau và người lập kế hoạch thiết kế chế tạo IC cần phải cân đối giữa các tiêu chí để có một phương án tối ưu nhất. Ví dụ cùng là vi xử lý nhưng nếu dùng thì không có yêu cầu đặc biệt về mặt tiêu hao năng lượng do nguồn cấp là cố định, khi đó Chip phải được thiết kế để có hiệu suất làm việc tối đa. Trong khi vi xử lý cho máy tính xách tay thì cần phải thiết kế để có mức tiêu thụ năng lượng thấp nhất có thể hoặc để có thể hoạt động ở nhiều mức tiêu thụ năng lượng khác nhau nhằm kéo dài thời gian sử dụng. Chip điều khiển cho các thiết bị di động thì cần phải tối ưu hết mức tiêu tốn năng lượng bằng cách thu gọn thiết kế, giảm thiểu những tập lệnh không cần thiết và sử dụng các phần tử tiết kiệm năng lượng nhất...

### **3. Các công nghệ thiết kế mạch logic số**

Vi mạch số đơn giản có thể được thiết kế thủ công (*Manual IC design*), nhưng với các vi mạch số cỡ lớn thì quá trình thiết kế buộc phải sử dụng các chương trình hỗ trợ thiết kế trên máy tính (*Design Automation*)

*Manual design:* Vi mạch số có thể được thiết kế bởi cách ghép nối các linh kiện bán dẫn rời rạc. Sự ra đời các IC đa dụng họ 74XX hay 40XX cho phép người sử dụng có thể tự thiết kế những mạch số cỡ nhỏ và cỡ vừa bằng cách ghép nối trên một bản mạch in. Nhờ có cấu trúc chuẩn hóa, có thể dễ dàng ghép nối, tạo những mạch chức năng khác nhau. Trên thực tế những mạch dạng này đã và vẫn đang được ứng dụng rộng rãi. Điểm hạn chế duy nhất của những thiết kế dạng này là chúng chỉ phù hợp cho những thiết kế SSI đơn giản do giới hạn về mật độ tích hợp và tốc độ làm việc thấp.



Hình 1-11. Phân loại thiết kế vi mạch số

*Design Automation* Máy tính là một sản phẩm đặc trưng nhất của nền công nghiệp sản xuất chế tạo bán dẫn nhưng ngay sau khi ra đời đã trở thành công cụ đặc lực cho việc thiết kế mô phỏng IC nói riêng và các thiết bị khác nói chung. Tự động hóa thiết kế không những giúp đơn giản hóa và rút ngắn đáng kể thời gian thiết kế sản phẩm mà còn đem lại những khả năng mà quá trình thiết kế thủ công bởi con người không làm được đó là:

- Khả năng làm việc với những thiết kế phức tạp tới cỡ hàng nghìn đến hàng tỷ *transistor*.
- Khả năng xử lý những bài toán tối ưu với nhiều tiêu chí và nhiều điều kiện ràng buộc phức tạp.
- Khả năng tự động tổng hợp thiết kế từ các mức trừu tượng cao xuống các mức trừu tượng thấp hơn một cách chính xác, nhanh chóng.
- Đơn giản hóa việc lưu trữ và trao đổi dữ liệu thiết kế.

Các phần mềm hỗ trợ thiết kế gọi chung là CAD Tools, trong lĩnh vực thiết kế ASIC có 3 hệ thống phần mềm phổ biến của Cadence®, Synopsys®, Magma® Design Automation Inc. Trong thiết kế trên FPGA phổ biến có Xilinx, Altera.

Trong tự động hóa thiết kế IC thường phân biệt thành những quy trình như sau:

*Full-custom ASIC*: là quy trình thiết kế IC có mức độ chi tiết cao nhất nhằm thu được sản phẩm có hiệu quả làm việc cao nhất trong khi vẫn đạt tối ưu về mặt tài nguyên trên nền một công nghệ bán dẫn nhất định. Để đạt được mục đích đó thiết kế không những được tối ưu ở những mức cao mà còn được tối ưu ở mức độ bố trí transistor và kết nối giữa chúng, ví dụ hai khối logic cùng thực hiện hàm OR nhưng phân bố ở hai vị trí khác nhau thì được cấu trúc bằng các mạch transistor khác nhau, phụ thuộc vào các thông số khác như tải đầu vào đầu ra, vị trí, ảnh hưởng các khối liền kề... Chính vì thế *Full-custom ASIC* đôi khi còn được gọi là *random-logic gate networks* nghĩa là mạch tạo bởi những cổng không đồng nhất.

*Semi-custom ASIC design*: Phân biệt với *Full-custom ASIC design*, khái niệm này chỉ quy trình thiết kế mà mức độ chi tiết không đạt đến tối đa, thông thường thiết kế đạt chi tiết đến mức cổng logic hoặc cao hơn. Do *Full-custom ASIC* có độ phức tạp cao nên không những chi phí cho quá trình thiết kế rất lớn mà khác thời gian dành cho thiết kế có thể kéo dài hàng vài năm trở lên, trong thời gian đó có thể đã có những công nghệ mới ra đời, mỗi một thay đổi nhỏ kéo theo việc phải làm lại gần như toàn bộ thiết kế và phát sinh thêm chi phí rất nhiều do vậy lợi nhuận sản phẩm bán ra thấp hay thậm chí thua lỗ. *Semi-custom ASIC* cân bằng giữa chi phí thiết kế và lợi nhuận thu được sản phẩm bằng cách đẩy nhanh và giảm thiểu chi phí cho quá trình thiết kế, dĩ nhiên bù lại sản phẩm làm ra không đạt được mức tối ưu lý thuyết như *Full-custom design*. Có nhiều dạng *Semi-custom design* nhưng một trong những kiểu cơ bản mà thường được sử dụng là thiết kế trên cơ sở thư viện cổng chuẩn (Standard Cell Library), thư viện này là tập hợp của các cổng logic như AND, OR, XOR, thanh ghi... và vì chúng có cùng kích thước chiều cao nên được gọi là cổng chuẩn.

*ASIC based on Programmable Device*: Thiết kế ASIC trên cơ sở IC khả trinh. Chíp khả trinh (*Programmable device*) được hiểu là IC chứa những phần tử logic có thể được lập trình can thiệp để tái cấu trúc nhằm thực hiện một chức năng nào đó. Quá trình tái cấu trúc thực hiện thông qua ngôn ngữ mô tả phần cứng nên thường được gọi ngắn gọn là lập trình.

IC khả trinh được chia thành các dạng sau:

*SPLD (Simple Programmable Logic Device)* Nhóm những IC khả trinh PROM, PAL, PLA, GAL. Đặc điểm chung của nhóm này là chứa một số lượng

cổng tương đương từ vài chục (PROM) đến vài trăm (PAL, GAL) cổng, nhóm này sử dụng cấu trúc của bộ nhớ ROM để lưu cấu hình IC, (vì vậy nhóm này còn gọi là *Memory-based PLD*), cấu trúc này bao gồm một mảng ma trận AND và một mảng ma trận OR có thể cấu trúc được. Trong các chip dạng này lại chia làm hai, thứ nhất là loại chỉ lập trình một lần, và loại có khả năng tái lập trình dùng các công nghệ như EEPROM hay EPROM. Cấu trúc cụ thể và nguyên lý làm việc của PROM, PAL, PLA, GAL, FPGA, CPLD sẽ được lần lượt được trình bày chi tiết ở phần tiếp theo.

**CPLD (Complex Programmable Logic Device)** CPLD là IC lập trình phức tạp thường được ghép từ nhiều các SPLD trên một chip đơn. Số cổng tương đương của CPLD đạt từ hàng nghìn đến hàng chục nghìn cổng.

**FPGA (Field-Programmable Gate Array)** là IC khả trinh cấu trúc từ mảng các khối logic lập trình được. Nếu như đối với các PLD khác việc tái cấu trúc IC được thực hiện trong điều kiện của nhà máy sản xuất bán dẫn, quá trình này cần những mặt nạ cho quang khắc nên sử dụng lớp những PLD này được gọi chung bằng thuật ngữ *Mask-Programmable Device*. FPGA phân biệt chính với các loại trên ở khả năng tái cấu trúc IC bởi người dùng cuối hay chính là người lập trình IC.

## 4. Kiến trúc của các IC khả trinh

Trong Kỹ thuật số ta đã chỉ ra mọi hàm logic tổ hợp đều có thể biểu diễn dưới dạng chuẩn tắc tuyến tíc 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 bằng cách ghép hai mảng ma trận nhân (AND) và ma trận cộng (OR). Nếu một trong các mảng này có tính khả trinh thì IC sẽ có tính khả trinh. Ta sẽ lần lượt nghiên cứu cấu trúc của một số loại IC hoạt động trên nguyên lý này.

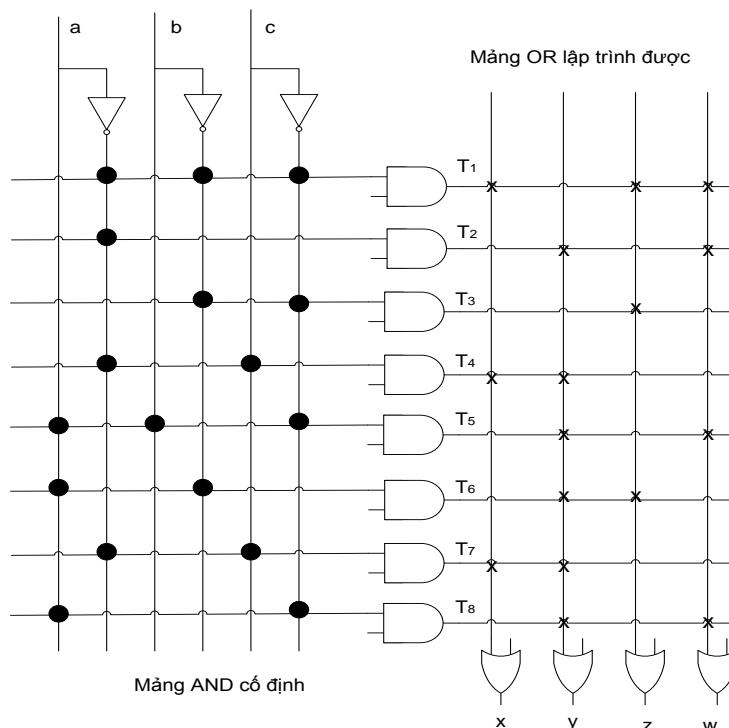
### 4.1. Kiến trúc PROM, PAL, PLA, GAL

#### 4.1.1. PROM

**PROM (Programmable Read-Only Memory)** được phát minh bởi Wen Tsing Chow năm 1956 khi làm việc tại Arma Division của công ty American Bosch Arma tại Garden, New York. PROM được chế tạo theo đơn đặt hàng từ lực lượng không quân của Mỹ lúc bấy giờ với mục đích có được một thiết bị lưu

trữ các tham số về mục tiêu một cách an toàn và linh động. Thiết bị này dùng trong máy tính của hệ thống phóng tên lửa Atlas E/F và được giữ bí mật trong vòng vài năm trước khi Atlas E/F trở nên phổ biến. PROM là vi mạch lập trình đầu tiên và đơn giản nhất trong nhóm các vi mạch bán dẫn lập trình được (*Programmable Logic Device*).

PROM có số đầu vào hạn chế, thông thường đến 16 đến 32 đầu vào, vì vậy chỉ thực hiện được những hàm đơn giản. Cấu trúc của PROM tạo bởi ma trận tạo bởi mảng cố định các phần tử AND nối với mảng các phần tử OR lập trình được.



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  $T_1, T_3$  như sau:

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

$$T_3 = \bar{b} \cdot \bar{c}$$

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:

$$Y = T_1 + T_3 = \bar{a} \cdot \bar{b} \cdot \bar{c} + \bar{b} \cdot \bar{c}$$
 (1.6)

Tính khả trìnhs 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 thế 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.

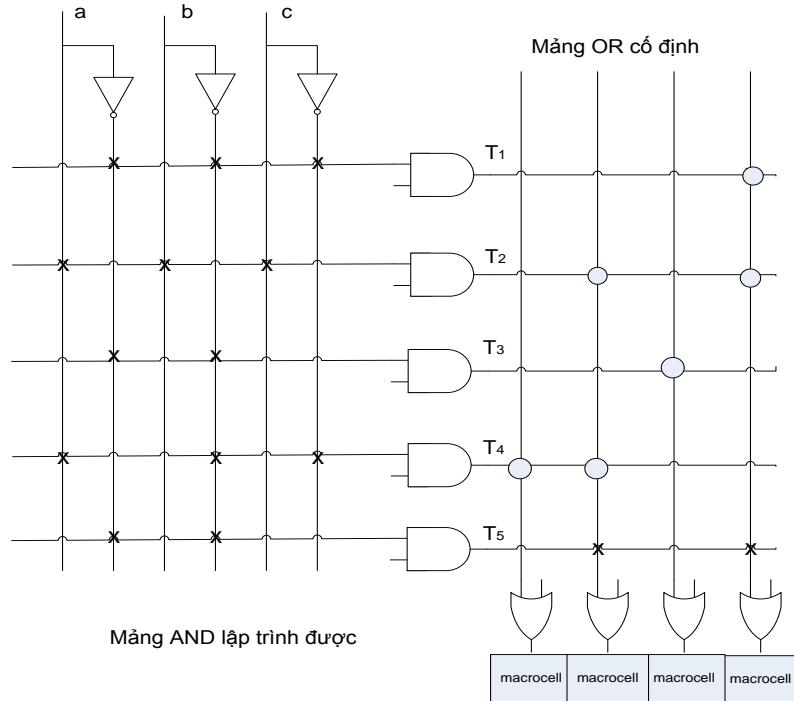
Những IC dạng PROM có khả năng tái lập trình là *UEPROM (Ultraviolet-Eraseable PROM)* sử dụng tia cực tím và *EEPROM (Electric-Eraseable PROM)* sử dụng hiệu điện thế ngưỡng cao để thiết lập lại các kết nối trong ma trận lập trình.

#### 4.1.2. PAL

PAL(*Programmable Array Logic*) ra đời cuối những năm 1970s. Cấu trúc của PAL kế thừa cấu trúc của PROM, 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.

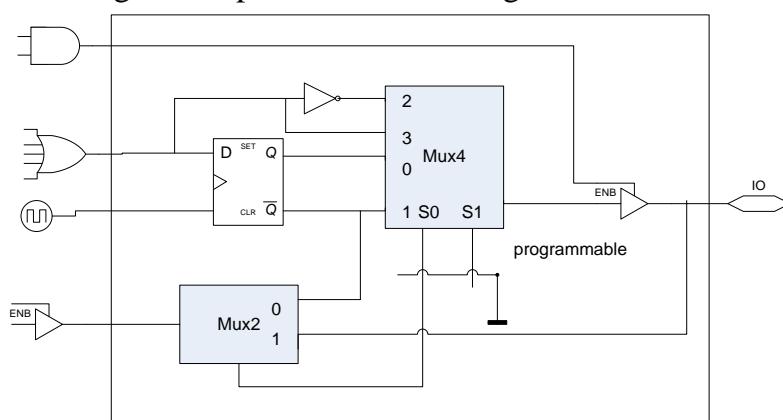
Ngoài ra cấu trúc cồng ra của PAL còn phân biệt với PROM ở mỗi đầu ra của mảng OR lập trình được được dẫn bởi khối logic gọi là *Macrocell*. Hình dưới đây minh họa cho cấu trúc của macrocell. Mỗi macrocell chứa 1 Flip-Flop Register, hai bộ *dòn kênh* (Multiplexers) 2 và 4 đầu vào Mux2, Mux4. Đầu ra của Mux2 thông qua một cồng 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.



Hình 1-13. Cấu trúc PAL

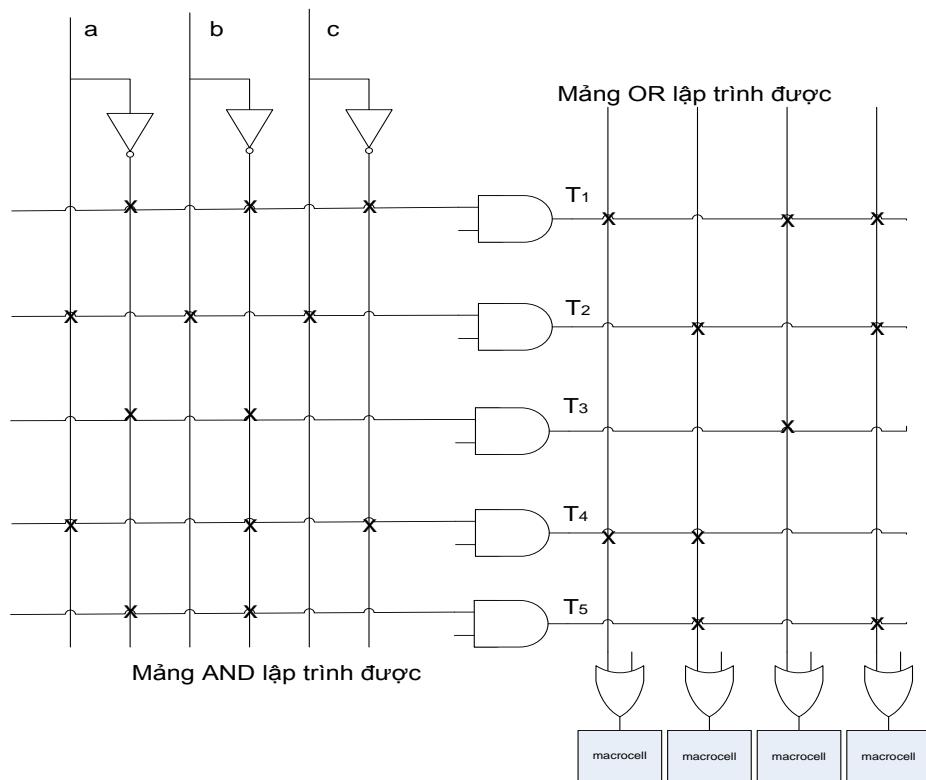
Đầu ra của macrocell cũng thông qua cổng 3 trạng thái có thể lập trình được để nối với cổng giao tiếp của PAL. Tín hiệu điều khiển của Mux4 có thể được lập trình để cho phép dẫn tín hiệu lần lượt qua các đầu vào 0,1,2,3 của Mux4 và gửi ra ngoài cổng giao tiếp IO, tùy thuộc vào cấu hình này mà tín hiệu tại IO có thể bị chặn (không gửi ra), dẫn trực tiếp từ mảng OR, thông qua thanh ghi Register. Nhờ cấu trúc macrocell PAL có thể được sử dụng không những để thực hiện các hàm logic tổ hợp mà cả các hàm logic tuần tự.



Hình 1-14. Cấu trúc Macrocell

#### 4.1.3. PLA

PLA (*Programmable Logic Array*) ra đời năm 1975 và là chíp lập trình thứ hai sau PROM. Cấu trúc của PLA không khác nhiều so với cấu trúc của PAL, ngoại trừ khả năng lập trình ở cả hai ma trận AND và OR. Nhờ cấu trúc đó PLA có khả năng lập trình linh động hơn, bù lại tốc độ của PLA thấp hơn nhiều so với PROM và PAL và các sản phẩm cùng loại khác. Thực tế PLA được ứng dụng không nhiều và nhanh chóng bị thay thế bởi những công nghệ mới hơn như PAL, GAL, CPLD...



Hình 1-15. Cấu trúc PLA

#### 4.1.4. GAL

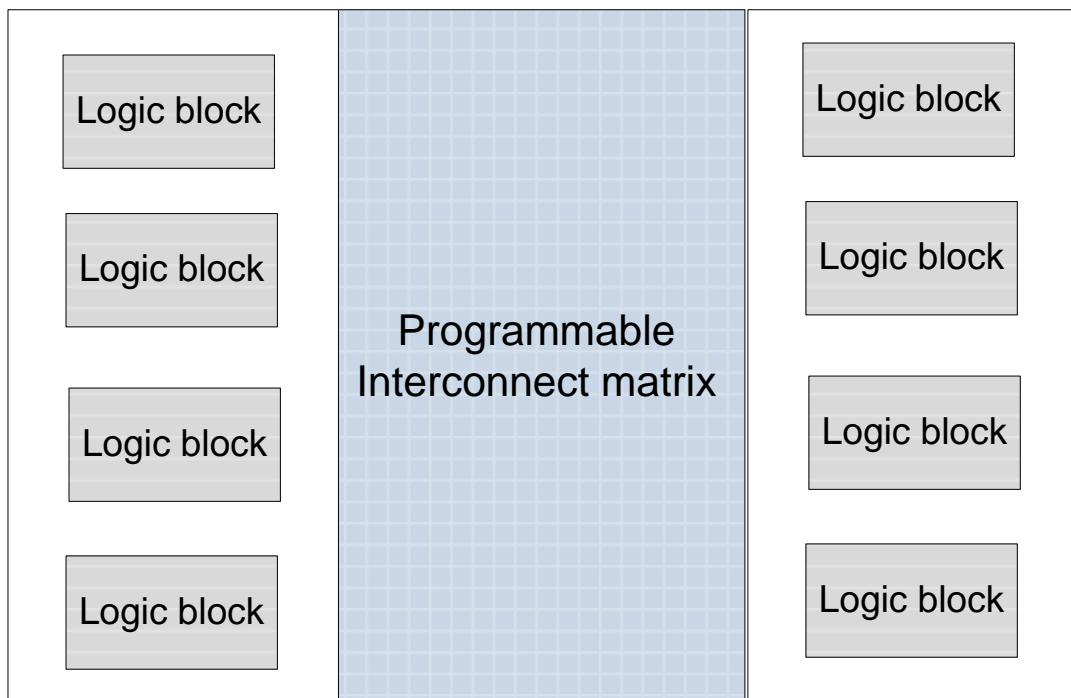
GAL (*Generic Array Logic*) được phát triển bởi Lattice Semiconductor company vào năm 1983, cấu trúc của GAL không khác biệt PAL nhưng thay vì lập trình sử dụng công nghệ antifuse thì ở GAL dùng *CMOS electrically erasable PROM*, chính vì vậy đôi khi tên gọi GAL ít được sử dụng thay vì đó GAL được hiểu như một dạng PAL được cải tiến.

## 4.2. Kiến trúc CPLD, FPGA

### 4.2.1. CPLD

Tất cả các chip khả trình PROM, PAL, GAL, thuộc nhóm *SPLD (Simple Programmable Logic Devices)* những IC này có ưu điểm là thiết kế đơn giản, chi phí thấp cho sản xuất cũng như thiết kế, có thể chuyển dễ dàng từ công nghệ này sang công nghệ khác tuy vậy nhược điểm là tốc độ làm việc thấp, số cổng logic tương đương nhỏ do đó không đáp ứng được những thiết kế phức tạp đòi hỏi nhiều về tài nguyên và tốc độ.

*CPLD (Complex Programmable Logic Devices)* được Altera tiên phong nghiên cứu chế tạo đầu tiên nhằm tạo ra những IC khả trình dung lượng lớn MAX5000, MAX7000, MAX9000 là họ những CPLD tiêu biểu của hãng này. Sau sự thành công của Altera một loạt các hãng khác cũng bắt tay vào nghiên cứu chế tạo CPLD, Xilinx với các sản phẩm XC95xx series, Lattice với isp Mach 4000 serise, ispMarch XO...



Hình 1-16. Cấu trúc CPLD

Một cách đơn giản nhất có thể hiểu CPLD được cấu trúc bằng cách ghép nhiều các chíp SPLD lại, thông thường là PAL. Tuy vậy về bản chất độ phức tạp của CPLD vượt xa so với các IC nhóm SPLD và cấu trúc của các CPLD cũng rất

đa dạng, phụ thuộc vào từng hãng sản xuất cụ thể. Dưới đây sẽ trình bày nguyên lý chung nhất của các chip họ này.

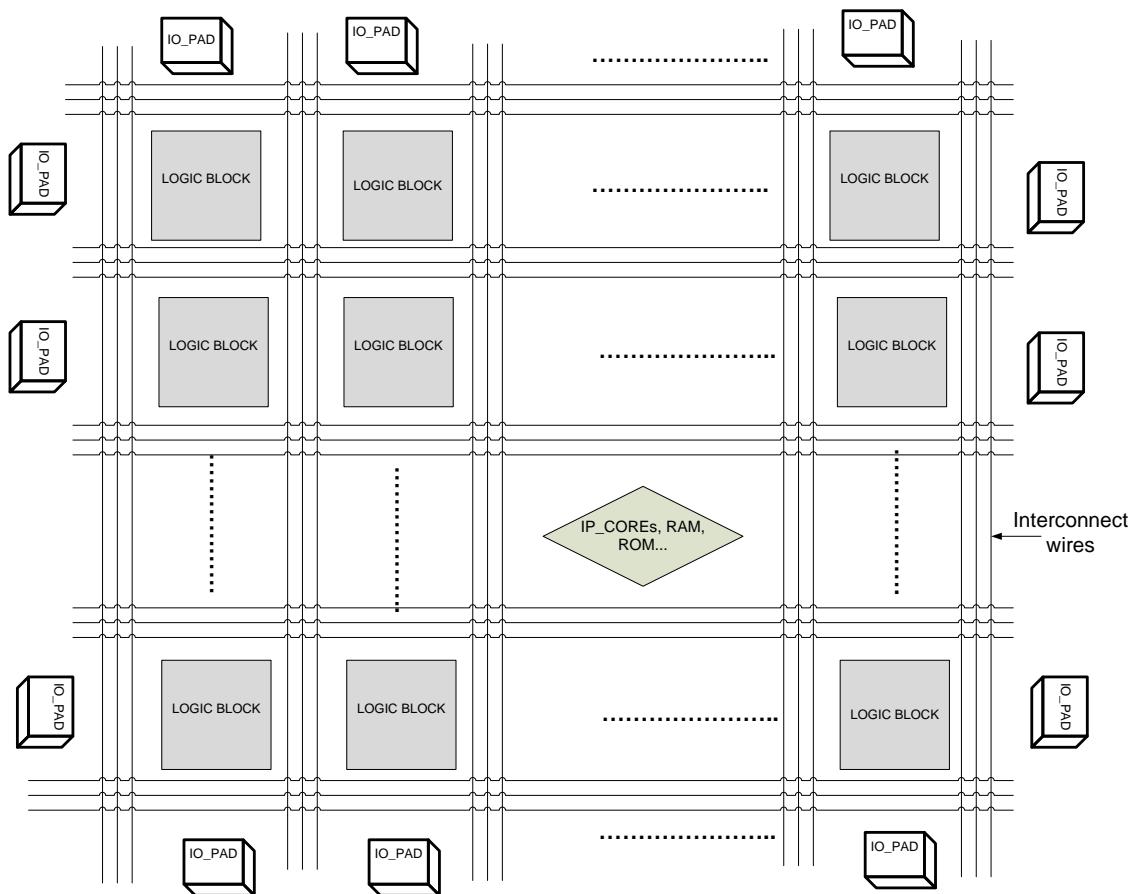
CPLD được tạo từ hai thành phần cơ bản là nhóm các khối logic (*Logic block*) và một ma trận kết nối khả trinh PIM (*Programmable Interconnect Matrix*). Logic block là các SPLD được cải tiến thường chứa từ 8 đến 16 macrocells. Tất cả các Logic block giống nhau về mặt cấu trúc. PIM là ma trận chứa các kết nối khả trinh, nhiệm vụ của ma trận này là thực hiện kết nối giữa các LB và các cổng vào ra IO của CPLD. Về mặt lý thuyết thì ma trận này có thể thực hiện kết nối giữa hai điểm bất kỳ.

CPLD thông thường sử dụng các công nghệ lập trình của EEPROM, điểm khác biệt là đối với CPLD thường không thể dùng những programmer đơn giản cho PAL, PLA... vì số chân giao tiếp của CPLD rất lớn. Để thực hiện cấu hình cho CPLD mỗi một công ty phát triển riêng cho mình một bộ công cụ và giao thức, thông thường các chip này được gắn trên một bo mạch in và dữ liệu thiết kế được tải vào từ máy vi tính. Tuy vậy các quy trình nạp trên đang dần bị thay thế bởi giao thức chuẩn JTAG (*Joint Test Action Group*) chuẩn, đây cũng là giao thức dùng để cấu trúc cho FPGA mà ta sẽ nghiên cứu kỹ hơn ở chương kế tiếp.

Nhờ kế thừa cấu trúc của SPLD nên CPLD không cần sử dụng bộ nhớ ROM ngoài để lưu cấu hình của IC, đây là một đặc điểm cơ bản nhất phân biệt CPLD với các IC khả trinh cỡ lớn khác như FPGA.

#### 4.2.2. FPGA

Về cấu trúc chi tiết và cơ chế làm việc của FPGA sẽ được dành riêng giới thiệu trong chương sau. Ở đây chỉ giới thiệu kiến trúc tổng quan nhất của IC dạng này. FPGA được cấu thành từ các khối logic (*Logic Block*) được bố trí dưới dạng ma trận, chúng được nối với nhau thông qua hệ thống các kênh kết nối lập trình được Hệ thống này còn có nhiệm vụ kết nối với các cổng giao tiếp IO\_PAD của FPGA.



Hình 1-17. Kiến trúc tổng quan của 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. Ngoài khả năng đó FPGA có mật độ tích hợp logic lớn nhất trong số các IC khả trình với số cổng tương đương lên tới hàng trăm nghìn, hàng triệu cổng. FPGA không dùng các mảng lập trình giống như trong cấu trúc của PAL, PLA mà dùng ma trận các khối logic. Điểm khác biệt cơ bản thứ ba của FPGA so với các IC kẽ trên là ở cơ chế tái cấu trúc, toàn bộ cấu hình của FPGA thường được lưu trong một bộ nhớ động (RAM), chính vì thế mà khi ứng dụng FPGA thường phải kèm theo một ROM ngoại vi để nạp cấu hình cho FPGA mỗi lần làm việc. Kiến trúc và cách thức làm việc của FPGA sẽ được nghiên cứu cụ thể ở chương thứ 3 của giáo trình này.

## **Câu hỏi ôn tập chương 1**

1. Transistor khái niệm, phân loại.
2. Khái niệm, phân loại vi mạch số tích hợp.
3. Cỗng logic cơ bản, tham số thời gian của cỗng logic tổ hợp.
4. Các loại Flip-flop cơ bản, tham số thời gian của Flip-flop.
5. Khái niệm mạch logic tổ hợp, cách xác định độ trễ trên mạch tổ hợp, khái niệm critical paths.
6. Khái niệm mạch dãy, cách tính thời gian trễ trên mạch dãy, khái niệm RTL, phương pháp tăng hiệu suất mạch dãy.
7. Các yêu cầu chung đối với thiết kế mạch logic số.
8. Các phương pháp thể hiện thiết kế mạch logic số.
9. Các công nghệ thiết kế mạch logic số, khái niệm, phân loại.
10. Trình bày sơ lược về các công nghệ thiết kế IC số trên chip khả trình.
11. Nguyên lý hiện thực hóa các hàm logic trên các IC khả trình dạng PROM, PAL, PLA, GAL.
12. Khái niệm thiết kế ASIC, các dạng thiết kế ASIC.
13. Khái niệm FPGA, đặc điểm FPGA.



## **Chương 2**

### **NGÔN NGỮ MÔ TẢ PHẦN CỨNG VHDL**

Chương 2 tập trung vào giới thiệu về ngôn ngữ mô tả phần cứng VHDL, đây là một ngôn ngữ mô tả phần cứng có tính ứng dụng cao nhưng cũng có cú pháp không quen thuộc và dễ tiếp cận. Nội dung kiến thức trình bày trong chương này theo định hướng như một tài liệu tra cứu hơn là bài giảng. Người học không nhất thiết phải theo đúng trình tự kiến thức trình bày mà có thể tham khảo tất cả các mục một cách độc lập, bên cạnh đó là tra cứu bằng các tài liệu khác cũng như tài liệu gốc bằng tiếng Anh. Các ví dụ có trong giáo trình đều có găng trình bày là các ví dụ đầy đủ có thể biên dịch và mô phỏng được ngay vì vậy khuyến khích người học tích cực thực hành song song với nghiên cứu lý thuyết.

Kết thúc nội dung của chương này yêu cầu người học phải có kỹ năng sử dụng VHDL ở cấp độ cơ bản, có khả năng thiết kế các khối số vừa và nhỏ như Flip-flop, khối chọn kênh, phân kênh, khối cộng, dịch, các khối giải mã... đã biết trong chương trình Điện tử số, đó cũng là các khối nền tảng cho các thiết kế lớn hơn và phức tạp hơn ở chương tiếp theo.

## 1. Giới thiệu về VHDL

VHDL viết tắt của VHSIC HDL (*Very-high-speed-integrated-circuit Hardware Description Language*) hay ngôn ngữ mô tả phần cứng cho các vi số mạch tích hợp tốc độ cao. Lịch sử phát triển của VHDL trải qua các mốc chính như sau:

- 1981: Phát triển bởi Bộ Quốc phòng Mỹ nhằm tạo ra một công cụ thiết kế phần cứng tiện dụng có khả năng độc lập với công nghệ và giảm thiểu thời gian cũng như chi phí cho thiết kế
- 1983-1985: Được phát triển thành một ngôn ngữ chính thống bởi 3 công ty Intermetrics, IBM and TI.
- 1986: Chuyển giao toàn bộ bản quyền cho Viện Kỹ thuật Điện và Điện tử (IEEE).
- 1987: Công bố thành một chuẩn ngôn ngữ IEEE-1076 1987.
- 1994: Công bố chuẩn VHDL IEEE-1076 1993.
- 2000: Công bố chuẩn VHDL IEEE-1076 2000.
- 2002: Công bố chuẩn VHDL IEEE-1076 2002
- 2007: công bố chuẩn ngôn ngữ Giao diện ứng dụng theo thủ tục VHDL IEEE-1076c 2007
- 2009: Công bố chuẩn VHDL IEEE-1076 2009

VHDL ra đời trên yêu cầu của bài toán thiết kế phần cứng lúc bấy giờ, nhờ sử dụng ngôn ngữ này mà thời gian thiết kế của sản phẩm bán dẫn giảm đi đáng kể, đồng thời với giảm thiểu chi phí cho quá trình này do đặc tính độc lập với công nghệ, với các công cụ mô phỏng và khả năng tái sử dụng các khối đơn lẻ. Các ưu điểm chính của VHDL có thể liệt kê ra là:

*Tính công cộng:* VHDL là ngôn ngữ được chuẩn hóa chính thức của IEEE do đó được sự hỗ trợ của nhiều nhà sản xuất thiết bị cũng như nhiều nhà cung cấp công cụ thiết kế mô phỏng hệ thống, hầu như tất cả các công cụ thiết kế của các hãng phần mềm lớn nhỏ đều hỗ trợ biên dịch VHDL.

*Được hỗ trợ bởi nhiều công nghệ:* VHDL có thể sử dụng mô tả nhiều loại vi mạch khác nhau trên những công nghệ khác nhau từ các thư viện rời rạc, CPLD, FPGA, tới thư viện công chuẩn cho thiết kế ASIC.

*Tính độc lập với công nghệ:* VHDL hoàn toàn độc lập với công nghệ chế tạo phần cứng. Một mô tả hệ thống chức năng dùng VHDL thiết kế ở mức thanh ghi truyền tải RTL có thể được tổng hợp thành các mạch trên các công nghệ bán dẫn khác nhau. Nói một cách khác khi một công nghệ phần cứng mới ra đời nó

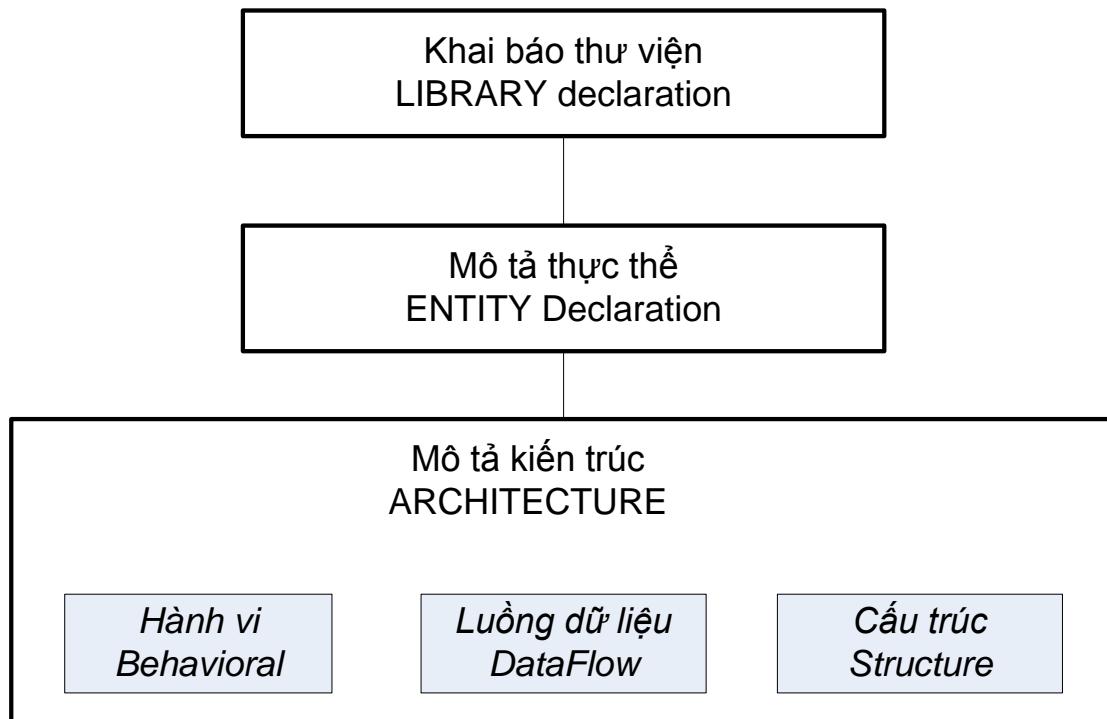
có thể được áp dụng ngay cho các hệ thống đã thiết kế bằng cách tổng hợp các thiết kế đó lại trên thư viện phần cứng mới.

*Khả năng mô tả mở rộng:* VHDL cho phép mô tả hoạt động của phần cứng từ mức thanh ghi truyền tải (*RTL–Register Transfer Level*) cho đến mức cổng (*Netlist*). Hiểu một cách khác VHDL có một cấu trúc mô tả phần cứng chặt chẽ có thể sử dụng ở lớp mô tả chức năng cũng như mô tả cổng trên một thư viện công nghệ cụ thể nào đó.

*Khả năng trao đổi, tái sử dụng:* Việc VHDL được chuẩn hóa giúp cho việc trao đổi các thiết kế giữa các nhà thiết kế độc lập trở nên hết sức dễ dàng. Bản thiết kế VHDL được mô phỏng và kiểm tra có thể được tái sử dụng trong các thiết kế khác mà không phải lặp lại các quá trình trên. Giống như phần mềm thì các mô tả HDL cũng có một cộng đồng mã nguồn mở cung cấp, trao đổi miễn phí các thiết kế chuẩn có thể ứng dụng ở nhiều hệ thống khác nhau.

## 2. Cấu trúc của chương trình mô tả bằng VHDL

Cấu trúc tổng thể của một khối thiết kế 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

## 2.1. Khai báo thư viện

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.

Ví dụ về khai báo thư viện

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

Khai báo thư viện bắt đầu bằng từ khóa **Library** Tên thư viện (chú ý là VHDL không phân biệt chữ hoa chữ thường). Sau đó trên từng dòng kế tiếp sẽ khai báo các gói thư viện con mà thiết kế sẽ sử dụng, mỗi dòng phải kết thúc bằng dấu “;”.

Tương tự như đối với các ngôn ngữ lập trình khác, người thiết kế có thể khai báo sử dụng các thư viện chuẩn hoặc thư viện người dùng. Thư viện IEEE gồm nhiều gói thư viện con khác nhau trong đó đáng chú ý có các thư viện sau:

- Gói IEEE.STD\_LOGIC\_1164 cung cấp các kiểu dữ liệu std\_ulogic, std\_logic, std\_ulogic\_vector, std\_logic\_vector, các hàm logic and, or, not, nor, xor... các hàm chuyển đổi giữa các kiểu dữ liệu trên. std\_logic, std\_ulogic hỗ trợ kiểu logic với 9 mức giá trị logic (xem 4.2)
- Gói IEEE.STD\_LOGIC\_ARITH định nghĩa các kiểu dữ liệu số nguyên SIGNED, UNSIGNED, INTEGER, SMALL INT cung cấp các hàm số học bao gồm “+”, “-”, “\*”, “/”, so sánh “<”, “>”, “<=”, “>=”, các hàm dịch trái, dịch phải, các hàm chuyển đổi từ kiểu vector sang các kiểu số nguyên.
- Gói IEEE.STD\_LOGIC\_UNSIGNED Cung cấp các hàm số học logic làm việc với các kiểu dữ liệu std\_logic, integer và trả về giá trị dạng std\_logic với giá trị không có dấu
- Gói IEEE.STD\_LOGIC\_SIGNED Cung cấp các hàm số học logic làm việc với các kiểu dữ liệu std\_logic, integer và trả về giá trị dạng std\_logic với giá trị có dấu
- Gói IEEE.NUMERIC\_BIT Cung cấp các hàm số học logic làm việc với các kiểu dữ liệu signed, unsigned được là chuỗi của các BIT.
- Gói IEEE.NUMERIC\_BIT Cung cấp các hàm số học logic làm việc với các kiểu dữ liệu signed, unsigned được là chuỗi của các STD\_LOGIC.

- Gói IEEE.STD\_LOGIC\_TEXTIO chứa các hàm, thủ tục vào ra READ/WRITE để đọc ghi dữ liệu cho các định dạng STD\_LOGIC, STD\_ULOGIC từ FILE, STD\_INPUT, STD\_OUTPUT, LINE.
- Gói STD.TEXTIO chứa các hàm vào ra READ/WRITE để đọc ghi dữ liệu với các định dạng khác nhau gồm, BIT, INTEGER, TIME, REAL, CHARACTER, STRING từ FILE, STD\_INPUT, STD\_OUTPUT, LINE.
- Gói IEEE.MATH\_REAL, IEEE.MATH\_COMPLEX cung cấp các hàm làm việc với số thực và số phức như SIN, COS, SQRT... hàm làm tròn, CIEL, FLOOR, hàm tạo số ngẫu nhiên SRAND, UNIFORM... và nhiều các hàm tính toán số thực khác.
- Gói STD.ENV cung cấp các hàm, thủ tục hệ thống phục vụ mô phỏng gồm stop – dừng mô phỏng, finish – thoát chương trình, resolution\_limit trả về bước thời gian mô phỏng.

Cụ thể và chi tiết hơn về các thư viện chuẩn của IEEE có thể tham khảo thêm trong tài liệu của IEEE (*VHDL Standard Language reference*), và xem thêm phần **Phụ lục 1** cuối sách liệt kê và phân loại đầy đủ các hàm của các thư viện chuẩn.

## 2.2. Mô tả thực thể

Khai báo thực thể (*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.

```
entity identifier is
    generic (generic_variable_declarations);
    port (input_and_output_variable_declarations);
end entity identifier;
```

Trong đó

- *identifier* là tên của khối thiết kế.
- khai báo *generic* là khai báo các tham số tĩnh của thực thể, khai báo này rất hay sử dụng cho những thiết kế có những tham số thay đổi như độ rộng kênh, kích thước ô nhớ, tham số bộ đếm... ví dụ chúng ta có thể thiết kế bộ cộng cho các hạng tử có độ dài bit thay đổi, số bit được thể hiện là hằng số trong khai báo generic (xem ví dụ dưới đây)
- Khai báo cổng vào ra: liệt kê tất cả các cổng giao tiếp của khối thiết kế, các cổng có thể hiểu là các kênh dữ liệu động của thiết kế để phân biệt với các tham số tĩnh trong khai báo generic. kiểu của các cổng có thể là:
  - **in:** cổng vào,

- **out**: công ra,
- **inout** vào ra hai chiều.
- **buffer**: công đệm có thể sử dụng như tín hiệu bên trong và **output**.
- **linkage**: Có thể làm bất kỳ các công nào kể trên.

Ví dụ cho khai báo thực thể như sau:

```
entity adder is
    generic ( N      : natural := 32 );
    port      ( A      : in  bit_vector(N-1 downto 0);
                B      : in  bit_vector(N-1 downto 0);
                cin   : in  bit;
                Sum   : out bit_vector(N-1 downto 0);
                Cout  : out bit );
end entity adder ;
```

Đoạn mã trên khai báo một thực thể cho khối cộng hai số, trong khai báo trên N là tham số tĩnh **generic** chỉ độ dài bit của các hạng tử, giá trị ngầm định N = 32, việc khai báo giá trị ngầm định là không bắt buộc. Khi khối này được sử dụng trong khối khác như một khối con khác thì có thể thay đổi giá trị của N để thu được thiết kế theo mong muốn. Về các cổng vào ra, khối cộng hai số nguyên có 3 cổng vào A, B N-bit là các hạng tử và cổng cin là bít nhớ từ bên ngoài. Hai cổng ra là Sum N-bit là tổng và bít nhớ ra Cout.

Khai báo thực thể có thể chứa chỉ mình khai báo cổng như sau:

```
entity full_adder is
    port (
        X, Y, Cin  : in  bit;
        Cout, Sum  : out bit
    );
end full_adder ;
```

Khai báo thực thể không chứa cả khai báo **generic** lẫn khai báo **port** vẫn được xem là hợp lệ, ví dụ những khai báo thực thể sử dụng để mô phỏng kiểm tra thiết kế thường được khai báo như sau:

```
entity TestBench is
end TestBench;
```

Ví dụ về cổng dạng **buffer** và **inout**: Cổng buffer được dùng khi tín hiệu được sử dụng như đầu ra đồng thời như một tín hiệu bên trong của khố thiết kế, điển hình như trong các mạch dãy làm việc đồng bộ. Xét ví dụ sau về bộ cộng tích lũy 4-bit đơn giản sau (accumulator):

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;
use IEEE.STD_LOGIC_arith.ALL;
```

```

-----
entity accumulator is
    port(
        data  : in      std_logic_vector(3 downto 0);
        nRST  : in      std_logic;
        CLK   : in      std_logic;
        acc   : buffer  std_logic_vector(3 downto 0)
    );
end accumulator;
-----
architecture behavioral of accumulator is
begin
    ac : process (CLK)
    begin
        if CLK = '1' and CLK'event then
            if nRST = '1' then
                acc <= "0000";
            else
                acc <= acc + data;
            end if;
        end if;
    end process ac;
end behavioral;
-----
```

Bộ cộng tích lũy sau mỗi xung nhịp CLK sẽ cộng giá trị hiện có lưu trong acc với giá trị ở đầu vào data, tín hiệu nRST dùng để thiết lập lại giá trị bằng 0 cho acc. Như vậy acc đóng vai trò như thanh ghi kết quả đầu ra cũng như giá trị trung gian được khai báo dưới dạng buffer. Trên thực tế thay vì dùng công buffer thường sử dụng một tín hiệu trung gian, khi đó công acc có thể khai báo như công ra bình thường, cách sử dụng như vậy sẽ tránh được một số lỗi có thể phát sinh khi tổng hợp thiết kế do khai báo buffer gây ra.

Ví dụ sau đây là mô tả VHDL của một khối đệm ba trạng thái 8-bit, sử dụng khai báo công INOUT. Công ba trạng thái được điều khiển bởi tín hiệu OE, khi OE bằng 0 giá trị của công là trạng thái trờ kháng cao “ZZZZZZZZ”, khi OE bằng 1 thì công kết nối đầu vào inp với outp.

```

-----
library ieee;
use ieee.std_logic_1164.all;
-----
entity bidir is
    port(
        bidir  : inout std_logic_vector (7 downto 0);
        oe, clk : in    std_logic_LOGIC;
-----
```

```

        inp      : in      std_logic_vector (7 downto 0);
        outp     : out     std_logic_vector (7 downto 0)
    );
END bidir;
-----
architecture behav of bidir is
signal a : std_logic_vector (7 downto 0);
signal b : std_logic_vector (7 downto 0);
begin
    process(clk)
    begin
        if clk = '1' and clk'event then
            a    <= inp;
            outp <= b;
        end if;
    end process;
    process (oe, bidir)
    begin
        if( oe = '0') then
            bidir <= "ZZZZZZZZ";
            b     <= bidir;
        else
            bidir <= a;
            b      <= bidir;
        end if;
    end process;
end maxpld;
-----
```

*\* 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. Chi tiết hơn về các thành phần khai báo này có thể xem trong [5] IEEE VHDL Standard Language reference (2002 Edition).*

### 2.3. Mô tả kiến trúc

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. Cấu trúc của mô tả kiến trúc tổng quát như sau:

```

architecture identifier of entity_name is
    [ declarations]
begin
```

```
[ statements ]
end identifier ;
```

Trong đó

- *identifier* là tên gọi của kiến trúc, thông thường để phân biệt các kiểu mô tả thường dùng các tên **behavioral** cho mô tả hành vi, **dataflow** cho mô tả luồng dữ liệu, **structure** cho mô tả cấu trúc tuy vậy có thể sử dụng một tên gọi hợp lệ bất kỳ nào khác.

- [declarations] có thể có hoặc không chứa các khai báo cho phép như sau:

Khai báo và mô tả chương trình con (subprogram)

Khai báo kiểu dữ liệu con (subtype)

Khai báo tín hiệu (signal), hằng số (constant), file

Khai báo khối con (component)

-[statements] phát biểu trong khối {begin end process;} chứa các phát biểu đồng thời (concurrent statements) hoặc các khối process chứa các phát biểu tuần tự (sequential statements).

Có ba dạng mô tả cấu trúc cơ bản là mô tả hành vi (*behavioral*), mô tả luồng dữ liệu (*dataflow*) và mô tả cấu trúc (*structure*). Trên thực tế trong mô tả kiến trúc của những khối phức tạp thì sử dụng kết hợp cả ba dạng mô tả này. Để tìm hiểu về ba dạng mô tả kiến trúc ta sẽ lấy ví dụ về khối *full\_adder* có khai báo entity như sau:

```
entity full_adder is
    port ( A      : in  std_logic;
           B      : in  std_logic;
           cin   : in  std_logic;
           Sum   : out std_logic;
           Cout  : out std_logic);
end entity full_adder;
```

### 2.3.1. Mô tả hành vi

Đối với thực thể *full\_adder* như trên kiến trúc hành vi (*behavioral*) được viết như sau

```
-----
architecture behavioral of full_adder is
begin
    add: process (A,B,Cin)
    begin
        if (a ='0' and b='0' and Cin = '0') then
            S      <= '0';
```

```

        Cout <='0';
elsif (a ='1' and b='0' and Cin = '0') or
      (a ='0' and b='1' and Cin = '0') or
      (a ='0' and b='0' and Cin = '1') then
  S    <= '1';
  Cout <='0';
elsif (a ='1' and b='1' and Cin = '0') or
      (a ='1' and b='0' and Cin = '1') or
      (a ='0' and b='1' and Cin = '1') then
  S    <= '0';
  Cout <= '1';
elsif (a ='1' and b='1' and Cin = '1') then
  S    <= '1';
  Cout <= '1';
end if;
end process add;
end behavioral;
-----
```

Mô tả hành vi gần giống như mô tả bằng lời cách thức tính toán kết quả đầu ra dựa vào các giá trị đầu vào. Toàn bộ mô tả hành vi phải được đặt trong một khối quá trình **{process** (sensitive 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.

Việc mô tả bằng hành vi không thể hiện rõ được cách thức cấu tạo phần cứng của vi mạch như các dạng mô tả khác và tùy theo những cách viết khác nhau thì có thể thu được những kết quả tổng hợp khác nhau.

Trong các mạch dãy đồng bộ, khối làm việc đồng bộ thường được mô tả bằng hành vi, ví dụ như trong đoạn mã sau mô tả thanh ghi sau:

```

process(clk)
begin
  if clk'event and clk='1' then
    Data_reg <= Data_in;
  end if;
end process;
```

### 2.3.2. Mô tả luồng dữ liệu

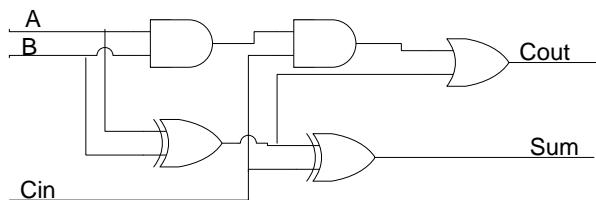
Mô tả luồng dữ liệu (*dataflow*) là dạng mô tả tương đối ngắn gọn và rất hay được sử dụng khi mô tả các khối mạch tổ hợp. Các phát biểu trong khối

`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. Ví dụ cho khói `full_adder` thì mô tả luồng dữ liệu như sau:

```
architecture dataflow of full_adder is
begin
    sum  <= (a xor b) xor Cin;
    Cout <= (a and b) or (Cin and (a xor b));
end dataflow;
```

### 2.3.3. Mô tả cấu trúc

Mô tả cấu trúc (*structure*) là mô tả sử dụng các mô tả có sẵn dưới dạng khói con (*component*). Dạng mô tả này cho kết quả sát với kết quả tổng hợp nhất. Theo mô tả luồng dữ liệu như ở trên ta thấy dùng hai cổng XOR, một cổng OR và 2 cổng AND để thực hiện thiết kế giống như hình dưới đây:



Hình 2-2. Sơ đồ logic của khói cộng đầy đủ (FULL\_ADDER)

Trước khi viết mô tả cho `full_adder` cần phải viết mô tả cho các phần tử cổng AND, OR, XOR như sau

```
----- 2 input AND gate -----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity AND2 is
    port(
        in1, in2 : in std_logic;
        out1      : out std_logic
    );
end AND2;
-----
architecture model_conc of AND2 is
begin
    out1 <= in1 and in2;
end model_conc;
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```

----- 2 input OR gate -----
entity OR2 is
port (
    in1, in2 : in std_logic;
    out1      : out std_logic
);
end OR2;
-----
architecture model_conc2 of AND2 is
begin
    out1 <= in1 or in2;
end model_conc2;
----- 2 input XOR gate -----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity XOR2 is
port (
    in1, in2 : in std_logic;
    out1      : out std_logic);
end XOR2;
-----
architecture model_conc2 of XOR2 is
begin
    out1 <= in1 xor in2;
end model_conc2;

```

Sau khi đã có các cổng trên có thể thực hiện viết mô tả cho full\_adder như sau

```

-----
architecture structure of full_adder is
signal t1, t2, t3: std_logic;

component AND2
port (
    in1, in2 : in std_logic;
    out1      : out std_logic
);
end component;
component OR2
port (
    in1, in2 : in std_logic;
    out1      : out std_logic);
end component;
component XOR2
port (
    in1, in2 : in std_logic;

```

```

        out1      : out std_logic
    );
end component;

begin
u1 : XOR2 port map (a, b, t1)
u2 : XOR2 port map (t1, Cin, Sum)
u3 : AND2 port map (t1, Cin, t2)
u4 : AND2 port map (a, b, t3)
u5 : OR2  port map (t3, t2, Cout);

end structure;
-----
```

Như vậy mô tả cấu trúc tuy khá dài nhưng là mô tả cụ thể về cấu trúc mạch, ưu điểm của phương pháp này là khi tổng hợp trên thư viện công sẽ cho ra kết quả đúng với ý tưởng thiết kế nhất. Với mô tả full\_adder như trên thì gần chắc chắn trình tổng hợp đưa ra sơ đồ logic sử dụng 2 cổng XOR, hai cổng AND và 1 cổng OR. Một khía cạnh khác mô tả cấu trúc cho phép gộp nhiều mô tả con vào một khối thiết kế lớn mà vẫn giữ được cấu trúc mã rõ ràng và khoa học. Nhược điểm là không thể hiện rõ ràng chức năng của mạch như hai mô tả ở các phần trên.

Ở ví dụ trên có sử dụng khai báo cài đặt khối con, chi tiết về khai báo này xem trong mục 7.5.

## 2.4. Khai báo cấu hình

Một thực thể có thể có rất nhiều kiến trúc khác nhau. Bên cạnh đó cấu trúc của ngôn ngữ VHDL cho phép sử dụng các khối thiết kế theo kiểu lồng ghép, vì vậy đối với một thực thể bất kỳ cần có thêm các mô tả để quy định việc sử dụng các kiến trúc khác nhau. Khai báo cấu hình (*Configuration declaration*) được sử dụng để chỉ ra kiến trúc nào sẽ được sử dụng trong thiết kế.

Cách thứ nhất để sử dụng khai báo cấu hình là sử dụng trực tiếp khai báo cấu hình bằng cách tạo một đoạn mã cấu hình độc lập không thuộc một thực thể hay kiến trúc nào theo cấu trúc:

```

configuration identifier of entity_name is
    [declarations]
    [block configuration]
end configuration identifier;
```

Ví dụ sau tạo cấu hình có tên add32\_test\_config cho thực thể add32\_test, cấu hình này quy định cho kiến trúc có tên circuits của thực thể add32\_test, khi cài đặt các khối con có tên add32 sử dụng kiến trúc tương ứng là WORK.add32(circuits), với mọi khía cạnh con add4c của thực thể add32 thì sử dụng

kiến trúc WORK.add4c(circuit), tiếp đó là quy định mọi khối con có tên fadd trong thực thể add4c sử dụng kiến trúc có tên WORK.fadd(circuits).

```
configuration add32_test_config of add32_test is
    for circuits -- of add32_test
        for all: add32
            use entity WORK.add32(circuits);
        for circuits -- of add32
            for all: add4c
                use entity WORK.add4c(circuits);
            for circuits -- of add4c
                for all: fadd
                    use entity WORK.fadd(circuits);
                end for;
            end for;
            end for;
        end for;
    end for;
end configuration add32_test_config;
```

Cặp lệnh cơ bản của khai báo cấu hình là cặp lệnh `for... use ... end for` có tác dụng quy định cách thức sử dụng các kiến trúc khác nhau ứng với các khối khác nhau trong thiết kế. Bản thân configuration cũng có thể được sử dụng như đối tượng của lệnh `use`, ví dụ:

```
configuration adder_behav of adder4 is
    for structure
        for all: full_adder
            use entity work.full_adder (behavioral);
        end for;
    end for;
end configuration;
```

Với một thực thể có thể khai báo nhiều cấu hình khác nhau tùy theo mục đích sử dụng. Sau khi được khai báo như trên và biên dịch thì sẽ xuất hiện thêm trong thư viện các cấu hình tương ứng của thực thể. Các cấu hình khác nhau xác định các kiến trúc khác nhau của thực thể và có thể được mô phỏng độc lập. Nói một cách khác cấu hình là một đối tượng có cấp độ cụ thể cao hơn so với kiến trúc.

Cách thức thứ hai để quy định việc sử dụng kiến trúc là dùng trực tiếp cặp lệnh `for... use ... end for;` như minh họa dưới đây, cách thức này cho phép khai báo cấu hình trực tiếp bên trong một kiến trúc cụ thể:

```
-----
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);
-- declaration of component full_adder
component full_adder
    port(
        A      : in  std_logic;
        B      : in  std_logic;
        Cin   : in  std_logic;
        S     : out std_logic;
        Cout  : out std_logic
    );
end component;

for u0: full_adder
use entity work.full_adder(behavioral);
for u1: full_adder
use entity work.full_adder(dataflow);
for u2: full_adder
use entity work.full_adder(structure);
for u3: full_adder
use entity work.full_adder(behavioral);

begin
    -- design of 4-bit adder
u0: full_adder
    port map (A => A(0), B => B(0),
              Cin => CI,    S =>Sum(0), Cout => C(0));
u1: full_adder
    port map (A => A(1), B => B(1),
              Cin => C(0), S =>Sum(1), Cout => C(1));
u2: full_adder
    port map (A => A(2), B => B(2),

```

```

        Cin => C(1), S => Sum(2), Cout => C(2));
u3: full_adder
    port map (A => A(3), B => B(3),
               Cin => C(2), S => Sum(3), Cout => CO);
end structure;
-----
```

Ở ví dụ trên một bộ cộng 4 bit được xây dựng từ 4 khối full\_adder nhưng với các kiến trúc khác nhau. Khối đầu tiên dùng kiến trúc hành vi (behavioral), khối thứ hai là kiến trúc kiểu luồng dữ liệu (dataflow), khối thứ 3 là kiến trúc kiểu cấu trúc (structure), và khối cuối cùng là kiến trúc kiểu hành vi.

### 3. Chương trình con và gói

#### 3.1. Thủ tục

Chương trình con (*subprogram*) là các đoạn mã dùng để mô tả một thuật toán, phép toán dùng để xử lý, biến đổi, hay tính toán dữ liệu. Có hai dạng chương trình con là thủ tục (*procedure*) và hàm (*function*).

Thủ tục thường dùng để thực hiện một tác vụ như biến đổi, xử lý hay kiểm tra dữ liệu, hoặc các tác vụ hệ thống như đọc ghi file, truy xuất kết quả ra màn hình, kết thúc mô phỏng, theo dõi giá trị tín hiệu.... Khai báo của thủ tục như sau:

```

procedure identifier [(formal parameter list)] is
    [declarations]
begin
    sequential statement(s)
end procedure identifier;
ví dụ:
procedure print_header ;
procedure build ( A : in      constant integer;
                  B : inout signal bit_vector;
                  C : out     variable real;
                  D : file);
```

Trong đó formal parameter list chứa danh sách các biến, tín hiệu, hằng số, hay dữ liệu kiểu FILE, kiểu ngầm định là biến. Các đối tượng trong danh sách này trừ dạng file có thể được khai báo là dạng vào (in), ra (out), hay hai chiều (inout), kiểu ngầm định là in. Xét ví dụ đầy đủ dưới đây:

```

-----  

library IEEE;  

use IEEE.STD_LOGIC_1164.ALL;  

use STD.TEXTIO.all;  

-----
```

```

entity compare is
    port(
        res1, res2 : in bit_vector(3 downto 0)
    );
end compare;
-----
architecture behavioral of compare is
procedure print_to_file(
    vall, val2 : in bit_vector(3 downto 0);
    FILE fout : text)
is
use STD.TEXTIO.all;
variable str: line;
begin
    WRITE (str, string'("vall = "));
    WRITE (str, vall);
    WRITE (str, string'(" val2 = "));
    WRITE (str, val2);
    if vall = val2 then
        WRITE (str, string'(" OK"));
    elsif
        WRITE (str, string'(" TEST FAILED"));
    end if;
    WRITELINE(fout, str);
    WRITELINE(output, str);
end procedure print_to_file;

FILE file_output : text open WRITE_MODE is
"test_log.txt";
-- start here
begin
    proc_compare: print_to_file(res1, res2,
file_output);
end behavioral;
-----
```

Trong ví dụ trên chương trình con dùng để so sánh và ghi kết quả so sánh của hai giá trị kết quả res1, res2 vào trong file văn bản có tên "test\_log.txt". Phần khai báo của hàm được đặt trong phần khai báo của kiến trúc nhưng nếu hàm được gọi trực tiếp trong kiến trúc như ở trên thì khai báo này có thể bỏ đi. Thân chương trình con được viết trực tiếp trong phần khai báo của kiến trúc và được gọi trực tiếp cặp **begin** **end behavioral**.

### 3.2. Hàm

Hàm (*function*) thường dùng để tính toán kết quả cho một tổ hợp đầu vào. Khai báo của hàm có cú pháp như sau:

```
function identifier [parameter list] return a_type;  
ví dụ
```

```
function random return float;  
function is even ( A : integer) return boolean;
```

Danh sách biến của hàm cũng được cách nhau bởi dấu “;” nhưng điểm khác là trong danh sách này không có chỉ rõ dạng vào/ra của biến mà ngầm định tất cả là đầu vào. Kiểu dữ liệu đầu ra của hàm được quy định sau từ khóa **return**. Cách thức sử dụng hàm cũng tương tự như trong các ngôn ngữ lập trình bậc cao khác. Xét một ví dụ đầy đủ dưới đây:

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
-----  
entity function_example is  
end function_example;  
-----  
architecture behavioral of function_example is  
type bv4 is array (3 downto 0) of std_logic;  
function mask(mask, val1 : in bv4) return bv4;  
  
signal vector1 : bv4 := "0011";  
signal mask1 : bv4 := "0111";  
signal vector2 : bv4;  
  
function mask(mask, val1 : in bv4) return bv4 is  
    variable temp : bv4;  
begin  
    temp(0) := mask(0) and val1(0);  
    temp(1) := mask(1) and val1(1);  
    temp(2) := mask(2) or val1(2);  
    temp(3) := mask(3) or val1(3);  
    return temp;  
end function mask;  
-- start here  
begin  
    masking: vector2 <= mask(vector1, mask1);  
end behavioral;
```

Ví dụ trên minh họa cho việc sử dụng hàm để thực hiện phép tính mặt nạ (mask) đặc biệt trong đó hai bit thấp của giá trị đầu vào được thực hiện phép logic OR với giá trị mask còn hai bit cao thì thực hiện mask bình thường với

phép logic AND. Phần khai báo của hàm được đặt trong phần khai báo của kiến trúc, nếu hàm được gọi trực tiếp trong kiến trúc như ở trên thì khai báo này có thể bỏ đi. Phần thân của hàm được viết trong phần khai báo của kiến trúc trước cặp {begin end behavioral}. Khi gọi hàm trong phần thân của kiến trúc thì giá trị trả về của hàm phải được gán cho một tín hiệu, ở ví dụ trên là vector2.

### 3.3. Gói

Gói (*package*) là tập hợp các kiểu dữ liệu, hằng số, biến, các chương trình con và hàm dùng chung trong thiết kế. Một cách đơn giản gói là một cấp thấp hơn của thư viện, một thư viện cấu thành từ nhiều gói khác nhau. Ngoài các gói chuẩn của các thư viện chuẩn như trình bày ở 2.1, ngôn ngữ VHDL cho phép người dùng tạo ra các gói riêng tùy theo mục đích sử dụng. Một gói bao gồm khai báo gói và phần thân của gói. Khai báo gói có cấu trúc như sau:

```
package identifier is
    [ declarations ]
end package identifier ;
```

Phần khai báo chỉ chứa các khai báo về kiểu dữ liệu, biến dùng chung, hằng và khai báo của hàm hay thủ tục nếu có.

Phần thân gói có cú pháp như sau:

```
package body identifier is
    [ declarations ]
end package body identifier ;
```

Phần thân gói chứa các mô tả chi tiết của hàm hay thủ tục. Ngoài ra gói còn được dùng để chứa các khai báo component dùng chung cho các thiết kế lớn. Ví dụ các phần tử như khối cộng, thanh ghi, khối dịch, thanh ghi dịch, bộ đếm, khối chọn kênh... là các khối cơ bản cấu thành của hầu hết các khối thiết kế phức tạp, vì vậy sẽ rất tiện dụng khi “đóng gói” tất cả các thiết kế này như một thư viện dùng chung giống như các gói chuẩn và sử dụng trong các thiết kế lớn khác nhau với điều kiện các khối này có khai báo sử dụng gói trên.

Ví dụ đầy đủ một gói có chứa hai chương trình con liệt kê ở 3.1 và 3.2 và chứa các khai báo thiết kế dùng chung như sau:

```
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use STD.TEXTIO.all;
-----
package package_example is
    type bv4 is array (3 downto 0) of std_logic;
    function mask(mask, val1 : in bv4) return bv4;
```

```

procedure print_to_file(val1, val2 : in
bit_vector(3 downto 0); FILE fout :text);
end package package_example;
-----
package body package_example is
function mask(mask, val1 : in bv4) return bv4;

signal vector1 : bv4 := "0011";
signal mask1    : bv4 := "0111";
signal vector2 : bv4;

component adder_sub is
generic (N: natural := 32);
port(
    SUB  : in std_logic;
    Cin   : in std_logic;
    A     : in std_logic_vector(N-1 downto 0);
    B     : in std_logic_vector(N-1 downto 0);
    SUM   : out std_logic_vector(N-1 downto 0) ;
    Cout  : out std_logic
);
end component;
component counter is
generic (N: natural := 3);
PORT(
    clk          : in std_logic;
    reset        : in std_logic;
    counter_enable : in std_logic;
    end_num      : in std_logic_vector (N-1 downto 0);
    cnt_end      : out std_logic
);
end component;
component reg is
generic (N: natural := 32);
port(
    D      : in std_logic_vector(N-1 downto 0);
    Q      : out std_logic_vector(N-1 downto 0);
    CLK    : in std_logic;
    RESET : in std_logic
);
end component;
component mux is
generic (N: natural := 32);
port(
    Sel     : in std_logic;
    Din1   : in std_logic_vector(N-1 downto 0);

```

```

        Din2      : in  std_logic_vector(N-1 downto 0);
        Dout      : out std_logic_vector(N-1 downto 0)
    );
end component;
function mask(mask, val1 : in bv4) return bv4 is
    variable temp : bv4;
begin
    temp(0) := mask(0) and val1(0);
    temp(1) := mask(1) and val1(1);
    temp(2) := mask(2) or  val1(2);
    temp(3) := mask(3) or  val1(3);
    return temp;
end function mask;
-----
procedure print_to_file(
    val1, val2 : in bit_vector(3 downto 0);
    FILE fout  : text)
is
use STD.TEXTIO.all;
variable str: line;
begin
    WRITE (str, string'("val1 = "));
    WRITE (str, val1);
    WRITE (str, string'(" val2 = "));
    WRITE (str, val2);
    if val1 = val2 then
        WRITE (str, string'(" OK"));
    elsif
        WRITE (str, string'(" TEST FAILED"));
    end if;
    WRITELINE(fout, str);
    WRITELINE(output, str);
end procedure print_to_file;

end package body package_example;
-----
```

Để sử dụng gói này trong các thiết kế thì phải khai báo thư viện và gói sử dụng tương tự như trong các gói chuẩn ở phần khai báo thư viện. Vì theo ngầm định các gói này được biên dịch vào thư viện có tên là work nên khai báo như sau:

```

-----  

library IEEE;  

use IEEE.STD_LOGIC_1164.ALL;  

use STD.TEXTIO.all;  

library work;
```

```

use work.package_example.all;
-----
entity pck_example is
port(
    res1, res2 : in bit_vector(3 downto 0)
);
end pck_example;
-----
architecture behavioral of pck_example is
signal vector2 : bv4;
signal vector1 : bv4 := "0011";
signal mask1 : bv4 := "0111";
FILE file_output : text open WRITE_MODE is
"test_log.txt";
begin
    proc_compare:
        print_to_file(res1, res2, file_output);
masking :
    vector2 <= mask(vector1, mask1);

end behavioral;
-----

```

## 4. Đối tượng dữ liệu, kiểu dữ liệu

### 4.1. Đối tượng dữ liệu

Trong VHDL có phân biệt 3 loại đối tượng dữ liệu là biến, hằng và tín hiệu. Các đối tượng được khai báo theo cú pháp

Object\_type identifier : type [: initial value];

Trong đó object\_type có thể là Variable, Constant hay Signal.

#### 4.1.1. Hằng

Hằng (*Constant*) là những đối tượng dữ liệu dùng khởi tạo để chứa các giá trị xác định trong quá trình thực hiện. Hằng có thể được khai báo trong các gói, thực thể, kiến trúc, chương trình con, các khối và quá trình.

Cú pháp

constant identifier : type [range value] := value;

Ví dụ:

constant PI : REAL := 3.141569;

constant vector\_1 : std\_logic\_vector(8 downto 0) :=  
"11111111";

#### 4.1.2. Biến

Biến (*Variable*) là những đối tượng dữ liệu dùng để chứa các kết quả trung gian, biến chỉ có thể được khai báo bên trong các quá trình hoặc chương trình con. Khai báo biến bao giờ cũng đi kèm với kiểu, có thể có xác định giới hạn giá trị và có giá trị khởi tạo ban đầu, nếu không có giá trị khởi tạo ban đầu thì biến sẽ nhận giá trị khởi tạo là giá trị nhỏ nhất trong miền giá trị.

Cú pháp

```
variable identifier : type [range value] [:= initial  
value];
```

Ví dụ

```
variable count      : integer range 0 to 15 := 0;  
variable vector_bit : std_logic_vector(8 downto 0);
```

#### 4.1.3. Tín hiệu

Tín hiệu (*Signal*) là các đối tượng dữ liệu dùng để kết nối giữa các khối logic hoặc để đồng bộ các quá trình. Tín hiệu có thể được khai báo trong phần khai báo gói, khi đó ta sẽ có tín hiệu là toàn cục, khai báo trong thực thể khi đó tín hiệu là tín hiệu toàn cục của thực thể, trong khai báo kiến trúc, và khai báo trong các khối. Các tín hiệu tuyệt đối không được khai báo trong các quá trình và các chương trình con mà chỉ được sử dụng trong chúng, đặc điểm này thể hiện sự khác biệt rõ nhất giữa biến và tín hiệu.

Cú pháp

```
signal identifier : type [range value] [:= initial  
value];
```

Ví dụ:

```
signal a          : std_logic := '0';  
signal vector_b : std_logic_vector(31 downto 0);
```

### 4.2. Kiểu dữ liệu

#### 4.2.1. Các kiểu dữ liệu tiền định nghĩa

Trong các kiểu dữ liệu của VHDL có chia ra dữ liệu tiền định nghĩa và dữ liệu người dùng định nghĩa. 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.

Các dữ liệu tiền định nghĩa được mô tả trong các thư viện STD, và IEEE, cụ thể như sau:

- BIT, và BIT\_VECTOR, được mô tả trong thư viện STD.STANDARD, BIT chỉ nhận các giá trị ‘0’, và ‘1’. Ngầm định nếu như các biến dạng BIT không được khởi tạo giá trị ban đầu thì sẽ nhận giá trị 0. Vì BIT chỉ nhận các giá trị tường minh nên không phù hợp khi sử dụng để mô tả thực thể phần cứng thật, thay vì đó thường sử dụng các kiểu dữ liệu STD\_LOGIC và STD\_ULOGIC. Tuy vậy trong một số trường hợp ví dụ các lệnh của phép dịch, hay lệnh WRITE chỉ hỗ trợ cho BIT và BIT\_VECTOR mà không hỗ trợ STD\_LOGIC và STD\_LOGIC\_VECTOR.

- STD\_ULOGIC và STD\_ULOGIC\_VECTOR, STD\_LOGIC, STD\_LOGIC\_VECTOR được mô tả trong thư viện IEEE.STD\_LOGIC\_1164, STD\_ULOGIC, và STD\_LOGIC có thể nhận một trong 9 giá trị liệt kê ở bảng sau:

Bảng 2-1

#### Giá trị của kiểu dữ liệu STD\_LOGIC/STD\_ULOGIC

‘U’	Không xác định (Unresolved)	-
‘X’	X	Bắt buộc
‘0’	0	Bắt buộc
‘1’	1	Bắt buộc
‘Z’	Trở kháng cao	-
‘W’	X	Yêu
‘L’	0	Yêu
‘H’	1	Yêu
‘-’	Không quan tâm	-

Tên đầy đủ của STD\_ULOGIC là Standard Unresolved Logic, hay kiểu logic chuẩn chưa quy định về cách kết hợp các giá trị logic với nhau, do vậy đối với kiểu STD\_ULOGIC thì khi thiết kế không cho phép để một tín hiệu có nhiều nguồn cấp. Kiểu STD\_LOGIC là kiểu dữ liệu cũng có thể nhận một trong 9 giá trị logic như trên nhưng đã có quy định cách thức các giá trị kết hợp với nhau bằng hàm resolved. Nếu quan sát trong file stdlogic.vhd ta sẽ gặp đoạn mã mô tả hàm này, hai tín hiệu giá trị kiểu STD\_LOGIC khi kết hợp với nhau sẽ thu được 1 tín hiệu giá trị logic theo quy tắc trong bảng *resolved table* dưới đây.

```
SUBTYPE std_logic IS resolved std_ulogic;
TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF
std_ulogic;
-----
```

```

-----
      CONSTANT resolution_table : stdlogic_table := (
-----
-- | U   X   0   1   Z   W   L   H   -
-----
( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), | U |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), | X |
( 'U', 'X', '0', 'X', '0', '0', '0', 'X', 'X' ), | 0 |
( 'U', 'X', 'X', '1', '1', '1', '1', 'X', 'X' ), | 1 |
( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), | Z |
( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), | W |
( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), | L |
( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), | H |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) | - |
);
FUNCTION resolved ( s : std_ulogic_vector ) RETURN
std_ulogic IS
  VARIABLE result : std_ulogic := 'Z'; -- weakest state
  default
  BEGIN
    IF      (s'LENGTH = 1) THEN      RETURN s(s'LOW);
    ELSE
      FOR i IN s'RANGE LOOP
        result := resolution_table(result, s(i));
      END LOOP;
    END IF;
  RETURN result;
END resolved;

```

Ví dụ đoạn mã sau đây sẽ gây ra lỗi biên dịch:

```

-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
-----
entity logic_exapmle is
port(
  A : in  std_ulogic_vector(8 downto 0);
  U : out std_ulogic_vector(8 downto 0)
);
end logic_exapmle;
-----
architecture dataflow of logic_exapmle is
begin
  U <= A;
  U <= "X01ZWLH-1";

```

```

end dataflow;
Lỗi biên dịch sẽ báo rằng tín hiệu U có hai nguồn đầu vào.
# ** Error: logic_example.vhd(19) : Nonresolved signal
'u' has multiple sources.

```

Như đối với đoạn mã trên nếu khai báo U là tín hiệu dạng STD\_LOGIC thì vẫn biên dịch được bình thường.

Trên thực tế tùy vào đối tượng cụ thể mà dùng các kiểu tương ứng nhưng để đảm bảo tránh việc ghép nhiều đầu vào chung (việc này bắt buộc tránh khi thiết kế mạch thật), thì nên sử dụng STD\_ULOGIC.

Các dữ liệu tiền định nghĩa khác có trong ngôn ngữ VHDL liệt kê ở dưới đây:

- BOOLEAN: có các giá trị TRUE, FALSE (đúng/sai).
- INTEGER: số nguyên 32 bits (từ -2.147.483.647 đến +2.147.483.647)
- NATURAL: số nguyên không âm (từ 0 đến +2.147.483.647)
- REAL: số thực nằm trong khoảng (từ -1.0E38 đến +1.0E38).
- TIME: sử dụng đối với các đại lượng vật lý, như thời gian, điện áp,...Hữu ích trong mô phỏng
- CHARACTER: ký tự ASCII.
- FILE\_OPEN\_KIND: kiểu mở file gồm các giá trị MODE, WRITE\_MODE, APPEND\_MODE.
- FILE\_OPEN\_STATUS: Trạng thái mở file với các giá trị OPEN\_OK, STATUS\_ERROR, NAME\_ERROR, MODE\_ERROR.
- SEVERITY\_LEVEL: trạng thái lỗi với các giá trị NOTE, WARNING, ERROR, FAILURE.

#### 4.2.2. Các kiểu dữ liệu vô hướng

Dữ liệu vô hướng trong VHDL (*Scalar types*) bao gồm kiểu liệt kê (enumeration), kiểu nguyên (integer), kiểu số thực dấu phẩy động (real), kiểu dữ liệu vật lý (physical type). Các kiểu dữ liệu dưới đây được xét như các đối tượng dữ liệu người dùng định nghĩa từ các đối tượng dữ liệu tiền định nghĩa ở trên.

##### Kiểu liệt kê

Kiểu liệt kê (*Enumeration*) được định nghĩa bằng cách liệt kê tất cả các giá trị có thể có của kiểu, khai báo như sau

```
type enum_name is (enum_literals list);
```

Ví dụ:

```
type MULTI_LEVEL_LOGIC is (LOW, HIGH, RISING,
FALLING, AMBIGUOUS);
```

```
type BIT is ('0','1');
type SWITCH_LEVEL is ('0','1','X');
```

Các kiểu liệt kê định sẵn trong VHDL là:

- kiểu ký tự (CHARACTER).
- kiểu (BIT) gồm hai giá trị 0, 1.
- kiểu logic (BOOLEAN) gồm các giá trị TRUE, FALSE.
- kiểu SEVERITY kiểu cảnh báo lỗi gồm các giá trị NOTE, WARNING, ERROR, FAILURE.

-kiểu dạng và trạng thái File (FILE\_OPEN\_KIND với các giá trị READ\_MODE, WRITE\_MODE, APPEND\_MODE, kiểu FILE\_OPEN\_STATUS với các giá trị OPEN\_OK, STATUS\_ERROR, NAME\_ERROR, MODE\_ERROR).

### Kiểu số nguyên

Kiểu số nguyên (*integer*) được định nghĩa sẵn trong VHDL là INTEGER có giới hạn từ -2147483647 đến +2147483647. Các phép toán thực hiện trên kiểu nguyên là +, -, \*, /. Các kiểu nguyên thứ sinh được khai báo theo cú pháp sau:

```
type identifier is range interger_range;
```

Trong đó interger\_range là một miền con của tập số nguyên, các giá trị giới hạn của miền con phải viết dưới dạng số nguyên và có thể nhận giá trị âm hoặc dương, ví dụ:

```
type word_index is range 30 downto 0;
type TWOS_COMPLEMENT_INTEGER is range -32768 to 32767;
```

### Kiểu số thực

Kiểu số thực (*Real*) được định nghĩa sẵn trong VHDL là Real có giới hạn từ -1.0E38 tới +1.0E38. Các kiểu thực thứ sinh được khai báo theo cú pháp sau:

```
type identifier is range real_range;
```

Trong đó real\_range là miền con của miền số thực các giá trị giới hạn của miền này có thể là các giá trị dương hoặc âm được viết bằng một trong những dạng cho phép của số thực như dạng dấu phẩy động hay dạng thập phân và không nhất thiết phải giống nhau, ví dụ:

```
type my_float1      is range 1.0      to 1.0E6;
type my_float2 is range -1.0e5 to 1.0E6;
```

### Kiểu giá trị đại lượng vật lý

Kiểu giá trị đại lượng vật lý (*physical*) được dùng để định nghĩa các đơn vị vật lý như thời gian, khoảng cách, diện tích, ... Chỉ có một kiểu giá trị đại lượng vật lý được định nghĩa sẵn trong VHDL là kiểu TIME

```

type Time is range --implementation defined-- ;
units
    fs;                      -- femtosecond
    ps = 1000 fs; -- picosecond
    ns = 1000 ps; -- nanosecond
    us = 1000 ns; -- microsecond
    ms = 1000 us; -- millisecond
    sec = 1000 ms; -- second
    min = 60 sec; -- minute
    hr = 60 min; -- hour
end units;

```

Các kiểu giá trị đại lượng vật lý được khai báo với cú pháp tương tự như trên, sau từ khóa **units** là đơn vị chuẩn và các đơn vị thứ sinh bằng một số nguyên lần các đơn vị chuẩn, danh sách các đơn vị được kết thúc bởi từ khóa **end units**, ví dụ:

```

type distance is range 0 to 1E16
units
    Ang;                      -- angstrom
    nm = 10 Ang;              -- nanometer
    um = 1000 nm;             -- micrometer (micron)
    mm = 1000 um;             -- millimeter
    cm = 10 mm;               -- centimeter
    dm = 100 mm;              -- decameter
    m = 1000 mm;              -- meter
    km = 1000 m;              -- kilometer
    mil = 254000 Ang;         -- mil (1/1000 inch)
    inch = 1000 mil;           -- inch
    ft = 12 inch;              -- foot
    yd = 3 ft;                -- yard
    fthn = 6 ft;               -- fathom
    frlg = 660 ft;              -- furlong
    mi = 5280 ft;              -- mile
    lg = 3 mi;                -- league
end units;

```

#### 4.2.2. Dữ liệu phức hợp

Dữ liệu phức hợp (*composite*) là dữ liệu tạo thành bằng các tổ hợp các dạng dữ liệu cơ bản trên theo các cách khác nhau. Có hai dạng dữ liệu phức hợp cơ bản là kiểu mảng dữ liệu khi các phần tử dữ liệu trong tổ hợp là đồng nhất và được sắp thứ tự, dạng thứ hai là dạng bản ghi khi các phần tử có thể có các kiểu dữ liệu khác nhau.

##### Kiểu mảng

Kiểu mảng (*Array*) trong VHDL có các tính chất như sau:

- Các phần tử của mảng có thể là mọi kiểu trong ngôn ngữ VHDL.
- Số lượng các chỉ số của mảng (hay số chiều của mảng) có thể nhận mọi giá trị nguyên dương.
- Mảng chỉ có một và một chỉ số dùng để truy cập tới phần tử.
- Miền biến thiên của chỉ số xác định số phần tử của mảng và hướng sắp xếp chỉ số trong của mảng từ cao đến thấp hoặc ngược lại.
- Kiểu của chỉ số là kiểu số nguyên hoặc liệt kê.

Mảng trong VHDL chia làm hai loại là mảng ràng buộc và mảng không ràng buộc. Mảng ràng buộc là mảng được khai báo tường minh có kích thước cố định. Cú pháp khai báo của mảng này như sau:

```
type array_name is array (index_range) of type;
```

Trong đó array\_name là tên của mảng, index\_range là miền biến thiên xác định của chỉ số nếu mảng là mảng nhiều chiều thì các chiều biến thiên cách nhau dấu “,”, ví dụ như sau:

```
type mem is array (0 to 31, 3 to 0) of std_logic;
type word is array (0 to 31) of bit;
type data is array (7 downto 0) of word;
```

Đối với mảng khai báo không tương minh thì miền giá trị của chỉ số không được chỉ ra mà chỉ ra kiểu của chỉ số:

```
type array_name is array (type of index range <>) of type;
```

Ví dụ:

```
type mem is array (natural range <>) of word;
type matrix is array (integer range <>,
                     integer range <>) of real;
```

Cách truy cập tới các phần tử của mảng của một mảng n chiều như sau:

```
array_name (index1, index 2,..., indexn)
```

ví dụ:

```
matrix(1,2), mem (3).
```

### Kiểu bản ghi

Bản ghi (*Record*) là nhóm của một hoặc nhiều phần tử thuộc những kiểu khác nhau và được truy cập tới như một đối tượng. Bản ghi có những đặc điểm như sau:

- Mỗi phần tử của bản ghi được truy cập tới theo trường.
- Các phần tử của bản ghi có thể nhận mọi kiểu của ngôn ngữ VHDL kể cả mảng và bản ghi.

ví dụ về bản ghi

```
type stuff is
  record
    I      : integer;
```

```

X      : real;
Day   : integer range 1 to 31;
name  : string(1 to 48);
prob  : matrix(1 to 3, 1 to 3);
end record;

```

Các phần tử của bản ghi được truy cập theo tên của bản ghi và tên trường ngăn cách nhau bằng dấu “.”, ví dụ:

```
node.data; stuff.day
```

## 5. Toán tử và biểu thức

Trong VHDL có tất cả 7 nhóm toán tử được phân chia theo mức độ ưu tiên và trật tự tính toán. Trong bảng sau liệt kê các nhóm toán tử theo trật tự ưu tiên tăng dần:

Bảng 2-2

**Các toán tử trong VHDL**

Toán tử logic	and, or, nand, nor, xor
Các phép toán quan hệ	=, /=, <, <=, >, >=
Các phép toán dịch	sll, srl, sla, sra, rol, ror
Các phép toán cộng, hợp	+, -, &
Toán tử dấu	+, -
Các phép toán nhân	*, /, mod, rem
Các phép toán khác	**, abs, not

Các quy định về trật tự các phép toán trong biểu thức được thực hiện như sau:

- Trong các biểu thức phép toán có mức độ ưu tiên lớn hơn sẽ được thực hiện trước. Các dấu ngoặc đơn “(“, “)” phân định miền ưu tiên của từng nhóm biểu thức.
- Các phép toán trong nhóm với cùng một mức độ ưu tiên sẽ được thực hiện lần lượt từ trái qua phải.

### 5.1. Toán tử logic

Các phép toán logic gồm and, or, nand, nor, xor, và not. Các phép toán này tác động lên các đối tượng kiểu BIT và Boolean và mảng một chiều kiểu BIT. Đối với các phép toán hai ngôi thì các toán hạng nhất định phải cùng kiểu, nếu hạng tử là mảng BIT một chiều thì phải có cùng độ dài, khi đó các phép toán logic sẽ thực hiện đối với các bit tương ứng của hai toán hạng có cùng chỉ số.

Phép toán not chỉ có một toán hạng, khi thực hiện với mảng BIT một chiều thì sẽ cho kết quả là mảng BIT lật đảo ở tất cả các vị trí của mảng cũ.

Ví dụ:

```
-----
library ieee;
use ieee.std_logic_1164.all;
-----
entity logic_example is
    port(
        in1    : in  std_logic_vector (5 downto 0);
        in2    : in  std_logic_vector (5 downto 0);
        out1   : out std_logic_vector (5 downto 0)
    );
end entity;
-----
architecture rtl of logic_example is
begin
    out1(0) <= in1(0) and in2 (0);
    out1(1) <= in1(1) or  in2 (1);
    out1(2) <= in1(2) xor  in2 (2);
    out1(3) <= in1(3) nor  in2 (3);
    out1(4) <= in1(4) nand in2 (4);
    out1(5) <= in1(5) and  (not in2 (5));
end rtl;
-----
```

## 5.2. Các phép toán quan hệ

Các phép toán quan hệ gồm  $=$ ,  $/=$ ,  $<$ ,  $\leq$ ,  $>$ ,  $\geq$  thực hiện các phép toán so sánh giữa các toán tử có cùng kiểu như Integer, real, character. Và cho kết quả dạng Boolean. Việc so sánh các hạng tử trên cơ sở miền giá trị của các đối tượng dữ liệu. Các phép toán quan hệ rất hay được sử dụng trong các câu lệnh rẽ nhánh

Ví dụ đầy đủ về phép toán so sánh thể hiện ở đoạn mã sau:

```
-----
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
-----
entity compare_example is
    port(
        val1    : in  std_logic_vector (7 downto 0);
        val2    : in  std_logic_vector (7 downto 0);
        res     : out std_logic_vector (2 downto 0)
    );
end entity;
-----
```

```

-----
architecture rtl of compare_example is
begin
    compare: process (val1, val2)
    begin
        if val1 > val2 then res(0) <= '1';
        else res (0) <= '0'; end if;
        if val1 = val2 then res(1) <= '1';
        else res (1) <= '0'; end if;
        if val1 < val2 then res(2) <= '1';
        else res (2) <= '0'; end if;
    end process compare;
end rtl;

```

### 5.3. Các phép toán dịch

Các phép toán quan hệ gồm sll, srl, sla, sra, rol, ror được hỗ trợ trong thư viện ieee.numeric\_bit, và ieee.numeric\_std. Cú pháp của các lệnh dịch có hai tham số là sho (shift operand) và shv (shift value), ví dụ cú pháp của sll như sau

```
sha sll shv;
```

Bảng 2-3

**Các phép toán dịch trong VHDL**

Toán tử	Phép toán	Kiểu của sho	Kiểu của shv	Kiểu kết quả
sll	Dịch trái logic	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
srl	Dịch phải logic	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
sla	Dịch trái số học	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
sra	Dịch phải số học	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
rol	Dịch vòng tròn sang trái	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
ror	Dịch vòng tròn phải	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho

Đối với dịch logic thì tại các vị trí bị trống sẽ được điền vào các giá trị '0' còn dịch số học thì các các vị trí trống được thay thế bằng bit có trọng số cao nhất (MSB) nếu dịch phải, và thay bằng bit có trọng số thấp nhất (LSB) nếu dịch

trái. Đối với dịch vòng thì các vị trí khuyết đi sẽ được điền bằng các bit dịch ra ngoài giới hạn của mảng. Quan sát ví dụ dưới đây

Giả sử có giá trị sho = “11000110” , shv = 2, xét đoạn mã sau

```
-----  
library ieee;  
USE ieee.Numeric_STD.all;  
USE ieee.Numeric_BIT.all;  
library STD;  
use STD.TEXTIO.ALL;  
-----  
entity shift_example is  
end entity;  
-----  
architecture rtl of shift_example is  
signal sho: bit_vector(7 downto 0) := "11000110";  
begin  
shifting: process (sho)  
    variable str:line;  
begin  
    write(str, string'("sho sll 2 = "));  
    write(str, sho sll 2);  
    writeline(OUTPUT, str);  
    write(str, string'("sho srl 2 = "));  
    write(str, sho srl 2);  
    writeline(OUTPUT, str);  
    write(str, string'("sho sla 2 = "));  
    write(str, sho sla 2);  
    writeline(OUTPUT, str);  
    write(str, string'("sho sra 2 = "));  
    write(str, sho sra 2);  
    writeline(OUTPUT, str);  
    write(str, string'("sho rol 2 = "));  
    write(str, sho rol 2);  
    writeline(OUTPUT, str);  
    write(str, string'("sho ror 2 = "));  
    write(str, sho ror 2);  
    writeline(OUTPUT, str);  
end process shifting;  
end architecture;
```

Với đoạn mã trên khi mô phỏng sẽ thu được kết quả:

```
# sho sll 2 = 00011000  
# sho srl 2 = 00110001  
# sho sla 2 = 00011000  
# sho sra 2 = 11110001
```

```
# sho rol 2 = 00011011
# sho ror 2 = 10110001
```

#### 5.4. Các phép toán cộng trừ và hợp

Các phép toán  $+$ ,  $-$  là các phép tính hai ngôi, các phép toán cộng và trừ có cú pháp thông thường, hạng tử của phép toán này là tất cả các kiểu dữ liệu kiểu số gồm INTEGER, REAL.

Bảng 2-4

#### Các phép toán cộng dịch và hợp

Toán tử	Phép toán	Kiểu toán tử trái	Kiểu của toán tử phải	Kiểu kết quả
$+$	Phép cộng	Dữ liệu kiểu số	Cùng kiểu	Cùng kiểu
$-$	Phép trừ	Dữ liệu kiểu số	Cùng kiểu	Cùng kiểu
$\&$	Phép hợp	Kiểu mảng hoặc phần tử mảng	Kiểu mảng hoặc phần tử mảng	Kiểu mảng

Phép toán hợp (concatenation)  $\&$  có đối số có thể là mảng hoặc phần tử mảng và kết quả hợp tạo ra một mảng mới có các phần tử ghép bởi các phần tử của các toán tử hợp, các ví dụ dưới đây sẽ minh họa rõ thêm cho phép hợp

```
-----
library ieee;
use ieee.std_logic_1164.all;
-----
entity duplicate is
    port(
        in1 : in std_logic_vector(3 downto 0);
        out1 : out std_logic_vector(7 downto 0)
    );
end entity;
-----
architecture rtl of duplicate is
begin
    out1 <= in1 & in1;
end rtl;
```

Trong ví dụ trên nếu gán giá trị  $in1 = "1100"$  thu được tương ứng ở đầu ra  $out1 = "11001100"$ .

#### 5.5. Các phép dấu

Các phép toán quan hệ gồm  $+, -, \text{and}, \text{or}$ , thực hiện với các giá trị dạng số và trả về giá trị dạng số tương ứng.

## 5.6. Các phép toán nhân chia, lấy dư

Các phép toán \*, /, mod, rem là các phép toán hai ngôi tác động lên các toán tử kiểu số theo như bảng sau:

Bảng 2-5

### Các phép toán nhân chia và lấy dư số nguyên

Toán tử	Phép toán	Toán tử trái	Toán tử phải	Kiểu kết quả
*	Phép nhân	Số nguyên Integer	Cùng kiểu	Cùng kiểu
		Số thực dấu phẩy động REAL	Cùng kiểu	Cùng kiểu
/	Phép chia	Số nguyên Integer	Cùng kiểu	Cùng kiểu
		Số thực dấu phẩy động REAL	Cùng kiểu	Cùng kiểu
Mod	Lấy module	Số nguyên Integer	Số nguyên Integer	Số nguyên Integer
Rem	Lấy phần dư (remainder)	Số nguyên Integer	Số nguyên Integer	Số nguyên Integer

Có thể kiểm tra các phép toán số học bằng đoạn mã sau:

```

-----
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
-----
entity mul_div_example is
    port(
        m1, m2          : in integer;
        res1, res2, res3, res4 : out integer
    );
end entity;
-----
architecture rtl of mul_div_example is
begin
    res1 <= m1 * m2;
    res2 <= m1 / m2;
    res3 <= m1 mod m2;
    res4 <= m1 rem m2;
end rtl;
-----
```

## 5.7. Các phép toán khác

Các phép toán **\*\***, **abs** là các phép toán lấy mũ e và lấy giá trị tuyệt đối một ngôi tác động lên các toán tử kiểu số theo như bảng sau:

Bảng 2-6

Các phép toán khác

Toán tử	Phép toán	Toán tử	Kiểu kết quả
**	Lấy mũ e	Số nguyên Integer	Cùng kiểu
	$** a = e^a$	Số thực dấu phẩy động REAL	Cùng kiểu
abs	Lấy trị tuyệt đối	Số nguyên Integer	Cùng kiểu
		Số thực dấu phẩy động REAL	Cùng kiểu

## 6. 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).

### 6.1. Phát biểu đợi

Phát biểu đợi (**wait**) có cấu trúc như sau

```
wait [sensitivity clause] [condition clause] [time clause];
```

Trong đó :

- sensitivity clause = **on** sensitivity list, danh sách các tín hiệu cần theo dõi, nếu câu lệnh wait dùng cấu trúc này thì nó có ý nghĩa bắt quá trình đợi cho tới khi có bất kỳ sự thay đổi nào của các tín hiệu trong danh sách theo dõi. Cấu trúc này tương đương với cấu trúc **process** dùng cho các phát biểu đồng thời sẽ tìm hiểu ở 7.2.

- Condition clause = until condition: trong đó condition là điều kiện dạng Boolean. Cấu trúc này bắt quá trình dừng cho tới khi điều kiện trong condition được thỏa mãn.
- Time clause = **for** time\_expression; có ý nghĩa bắt quá trình dừng trong một khoảng thời gian xác định chỉ ra trong tham số lệnh.

Ví dụ về các kiểu gọi lệnh wait:

```
wait for 10 ns;      -- timeout clause, specific time
delay
  wait until clk='1';  -- Boolean condition
  wait until S1 or S2; -- Boolean condition
  wait on sig1, sig2;-- sensitivity clause, any event
on any
```

Câu lệnh wait nếu sử dụng trong process thì process không được có danh sách tín hiệu theo dõi (sensitive list), lệnh wait on tương đương với cách sử dụng process có danh sách tín hiệu theo dõi. Xem xét ví dụ đầy đủ dưới đây.

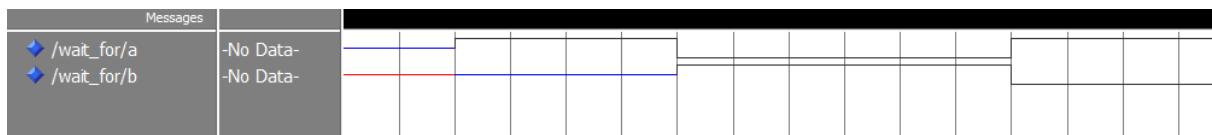
```
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity AND3 is
port (
  A : in std_logic;
  B : in std_logic;
  C : in std_logic;
  S : out std_logic
);
end AND3;
-----
architecture wait_on of AND3 is
begin
  andd: process
  begin
    S <= A and B and C;
    wait on A, B, C;
  end process andd;
end wait_on;
-----
architecture use_process of AND3 is
begin
  andd: process (A, B, C)
  begin
    S <= A and B and C;
  end process andd;
```

```
end use_process;
```

Khối mô tả ở trên thực hiện một hàm AND 3 đầu vào, với kiến trúc wait\_on sử dụng cấu trúc câu lệnh **wait on** kèm theo danh sách tín hiệu theo dõi A, B, C. Cách sử dụng đó tương đương với cách viết trong kiến trúc use\_process khi không dùng wait\_on mà sử dụng danh sách tín hiệu theo dõi ngay sau từ khóa process.

Với lệnh wait for xem xét ví dụ đầy đủ dưới đây:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity wait_for is
    port(
        A : out std_logic;
        B : out std_logic
    );
end wait_for;
-----
architecture behavioral of wait_for is
begin
    waiting: process
        begin
            A <= 'Z'; B <= 'X';
            wait for 100 ns;
            A <= '1'; B <= 'Z';
            wait for 200 ns;
            A <= '0'; B <= '1';
            wait for 300 ns;
            A <= '1'; B <= '0';
            wait;
        end process waiting;
    end behavioral;
```



Hình 2-3. Kết quả mô phỏng gán tín hiệu và lệnh Wait

Trong ví dụ này các tín hiệu A, B, được gán các giá trị thay đổi theo thời gian, khoảng thời gian được quy định bằng các lệnh wait for. Ở ví dụ trên ban

đầu A, B nhận các giá trị là Z và X sau 100 ns A bằng 1 còn B bằng Z, sau tiếp 200 ns A nhận giá trị bằng 0 và B bằng 1.

Ở đây ta cũng gặp cấu trúc lệnh wait không có tham số, lệnh này tương đương lệnh đợi trong khoảng thời gian là vô hạn, các tín hiệu A, B khi đó được giữ nguyên giá trị được gán ở câu lệnh trước. Cấu trúc này là các cấu trúc chỉ dùng cho mô phỏng, đặc biệt hay dùng trong các khối kiểm tra, phục vụ cho việc tạo các xung đầu vào của các khối kiểm tra.

Ví dụ sau đây là ví dụ về lệnh wait until

```
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity wait_until is
port (D : in std_logic;
      Q : out std_logic
     );
end wait_until;
-----
architecture behavioral of wait_until is
signal clk : std_logic := '0';
begin
  create_clk: process
begin
  wait for 100 ns;
  clk <= not clk after 100 ns;
  end process create_clk;
  latch: process
begin
  wait until clk = '1';
  Q <= D;
  end process latch;
end behavioral;
```

Ví dụ này mô tả phần tử latch, giá trị đầu ra nhận giá trị đầu vào tại mỗi thời điểm mà giá trị tín hiệu đồng bộ bằng 1 trong đó xung nhịp clk được tạo bằng cấu trúc `clk <= not clk after 100 ns;` nghĩa là chu kỳ tương ứng bằng 200 ns.

## 6.2. Phát biểu xác nhận và báo cáo

Phát biểu xác nhận và báo cáo (*assert and report statement*) có cấu trúc như sau

```
assert boolean_condition [report string] [severity name];
```

Trong đó :

boolean\_condition: điều kiện dạng Boolean. Cấu trúc này kiểm tra giá trị của boolean\_condition.

report string: báo cáo lỗi nếu được sử dụng thì phải đi cùng một chuỗi thông báo lỗi

severity name: thông báo mức độ lỗi nếu được sử dụng thì phải đi kèm với các giá trị định sẵn bao gồm NOTE, WARNING, ERROR, FAILURE

Lệnh report có thể được sử dụng độc lập với assert khi đó nó có cấu trúc report string [severity name];

Các ví dụ về lệnh **assert** và **report**:

```
assert a = (b or c);
assert j < i report "internal error, tell someone";
assert clk='1' report "clock not up" severity WARNING;
report "finished pass1"; -- default severity name is
NOTE
report "inconsistent data." severity FAILURE;
```

Kết hợp giữa lệnh assert và cảnh báo severity ở cấp độ FAILURE có thể dùng để ngắt quá trình mô phỏng, khi trong quá trình mô phỏng mà lệnh này được thực thi thì chương trình sẽ dừng lại. Cấu trúc này vì thế được sử dụng trong quá trình kiểm tra tự động các thiết kế. (Xem thêm mục 9.1.1)

### 6.3. Phát biểu gán biến

Trong ngôn ngữ VHDL cú pháp của phát biểu gán biến (*variable assignment statement*) tương tự như phép gán giá trị của biến như trong các ngôn ngữ lập trình khác, cú pháp như sau:

```
variable := expression;
```

Trong đó variable là các đối tượng chỉ được phép khai báo trong các chương trình con và các quá trình, các biến chỉ có tác dụng trong chương trình con hay quá trình khai báo nó, expression có thể là một biểu thức hoặc một hằng số có kiểu giống kiểu của variable. Quá trình gán biến diễn ra tức thì với thời gian mô phỏng bằng 0 vì biến chỉ có tác dụng nhận các giá trị trung gian. Ví dụ về gán biến.

```
A := -B + C * D / E mod F rem G abs H;
```

```
Sig := Sa and Sb or Sc nand Sd nor Se xor Sf xnor Sg;
```

Ví dụ sau đây minh họa cho phát biểu gán biến và phát biểu **assert**. Trong ví dụ này một bộ đếm được khai báo dưới dạng biến, tại các thời điểm

sùn lên của xung nhịp đồng hồ giá trị counter được tăng thêm 1 đơn vị. Lệnh **assert** sẽ kiểm soát và thông báo khi nào giá trị counter vượt quá 100 và thông báo ra màn hình.

```

-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity assert_example is
end assert_example;
-----
architecture behavioral of assert_example is
signal clk: std_logic := '0';
begin
    create_clk: process
    begin
        wait for 100 ns;
        clk <= not clk after 100 ns;
    end process create_clk;
    count: process (clk)
    variable counter : integer := 0;
    begin
        if clk'event and clk = '1' then
            counter := counter + 1;
        end if;
        assert counter <100 report "end of counter"
severity NOTE;
    end process count;
end behavioral;
-----
Khi mô phỏng ta thu được thông báo sau ở màn hình khi counter vượt quá
giá trị 100.
# ** Note: end of counter
#      Time: 20 us  Iteration: 0  Instance:
/assert_example

```

#### 6.4. Phát biểu gán tín hiệu

Phát biểu gán tín hiệu (*signal assignment statement*) có cấu trúc như sau

target <= [ delay\_mechanism ] waveform;

Trong đó :

- target: đối tượng cần gán tín hiệu.

- Delay mechanism: cơ chế làm trễ tín hiệu có cấu trúc như sau:

transport [ reject\_time\_expression ] inertial

Trong đó nếu có từ khóa **transport** thì tín hiệu được làm trễ theo kiểu đường truyền là dạng làm trễ mà không phụ thuộc vào dạng tín hiệu đầu vào, xung đầu vào dù có độ rộng xung như thế nào vẫn được truyền tải đầy đủ. Nếu không có cơ chế làm trễ nào được chỉ ra hoặc sử dụng cấu trúc **inertial** thì cơ chế làm trễ là cơ chế quán tính (inertial delay), theo cơ chế này thì sẽ có một giới hạn độ rộng xung đầu vào được chỉ ra gọi là pulse reject limit được chỉ ra cùng từ khóa **reject**, nếu tín hiệu vào có độ rộng xung bé hơn giới hạn này thì không đủ cho phần tử logic kế tiếp thực hiện thao tác chuyển mạch và gây ra lỗi. Nếu giới hạn thời gian chỉ ra sau **reject** là giá trị âm cũng gây ra lỗi.

```
waveform := wave_form elements := signal + {after
time_expression};
```

- Chỉ ra tín hiệu gán giá trị và thời gian áp dụng cho cơ chế làm trễ đã trình bày ở phần trên.

Ví dụ về gán tín hiệu tuần tự:

```
-- gán với trễ quán tính, các lệnh sau tương đương
O_pin <= i_pin after 10 ns;
O_pin <= inertial i_pin after 10 ns;
O_pin <= reject 10 ns inertial I_pin after 10 ns;
-- gán với kiểm soát độ rộng xung đầu vào
O_pin <= reject 5 ns inertial I_pin after 10 ns;
O_pin <= reject 5 ns inertial I_pin after 10 ns, not
I_pin after 20 ns;
-- Gán với trễ transport
O_pin <= transport I_pin after 10 ns;
O_pin <= transport I_pin after 10 ns, not I_pin after
20 ns;
-- tương đương các lệnh sau
O_pin <= reject 0 ns inertial I_pin after 10 ns;
O_pin <= reject 0 ns inertial I_pin after 10 ns, not
I_pin after 10 ns;
```

Xét một ví dụ đầy đủ về lệnh này như sau như sau:

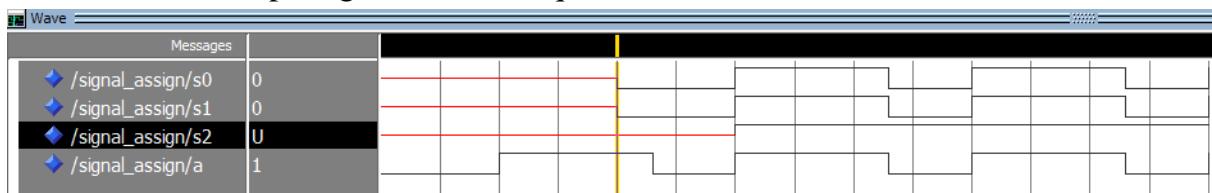
```
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity signal_assign is
port (
    S0 : out std_logic;
    S1 : out std_logic;
    S2 : out std_logic
);
end signal_assign;
```

```

-----
architecture behavioral of signal_assign is
signal A: std_logic:= '0';
begin
    create_data: process
    begin
        wait for 100 ns;
        A <= '1';
        wait for 100 ns;
        A <= not A after 30 ns;
    end process create_data;
    andd: process (A)
    begin
        S0 <= transport A after 200 ns;
        S1 <= reject 30 ns inertial A after 200 ns;
        S2 <= reject 100 ns inertial A after 200 ns;
    end process andd;
end behavioral;

```

Sau khi mô phỏng thu được kết quả trên waveform như sau:



Hình 2.4. Kết quả mô phỏng ví dụ gán tín hiệu

Tín hiệu A là dạng tín hiệu xung nhịp có chu kỳ 60 ns, A thay đổi giá trị sau mỗi nửa chu kỳ tức là sau mỗi 30 ns. Tín hiệu S0 được gán bằng A theo phương thức **transport** vì vậy trùng lặp hoàn toàn với A. Tín hiệu S1 gán theo phương thức **inertial** với thời gian reject bằng 30 ns do vậy vẫn thu được giá trị giống A. Tín hiệu S2 cũng được gán bằng phương thức **inertial** như với **reject** time bằng 100 ns > 30 ns nên có giá trị không thay đổi.

## 6.5. Lệnh rẽ nhánh và lệnh lặp

Lệnh rẽ nhánh là lệnh lặp là các phát biểu điều khiển quá trình trong các chương trình con hoặc trong các quá trình. Các phát biểu này có cú pháp giống như các phát biểu tương tự trong các ngôn ngữ lập trình khác.

### 6.5.1. Lệnh rẽ nhánh if

```

if condition1 then
    sequence-of-statements
elsif condition2 then
    [sequence-of-statements ]

```

```

elsif condition3 then
    [sequence-of-statements]
    ..
else
    [sequence-of-statements]
end if;

```

Trong đó :

- condition : các điều kiện dạng boolean.
- [sequence-of-statements] : khối lệnh thực thi nếu điều kiện được thỏa mãn.

Ví dụ sau minh họa cách sử dụng lệnh `if` để mô tả D-flipflop bằng VHDL, điều kiện sùn dương của tín hiệu xung nhịp được kiểm tra bằng cấu trúc `clk = '1' and clk'event` trong đó `clk'event` thể hiện có sự kiện thay đổi giá trị trên `clk` và điều kiện `clk = '1'` xác định sự thay đổi đó là sùn dương của tín hiệu.

```

-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity D_flipflop is
port(
    D      : in  std_logic;
    CLK   : in  std_logic;
    Q      : out std_logic
);
end D_flipflop;
-----
architecture behavioral of D_flipflop is
begin
DFF: process (clk, D)
begin
    if clk = '1' and clk'event then
        Q <= D;
    end if;
end process DFF;
end behavioral;
-----
```

### 6.5.2. Lệnh chọn case

Lệnh lựa chọn (*case*) có cú pháp như sau:

```

case expression is
    when choice1 =>
        [sequence-of-statements]
```

```

when choice2 =>
    [sequence-of-statements]
    ...

when others => -- optional if all choices covered
    [sequence-of-statements]
end case;

```

Lệnh case bắt đầu bằng từ khóa **case** theo sau là một biểu thức (expression). Các trường hợp được chọn bắt đầu bằng từ khóa **when** giá trị có thể của expression và mã thực thi tương ứng bắt đầu sau dấu “=>”. When others sẽ quét hết tất cả các giá trị có thể có của expression mà chưa được liệt kê ra ở trên (tương đương với từ khóa **default** trong ngôn ngữ lập trình C).

Ví dụ về lệnh **case**:

```

-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity mux2 is
port(
    A      : in  std_logic;
    B      : in  std_logic;
    Sel   : in  std_logic;
    S      : out std_logic
);
end mux2;
-----
architecture behavioral of mux2 is
begin
mux: process (A, B, sel)
begin
CASE sel IS
    WHEN '0' => S <= A;
    WHEN others => S <= B;
end case;
end process mux;
end behavioral;
-----
```

### 6.5.3. Lệnh lặp

Có ba dạng lệnh lặp dùng trong VHDL là lệnh loop, lệnh while, và lệnh for:

```

loop
    sequence-of-statements -- use to get out

```

```

end loop;
for variable in range loop
    sequence-of-statements
end loop;
while condition loop
    sequence-of-statements
end loop;

```

Đối với lệnh lặp dạng **loop** thì vòng lặp chỉ kết thúc nếu nó gặp lệnh **exit** ở trong đó. Đối với lệnh lặp dạng **for** thì vòng lặp kết thúc khi đã quét hết tất cả các giá trị của biến chạy. Với vòng lặp dạng **while** thì vòng lặp kết thúc khi điều kiện trong **condition** là FALSE.

Ví dụ đầy đủ về ba dạng lệnh lặp:

```

-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity loop_example is
port(
    vector_in : in std_logic_vector(7 downto 0);
    out1      : out std_logic
);
end loop_example;
-----
architecture loop1 of loop_example is
begin
    loop_p: process (vector_in)
        variable I : integer;
        variable p : std_logic := '1';
    begin
        for i in 7 downto 0 loop
            p := p and vector_in(i);
        end loop;
        out1 <= p;
    end process loop_p;
end loop1;
-----
architecture loop2 of loop_example is
begin
    loop_p: process (vector_in)
        variable i: integer    := 7;
        variable p: std_logic := '1';
    begin
        while i > 0 loop
            p := p and vector_in(i);
            i := i-1;
        end loop;
    end process loop_p;
end loop2;

```

```

        end loop;
        i    := 7;
        out1 <= p;
    end process loop_p;
end loop2;
-----
architecture loop3 of loop_example is
begin
    loop_p: process (vector_in)
    variable i: integer := 7;
    variable p: std_logic := '1';
    begin
        loop
            p := p and vector_in(i);
            i := i-1;
            exit when i < 0;
        end loop;
        i    := 7;
        out1 <= p;
    end process loop_p;
end loop3;
-----
```

Ví dụ trên thể hiện ba phương pháp khác nhau dùng lệnh để hiện thực hóa hàm AND cho tổ hợp 8 bit đầu vào. Phương pháp thứ nhất sử dụng lệnh lặp xác định, phương pháp thứ hai và thứ ba sử dụng vòng lặp kiểm soát điều kiện đối với biến chạy i, về cơ bản cả ba phương pháp đều cho hiệu quả mô tả giống nhau.

## 7. 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 phần 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.

Vị trí của các phát biểu đồng thời nằm trực tiếp trong khối **begin end** của mô tả kiến trúc:

```

architecture identifier of design is
    {declarative_part}
begin
    {concurrent_statements}
```

```
end identifier;
```

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*).

## 7.1. Phát biểu khối

Phát biểu khối (*Block statement*) là một khối các câu lệnh song song thể hiện một phần của thiết kế, các khối có thể được khai báo lồng ghép trong nhau và có tính chất kế thừa. Phát biểu khai báo block có cấu trúc như sau:

```
block [ (guard_expression) ] [is]
    block_header
    block_declarative_part
begin
    block_statement_part
end block;
```

Trong đó :

- guard\_expression: nếu được sử dụng thì có một tín hiệu tên là GUARD có kiểu Boolean được khai báo một cách không tường minh bằng một biểu thức gán giá trị cho tín hiệu đó, tín hiệu GUARD có thể sử dụng để điều khiển các thao tác bên trong khối.
- block\_header và block\_declarative\_part: là các khai báo và gán giá trị cho các tham số generics, ports.
- block\_statement\_part: khối chính của block chứa các lệnh đồng thời.

Ví dụ về các sử dụng block:

```
clump : block
begin
    A <= B or C;
    D <= B and not C;
end block clump ;

maybe : block ( B'stable(5 ns) ) is
    port (A, B, C : inout std_logic );
    port map ( A => S1, B => S2, C => outp );
    constant delay: time := 2 ns;
    signal temp: std_logic;
begin
    temp <= A xor B after delay;
    C <= temp nor B;
end block maybe;
```

Xét một ví dụ đầy đủ về sử dụng **guarded block** để mô phỏng một bus ba trạng thái như sau:

```
-----
library ieee;
use ieee.std_logic_1164.all;
-----
entity bus_drivers is
end bus_drivers;
-----
architecture behavioral of bus_drivers is
signal TBUS: STD_LOGIC := 'Z';
signal A, B, OEA, OEB : STD_LOGIC:= '0';
begin
    process
    begin
        OEA <= '1' after 100 ns, '0' after 200 ns;
        OEB <= '1' after 300 ns;
        wait;
    end process;
-----
    B1 : block (OEA = '1')
    begin
        TBUS <= guarded not A after 3 ns;
    end block;
-----
    B2 : block (OEB = '1')
    begin
        TBUS <= guarded not B after 3 ns;
    end block;
end behavioral;
```

Trong ví dụ trên TBUS được kết nối giá trị đảo của A, B chỉ trong trường hợp các tín hiệu cho phép tương ứng OEA, OEB bằng 1. Điều kiện này được khai báo ẩn trong tín hiệu GUARD của hai block B1 và B2.

## 7.2. Phát biểu quá trình

Quá trình (process) là một khối chứa các khai báo tuần tự nhưng thể hiện cách thức hoạt động của một phần thiết kế. Trong mô tả kiến trúc của khối thiết kế có thể có nhiều khai báo process và các khai báo này được thực hiện song song, độc lập với nhau không phụ thuộc vào thứ tự xuất hiện trong mô tả. Chính vì thế mặc dù trong khối quá trình chỉ chứa các khai báo tuần tự nhưng một khối được coi là một phát biểu đồng thời. Cấu trúc của quá trình như sau:

```
[postponed] process [(sensitivity_list)] [is]
    process_declarative_part
```

```

begin
    [process_statement_part]
end [ postponed ] process [ process_label ];

```

Trong đó :

- **postponed**: nếu được sử dụng thì quá trình là một quá trình được trì hoãn, nếu không đó là một quá trình thông thường. Quá trình trì hoãn là quá trình được thực hiện chỉ sau khi tất cả các quá trình không trì hoãn (**nonpostponed**) đã bắt đầu và kết thúc.
- **sensitivity\_list**: tương tự như danh sách tín hiệu được theo dõi đối với lệnh **wait**. Nó có ý nghĩa là mọi thay đổi của các tín hiệu trong danh sách này thì các lệnh trong quá trình này sẽ được thực hiện. Khi mô tả các khối logic tổ hợp thì các danh sách này chính là các tín hiệu đầu vào, còn đối với mạch dãy thì là tín hiệu xung nhịp đồng bộ clk.
- **process\_declarative\_part**: phần khai báo của quá trình, lưu ý là các khai báo tín hiệu không được đặt trong phần này.
- **process\_statement\_part**: Phần nội dung của **process**.

Ví dụ về cách sử dụng process:

```

-----
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-----
entity counter4 is
    port(
        count    : out std_logic_vector( 3 downto 0);
        enable   : in  std_logic;
        clk      : in  std_logic;
        reset    : in  std_logic
    );
end entity;
-----
architecture rtl of counter4 is
signal cnt :std_logic_vector ( 3 downto 0) := "0000";
begin
    process (clk, reset)
    begin
        if (reset = '1') then
            cnt <= "0000";
        elsif (rising_edge(clk)) then
            if (enable = '1') then
                cnt <= cnt + 1;
            end if;
        end if;
    end process;
end architecture;

```

```

        end if;
    end if;
end process;
count <= cnt;
end architecture;
-----
```

Ở ví dụ trên mô tả cấu trúc một bộ đếm 4 bit bằng cú pháp của process, danh sách sensitive list bao gồm tín hiệu đồng bộ CLK và tín hiệu RESET, tín hiệu RESET làm việc không đồng bộ, mỗi khi RESET bằng 1 thì giá trị đếm count được đặt giá trị 0. Nếu RESET bằng 0 và có sườn dương của xung nhịp CLK thì giá trị đếm được tăng thêm 1.

Khối process còn rất hay được sử dụng để mô tả các tổ hợp, khi đó tất cả các đầu vào của mạch tổ hợp phải được đưa vào danh sách sensitive list, nếu bỏ sót tín hiệu đầu vào trong danh sách này thì mặc dù cú pháp lệnh vẫn đúng nhưng chắc chắn mạch sẽ hoạt động sai về chức năng. Các lệnh bên trong khối process sẽ mô tả sự phụ thuộc logic của đầu ra với các đầu vào.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity mux2 is
port(A  : in  std_logic;
      B  : in  std_logic;
      sel : in  std_logic;
      S   : out std_logic);
end mux2;
-----
architecture behavioral of mux2 is
begin
mux: process (A, B, sel)
begin
    CASE sel IS
        WHEN '0'      => S <= A;
        WHEN others => S <= B;
    end case;
end process mux;

end behavioral;
```

Ở ví dụ trên mô tả một khối chọn kênh (MUX) dùng lệnh case, các đầu vào của gồm A, B và tín hiệu chọn Sel, đầu ra là S, S sẽ chọn bằng A nếu sel = 0 và bằng B nếu sel = 1.

### 7.3. Phát biểu gán tín hiệu đồng thời

Phát biểu gán tín hiệu đồng thời

```
[label :] [postponed] conditional_signal_assignment  
| [label:] [postponed] selected_signal_assignment  
options ::= [ guarded ] [ delay_mechanism ]
```

Trong đó :

- postponed: nếu được sử dụng thì lệnh gán được trì hoãn, nếu không đó là một lệnh gán thông thường. Lệnh gán trì hoãn được thực hiện chỉ sau khi tất cả các lệnh không trì hoãn (non-postponed) đã bắt đầu và kết thúc.
- options: trong mục này nếu dữ liệu được khai báo tùy chọn bảo vệ (guarded) thì sẽ thực hiện dưới sự điều khiển của tín hiệu ẩn GUARD như đã trình bày ở trên. delay\_mechanism là phương thức làm việc theo trễ của tín hiệu, có hai dạng là **transport** và **inertial** như đã trình bày ở 6.4.

Phát biểu gán tín hiệu đồng thời thường sử dụng trong các kiến trúc dạng dataflow. Các lệnh gán này cũng có thể sử dụng các cú pháp có điều kiện kiện sử dụng WHEN hoặc sử dụng tổ hợp WITH/SELECT/WHEN.

#### 7.3.1. Gán tín hiệu dùng WHEN

Cú pháp tổng quát như sau:

```
target <= waveform1 when condition1 else  
    waveform2 when condition2 else  
    •  
    •  
    •  
    waveformN-1 when conditionN-1 else  
    waveformN when conditionN else  
    default_waveform;
```

Trong đó default\_waveform có thể chọn giá trị UNAFFECTED. Tương đương với đoạn mã trên có thể sử dụng process và lệnh tuần tự if như sau:

```
process (sel1, waveform)  
begin  
if condition1 then  
    target <= waveform1  
elsif condition2 then  
    target <= waveform2  
    •  
    •  
    •
```

```

elsif conditionN-1 then
    target <= waveformN-1
elsif conditionN then
    target <= waveformN
end if ;
end process;

```

### 7.3.2. Gán tín hiệu dùng WITH/SELECT/WHEN

Cú pháp tổng quát như sau:

```

WITH expression SELECT
target <= options waveform1 WHEN choice_list1 ,
    waveform2 WHEN choice_list2 ,
    .
    .
    .
    waveformN-1 WHEN choice_listN-1,
    waveformN WHEN choice_listN
    default_value WHEN OTHERS ;

```

Trong đó default\_value có thể nhận giá trị UNAFFECTED Đoạn mã trên cũng có thể thay thế bằng lệnh case trong một process như sau:

```

CASE expression IS
WHEN choice_list1 =>
    target <= waveform1;
WHEN choice_list2 =>
    target <= waveform2;
    .
    .
    .
WHEN choice_listN-1 =>
    target <= waveformN_1;
WHEN choice_listN =>
    target <= waveformN;
END CASE;

```

Để minh họa cho sử dụng cấu trúc WHEN, và WITH/SELECT/WHEN xét mô tả một bộ chọn kênh như sau, khôi thiết kế gồm hai bộ giải mã kênh 4 đầu vào và 1 đầu ra độc lập với nhau, điều khiển bởi các tín hiệu sel1, sel2, bộ giải mã thứ nhất sử dụng cấu trúc WHEN, còn tổ hợp thứ hai sử dụng cấu trúc WITH/SELECT/WHEN.

```

-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity concurrent_expample is
port(

```

```

A           : in  std_logic_vector(3 downto 0);
B           : in  std_logic_vector(3 downto 0);
sel1, sel2 : in  std_logic_vector(1 downto 0);
O1, O2     : out std_logic
);
end concurrent_expample;
-----
architecture dataflow of concurrent_expample is
begin
    O1 <= A(0) WHEN sel1 = "00" else
                  A(1) WHEN sel1 = "01" else
                  A(2) WHEN sel1 = "10" else
                  A(3) WHEN sel1 = "11" else
                  UNAFFECTED;

    WITH sel2  SELECT
        O2 <= B(0) WHEN "00",
                      B(1) WHEN "01",
                      B(2) WHEN "10",
                      B(3) WHEN "11",
                      UNAFFECTED WHEN others;
end dataflow;
-----
```

Mã nguồn trên có thể thay thế bằng mã sử dụng các phát biểu tuần tự tương đương như sau:

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity sequential_expample is
port(
    A           : in  std_logic_vector(3 downto 0);
    B           : in  std_logic_vector(3 downto 0);
    sel1, sel2 : in  std_logic_vector(1 downto 0);
    O1, O2     : out std_logic
);
end sequential_expample;
-----
architecture dataflow of sequential_expample is
begin
    decodeA: process (A, sel1)
    begin
        if sel1 = "00" then O1 <= A(0); elsif
                           sel1 = "01" then O1 <= A(1); elsif
                           sel1 = "10" then O1 <= A(2); elsif
                           sel1 = "11" then O1 <= A(3);
```

```

        end if;
        end process decodeA;
decodeB: process (B, sel2)
begin
    CASE sel1 IS
        WHEN "00" => O2 <= B(0);
        WHEN "01" => O2 <= B(1);
        WHEN "10" => O2 <= B(2);
        WHEN "11" => O2 <= B(3);
        WHEN others => O2 <= 'X';
    END CASE;
end process decodeB;
end dataflow;
-----
```

#### 7.4. Phát biểu generate

Phát biểu generate (*generate statement*) là một cú pháp lệnh song song khác. Nó tương đương với khối lệnh tuần tự LOOP trong việc cho phép các đoạn lệnh được thực hiện lặp lại một số lần nào đó. Mẫu dùng của phát biểu này như sau:

```

for generate_parameter_specification
| if condition
generate
[ { block_declarative_item }
begin ]
{ concurrent_statement }
end generate [label];
```

Lưu ý rằng các tham số dùng cho vòng lặp **for** phải là các tham số tĩnh, nếu tham số dạng động sẽ gây ra lỗi biên dịch. Dưới đây là đoạn mã minh họa mô tả bộ cộng 4 bit, thay vì việc mô tả tường minh tất cả các bít của đầu ra ta dùng vòng lặp FOR/GENERATE để gán các giá trị cho chuỗi nhớ C và tổng O theo sơ đồ đệ quy:

```

-----  

library IEEE;  

use IEEE.STD_LOGIC_1164.ALL;  

-----  

entity adder4_gen 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_gen;
-----
architecture dataflow of adder4_gen is
signal C: std_logic_vector (4 downto 0);
begin
    C(0) <= CI;
    CO <= C(4);
    Carry: for i in 1 to 4 generate
        C(i) <= (A(i-1) and B(i-1)) or (C(i-1) and
(A(i-1) or B(i-1)));
    end generate Carry;
    Suma: FOR i IN 0 to 3 GENERATE
        SUM(i) <= A(i) xor B(i) xor C(i);
    END GENERATE Suma;
end dataflow;
-----
```

Câu lệnh GENERATE còn có thể được sử dụng dưới dạng cấu trúc IF/GENERATE . Ví dụ dưới đây minh họa cho cách sử dụng đó. Khối thiết kế gồm hai đầu vào 4 bit và một đầu ra 2 bít, tại các vị trí bít chẵn của a, b thì đầu ra bằng phép AND của các tín hiệu đầu vào:

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity generate_exapmle2 is
port(
    A : in  std_logic_vector(3 downto 0);
    B : in  std_logic_vector(3 downto 0);
    O : out std_logic_vector(1 downto 0)
);
end generate_exapmle2;
-----
architecture dataflow of generate_exapmle2 is
begin
    msk: FOR i IN 0 to 3 GENERATE
        ifgen: IF i rem 2 = 0 GENERATE
            O(i/2) <= A(i) and B(i);
        END GENERATE ifgen;
    END GENERATE msk;
end dataflow;
```

Như thấy ở các ví dụ trên lệnh GENERATE thường được sử dụng để cài đặt các khối xử lý logic giống nhau về mặt cấu trúc, số lượng các khối này có thể

rất nhiều cú pháp generate cho phép viết mã ngắn gọn. Ngoài ra trong một số trường hợp số lượng các khối cài đặt phụ thuộc vào các tham số tĩnh mà không phải hằng số thì việc cài đặt bắt buộc phải thực hiện bằng lệnh này.

## 7.5. Phát biểu cài đặt khối con

Phát biểu cài đặt khối con (*component installation statement*) sử dụng cho mô tả dạng cấu trúc của thiết kế khi khối thiết kế tổng được cấu tạo từ nhiều những khối nhỏ (xem ví dụ thêm ở 2.2.1).

Cú pháp tổng quát như sau:

```
component component_name is
    generic (generic_variable_declarations );
    port    (input_and_output_variable_declarations);
end component component_name;
```

Ở đoạn mã trên ta đã khai báo sử dụng các cổng AND2, OR2 và XOR2. Sau khi được khai báo component thì các cổng này có thể được sử dụng nhiều lần với cú pháp đầy đủ như sau:

```
identifier : component_name
    generic map( generic_variables => generic_values)
    port map(input_and_output_variables => signals);
```

Trong đó

-*identifier* là tên của khối con sẽ sử dụng cho cài đặt có tên là *component\_name*

-danh sách các biến generic và các cổng được gán tường minh bằng toán tử “=>”, khi đó thứ tự gán các biến hoặc cổng không quan trọng. Cách thứ hai là gán các cổng không tường minh, khi đó thứ tự các cổng phải đúng như thứ tự khai báo trong component. Tuy vậy không khuyến khích viết như vậy vì khi đó chỉ cần nhầm lẫn về thứ tự có thể làm sai chức năng của thiết kế. Để minh họa rõ hơn quay lại ví dụ về bộ cộng nhiều bit ở trên:

```
entity adder is
    generic ( N : natural := 32 ) ;
    port ( A      : in  std_logic_vector(N-1 downto 0);
           B      : in  std_logic_vector(N-1 downto 0);
           cin   : in  std_logic;
           Sum   : out std_logic_vector(N-1 downto 0);
           Cout  : out std_logic );
end entity adder ;
```

Khai báo component tương ứng sẽ là

```
component adder is
    generic (N : natural := 32) ;
    port ( A      : in  std_logic_vector(N-1 downto 0);
```

```

        B      : in  std_logic_vector(N-1 downto 0);
        cin   : in  std_logic;
        Sum   : out std_logic_vector(N-1 downto 0);
        Cout  : out std_logic );
end component adder ;
-----
```

Khai báo sử dụng component adder 16 bit dạng tường minh sẽ là:

```

U1: adder
generic map (N => 16)
port map (A => in_a, B=> in_b, Cin => in_c
          Sum => out_s, Cout => out_C
          ) ;
```

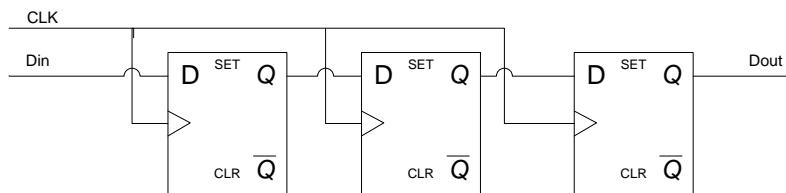
Khai báo sử dụng component adder 16 bit dạng không tường minh sẽ là:

```

U1: adder
generic map (16)
port map (
            in_a, in_b, in_c
            out_s, out_C
            );
```

Xét một ví dụ đầy đủ về cài đặt khói con dưới đây, ở phần 6.4 khi nghiên cứu về phát biểu gán tín hiệu tuần tự ta đã có mô tả VHDL của D\_flipflop, ví dụ sau dùng phát biểu cài đặt component để mô tả một thanh ghi dịch gồm 3 D-flipflop.

Sơ đồ logic của shift\_reg:



Hình 2-5. Sơ đồ khói của thanh ghi dịch

Mô tả VHDL:

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity shift_reg is
port(
    Din  : in  std_logic;
    CLK  : in  std_logic;
    Dout : out std_logic
);
```

```

end shift_reg;
-----
architecture structure of shift_reg is

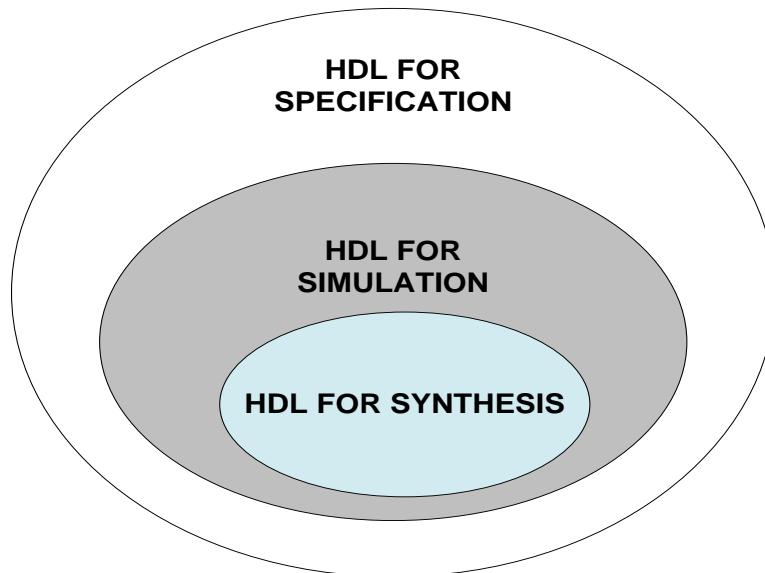
signal Q1, Q2 : std_logic;

component D_flipflop port(
    D   : in  std_logic;
    CLK : in  std_logic;
    Q   : out std_logic
);
end component;
begin

DFF1: D_flipflop
port map (D => Din, CLK => CLK, Q => Q1);
DFF2: D_flipflop
port map (D => Q1, CLK => CLK, Q => Q2);
DFF3: D_flipflop
port map (D => Q2, CLK => CLK, Q => Dout);
end structure;
-----
```

Ở ví dụ trên các 3 flipflop D được đặt nối tiếp và ta sẽ cài đặt cổng sao cho đầu ra của thanh ghi trước sẽ là đầu vào của thanh ghi sau và sau mỗi xung nhịp chuỗi bit Q1, Q2, Q3 sẽ dịch sang bên phải một bit.

## 8. Phân loại mã nguồn VHDL



Hình 2-6. Các dạng mã nguồn VHDL

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 nhắm tới 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.

Một số dạng mã nguồn không tổng hợp được mà chỉ dành cho mô phỏng được liệt kê ở dưới đây:

- Các mã mô tả độ trễ thời gian của tín hiệu, trên thực tế độ trễ các mạch do tính chất vật lý của phần cứng quy định, mã nguồn VHDL độc lập với phần cứng nên các mã quy định độ trễ đều không tổng hợp được mà chỉ dành cho mô phỏng:

```
wait for 5 ns;  
A <= B after 3 ns;  
- Các mã cài đặt giá trị, ban đầu  
signal a :std_logic_vector (3 downto 0) := "0001";  
signal n : BIT := '0';  
- Mô tả flip-flop làm việc ở cả hai sườn xung nhịp, trên thực tế các Flip-flop chỉ làm việc ở một sườn âm hoặc sườn dương, đoạn mã sau không tổng hợp được thành mạch.  
PROCESS ( Clk )  
BEGIN
```

```

        IF rising_edge(Clk) or falling_edge(CLk) THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
    - Các mã làm việc với kiểu số thực, các trình tổng hợp hiện tại mới chỉ
      hỗ trợ các mạch tổng hợp với số nguyên, các code mô tả sau không thể
      tổng hợp được:
    signal a, b, c: real
begin
    C <= a + b;
end architechture;
    - Các lệnh báo cáo và theo dõi (report, assert), các lệnh này
      phục vụ quá trình mô phỏng kiểm tra mạch, không thể tổng hợp được:
    assert a='1' report "it OK" severity NOTE;
    report "finished test";

```

## 9. Kiểm tra thiết kế bằng VHDL.

Một trong những phần công việc khó khăn và chiếm nhiều thời gian trong quá trình thiết kế vi mạch là phần kiểm tra thiết kế. Trên thực tế đã có rất nhiều phương pháp, công cụ khác nhau ra đời nhằm đơn giản hóa và tăng độ tin cậy quá trình kiểm tra. Một trong những phương pháp phổ biến và dễ dùng là phương pháp viết khói kiểm tra trên chính HDL, bên cạnh đó một số hướng phát triển hiện nay là thực hiện các khói kiểm tra trên các ngôn ngữ bậc cao như System C, System Verilog, các ngôn ngữ này hỗ trợ nhiều hàm bậc cao cũng như các hàm hệ thống khác nhau.

Trong khuôn khổ của chương trình chúng ta sẽ nghiên cứu sử dụng phương pháp thứ nhất là sử dụng mô tả VHDL cho khói kiểm tra. Lưu ý rằng mô hình kiểm tra ở dưới đây là không bắt buộc phải theo, người thiết kế sau khi đã nắm được phương pháp có thể viết các mô hình kiểm tra khác nhau tùy theo mục đích và yêu cầu.

Chúng ta bắt đầu với việc kiểm tra cho khói khói cộng 4 bit được viết bằng câu lệnh generate đã có ở phần 7.4:

```

-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity adder4_gen is
port(
    A    : in std_logic_vector(3 downto 0);
    B    : in std_logic_vector(3 downto 0);

```

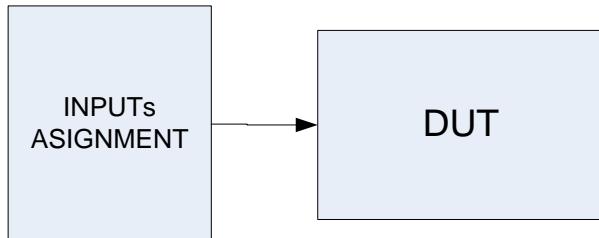
```

    CI  : in std_logic;
    SUM : out std_logic_vector(3 downto 0);
    CO   : out std_logic
  );
end adder4_gen;
-----
architecture dataflow of adder4_gen is
signal C: std_logic_vector (4 downto 0);
begin
  C(0) <= CI;
  CO <= C(4);
  Carry: for i in 1 to 4 generate
    C(i) <= (A(i-1) and B(i-1)) or (C(i-1) and
(A(i-1) or B(i-1)));
  end generate Carry;
  --SUM(0) <= A(0) xor B(0) xor CI;
  Suma: FOR i IN 0 to 3 GENERATE
    SUM(i) <= A(i) xor B(i) xor C(i);
  END GENERATE Suma;
end dataflow;
-----
```

Nhiệm vụ của chúng ta là cần kiểm tra xem với mô tả như trên thì bộ cộng có làm việc đúng chức năng không. Quy trình thiết kế chia làm hai bước, bước thứ nhất sẽ tiến hành kiểm tra nhanh với một vài giá trị đầu vào, bước thứ hai là kiểm tra toàn bộ thiết kế:

## 9.1. Kiểm tra nhanh

Sơ đồ kiểm tra nhanh như sau:



Hình 2-7. *Kiểm tra nhanh thiết kế*

Ở sơ đồ trên DUT là viết tắt của *Device Under Test* nghĩa là đối tượng bị kiểm tra, trong trường hợp này là bộ cộng 4 bit, khối Input generator sẽ tạo ra một hoặc một vài tổ hợp đầu vào để gán cho các cổng input của DUT.

Để hiểu kỹ hơn ta phân tích mã nguồn của khối kiểm tra như sau:

```
-----
library IEEE;
```

```

use IEEE.STD_LOGIC_1164.ALL;
-----
entity test_adder4_gen is
end test_adder4_gen;
-----
architecture testbench of test_adder4_gen is

component adder4_gen 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 component;
-- khai bao cac tin hieu vao ra cho DUT
signal A : std_logic_vector(3 downto 0) := "0101";
signal B : std_logic_vector(3 downto 0) := "1010";
signal CI : std_logic                      := '1';
-- output---
signal SUM : std_logic_vector(3 downto 0);
signal CO : std_logic;

begin
    DUT: component adder4_gen
        port map (
            A => A, B=> B, CI => CI,
            SUM => SUM, CO =>CO
        );
end testbench;
-----
```

Khối kiểm tra thường là một thiết kế mà không có cổng vào hoặc ra giống như ở trên. Trong khối kiểm tra sẽ khai báo DUT như một khối thiết kế con do vậy component adder4\_gen được khai báo trong phần khai báo của kiến trúc.

Bước tiếp theo cẩn cứ vào các cổng vào ra của DUT sẽ tiến hành khai báo các tín hiệu tương ứng, để tránh nhầm lẫn cho phép dùng tên các tín hiệu này trùng với tên các tín hiệu vào ra của DUT.

Các tín hiệu tương ứng với các chân input được gán giá trị khởi tạo là các hằng số hợp lệ bất kỳ, việc gán này tương đương với ta sẽ gửi tới DUT các giá trị đầu vào xác định.

Các tín hiệu tương ứng với chân output sẽ để trống và không được phép gán giá trị nào cả.

Bước cuối cùng là cài đặt DUT là một khối con adder4\_gen với các chân vào ra được gán tương ứng với các tín hiệu khai báo ở trên. Khi chạy mô phỏng ta thu được kết quả như sau:

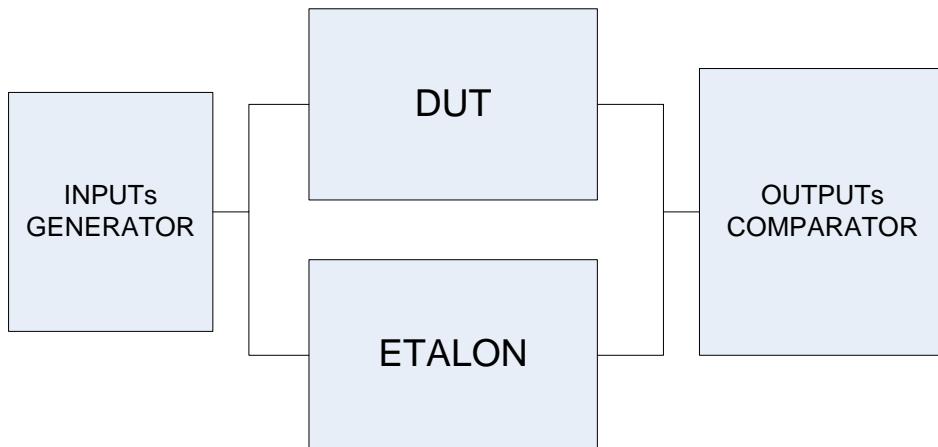
Messages								
+◆ /test_adder4_gen/a	0101	0101						
+◆ /test_adder4_gen/b	1010		1010					
◆ /test_adder4_gen/ci	1							
+◆ /test_adder4_gen/sum	0000	0000						
◆ /test_adder4_gen/co	1							

Hình 2-8. *Giản đồ sóng kiểm tra khối công 4 bit*

### 9.1. Kiểm tra tự động nhiều tổ hợp đầu vào

Việc kiểm tra nhanh cho phép phát hiện những lỗi đơn giản về mặt chức năng và không thể dựa vào kết quả của bước kiểm tra này để kết luận khối của chúng ta làm việc đúng hay chưa. Về mặt lý thuyết, khối thiết kế được coi là làm việc đúng nếu nó cho ra kết quả đúng với mọi tổ hợp đầu vào. ví dụ trên ta có 3 tín hiệu đầu vào là a(3:0), b(3:0), CI, vì vậy tổ hợp tất cả các tín hiệu đầu vào là  $2^4 * 2^4 * 2^1 = 2^9 = 512$  tổ hợp. Rõ ràng đối với khả năng của máy tính hiện đại đây là một con số rất nhỏ. Như vậy khối adder4 có thể được kiểm tra với độ tin cậy lên tới 100%.

Sơ đồ của kiểm tra tự động như sau:



Hình 2-9. *Mô hình kiểm tra tự động*

Hình 16: Mô hình chung của khối kiểm tra thiết kế tự động

Trong đó :

- DUT (device under test) đối tượng kiểm tra, ví dụ như trong trường hợp của chúng ta là adder4

- ETALON: là một thiết kế chuẩn thực hiện chức năng giống như DUT nhưng có thể sử dụng các cấu trúc bậc cao dạng không tổng hợp được (simulation only), sử dụng thuật toán đơn giản nhất có thể, etalon thường được mô tả để làm việc như một function để tránh các cấu trúc phụ thuộc thời gian hay phát sinh ra lỗi. Nhiệm vụ của Etalon là thực hiện chức năng thiết kế một cách đơn giản và chính xác nhất để làm cơ sở so sánh với kết quả của DUT.
- INPUTs GENERATOR: khối tạo đầu vào, khối này sẽ tạo các tổ hợp đầu vào khác nhau và đồng thời gửi đến hai khối là DUT và ETALON.
- OUTPUTs COMPARATORs: Khối này sẽ so sánh kết quả đầu ra tại những thời điểm nhất định và đưa ra tín hiệu cho biết là hai kết quả này có nhau không.

Để có thể gán nhiều đầu vào tại các điểm khác nhau trong khối kiểm tra phải tạo ra một xung nhịp đồng hồ dạng như CLK, các tổ hợp giá trị đầu vào mới sẽ được gán tại các thời điểm nhất định phụ thuộc CLK, thường là vào vị trí sườn âm hoặc sườn dương của xung nhịp.

Quay trở lại với module cộng 4 bit adder4\_gen ở trên, với thiết kế này ta có thể thực hiện kiểm tra cho tất cả các tổ hợp đầu vào (512 tổ hợp), trước hết ta viết một khối chuẩn etalon cho bộ cộng 4 bit như sau:

```
----- adder 4-bit etalon -----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-----
entity adder4_etalon 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_etalon;
-----
architecture behavioral of adder4_etalon is
signal s_sig: std_logic_vector(4 downto 0);
signal a_sig: std_logic_vector(4 downto 0);
signal b_sig: std_logic_vector(4 downto 0);
begin
    a_sig <= '0' & A;
```

```

b_sig <= '0' & B;
plus: process (a_sig, b_sig, CI)
begin
    s_sig <= a_sig + b_sig + CI;
end process plus;
SUM <= s_sig (3 downto 0);
CO   <= s_sig (4);
end behavioral;
-----
```

adder4\_etalon thực hiện phép cộng 4 bit bằng lệnh + của VHDL, kết quả phép cộng này là tin cậy 100% nên có thể sử dụng để kiểm tra thiết kế adder4\_gen ở trên của chúng ta. Để thực hiện các thao tác kiểm tra tự động, sử dụng một khối kiểm tra có nội dung như sau:

```

-----adder 4 testbench_full -----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
library STD;
use STD.TEXTIO.ALL;
-----
entity adder4_testbench is
end adder4_testbench;
-----
architecture testbenchfull of adder4_testbench is
signal a_t  : std_logic_vector(3 downto 0) := "0000";
signal b_t  : std_logic_vector(3 downto 0) := "0000";
signal sum_t : std_logic_vector(3 downto 0);
signal sum_e : std_logic_vector(3 downto 0);
signal ci_t  : std_logic           := '0';
signal co_t  : std_logic;
signal co_e  : std_logic;
signal clk   : std_logic           := '0';

component adder4_gen
    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 component;

component adder4_etalon
    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 component;
BEGIN
  --create clock
  create_clock: process
begin
  wait for 15 ns;
  CLK <= not CLK after 50 ns;
end process create_clock;

check:
process (clk)
variable info: line;
variable test_cnt: integer := 0;
begin
if clk = '1' and clk'event then
  write(info, string'("Test # "));
  write(info, integer'(test_cnt + 1));
  write(info, string'(" a = "));
  write(info, integer'(conv_integer(a_t)));
  write(info, string'(" b = "));
  write(info, integer'(conv_integer(b_t)));
  write(info, string'(" CI = "));
  write(info, integer'(conv_integer(ci_t)));
  write(info, string'(" sum = "));
  write(info, integer'(conv_integer(sum_t)));
  write(info, string'(" CO = "));
  write(info, integer'(conv_integer(co_t)));
  write(info, string'(" sum_e = "));
  write(info, integer'(conv_integer(sum_e)));
  write(info, string'(" CO_e = "));
  write(info, integer'(conv_integer(co_e)));
  if sum_e /= sum_t or co_e /= co_t then
    write(info, string'("FAILURE"));
  else
    write(info, string'(" OK"));
  end if;
  writeline(output, info);
  -- input data generator.
  test_cnt := test_cnt + 1;
  ci_t <= not ci_t;
  if ci_t = '1' then

```

```

    a_t <= a_t +1;
end if;
if a_t = "1111" then
    b_t <= b_t + 1;
end if;
assert test_cnt < 512
report "end simulation" severity NOTE;
end if;
end process check;
-- component installation
dut: adder4_gen
port map (
    A => a_t, B => b_t, CI => ci_t,
    SUM =>sum_t, CO => co_t);
etalon: adder4_etalon
port map (A => a_t, B => b_t, CI => ci_t,
    SUM =>sum_e, CO => co_e);
END testbenchfull;
-----
```

Khối kiểm tra cài đặt đồng thời các khối con là adder4\_gen (DUT) và adder4\_etalon (ETALON), các khối này có các đầu vào dữ liệu y hệt nhau lấy từ các tín hiệu tương ứng là a\_t, b\_t, ci\_t. Các đầu ra của hai khối này tương ứng là sum\_t, co\_t cho DUT và sum\_e, co\_e cho ETALON.

Khối kiểm tra tự động này thực chất lặp lại chức năng của khối kiểm tra nhanh nhiều lần với nhiều tổ hợp đầu vào, mặt khác sử dụng kết quả tính toán từ khối thiết kế chuẩn ETALON để kiểm tra tự động tính đúng đắn của kết quả đầu ra từ khối DUT cần kiểm tra.

Để làm được việc đó, một xung nhịp clk được tạo ra, chu kỳ của xung nhịp này có thể nhận giá trị bất kỳ, ví dụ ở đoạn mã trên ta tạo xung nhịp clk có chu kỳ T = 2 x 50 ns = 100 ns. Có xung nhịp này ta sẽ tạo một bộ đếm theo xung đếm là clk để đếm số lượng test, khi đếm đủ 512 test thì sẽ thông báo ra màn hình việc thực hiện test đã xong.

Tại mỗi thời điểm sùn dương của clk giá trị đếm này tăng thêm 1 đồng thời tại thời điểm đó tổ hợp giá trị đầu vào thay đổi tuần tự để quét hết 512 tổ hợp giá trị khác nhau. Đầu tiên các giá trị a\_t, b\_t, ci\_t nhận giá trị khởi tạo là 0, 0, 0. Tại các sùn dương của xung nhịp thay đổi giá trị ci\_t trước, nếu ci\_t = 1 thì tăng a\_t thêm 1 đơn vị, sau đó kiểm tra nếu a\_t = "1111" thì sẽ tăng b\_t thêm 1 đơn vị.

Các đầu ra sum\_t, sum\_e, co\_t, co\_e sẽ được so sánh với nhau sau mỗi xung nhịp, nếu như sum\_t = sum\_e và co\_t = co\_e thì kết luận là adder4\_gen làm việc đúng (TEST OK) và ngược lại là làm việc sai (TEST FAILURE)

Ví số lượng tổ hợp đầu vào là lớn nên không thể quan sát bằng waveform nứa, kết quả khi đó được thông báo trực tiếp ra màn hình. Khi chạy mô phỏng ta thu được thông báo như sau:

```
#Test#1a = 0b = 0 CI = 0 sum = 0 CO = 0sum_e = 0CO_e = 0 OK
#Test#2a = 0b = 0 CI = 1 sum = 1 CO = 0sum_e = 1CO_e = 0 OK
#Test#3a = 1b = 0 CI = 0 sum = 1 CO = 0sum_e = 1CO_e = 0 OK
#Test#4a = 1b = 0 CI = 1 sum = 2 CO = 0sum_e = 2CO_e = 0 OK
#Test#5a = 2b = 0 CI = 0 sum = 2 CO = 0sum_e = 2CO_e = 0 OK
#Test#6a = 2b = 0 CI = 1 sum = 3 CO = 0sum_e = 3CO_e = 0 OK
#Test#7a = 3b = 0 CI = 0 sum = 3 CO = 0sum_e = 3CO_e = 0 OK
#Test#8a = 3b = 0 CI = 1 sum = 4 CO = 0sum_e = 4CO_e = 0 OK
#Test#9a = 4 b = 0 CI = 0 sum = 4 CO = 0sum_e = 4CO_e = 0 OK
#Test#10a = 4 b = 0 CI = 1sum = 5 CO = 0sum_e = 5CO_e = 0 OK
#Test#11a = 5 b = 0 CI = 0sum = 5 CO = 0sum_e = 5CO_e = 0 OK
...
...
#Test#511a =15b = 14CI = 0sum =13CO = 1sum_e =13CO_e= 1OK
#Test#512a =15b = 15CI = 1sum= 15CO = 1sum_e =15CO_e = 1 OK
# ** Note: end simulation
# Time: 61385 ns Iteration: 0 Instance: /adder4_testbench
```

Nếu như tất cả các trường hợp đều thông báo là OK thì có thể kết luận DUT làm việc đúng, quá trình kiểm tra hoàn tất.

Lưu ý rằng khi thực hiện kiểm tra tự động thì ETALON không phải lúc nào cũng có độ chính xác 100% như ở trên và đôi khi rất khó để viết được khôi này này. Khi đó ta phải có phương án kiểm tra khác.

Điểm lưu ý thứ hai là trên thực tế việc kiểm tra với mọi tổ hợp đầu vào (*full\_tb*) thường là không thể vì số lượng các tổ hợp này trong đa số các trường hợp rất lớn ví dụ như nếu không phải bộ cộng 4 bit mà là bộ cộng 32 bit thì số lượng tổ hợp đầu vào là  $2^{32 \times 2 + 1} = 2^{65}$  là một con số quá lớn để kiểm tra hết dù bằng các máy tính nhanh nhất. Khi đó quá trình kiểm tra chia thành hai bước:

Bước 1 sẽ tiến hành kiểm tra bắt buộc với các tổ hợp có tính chất riêng như bằng 0 hay bằng số lớn nhất, hoặc các tổ hợp gây ra các ngoại lệ, các tổ hợp đã gây phát sinh lỗi.

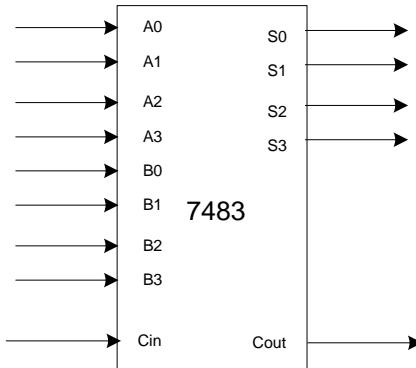
Nếu kiểm tra với các tổ hợp này không có lỗi sẽ chuyển sang bước thứ hai là RANDOM\_TEST. Bộ phân kiểm tra sẽ cho chạy RANDOM\_TEST với số lượng đầu vào không giới hạn. Nếu trong quá trình này phát hiện ra lỗi thì tổ hợp

giá trị gây ra lỗi sẽ được bổ xung vào danh sách các tổ hợp bắt buộc phải kiểm tra ở bước 1 và làm lại các bước kiểm tra từ đầu sau khi sửa lỗi. Kiểm tra được coi là thực hiện xong nếu như với một số lượng rất lớn RANDOM\_TEST mà không tìm thấy lỗi.

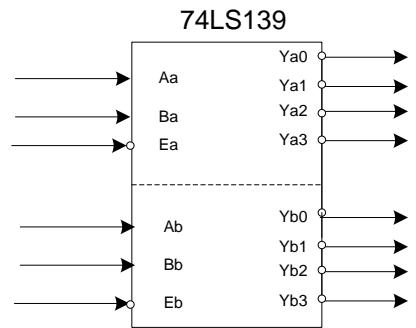
## Bài tập chương 2

### Bài tập

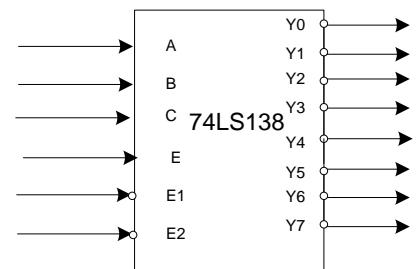
- Thiết kế full\_adder trên VHDL, trên cơ sở đó thiết kế bộ cộng 4 bit tương tự IC 7483.



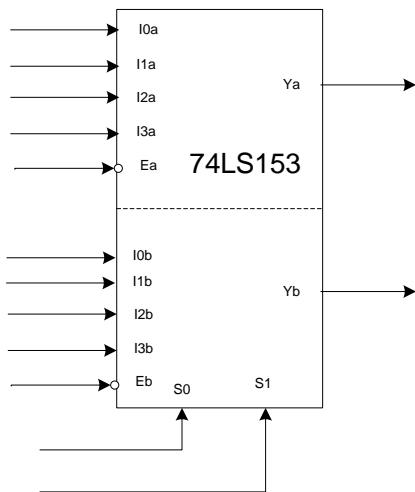
- Thiết kế bộ giải mã nhị phân 2\_to\_4 có đầu ra thuận, nghịch tương tự IC 74LS139



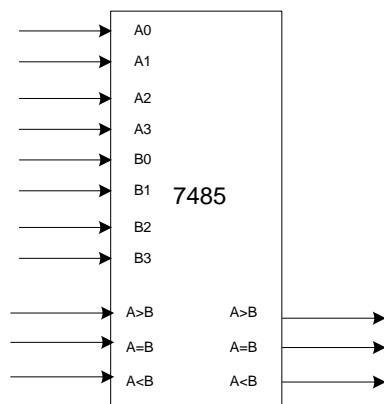
- Thiết kế bộ giải mã nhị phân 3\_to\_8 có đầu ra thuận, nghịch tương tự IC 74LS138.



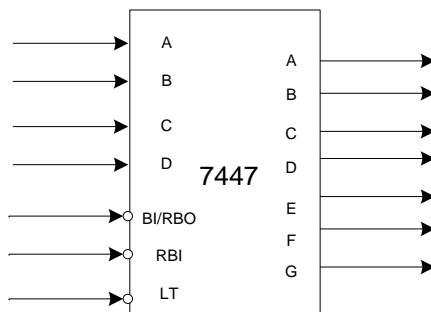
- Thiết kế bộ chọn kênh 4 đầu vào 1 đầu ra MUX4\_1 tương tự IC 74153 nhưng chỉ hỗ trợ một kênh chọn (IC này có hai kênh chọn riêng biệt như hình vẽ)



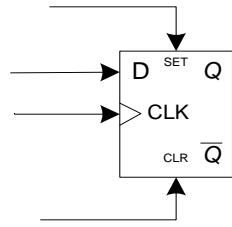
5. Thiết bộ phân kênh 1 đầu vào 4 đầu ra DEMUX1\_4.
6. Thiết kế bộ cộng/ trừ 4 bit sử dụng toán tử cộng trên VHDL.
7. Thiết kế bộ so sánh hai số không dấu 4 bit tương tự IC 7485.



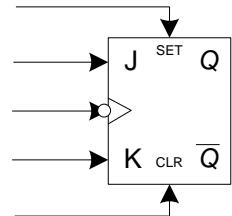
8. Thiết kế các bộ chuyển đổi mã từ NBCD – 7-SEG(LED 7 đoạn) tương tự IC 7447, hỗ trợ cổng LamTest, khi cổng này có giá trị bằng 1, tất cả đèn phải sáng không phụ thuộc mã đầu NBCD đầu vào. Để đơn giản, các chân RBI, RBO không cần thiết kế.



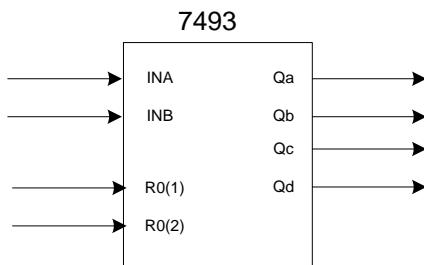
9. Thiết kế các flip-flop đồng bộ D với đầy đủ các chân tín hiệu như sau:



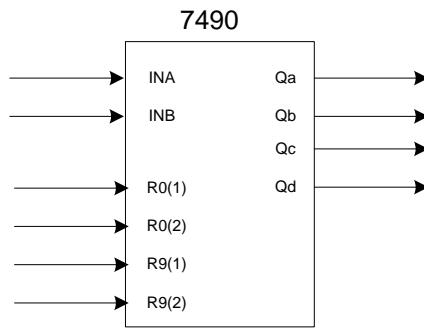
10. Thiết kế các flip-flop đồng bộ JK với đầy đủ các chân tín hiệu như sau:



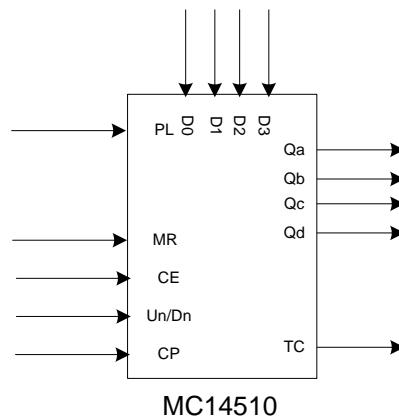
11. Thiết kế trên VHDL thanh ghi dịch trái qua phải 32-bit, số lượng bit dịch là một số nguyên từ 1-31 trên VHDL (sử dụng toán tử dịch).
12. Thiết kế thanh ghi dịch đồng bộ nối tiếp 4 bit sang bên trái, với đầu vào nối tiếp SL, hỗ trợ tín hiệu Reset không đồng bộ và tín hiệu Enable.
13. Thiết kế thanh 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.
14. 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.



15. 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.



16. Thiết kế IC đếm theo cấu trúc của IC 4510, 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:



17. Thiết kế bộ đếm thập phân đồng bộ, RESET không đồng bộ, có tín hiệu ENABLE.
18. 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.
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 ‘1’. Trường hợp không có bít ‘1’, thì đầu ra nhận giá trị không xác định. (“XX”).
20. 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 bít ‘0’, thì đầu ra nhận giá trị không xác định. (“XX”).
21. Thiết kế khối tính giá trị Parity cho một chuỗi 8 bit với quy định Parity = 1 nếu số bít bằng 1 là số lẻ.
22. 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.

23. Thiết kế các bộ chuyển đổi mã từ

BINARY – BCD, BCD – BINARY cho các số có giá trị từ 0-99.

BCD – GRAY, GRAY – BCD.

BCD – 7SEG, 7SEG – BCD.

7SEG– GRAY, GRAY-7SEG

24. Thiết kế bộ đếm nhị phân Kđ = 12 dùng JK Flip-flop

25. Hiện thực sơ đồ mã CRC nối tiếp và song song bằng VHDL.

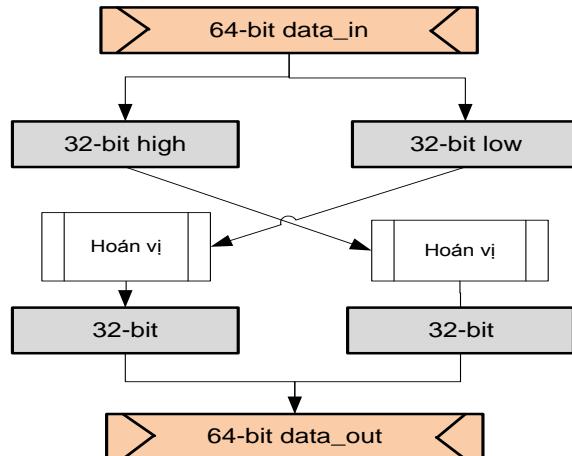
26. Thiết kế khối cộng cho 2 số NBCD, kết quả thu được là một số có hai chữ số biểu diễn dưới dạng NBCD.

27. Thiết kế khối thực thi nhiệm vụ hoán vị các vị trí bit của một chuỗi 32 bit theo ma trận hoán vị sau:

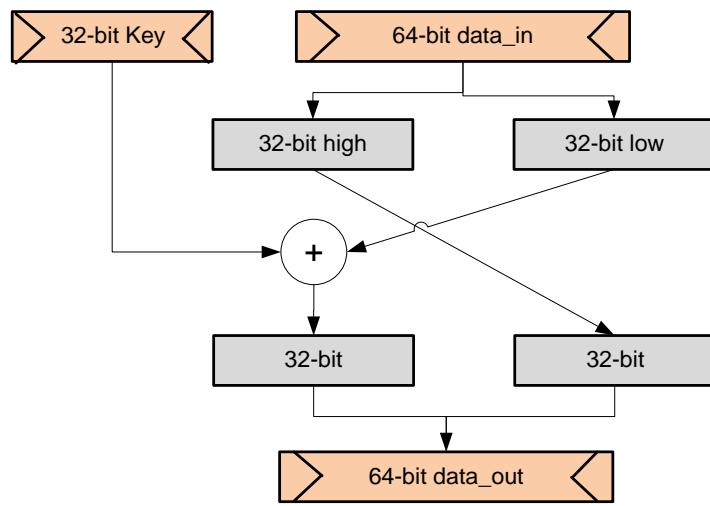
18	2	3	28	5	19	30	8
9	15	23	16	13	27	10	12
17	1	6	20	22	21	11	24
26	25	14	4	29	7	31	0

28. Theo ma trận trên thì bit thứ 31 của kết quả bằng bit thứ 18 của chuỗi đầu vào, bit thứ 30 là bit thứ 2 và tiếp tục như vậy cho tới hết.

29. Viết mô tả VHDL cho khối thiết kế có sơ đồ sau, trong đó các ma trận hoán vị lấy từ ma trận bài 25.



30. Viết mô tả VHDL cho khối biến đổi dữ liệu sau, dữ liệu đầu vào được tác thành hai phần 32 bit thấp và 32 bit cao, sau đó từng phần được xử lý riêng như trên sơ đồ và kết quả đầu ra được hợp bởi kết quả của từng phần này.



## Câu hỏi ôn tập lý thuyết

1. Trình bày sơ lược về ngôn ngữ mô tả phần cứng, các ngôn ngữ phổ biến, ưu điểm của phương pháp dùng HDL để thiết kế phần cứng
2. Cấu trúc của thiết kế bằng VHDL.
3. Các dạng mô tả kiến trúc khác nhau trong VHDL, ưu điểm, nhược điểm và ứng dụng của từng loại.
4. Trình bày về đối tượng dữ liệu trong VHDL.
5. Trình bày về các kiểu dữ liệu trong VHDL, kiểu dữ liệu tiền định nghĩa và dữ liệu định nghĩa bởi người dùng.
6. Toán tử và biểu thức trong VHDL.
7. Phát biểu tuần tự, bản chất, ứng dụng, lấy ví dụ VHDL đơn giản về phát biểu này.
8. Phát biểu đồng thời, bản chất, ứng dụng, lấy ví dụ VHDL đơn giản về phát biểu này.
9. Phân loại mã nguồn VHDL, thế nào là mã tổng hợp được và mã chỉ dùng mô phỏng.
10. Yêu cầu chung đối với kiểm tra thiết kế trên VHDL, các dạng kiểm tra thiết kế trên VHDL.
11. Vai trò và phương pháp tổ chức kiểm tra nhanh khỏi thiết kế VHDL.
12. Vai trò và phương pháp tổ chức kiểm tra tự động khỏi thiết kế VHDL.
13. Mô tả khối tổ hợp trên VHDL, lấy ví dụ.
14. Mô tả mạch tuần tự trên VHDL, lấy ví dụ.

## **Chương 3**

# **THIẾT KẾ CÁC KHỐI MẠCH DÃY VÀ TỔ HỢP THÔNG DỤNG**

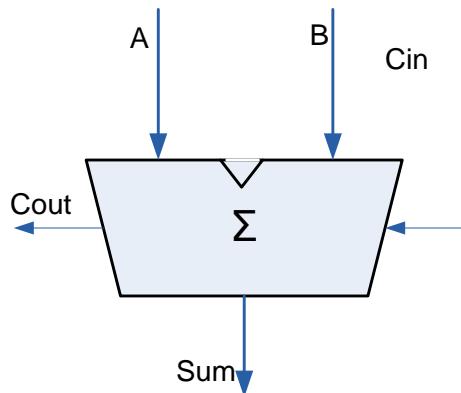
Nội dung của chương II cung cấp cho người học những kỹ năng cơ bản của VHDL, trên cơ sở đó chương III sẽ giới thiệu tiếp cho người học bắt đầu thiết kế các khối số phức tạp bằng cách ghép nối các khối cơ bản lại để thành thiết kế lớn hoàn chỉnh. Bên cạnh đó chương III cũng cung cấp một số lượng lớn các thuật toán khác nhau cho những khối thiết kế thông dụng từ bộ cộng, bộ đếm, thanh ghi, khối nhân, chia cho tới các khối làm việc với số thực. Người học có thể trên cơ sở đó để nghiên cứu hoàn thiện, phát triển về mặt thuật toán và cấu trúc cũng biết cách tiếp cận với các thuật toán phức tạp hơn trong tài liệu tham khảo, bên cạnh đó là khả năng sử dụng các khối thiết kế làm nền tảng cho các thiết kế cấp độ phức tạp hơn gấp trong các bài toán chuyên ngành.

Những kỹ năng người học phải có được sau khi thực hiện các bài tập thực hành ở chương này là làm chủ được thiết kế về mặt cấu trúc cũng như về mặt hành vi, nói một cách khác là hiểu rõ cấu tạo của mạch và cách thức mạch này thực hiện chức năng thiết kế hay là giản đồ thời gian làm việc của mạch. Phần thực hành của chương III bao gồm những bài tập từ dễ đến khó, với mục đích giúp người đọc rèn luyện tư duy thiết kế các khối mạch số từ việc xây dựng sơ đồ thuật toán, sơ đồ hiện thực hóa cho tới mô tả bằng VHDL và mô phỏng kiểm tra.

# 1. Các khối cơ bản

## 1.1. Khối cộng đơn giản

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

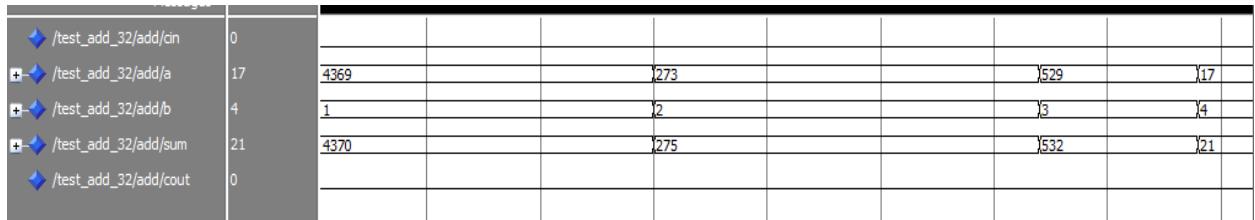
Hàm cộng có thể được mô tả trực tiếp bằng toán tử “+” mặc dù với kết quả này thì mạch cộng tổng hợp ra sẽ không đạt được tối ưu về tốc độ cũng như tài nguyên, mô tả VHDL của bộ cộng như sau:

```
----- 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.



Hình 3-2. Kết quả mô phỏng bộ cộng

## 1.2. Khối trừ

Vì các số có dấu trên máy tính được biểu diễn dưới dạng số bù 2 ( $2^r$  complement), do đó để thực hiện phép trừ  $A - B$  thì tương đương với thực hiện  $A + \bar{B} + 1$

Xét ví dụ  $A = 10 = 1010$ ,  $B = 5 = 0101$  biểu diễn dưới dạng số có dấu 5-bit ta phải thêm bit dấu bằng 0 vào trước.

$$\begin{aligned} A &= 01010, \quad \bar{B}_2(A) = \text{not } (A) + 1 = 10101 + 1 = 10110 \\ B &= 00101, \quad \bar{B}_2(B) = \text{not } (B) + 1 = 11010 + 1 = 11011 \end{aligned}$$

Tính  $A - B$ :

$$\begin{array}{r} A \quad 01010 \quad 01010 \\ - = - \quad = + \\ B \quad 00101 \quad 11011 \\ \hline 1 \quad 00101 \end{array}$$

Loại bỏ bit nháy ở kết quả cuối cùng ta được  $A - B = 00101 = 5$ .

Tính  $B - A$ :

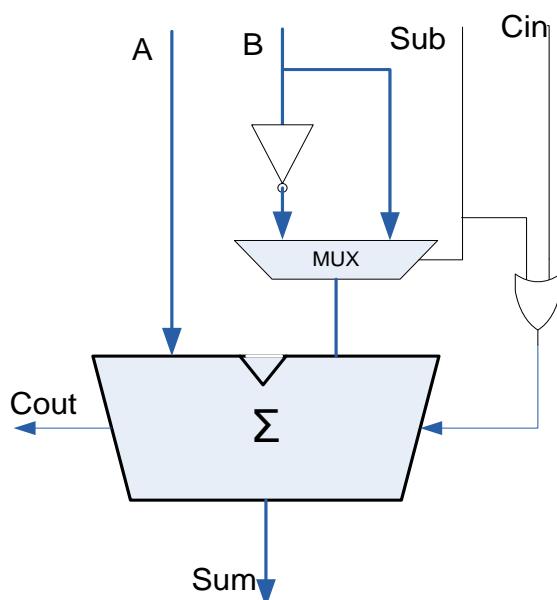
$$\begin{array}{r} B \quad 00101 \quad 00101 \\ - = - \quad = + \\ A \quad 01010 \quad 10110 \\ \hline 0 \quad 11011 \end{array}$$

Loại bỏ bit nhớ ta được  $B - A = 11101$ , đây là số âm, muốn tính giá trị tuyệt đối để kiểm tra lại lấy bù 2 của  $11101$

$$\text{Bù } 2 \text{ (11101)} = 00100 + 1 = 00101 = 5$$

$$\text{vậy } B - A = -5$$

Dựa trên tính chất trên của số bù hai ta chỉ cần thực hiện một thay đổi nhỏ trong cấu trúc của bộ cộng để nó có khả năng thực hiện cả phép cộng lẫn phép trừ mà không làm thay đổi lớn về tài nguyên logic cũng như độ trễ của mạch. Tại đầu vào sẽ bổ xung thêm tín hiệu  $SUB$ , tín hiệu này quyết định sẽ thực hiện phép cộng hay phép trừ. Khi  $SUB = 1$  để lấy bù 2 của  $B$  sẽ lấy đảo  $B$  và cho giá trị đầu vào  $Cin = 1$ , để hiện thực trên mạch cấu trúc bộ cộng được bổ xung một khối MUX trước cổng  $B$ , khối này có hai đầu vào là  $B$  và  $\text{not } B$ , nếu  $SUB = 0$  thì  $B$  được chọn, nếu  $SUB = 1$  thì  $\text{not } B$  được chọn. Đầu vào  $Cin$  được OR với  $SUB$  trước khi vào bộ cộng.



Hình 3-3. Sơ đồ khối bộ cộng trừ đơn giản

Trong mã nguồn cho khối bộ cộng/trừ adder\_sub.vhd sử dụng bộ cộng adder32 như một khối con (component).

```

----- Bo cong tru doan gian -----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-----
entity adder_sub is

```

```

port(
    SUB : in std_logic;
    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 adder_sub;
-----
architecture rtl of adder_sub is
signal B_temp      : std_logic_vector(31 downto 0);
signal Cin_temp    : std_logic;

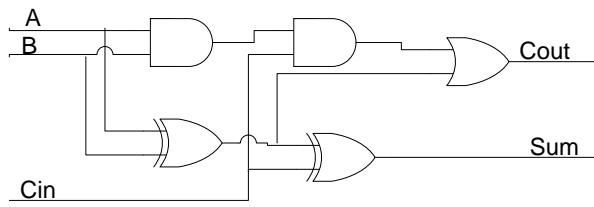
component adder32 is

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 component;
-----
begin
    Cin_temp <= SUB or Cin;
    MUX32: process (B, SUB)
    begin
        if SUB = '1' then
            B_temp <=  not B;
        else
            B_temp <= B;
        end if;
    end process MUX32;

    add: component adder32
        port map (Cin_temp, A, B_temp, SUM, Cout);
end rtl;
-----
```

### 1.3. Khối cộng thấy nhớ trước.

Độ trễ tổ hợp của khối cộng gây ra bởi chuỗi bit nhớ, bộ cộng nối tiếp có đặc điểm là độ trễ cao do đặc điểm của chuỗi bit nhớ là bit nhớ sau phải đợi bit nhớ trước nó.



Hình 3-4. Sơ đồ khối bộ cộng 1 bit đầy đủ

Như thấy trên hình vẽ thì thời gian trễ của chuỗi bit nhớ phải thông qua tối thiểu một cổng AND và một cổng OR, nếu là bộ cộng 32-bit thì tổng thời gian trễ là thời gian trễ của 32 cổng AND và 32 cổng OR. Trên thực tế độ trễ của cổng AND, cổng OR gần tương đương nên để đơn giản ta xem độ trễ của một trong hai cổng này là một lớp trễ hay một “level” logic.

Như vậy bộ cộng nối tiếp có  $32 \times 2 = 64$  lớp trễ.

Phép cộng là một phép toán cơ bản và sử dụng nhiều do vậy việc nghiên cứu, sử dụng các thuật toán tăng tốc bộ cộng đã và đang được thực hiện rất nhiều. Trong phần này ta xem xét một thuật toán phổ biến nhằm rút ngắn thời gian thực hiện tính toán chuỗi bit nhớ là thuật toán Cộng thấy nhớ trước (*Carry Look-Ahead Adder*). 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:

$g_i = a_i \text{ and } b_i = a_i \cdot b_i$  – 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.

$p_i = a_i \text{ or } b_i = a_i + b_i$  – 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  $-1 \leq j < i$   $g_j = 1$  (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  $g_j \cdot p_{j+1} \cdot p_{j+2} \cdot p_i = 1$ .

Ví dụ bit nhớ ở vị trí thứ 0 là  $c_0 = 1$  nếu như có nhớ sang vị trí thứ 1 và bằng 0 nếu như không có nhớ.  $C_0$  bằng 1 nếu như hoặc tại vị trí 0 có sinh nhớ ( $g_0 = 1$ ), hoặc có nhớ của Cin và nhớ này được “lan truyền” qua vị trí thứ 0 (Cin = 1 và  $P_0 = 1$ ).

Công thức cho các bit nhớ có thể viết theo quy luật sau:

$$\begin{aligned}
 c_0 &= g_0 + C_{in} \cdot p_0, \\
 c_1 &= g_1 + g_0 \cdot p_1 + C_{in} \cdot p_0 \cdot p_1 = g_1 + c_0 \cdot p_1, \\
 c_2 &= g_2 + g_0 \cdot p_1 \cdot p_2 + g_1 \cdot p_2 + C_{in} \cdot p_0 \cdot p_1 \cdot p_2 = g_2 + c_1 \cdot p_2, \\
 c_3 &= g_3 + g_0 \cdot p_1 \cdot p_2 \cdot p_3 + g_1 \cdot p_2 \cdot p_3 + g_2 \cdot p_3 + C_{in} \cdot p_0 \cdot p_1 \cdot p_2 \cdot p_3 = g_3 + \\
 c_2 \cdot p_3, \\
 &\dots
 \end{aligned} \tag{3.1}$$

Các công thức ở về thực chất là một cách trình bày khác của thuật toán cộng nối tiếp. Nhưng cách trình bày này gợi ý cho một sơ đồ có thể rút ngắn thời gian tính các giá trị c: theo công thức trên thì các giá trị bit nhớ sau vẫn phụ thuộc vào giá trị bit nhớ trước, để tính  $c_0, c_1, c_2, c_3$  thì phải có Cin xác định. Sự phụ thuộc này là tự nhiên và không thể thay đổi. Nếu bỏ hết các yếu tố phụ thuộc vào Cin và tính các đại lượng trung gian sau:

$$\begin{aligned}
 p_i &= a_i + b_i \text{ (với } i=0 \text{ -3)} \\
 g_i &= a_i \cdot b_i \text{ (với } i=0 \text{ -3)} \\
 p_{01} &= p_0 \cdot p_1 \\
 p_{02} &= p_0 \cdot p_1 \cdot p_2 \\
 p_{03} &= p_0 \cdot p_1 \cdot p_2 \cdot p_3 \\
 g_{01} &= g_1 + g_0 \cdot p_1 \\
 g_{02} &= g_2 + g_0 \cdot p_1 \cdot p_2 + g_1 \cdot p_2 \\
 g_{03} &= g_3 + g_0 \cdot p_1 \cdot p_2 \cdot p_3 + g_1 \cdot p_2 \cdot p_3 + g_2 \cdot p_3 = g_{01}p_{23} + g_{23} \tag{3.2}
 \end{aligned}$$

Khi đó nếu chia khối cộng N-bit thành các khối 4 bit thì các đại lượng trên của tất cả các khối sẽ được tính song song với nhau. Ta gọi khối thực thi các tính toán đó là CLA (Carry Look Ahead), khi đó có thể phân tích được độ trễ của một CLA như sau

Bảng 3-1

#### Tính số lớp trễ của CLA

Lớp trễ	p	G
1	$p_i = a_i + b_i$ (với $i=0 \text{ -3}$ )	$g_i = a_i \cdot b_i$ (với $i=0 \text{ -3}$ )
2	$p_{01}, p_{12}, p_{23}$	$g_0p_1, g_1p_2, g_2, p_3$
3	$P_{03}$	$g_{01}, g_{23} = (g_3 + g_2p_3)$
4	--	$g_{02}, g_{01}p_{23}$
5	--	$g_{03}$

Có thể tính được để tính xong  $p_{03}$  phải cần 3 lớp trẽ logic, để tính được  $g_{03}$  cần 5 lớp trẽ logic. Nghĩa là các giá trị  $p$  luôn được tính trước  $g$ .

Một cách tổng quát nếu nhìn lại sẽ thấy ý nghĩa của các đại lượng  $p, g$  tính bởi CLA vẫn không thay đổi, ví dụ  $p_{03} = 1$  nói lên rằng cụm các bit từ 0 tới 3 có khả năng lan truyền bit nhớ,  $g_{02} = 1$  nói lên rằng cụm các bit từ 0 tới 2 sinh ra bit nhớ.

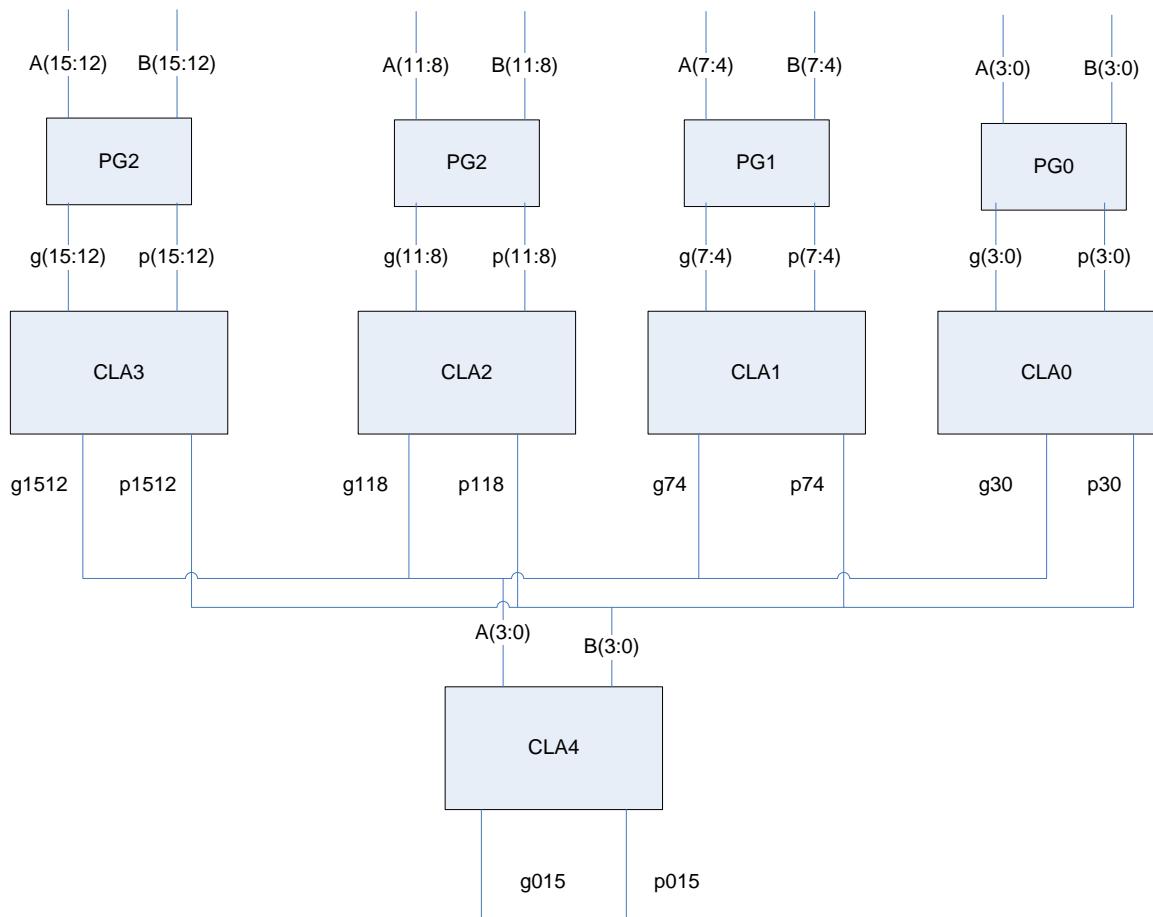
Ta viết lại công thức cho các bit nhớ  $c_3$ , và tương tự cho  $c_7, c_{11}, c_{15} \dots$  như sau

$$\begin{aligned}c_3 &= g_{03} + C_{in} \cdot p_{03} \\c_7 &= g_{47} + g_{03} \cdot p_{47} + C_{in} \cdot p_{03} \cdot p_{47} \\c_{12} &= g_{811} + g_{47} \cdot p_{811} + g_{03} \cdot p_{47} \cdot p_{811} + C_{in} \cdot p_{03} \cdot p_{47} \cdot p_{811} \\c_{15} &= g_{1215} + g_{811} \cdot p_{1215} + g_{47} \cdot p_{811} \cdot p_{1215} + g_{03} \cdot p_{47} \cdot p_{811} \cdot p_{1215} + C_{in} \cdot p_{03} \cdot p_{47} \cdot p_{811} \cdot p_{1215}\end{aligned}\quad (3.3)$$

Trong đó các đại lượng  $g_{03}, p_{03}, g_{47}, p_{47}, g_{811}, p_{811}, g_{1215}, p_{1215}$  được tính song song. Các công thức trên hoàn toàn trùng lặp với công thức tính bit nhớ cho một bộ cộng 4 bit (1), do vậy khối trên lại được xây dựng trên cơ sở một khối CLA.

Từ bảng tính độ trễ của một CLA có thể suy ra để tính được  $c_3$  cần 6 lớp trẽ logic, tính được  $c_7$  cần 7 lớp trẽ logic,  $c_{11}$  cần 8 lớp trẽ logic,  $c_{15}$  cần 9 lớp trẽ logic. Nếu so sánh với bộ cộng 16 bít cần  $16 \times 2 = 32$  lớp trẽ 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.

Trên thực tế bộ cộng Carry Look Ahead Adder thường được xây dựng từ các bộ 4 bít CLA, mỗi bộ này có nhiệm vụ tính toán các giá trị trung gian. Sơ đồ khối của chuỗi bit nhớ như sau.



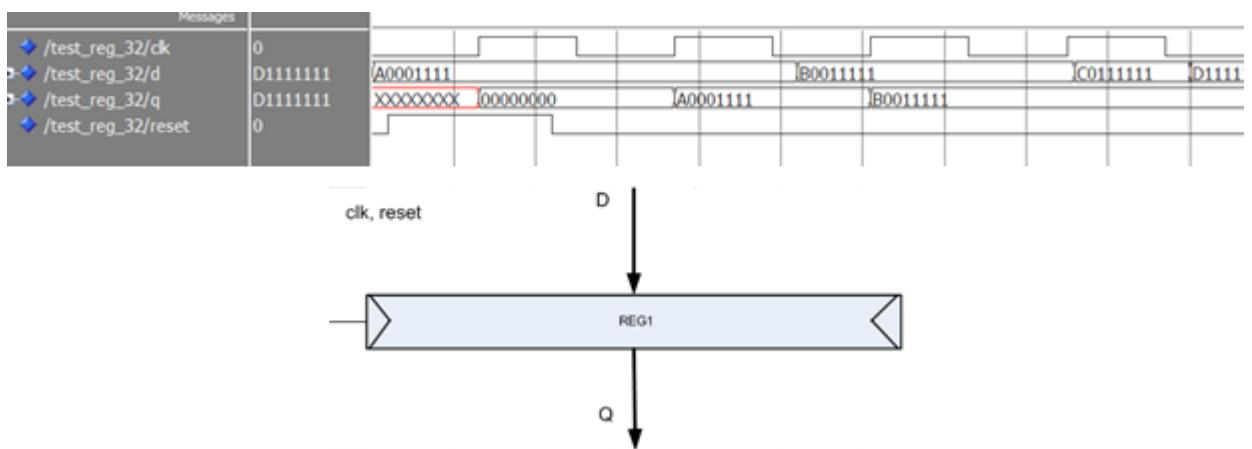
Hình 3-5. Sơ đồ khối chuỗi bit nhớ dùng CLA

Lưu ý là sơ đồ trên là sơ đồ của chuỗi bit nhớ, còn để hiện thực hóa khối cộng thì cần phải thêm các thành phần để tính giá trị của các bit tổng (SUM), người đọc có thể tự tìm hiểu để hoàn thiện bước này.

#### 1.4. Thanh ghi

Thanh ghi là chuỗi các phần tử nhớ được ghép với nhau và là thành phần không thể thiếu của các thiết kế mạch dãy, đặc điểm quan trọng nhất để phân biệt thanh ghi với các khối tổ hợp là thanh ghi bao giờ cũng chịu sự điều khiển của xung nhịp đồng bộ, giá trị đầu ra là giá trị lưu trong các ô nhớ của thanh ghi được gán bằng giá trị của đầu vào tại các thời điểm nhất định (sườn dương hoặc sườn âm) theo điều khiển xung nhịp đồng bộ, nếu so sánh với khối tổ hợp thì giá trị đầu ra của mạch tổ hợp thay đổi tức thì ngay sau khi có sự thay đổi của các đầu vào.

Thường gặp và phổ biến nhất là các thanh ghi sử dụng D-flipflop và làm việc đồng bộ theo sườn dương của xung nhịp hệ thống. Giản đồ sóng và biểu diễn của thanh ghi thể hiện ở hình dưới đây:



Hình 3-6. Sơ đồ khối và giản đồ sóng của thanh ghi

Như quan sát trên giản đồ sóng, giá trị đầu ra Q thay đổi chỉ tại các thời điểm có sườn dương của tín hiệu clk, tại thời điểm đó giá trị của Q sẽ được gán bằng giá trị đầu vào D của thanh ghi. Tại các thời điểm khác giá trị của Q được giữ không đổi. Mô tả thanh ghi trên VHDL khá đơn giản như sau:

```

----- register 32-bit -----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity 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 reg_32;
-----
architecture behavioral of reg_32 is
begin
    reg_p: process (CLK, RESET)
    begin
        if RESET = '1' then
            Q <= (others => '0');
        elsif CLK = '1' and CLK'event then
            Q <= D;
        end if;
    end process reg_p;
end behavioral;
-----
Cấu trúc

```

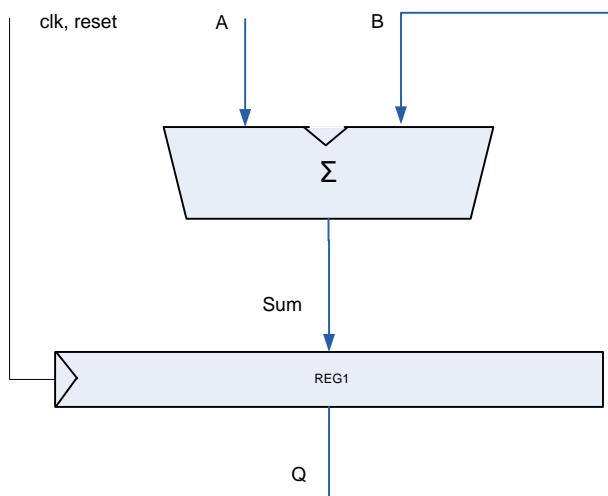
```

if CLK = '1' and CLK'event then...
    quy định thanh ghi làm việc theo tín hiệu sườn dương của xung nhịp clk, một
cách viết khác tương đương là
    if rising_edge(clk) then...

```

### 1.5. Bộ cộng tích lũy

Bộ cộng tích lũy là sự kết hợp giữa bộ cộng và thanh ghi, cấu trúc của khối này thể hiện ở hình dưới đây:



Hình 3-7. Sơ đồ khối bộ cộng tích lũy

Đầu ra của bộ cộng được nối với đầu vào của thanh ghi, còn đầu ra của thanh ghi được dẫn vào cổng B của bộ cộng, sau mỗi xung nhịp đồng hồ giá trị này được cộng thêm giá trị ở cổng A và lưu lại vào thanh ghi. Với mô tả của bộ cộng và thanh ghi ở trên, mô tả của bộ cộng tích lũy như sau:

```

-----accumulator-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-----
entity accumulator_32 is
port(
    A      : in      std_logic_vector(31 downto 0);
    Q      : buffer   std_logic_vector(31 downto 0);
    CLK    : in      std_logic;
    RESET : in      std_logic
);
end accumulator_32;
-----
```

```

architecture structure of accumulator_32 is
signal sum32 : std_logic_vector(31 downto 0);
signal Q_sig : std_logic_vector(31 downto 0);
signal Cout : std_logic;
signal Cin : std_logic;

----COMPONENT ADD_32----
component adder32 is

port (
    Cin: 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 component;

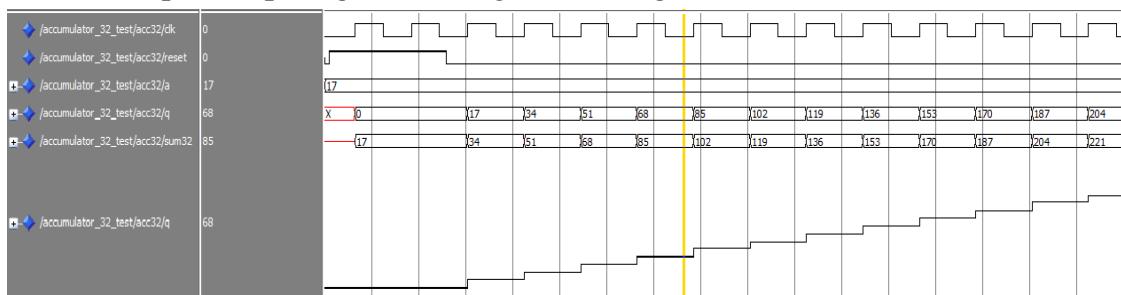
----COMPONENT REG_32----
component reg_32 is
port ( D : in std_logic_vector(31 downto 0);
       Q : out std_logic_vector(31 downto 0);
       CLK : in std_logic;
       RESET: in std_logic
);
end component;

begin
    Q_sig <= Q;
    Cin <= '0';
    add32: component adder32
        port map (Cin, A, Q_sig, sum32, Cout);
    reg32: component reg_32
        port map (sum32, Q, CLK, RESET);

end structure;

```

Kết quả mô phỏng thu được giản đồ sóng như sau:



Hình 3-8. Kết quả mô phỏng bộ cộng tích lũy

Sau xung nhịp reset giá trị q của thanh ghi bằng 0, sau đó cứ mỗi xung nhịp giá trị này tăng thêm 1, bằng giá trị đầu vào của A. Quan sát trên giản đồ sóng cũng dễ dàng nhận thấy giá trị tại đầu ra q của thanh ghi bao giờ cũng chậm hơn giá trị đầu vào sum của thanh ghi một xung nhịp clk.

## 1.6. Bộ đếm

Bộ đếm là một trường hợp đặc biệt của bộ cộng tích lũy, nếu ta cho đầu vào của bộ cộng A luôn nhận giá trị bằng 1 thì sau mỗi xung nhịp giá trị trong thanh ghi tăng thêm 1. Trong trường hợp đếm ngược thì cho giá trị của A bằng -1. Giá trị đếm là giá trị lưu trong thanh ghi còn xung đếm chính là xung nhịp hệ thống. Cách mô tả bộ đếm trên VHDL như sau:

```
-----Counter-----
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-----
entity counter4 is
    port (
        count  :out std_logic_vector( 3 downto 0 );
        enable :in  std_logic;
        clk     :in  std_logic;  -- Dau vao xung dem clock
        reset   :in  std_logic
    );
end entity;
-----
architecture rtl of counter4 is
signal cnt :std_logic_vector ( 3 downto 0 ) := "0000";
begin
    process (clk, reset) begin
        if (reset = '1') then
            cnt <= "0000";
        elsif (rising_edge(clk)) then
            if (enable = '1') then
                cnt <= cnt + 1;
            end if;
        end if;
    end process;
    count <= cnt;
end architecture;
```

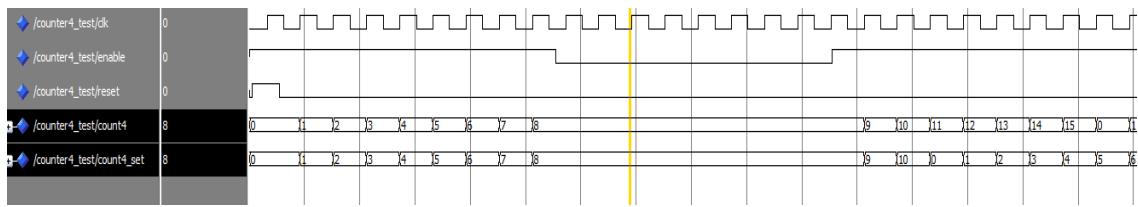
Trong đoạn mã trên tín hiệu reset được mô tả ở chế độ không đồng bộ, nghĩa là khi reset = 1 thì ngay lập tức giá trị đếm cnt bị reset về 0. Trong trường

hợp đồng bộ thì giá trị đếm bị reset chỉ tại sườn dương của xung nhịp clk. Ngoài tín hiệu reset, bộ đếm còn được điều khiển bởi enable, nếu enable =1 thì bộ đếm làm việc, nếu enable = 0 thì giá trị đếm được giữ nguyên.

Bộ đếm trên ở chế độ đếm sẽ đếm nhị phân với  $K_d = 2^i$ , các giá trị đếm thay đổi từ từ 0 đến 15 sau đó lại bắt đầu đếm từ 0, trong trường hợp muốn đặt  $K_d$  bằng một giá trị khác cần thêm một khối tổ hợp làm nhiệm vụ so sánh giá trị đếm với  $K_d$  để đặt lại giá trị đếm như ở mô tả dưới đây, bộ đếm với  $K_d = 10$ :

```
-----Counter set-----
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-----
entity counter4_set is
    port (
        count    :out std_logic_vector( 3 downto 0);
        enable   :in  std_logic;
        clk      :in  std_logic;
        reset    :in  std_logic
    );
end entity;
-----
architecture rtl of counter4_set is
    signal cnt :std_logic_vector ( 3 downto 0) :=
"0000";
begin
    begin
        process (clk, reset) begin
            if (reset = '1') then
                cnt <= "0000";
            elsif (rising_edge(clk)) then
                if (enable = '1') then
                    if cnt = "1010" then --cnt = 10-reset
                        cnt <= "0000";
                    else
                        cnt <= cnt + 1;
                    end if;
                end if;
            end if;
        end process;
        count <= cnt;
    end architecture;
```

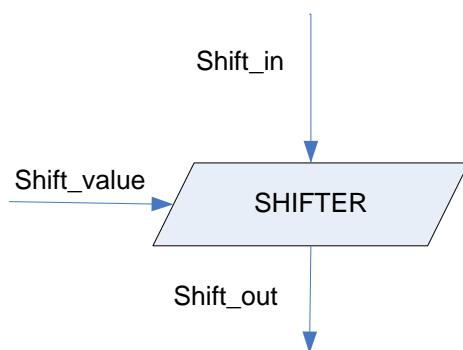
Kết quả mô phỏng của hai bộ đếm như sau:



Hình 3-9. Kết quả mô phỏng các bộ đếm

Các giá trị đếm bị reset đồng bộ về 0, sau đó đếm lần lượt từ 0 đến 8, tín hiệu enable sau đó bằng 0, nên giá trị này giữ nguyên, khi enable =1 các bộ đếm tiếp tục đếm, bộ đếm đặt lại trạng thái (*count4\_set*) chỉ đếm đến 10 rồi quay lại đếm từ 0, trong khi bộ đếm bình thường (*count2*) đếm đến 15.

### 1.7. Bộ dịch



Hình 3-10. Sơ đồ khối dịch

Bộ dịch là khối logic tổ hợp thực hiện động tác dịch chuỗi bít đầu vào, có tất cả 6 phép toán dịch gồm dịch trái logic, dịch trái số học, dịch phải logic, dịch phải số học, dịch tròn trái, dịch tròn phải, chi tiết về các lệnh dịch này xem trong mục 5.3 của chương 2.

Đầu vào của khối dịch gồm chuỗi nhị phân cần phải dịch *shift\_in* và giá trị số lượng bit cần phải dịch *shift\_value*. Đầu ra là giá trị chuỗi nhị phân sau khi thực hiện dịch. Khi viết mã cho bộ dịch lưu ý nếu dùng các toán tử dịch của VHDL thì các chuỗi nhị phân dịch phải được khai báo dưới dạng *bit\_vector*. Ví dụ dưới đây là một bộ dịch với đầu vào 32-bit:

```

-----
----- SHIFTER -----
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;
  
```

```

USE ieee.Numeric_BIT.all;
-----
entity 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 shifter_32;
-----
architecture behavioral of shifter_32 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 sll sa;
    shift_out <= TO_STDLOGICVECTOR(sho);
end behavioral;

```

Ở ví dụ trên vì đầu vào dịch shift\_in là giá trị 32-bit nên số bít dịch tối đa là 31, biểu diễn dưới dạng nhị phân cần 5-bit nên shift\_value được khai báo là std\_logic\_vector(4 downto 0). Kết quả mô phỏng như sau:

+◆ /test_shift_32/sh32/shift_in	00011001000100000111000100010001	U...00011001000100000111000100010001
+◆ /test_shift_32/sh32/shift_value	3	3
+◆ /test_shift_32/sh32/shift_out	11001000100000111000100010001000	00...11001000100000111000100010001000

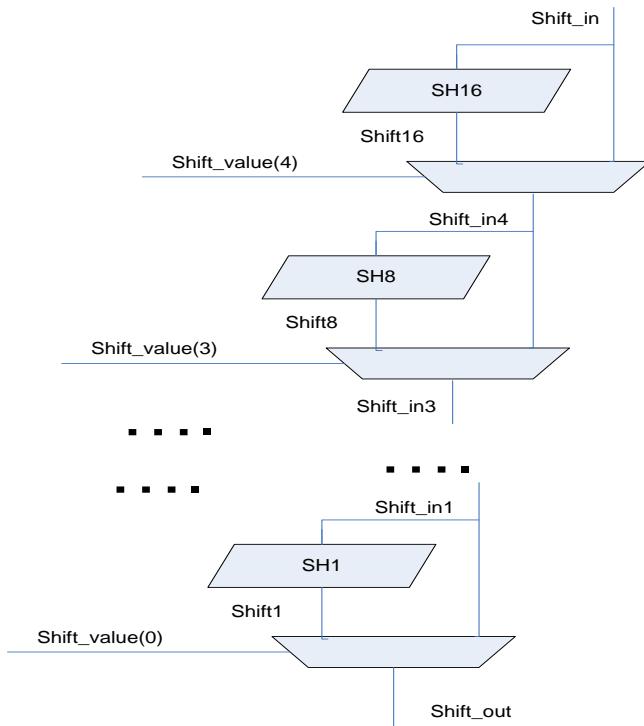
Hình 3-11. Kết quả mô phỏng khối dịch tổ hợp

Tương ứng với phần mô tả, giá trị shift\_out bằng shift\_in được dịch logic sang bên trái 3-bit.

Phương pháp sử dụng trực tiếp toán tử dịch của VHDL không được một số trình tổng hợp hỗ trợ, nghĩa là không tổng hợp được mạch, trong trường hợp đó khối dịch phải được viết chi tiết hơn. Nhận xét rằng độ phức tạp của khối dịch trên nằm ở chỗ giá trị dịch là không xác định, nếu giá trị dịch xác định thì phép dịch có thể thực hiện hết sức dễ dàng bằng toán tử hợp &. Ví dụ để dịch chuỗi bit đi 4 bit logic sang phải

```
shift_out = "0000" & shift_in(31 downto 4);
```

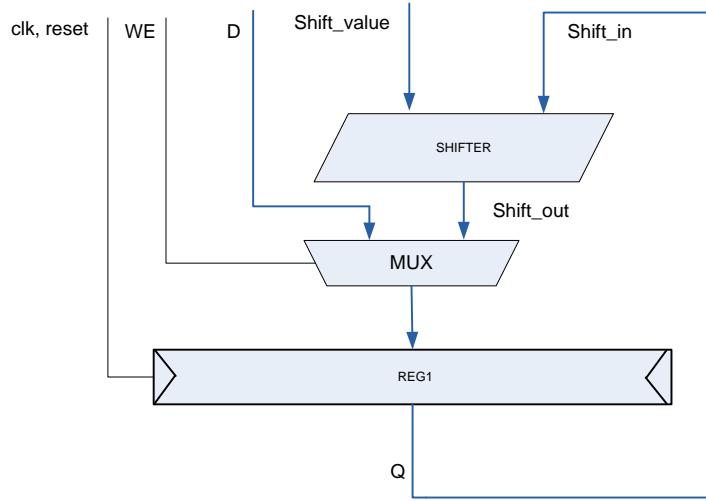
Từ đó có thể xây dựng khối dịch bằng sơ đồ thuật toán đơn giản như sau. Giả sử ta cần thiết kế khối dịch cho dữ liệu dịch shift\_in 32 bit, giá trị dịch shift\_value được biểu diễn là 5 bit. Các bit của shift\_value từ cao nhất tới thấp nhất sẽ được xét. Ví dụ, với bit đầu tiên shift\_value(4) được đưa vào làm tín hiệu điều khiển cho khối chọn kênh thứ nhất, nếu shift\_value(4) = 1 giá trị được chọn sẽ là đầu vào shift\_in được dịch đi 16 bit bởi bộ dịch SH16, nếu shift\_value(4) = 0 thì giá trị shift\_in được chọn, nghĩa là đầu vào không bị dịch đi. Sơ đồ cứ tiếp tục như vậy cho đến bit cuối cùng shift\_value(0), đầu ra của khối chọn kênh cuối cùng chính là giá trị shift\_out.



Hình 3-12. Sơ đồ thuật toán khói dịch đơn giản

### 1.8. Thanh ghi dịch

Tương tự như trường hợp của khói cộng tích lũy, kết hợp khói 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.

Mã mô tả thanh ghi dịch như sau:

```

----- 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);

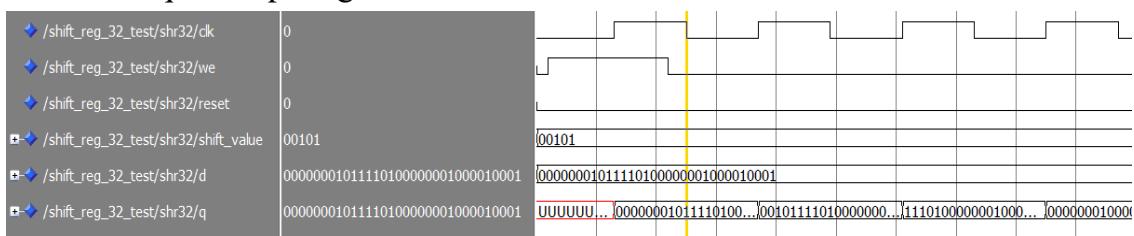
```

```

        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;
-----
```

Kết quả mô phỏng ra như sau:



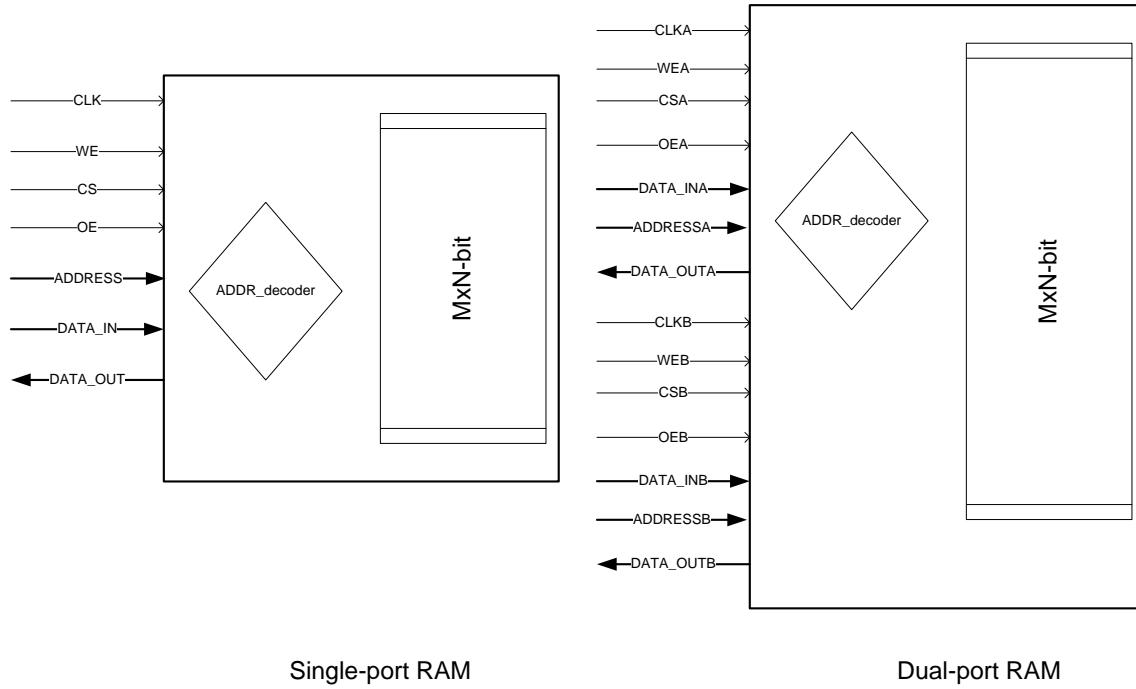
Hình 3-14. Kết quả mô phỏng thanh ghi dịch

Khi tín hiệu WE bằng 1 (mức tích cực cao) giá trị thanh ghi được gán bằng giá trị của cổng D, su đó E chuyển về thấp, giá trị trong thanh ghi Q được dịch sang trái mỗi lần shift\_value = “00101” = 5 bit.

## 2. Các khối nhớ

### 2.1. Bộ nhớ RAM

RAM (*Random Access Memory*) là một phần tử rất hay được sử dụng trong thiết kế các hệ thống số. RAM có thể phân loại theo số lượng cổng và cách thức làm việc đồng bộ hay không đồng bộ của các thao tác đọc và ghi.



Hình 3-15. Sơ đồ khối *Single-port RAM* và *Dual-port RAM*

- *Single port RAM* là RAM chỉ có một kênh đọc và ghi, một đường vào địa chỉ, các động tác đọc ghi trên kênh này chỉ có thể thực hiện lần lượt.

- *Dual-port RAM* là RAM có hai kênh đọc ghi riêng biệt tương ứng hai kênh địa chỉ, các kênh đọc ghi này có thể dùng chung xung nhịp đồng bộ cũng có thể không dùng chung. Đôi với Dual-port RAM có thể đọc và ghi đồng thời trên hai kênh.

- *Synchronous RAM* - RAM đồng bộ là RAM thực hiện thao tác đọc hoặc ghi đồng bộ.

- *Asynchronous RAM* - RAM không đồng bộ là RAM thực hiện thao tác đọc hoặc ghi không đồng bộ, thời gian kể từ khi có các tín hiệu yêu cầu đọc ghi cho tới khi thao tác thực hiện xong thuần túy là trễ tổ hợp.

Kết hợp cả hai đặc điểm trên có thể tạo thành nhiều kiểu RAM khác nhau, ví dụ *single-port RAM synchronous READ synchronous WRITE* nghĩa là RAM một cổng đọc ghi đồng bộ, hay *Dual-port RAM synchronous WRITE*

*asynchronous READ* là RAM hai cổng ghi đồng bộ đọc không đồng bộ... Khối RAM được cấu thành từ hai bộ phận là khối giải mã địa chỉ và dãy các thanh ghi, khối giải mã địa chỉ sẽ đọc địa chỉ và quyết định sẽ truy cập tới vị trí thanh ghi nào để thực hiện thao tác đọc hoặc ghi. Kích thước của khối RAM thường được ký hiệu là  $M \times N$ -bit trong đó  $M$  là số lượng thanh ghi,  $N$  là số bit trên 1 thanh ghi. Ví dụ  $128 \times 8$  bit là khối RAM gồm 128 thanh ghi, mỗi thanh ghi 8 bit. Số bit cần thiết cho kênh địa chỉ là  $\text{ADDR\_WIDTH} = [\log_2 M]$ , nếu  $M = 128$  nên số  $\text{ADDR\_WIDTH} = 7$ .

Mô tả VHDL một khối *single-port RAM synchronous READ/WRITE* ở dưới đây:

```

----- simple RAM unit -----
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
-----
entity simple_ram is
generic (
    DATA_WIDTH :integer := 8;
    ADDR_WIDTH :integer := 4
);
port(
    clk      :in  std_logic;
    address  :in  std_logic_vector (ADDR_WIDTH-1 downto 0);
    data_in  :in  std_logic_vector (DATA_WIDTH-1 downto 0);
    data_out :out std_logic_vector (DATA_WIDTH-1 downto 0);
    cs       :in  std_logic; -- Chip Select
    we       :in  std_logic; -- we = 1 write, we = 0 read
    oe       :in  std_logic -- OutputEnable when reading
);
end entity;
-----
architecture rtl of simple_ram is
constant RAM_DEPTH :integer := 2**ADDR_WIDTH;

    type RAM is array (integer range <>)of
std_logic_vector (DATA_WIDTH-1 downto 0);
    signal mem : RAM (0 to RAM_DEPTH-1);
begin

    WRITTING:
    process (clk) begin
        if (rising_edge(clk)) then
            if (cs = '1' and we = '1') then

```

```

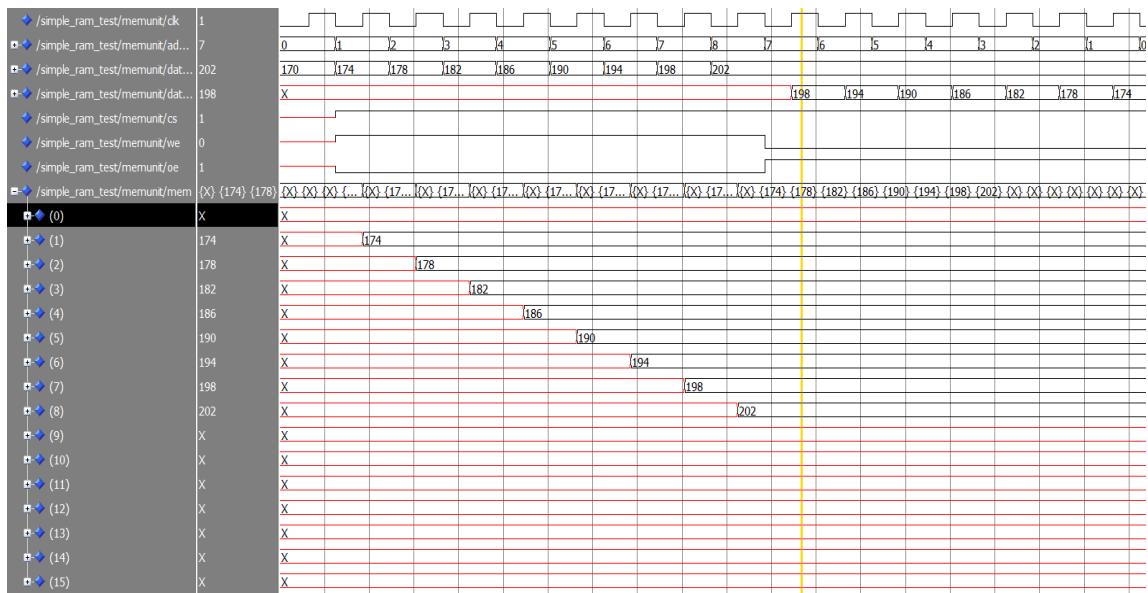
        mem(conv_integer(address)) <= data_in;
    end if;
end if;
end process;

READING:
process (clk) begin
    if (rising_edge(clk)) then
        if (cs = '1' and we = '0' and oe = '1')
then
            data_out <= mem(conv_integer(address));
        end if;
    end if;
end process;
end architecture;

```

Các tín hiệu điều khiển thao tác đối với RAM bao gồm CS – chip select, với mọi thao tác thì đều yêu cầu CS phải ở mức cao. WE – write enable bằng 1 nếu cần ghi dữ liệu vào RAM. Tín hiệu OE – output enable bằng 1 nếu là đọc dữ liệu từ RAM, với thiết kế như vậy thì WE và OE không bao giờ đồng thời bằng 1.

Hình dưới đây mô phỏng làm việc của một khối RAM.



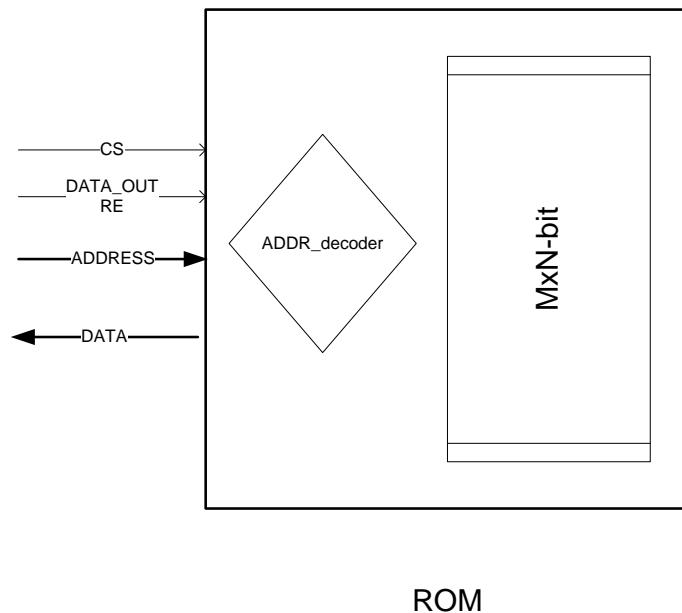
Hình 3-16. Kết quả mô phỏng khối RAM

Khi tín hiệu CS, WE ở mức tích cực cao, thực hiện động tác ghi vào RAM, mỗi xung nhịp ta ghi một giá trị bất kỳ, sau đó giá trị địa chỉ được tăng lên một đơn vị để thực hiện ghi vào ô nhớ kế tiếp. Cứ như thế khối RAM ta sẽ thực

hiện ghi 8 giá trị 174, 178, 182 ...202 vào các thanh ghi có địa chỉ tương ứng là 1,2,...8. Sau đó thực hiện động tác đọc giá trị từ RAM, tín hiệu CS giữ ở mức cao, WE bây giờ có mức thấp còn OE được đẩy lên mức cao, lần lượt thực hiện đọc giá trị từ thanh ghi thứ 7 đến thanh ghi thứ 1, mỗi lần đọc ta giảm giá trị địa chỉ đi một đơn vị.

## 2.2. Bộ nhớ ROM

ROM (*Read-only Memory*) là cấu trúc nhớ chỉ đọc trong đó các giá trị ô nhớ được lưu cố định khi khởi tạo ROM và không thay đổi trong quá trình sử dụng, thậm chí khi không có nguồn cấp giá trị trong ROM vẫn được giữ nguyên.



Hình 3-17. Sơ đồ khối ROM

Cấu trúc của ROM về cơ bản giống như của RAM, có một khối chứa dữ liệu cố định, không nhất thiết phải sử dụng xung nhịp CLK, khối giải mã địa chỉ. Các tín hiệu đầu vào là địa chỉ ADDRESS, tín hiệu CS – chip select, tín hiệu cho phép đọc RE – read enable. Đầu ra của khối ROM là giá trị DATA

Mô tả VHDL của khối ROM khá đơn giản nếu so sánh với khối RAM, thông thường khối ROM được mô tả dưới dạng một khối tổ hợp, đoạn mã dưới đây mô tả một khối ROM 16x8-bit

```
----- Simple ROM unit -----
library ieee;
use ieee.std_logic_1164.all;
-----
entity simple_rom is
```

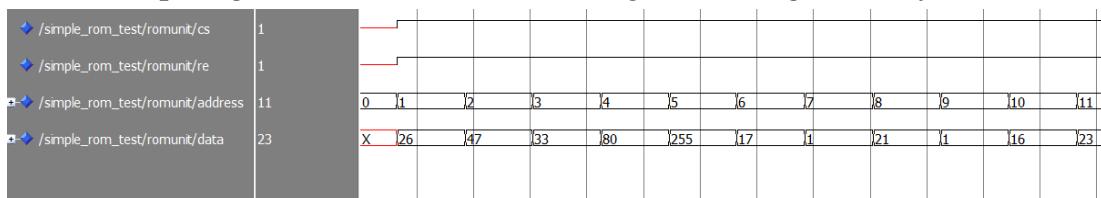
```

port (
    cs      :in std_logic;
    re :in std_logic;
    address :in std_logic_vector (3 downto 0);
    data    :out std_logic_vector (7 downto 0)
);
end entity;

-----
architecture behavioral of simple_rom is
-- Du lieu trong ROM duoc nap cac gia tri co dinh
begin
    process (re, cs, address) begin
        if re = '1' and cs = '1' then
            case (address) is
                when x"0"    => data <= x"00";
                when x"1"    => data <= x"1a";
                when x"2"    => data <= x"2f";
                when x"3"    => data <= x"21";
                when x"4"    => data <= x"50";
                when x"5"    => data <= x"ff";
                when x"6"    => data <= x"11";
                when x"7"    => data <= x"01";
                when x"8"    => data <= x"15";
                when x"9"    => data <= x"01";
                when x"A"    => data <= x"10";
                when x"B"    => data <= x"17";
                when x"C"    => data <= x"50";
                when x"D"    => data <= x"80";
                when x"E"    => data <= x"e0";
                when x"F"    => data <= x"f0";
                when others => data <= x"0f";
            end case;
        end if;
    end process;
end architecture;

```

Mô phỏng thao tác đọc dữ liệu như ở giản đồ sóng dưới đây:

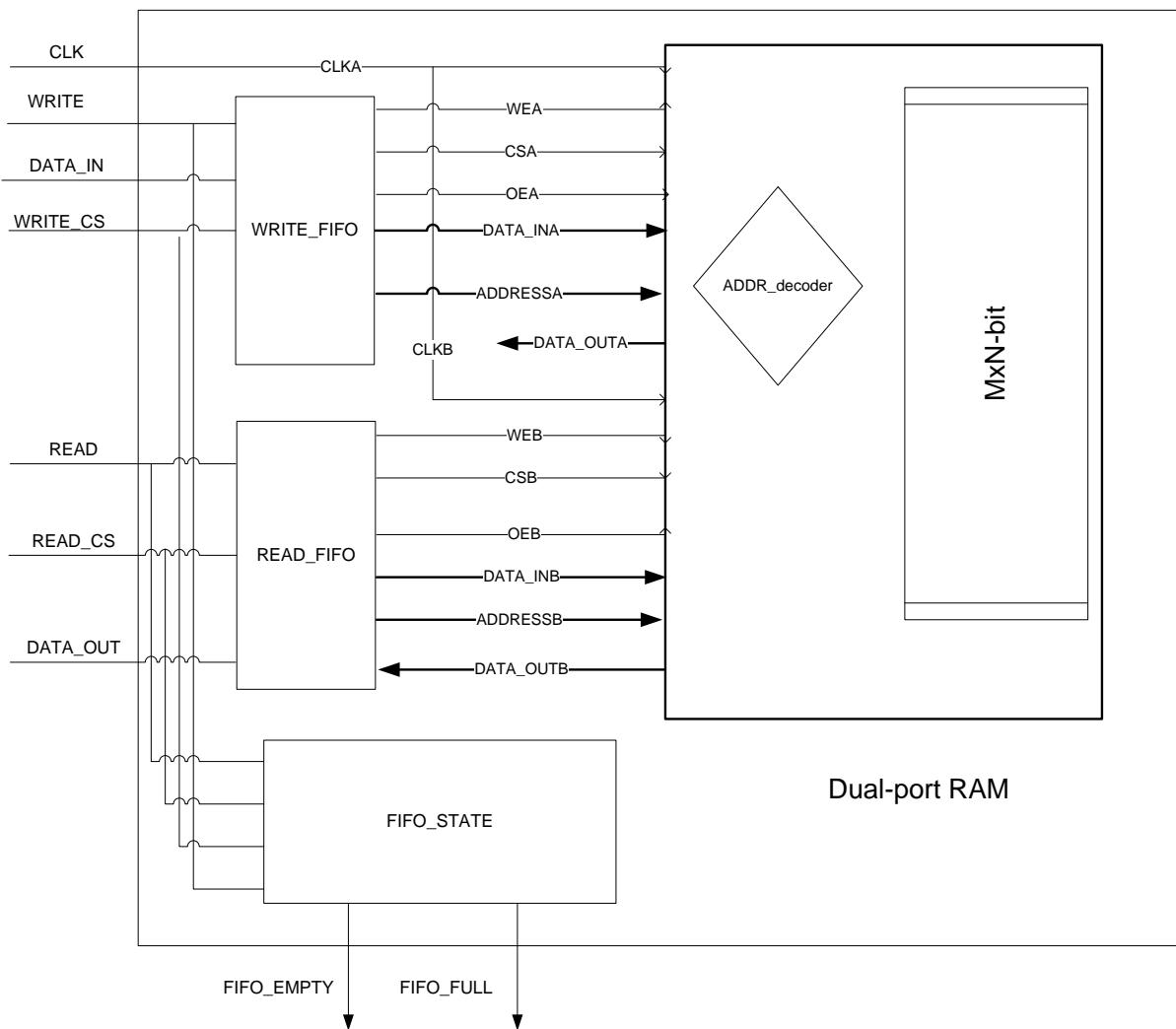


Hình 3-18. Kết quả mô phỏng khối ROM

Ở ví dụ trên khối ROM được đọc lần lượt các ô nhớ từ thứ 1 đến thứ 11, để thực hiện thao tác đọc thì tín hiệu CS và RE phải đồng thời để ở mức cao.

### 2.3. 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  $2^{\text{addr\_width}} - 1$

- 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  $2^{\text{addr\_width}} - 1$  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 (*FIFO\_EMPTY* = 0).

## 2.4. 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 trả tới vị trí giá trị N.

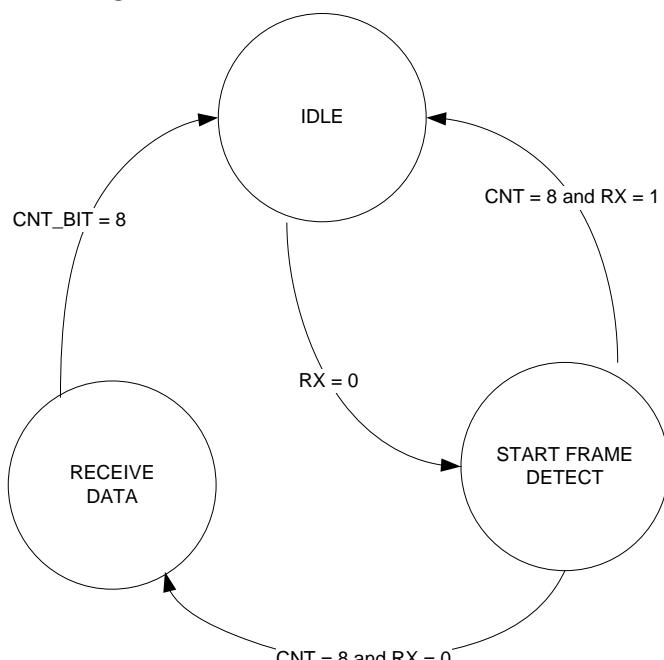
- Nếu có thao tác ghi dữ liệu mà không đọc thì dữ liệu được lưu vào ô có địa chỉ N và N tăng thêm 1.
- Nếu có thao tác đọc dữ liệu mà không ghi thì dữ liệu được đọc ở ô có địa chỉ N-1 và N giảm đi 1.
- Nếu có thao tác đọc, ghi dữ liệu đồng thời thì N không thay đổi và dữ liệu đọc chính là dữ liệu ghi.

Trạng thái của LIFO cũng được xác định thông qua giá trị của N, nếu  $N=0$  thì  $LIFO\_EMPTY = 1$  và nếu  $N = 2^{\text{addr\_width}} - 1$  thì  $LIFO\_FULL = 1$ .

### 3. Máy trạng thái hữu hạn

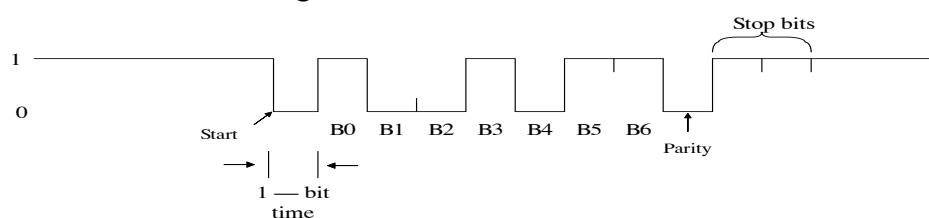
FSM (Finish-state machine) máy trạng thái hữu hạn là mạch có hữu hạn các trạng thái và hoạt động của mạch là sự chuyển đổi có điều kiện giữa các trạng thái đó.

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 thông 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 3-20. 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 3-21. 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 khói tín hiệu RX ở mức cao và khói nhận nằm ở trạng thái chờ IDLE. Khi RX chuyển từ mức cao xuống mức thấp thì khói nhận chuyển sang trạng thái thứ hai gọi là trạng thái dò tín hiệu start, START\_FRAME\_DETECT. Bít START được xem là hợp lệ nếu như mức 0 của RX được giữ trong một khoảng thời gian đủ lâu xác định, khói 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, khói nhận sẽ quay về trạng thái chờ IDLE. Ở trạng thái nhận dữ liệu thì khói 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;
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 bít 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 khôi điều khiển trạng thái bộ nhận UART phức tạp hơn do còn phần điều khiển trạng thái các bộ đếm, thanh ghi*

#### **4. Khối nhân số nguyên**

Phép nhân cũng là một phép toán rất hay sử dụng trong tính toán xử lý, việc thiết kế khối nhân phức tạp và cần nhiều tài nguyên hơn khối cộng, trên thực tế ở một số dòng vi xử lý đơn giản phép nhân được thực hiện thông qua khối cộng bằng phần mềm. Trong các vi xử lý hiện đại thì khối nhân được hỗ trợ phần cứng hoàn toàn. Dưới đây ta sẽ lần lượt nghiên cứu hai thuật toán cơ bản của khối nhân. Các thuật toán khác có thể tham khảo trong các tài liệu [11], [12]

#### 4.1. Khối nhân số nguyên không dấu dùng phương pháp cộng dịch

Khối nhân dùng thuật toán cộng dịch (shift-add multiplier), xét phép nhân hai số 4 bit không dấu như sau:

$$x \cdot a = x_0.a + 2.x_1.a + 2^2.x_2.a + 2^3.x_3.a \quad (3.4)$$

với  $x = x_3x_2x_1x_0$ ,  $b = b_3b_2b_1b_0$

0101	- số bị nhân	multiplicand
0111	- số nhân	multiplier
<hr/>		
0101	- tích riêng	partial products
0101		
0101		
0000		
<hr/>		
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 bít nhân thứ 0 là 0 bit, thứ 1 là 1 bít, thứ 2 là hai bit và thứ 3 là 3 bít. Kết quả nhân (*product*) thu được sau khi thực hiện phép cộng cho 4 tích riêng.

Sơ đồ trên giúp chúng ta hiểu phép nhân được thực hiện như thế nào nhưng khi xây dựng phần cứng dựa trên sơ đồ này có một nhược điểm là các tích riêng bị dịch bởi các giá trị dịch khác nhau. Nếu xây dựng khối nhân thuần túy là khối tổ hợp thì để nhân hai số 4 bit cần 4 khối dịch và 3 khối cộng 8 bit, nếu số lượng bit của các nhân tử tăng lên thì tài nguyên phần cứng tăng lên rất nhiều và không phù hợp với thực tế. Nếu xây dựng khối nhân theo dạng mạch tổ hợp kết hợp thanh ghi dịch và bộ cộng tích lũy thì cần thêm một bộ đếm để đếm giá trị dịch, việc này làm cho khối nhân trở nên phức tạp.

Giải pháp để đơn giản hóa cấu trúc của khối nhân có thể sử dụng một trong hai sơ đồ thuật toán cộng dịch trái và cộng dịch phải. Với sơ đồ cộng dịch phải phép nhân được thực hiện theo công thức sau:

$$\begin{aligned} x \cdot a &= x_0.a + 2.x_1.a + 2^2.x_2.a + 2^3.x_3.a \\ &= x_0.a + 2.(x_1.a + 2(x_2.a + 2.x_3.a)) \end{aligned} \quad (3.5)$$

### 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

```

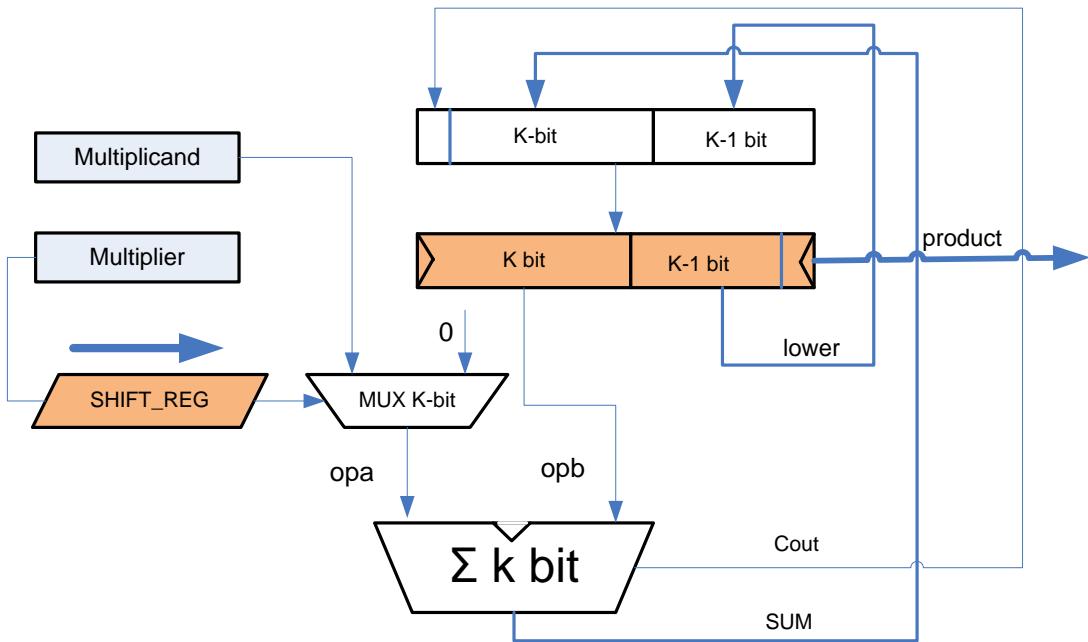
Với sơ đồ này thì số nhân được dịch từ trái qua phải, tức là số bị nhân sẽ được nhân lần lượt với các bit từ thấp đến cao  $x_0, x_1, x_2, x_3$ . Các giá trị  $p(i)$  là giá trị tích lũy của các tích riêng. Giá trị  $p(0)$  được khởi tạo bằng 0.  $P(1) = p(0) + x_0.a$ , đây là phép cộng 4 bit và  $p(1)$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

Sơ đồ hiện thực hóa khối nhân dùng thuật toán cộng dịch phải cho K bit sử dụng thanh ghi dịch và bộ cộng tích lũy như sau:



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

Khối cộng có một hạng tử 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 qua 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.

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

$$x \cdot a = x_0.a + 2.x_1.a + 2^2x_2.a + 2^3.x_3.a$$

$$= ((x_3.a.2 + x_2.a).2 + x_1.a).2 + x_0.a \quad (3.6)$$

Ví dụ số cho thuật toán cộng dịch trái:

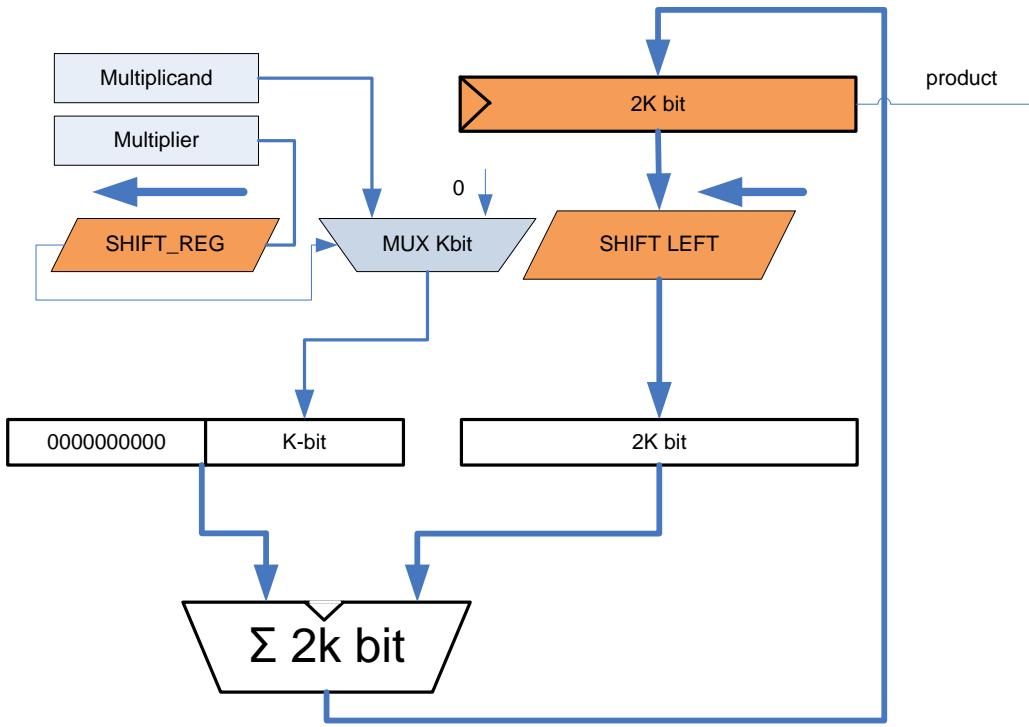
a	0	1	0	1
x	0	1	1	1
<hr/>				
p(0)	0	0	0	0
2 p(0)	0	0	0	0
+ x3.a	0	0	0	0
<hr/>				
p(1)	0	0	0	0
2 p(1)	0	0	0	0
+ x2.a	0	1	0	1
<hr/>				
2p(2)	0	0	0	1
p(2)	0	0	0	1
+ x2.a	0	1	0	1
<hr/>				
2p(3)	0	0	0	1
p(2)	0	0	0	1
+ x3.a	0	1	0	1
<hr/>				
P	0	0	1	0
	0	0	0	0
	1	1	1	1

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  $x3.a$  để thu được  $p(1)$  là một số 5 bit.  $p(1)$  tiếp tục được dịch trái 1 bit và cộng với  $x2.a$  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 bit bằng khối dịch trước khi đi vào khối cộng qua cổng opb. Giá trị cộng SUM được lưu trở lại vào thanh ghi trong xung nhịp kế tiếp

#### 4.2. Khối nhân số nguyên có dấu

Số nguyên có dấu trong máy tính được biểu diễn dưới dạng bù 2 ( $2s'$  complements) theo đó giá trị của số biểu diễn bằng một chuỗi bit nhị phân được tính theo công thức:

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

trong đó  $x_{n-1}x_{n-2}\dots x_1x_0$  là biểu diễn dưới dạng nhị phân.

Công thức trên đúng cho cả số âm lẫn số dương, nếu số là âm thì bit có trọng số lớn nhất  $x_{n-1}$  bằng 1 và ngược lại. Trong hệ thống biểu diễn này định nghĩa:

Bù 1 của số đó  $x_{n-1}x_{n-2}\dots x_1x_0$  là số nhận bởi các bit này nhưng lấy đảo

Bù 2 của số đó  $x_{n-1}x_{n-2}\dots x_1x_0$  là số bù 1 của số đó cộng thêm 1.

Một tính chất quan trọng của số bù 2 là số bù hai của một số A bằng số đối của số đó, hay nói cách khác  $b\bar{u}2(A) + A = 0$ . Tính chất này cho phép ta xây dựng khối nhân số có dấu bằng cách đơn giản nhất là đưa các số về dạng biểu diễn không âm, hay trị tuyệt đối của các số đó và nhân với nhau sử dụng một trong các sơ đồ cộng dịch trái hay cộng dịch phải ở trên. Riêng bit dấu của kết quả được tính riêng. Nếu như hai số nhân và số bị nhân khác dấu để đưa về dạng biểu diễn đúng của kết quả cần lấy bù 2 của kết quả phép nhân không dấu.

Cách thực hiện trên khá đơn giản tuy vậy đòi hỏi nhiều tài nguyên logic cũng như tăng độ trễ của khối nhân do phải bổ xung các khối tính bù 2 vào đầu và cuối chuỗi tính toán.

Ta lại có nhận xét rằng nếu biểu diễn giá trị a dưới dạng số bù hai bằng k bit, giá trị này được biểu diễn tương ứng bằng  $k+m$  bit bằng cách điền vào m bit mở rộng bên trái giá trị dấu (bit thứ  $k-1$ ) của số đó, toàn bộ k bit bên phải giữ nguyên. Động tác bổ xung các dấu như thế được gọi là mở rộng có dấu (signed-extend)

Ví dụ

$$-4 = (1100)_{4 \text{ bit}} = (11111100)_{8 \text{-bit}}$$

Với nhận xét này có thể thực hiện một vài điều chỉnh nhỏ trong sơ đồ nhân công dịch để sơ đồ này cũng đúng với số có dấu. Yêu cầu là khi mở rộng các kết quả tích lũy cần phải mở rộng theo kiểu có dấu (signed-extend), cụ thể hơn nếu bit mở rộng ở trường hợp nhân số không dấu là bit nhớ thì trong trường hợp này là bit dấu. Yêu cầu khác là khi số nhân là số âm thì nhân với bit dấu (bit cao nhất) thì theo công thức (2) kết quả thu được là số đối của số bị nhân hay là số bù 2 của số bị nhân.

Nhược điểm của phương pháp này là phải thiết kế hai khối nhân riêng cho số có dấu và không dấu.

Xét ví dụ về phép nhân của hai số có dấu 4 bit dùng thuật toán cộng dịch phải.

---

a	1 1 0 1	Bù 2 a = 0011 (a = -3)	
x	1 1 0 0	x = -4	

$P(0) \quad 0 \ 0 \ 0 \ 0$   
 $+x.a \quad 0 \ 0 \ 0 \ 0$

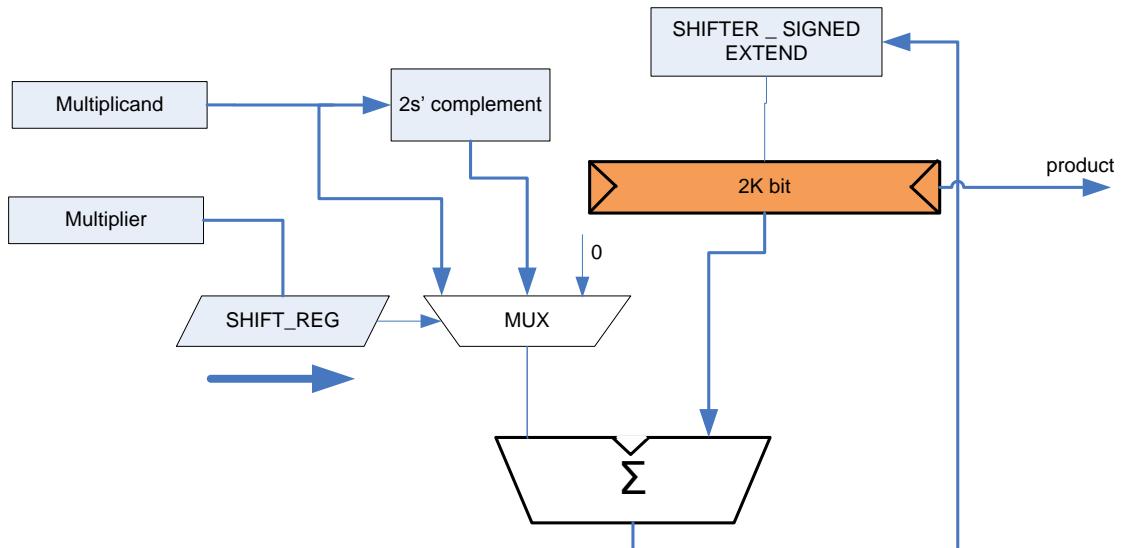
$-----$   
 $2p(1) \ 0 \ 0 \ 0 \ 0 \ 0 \quad$  vì  $p(1) \geq 0$  chèn thêm 0 vào trái  
 $P(1) \quad 0 \ 0 \ 0 \ 0 \ 0$   
 $+x1.a \quad 0 \ 0 \ 0 \ 0$

$-----$   
 $2p(2) \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \quad$  vì  $p(2) \geq 0$  chèn thêm 0 vào trái  
 $P(2) \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0$   
 $+x2.a \quad 1 \ 1 \ 0 \ 1$

$-----$   
 $2p(3) \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \quad$  vì  $p(2) < 0$  chèn thêm 1 vào trái  
 $P(3) \quad 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0$   
 $-x3.a \quad 0 \ 0 \ 1 \ 1$

$-----$   
 $P(4) \quad 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \quad$  vì  $p(2) \geq 0$  chèn thêm 0 vào trái  
 $P \quad 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \quad = +12$

Sơ đồ hiện thực khói nhân có dấu trên phần cứng như sau:



Hình 3-24. Sơ đồ hiện thực hóa thuật toán nhân số có dấu cho 2 số K-bit

Về cơ bản sơ đồ này không khác nhiều so với sơ đồ nhân không dấu, điểm khác biệt là ở đây trước khi kết quả tích lũy được lưu vào thanh ghi trung gian thì được dịch và chèn các bit dấu mở rộng theo quy tắc đối với số có dấu ở khối SHIFTER SIGNED-EXTEND. Và MUX thực hiện lựa chọn giữa 3 giá trị là số bị nhân, 0 và số đối của nó. Số đối của số bị nhân được chọn nếu bit nhân là bit dấu và bit này bằng 1.

Một phương pháp khác để thực hiện phép nhân với số có dấu biểu diễn dưới dạng bù 2 là sử dụng mã hóa Booth. 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:

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

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

$$b_i = (-x_{i+1} + x_i) \text{ với } i = -1, n-2, \text{ và } x_{-1} = 0.$$

Công thức trên được viết thành:

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

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:

Bảng 3-2

### 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

Theo bảng mã trên hai bit nhị phân tương ứng sẽ được mã hóa bằng các giá trị 0, 1, -1.

Ví dụ chuỗi bit và mã hóa Booth cơ số hai tương ứng:

2's complement 1 1 1 0 0 1 0 1 1 0 1 0 0 1 (0)  
Radix-2 booth 0 0-1 0 1-1 1 0-1 1-1 0 1-1

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.

Sơ đồ trên có thể sửa đổi một chút để nó vẫn đúng với 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.

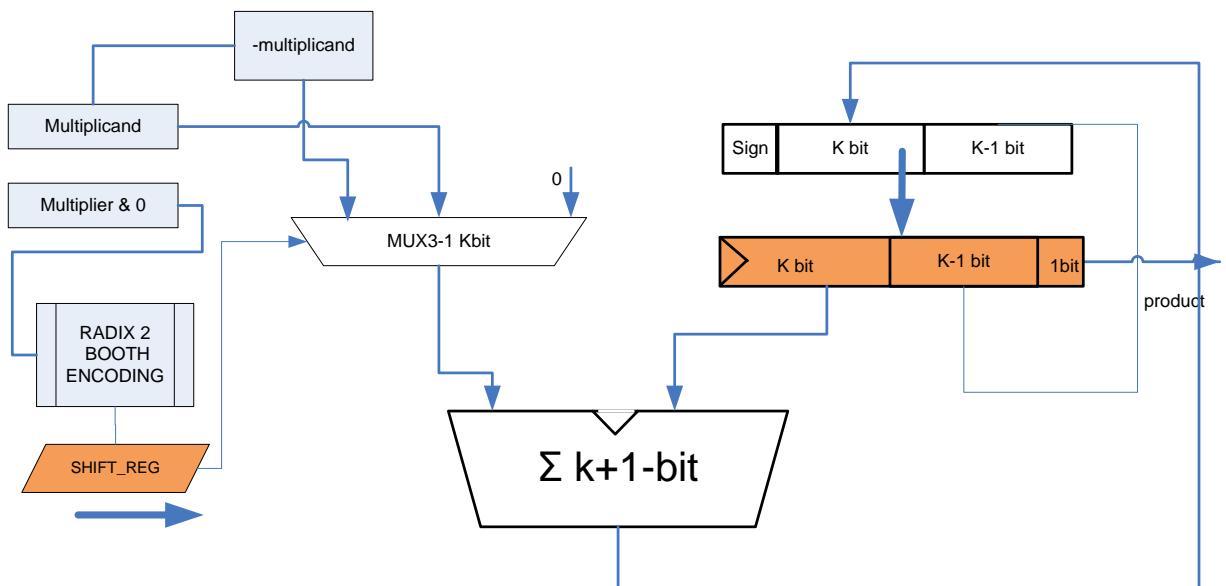
$2^r$ s complemnt (0) 1 1 1 0 0 1 0 1 1 0 1 0 0 1 (0)  
 Radix-2 Booth (1) 0 0-1 0 1-1 1 0-1 1-1 0 1-1

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
-----
P(0)   0 0 0 0
+b0.a  0 0 1 1

-----
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
-----
2p(2)  0 0 0 0 1 1       mở rộng với bit dấu 0
P(2)   0 0 0 0 1 1
+*2.a  0 0 0 0
-----
2p(2)  0 0 0 0 0 1 1       mở rộng với bit dấu 0
P(3)   0 0 0 0 0 1 1
+*3.a  1 1 0 1
-----
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 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 thanh ghi dịch  $2^*K$  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.

#### 4.3. Khối nhân dùng mã hóa Booth cơ số 4

Mã hóa Booth cơ số 2 không được ứng dụng thực tế nhưng là cơ sở giúp hiểu các dạng mã hóa Booth cao hơn. Trong nỗ lực tăng tốc cho phép nhân trong phần cứng một trong những phương pháp là thực hiện nhân không phải từng bit của số nhân mà nhân với một nhóm bit cùng một lúc.

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:

$$\begin{aligned}
 x_{2n-1}x_{2n-2}\dots x_1x_0 &= -2^{2n-1}x_{2n-1} + 2^{2n-2}x_{2n-2} + \dots + 2x_1 + x_0 \\
 &= -2^{2n-2}2.x_{2n-1} + 2^{2n-2}x_{2n-2} + 2^{2n-2}x_{2n-3} - 2^{2n-4}2.x_{2n-3} + 2^{2n-4}x_{2n-4} + 2^{2n-4}x_{2n-5} + \dots \\
 -2.2.x_1 + 2.x_0 + 2.0 &= 2^{2n-2}(-2x_{2n-1} + x_{2n-2} + x_{2n-3}) + 2^{2n-4}(-2x_{2n-3} + x_{2n-4} + x_{2n-5}) + \dots + (-2x_1 + x_0 + 0)
 \end{aligned} \tag{3.10}$$

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

$$b_i = (-2x_{2i+1} + x_{2i} + x_{2i-1}) \text{ với } i = 0, 1, 2, \dots, n-1 \tag{3.11}$$

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

Bảng 3-3

**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:

$$\begin{array}{r}
 a \quad \quad \quad 1 \ 1 \ 0 \ 1 \ 0 \ 1 \quad \quad \quad \text{Bù 2 } a = 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ (a = -11) \\
 \times \quad \quad \quad 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ (0) \quad \quad \quad x = + 26 \\
 b \quad \quad \quad 2 \ -1 \ -2 \quad \quad \quad \text{Radix 4 booth encoding of } x \\
 \hline
 4P(0) \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
 P(0) \quad \quad \quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
 +b0.a \quad \quad \quad 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \\
 \hline
 \end{array}$$

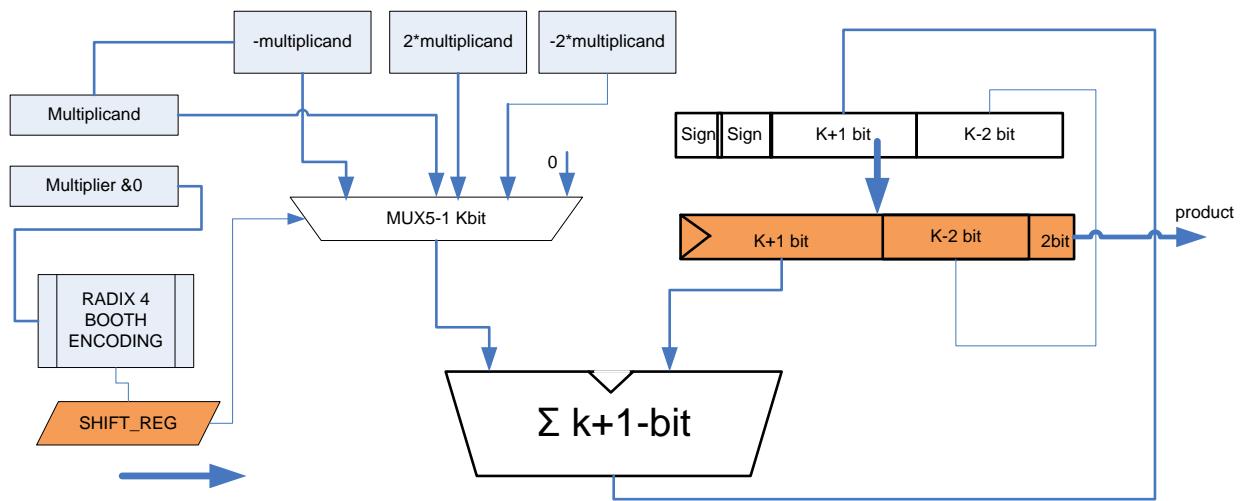
```

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 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 chưa 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 diền 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 chưa 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

Bằng cách tương tự có thể xây dựng sơ đồ nhân với mã hóa Booth cho các cơ số cao hơn.

## 5. Khối chia số nguyên

Khối chia là một khối số học khá phức tạp nếu so sánh với khối cộng hay khối nhân, trên thực tế các vi điều khiển và vi xử lý trước đây thường không hỗ trợ phép chia ở cấp độ phần cứng mà được thực hiện bằng phần mềm. Ở các vi xử lý hiện đại thì phép chia được hỗ trợ hoàn toàn, tuy vậy nếu so sánh về tốc độ thực hiện thì phép chia vẫn chậm hơn nhiều so với các phép toán khác, hay ngay cả đôi với các phép toán trên số thực. Thông thường phép nhân với số nguyên 32 bit dùng 4 xung nhịp Clk, còn phép chia dùng tới 35 xung nhịp.

Dưới đây ta sẽ nghiên cứu các thuật toán cơ bản để thực hiện phép chia trên phần cứng, đây chưa phải là những thuật toán nhanh nhất của phép chia nhưng là là những thuật toán cơ sở để người đọc nghiên cứu các thuật toán khác.

Ký hiệu

$z$ : số bị chia (*dividend*)

$d$ : số chia (*divisor*)

$q$ : thương số (*quotient*)

$s$ : số dư (*remainder*)

Sơ đồ thực hiện của phép chia số nhị phân tương tự như sơ đồ thực hiện phép chia đôi với số thập phân. Ví dụ dưới đây minh họa một phép chia số 8 bit cho số 4 bit không có dấu.

$$\begin{array}{r}
 z = 197 \\
 d = 14 \\
 \hline
 110000101 \\
 1110 \\
 \hline
 00001101 \\
 00000000 \\
 \hline
 1101
 \end{array}$$

```

-----
s (0)      1 1 0 0 0 1 0 1
2s (0)    1 1 0 0 0 1 0 1
-q3.2^4d   1 1 1 0
-----          q3 = 1

-----
s (1)      1 0 1 0 1 0 1
2s (1)    1 0 1 0 1 0 1
-q2.2^4d   1 1 1 0
-----          q2 = 1

-----
s (2)      0 1 1 1 0 1
2s (2)    0 1 1 1 0 1
-q1.2^4d   1 1 1 0
-----          q1 = 1

-----
s (3)      0 0 0 0 1
2s (3)    0 0 0 0 1
-q.0^4d    0 0 0 0
-----          q0 = 0

-----
s =        0 0 0 1 = 1           q = 1 1 1 0 = 14

```

Phép chia được thực hiện bằng cách dịch số chia sang bên phải 4 bit để vị trí bit có trọng số cao nhất trùng với bít có trọng số cao nhất của số bị chia. Tiếp đó khởi tạo số dư bằng số bị chia, nếu số này lớn hơn  $2^4.d$  thì bit thứ 4 tương ứng của thương số bằng  $q_3 = 1$ , Trong trường hợp ngược lại  $q_3 = 0$ . Thực hiện phép trừ số dư hiện tại cho  $q_3.2^4.d$ .

Kết quả phép trừ được dịch sang phải 1 bit để thu được số dư trung gian mới và lặp lại quá trình trên. Cho tới bước thứ 4, số dư ở bước cuối cùng thu được là số dư cần tìm. Thương số được ghép bởi các bit từ cao đến thấp từ bước 1 đến 4.

### 5.1. Khối chia dùng sơ đồ khôi phục phần dư

Khối chia trên phần cứng được xây dựng trên cơ sở nguyên lý như trên, điểm khác biệt là phép trừ được thay 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:

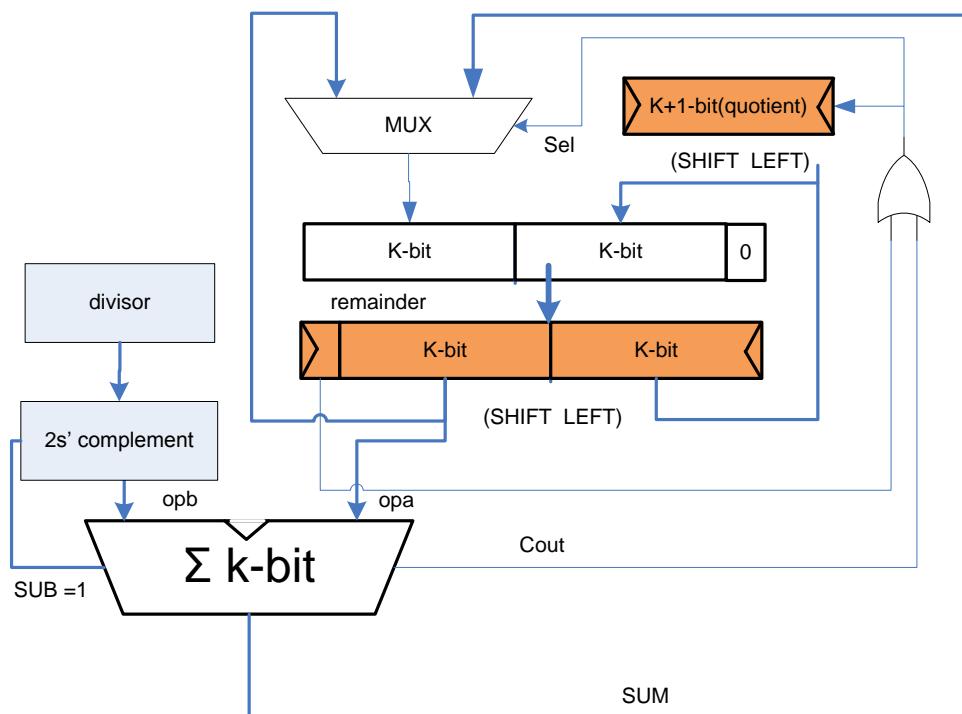
```

-----
z      1 1 1 0 0 1 0 1      z = 197
2^4d  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	
<hr/>		
s (0)	(0) 1 1 1 1 0 1 0 1	kết quả âm q4 = 0
2s (0)	1 1 0 0 0 1 0 1	phục hồi.
+ (-2^4d)	1 0 0 1 0	
<hr/>		
s (1)	(0) 1 0 1 0 1 0 1	kết quả dương q3 = 1
2s (1)	1 0 1 0 1 0 1	
+ (-2^4d)	1 0 0 1 0	
<hr/>		
s (2)	(0) 0 1 1 1 0 1	kết quả dương q2 = 1
2s (2)	0 1 1 1 0 1	
+ (-2^4d)	1 0 0 1 0	
<hr/>		
s (3)	(0) 0 0 0 0 1	kết quả dương q1 = 1
2s (3)	0 0 0 0 1	
+ (-2^4d)	1 0 0 1 0	
<hr/>		
s (4) =	(0) 0 0 1 1 = 1	kết quả âm q0 = 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 phần 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 dấu 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 cỗng logic OR hai đầu vào.

Thanh ghi đích 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.

## 5.2. Khối chia dùng sơ đồ không khôi phục phần dư

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 khôi 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  $s - 2^4d$ , 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  $2.s - 2^4d$ . Tuy vậy nếu lưu ý rằng  $2.s - 2^4d = 2(s - 2^4d) + 2^4d$ . 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  $2^4d$  ta sẽ cộng trực tiếp với  $2^4d$ , 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  $2^4d$ .

Nếu giá trị trong thanh ghi là âm  $\rightarrow$  thực hiện trừ với  $2^4d$ .

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

$$\begin{array}{r}
 z & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
 2^4d & 1 & 1 & 1 & 0 \\
 \hline
 s(0) & 0 & | & 0 & 0 & 1 & 0 & 0 & 1 \\
 2s(0) & 0 & | & 0 & 1 & 0 & 0 & | & 0 & 1 \\
 +(-2^4d) & 1 & | & 1 & 0 & 0 & 1 & 0 \\
 \hline
 s(1) & (0) & | & 1 & 1 & 0 & 1 & 0 & | & 0 & 1 & 0 & 1 & \text{kết quả âm } q4 = 0 \\
 2s(1) & 1 & | & 1 & 0 & 1 & 0 & 0 & | & 1 & 0 & 1 \\
 +(+2^4d) & 0 & | & 0 & 1 & 1 & 1 & 0 \\
 \hline
 s(2) & (1) & | & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & \text{kết quả dương } q3 = 1
 \end{array}$$

2s(2)	0   0 0 1 0 1 0 1	
+ (-2^4d)	1   1 0 0 1 0	
<hr/>		
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	
<hr/>		
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	
<hr/>		
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.		

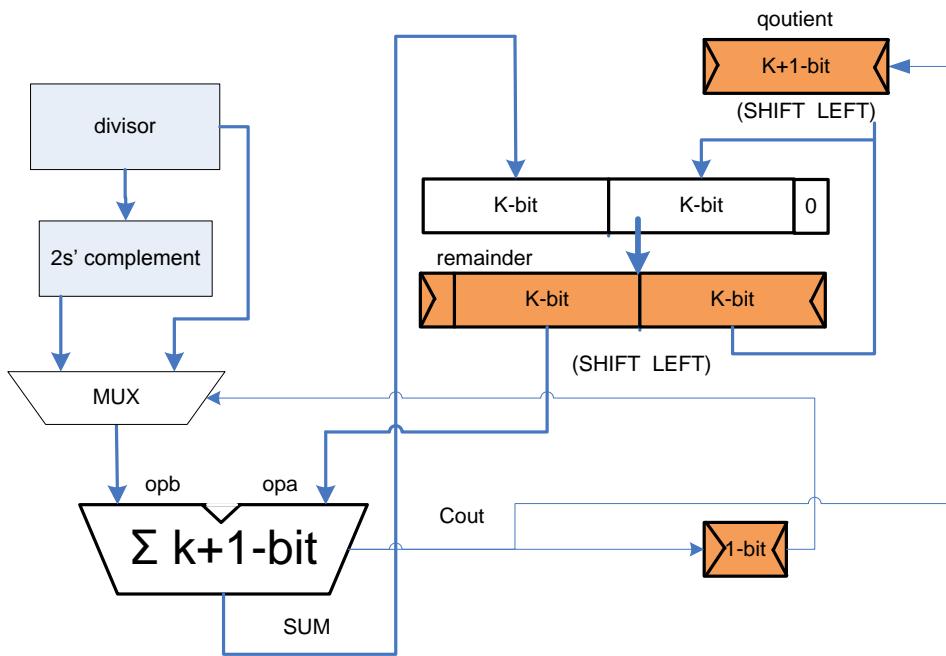
- Các bit tận cùng bên trái dấu | là bit dấu của các toán hạng, liệt kê chỉ để làm rõ giá trị cộng, các bit này luôn là 1,0 hoặc 0,1.

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 dấu của các hạng tử này luôn trái ngược nhau. Lợi dụng đặc điểm đó ta có thể lược bỏ giá trị bit dấu 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 dễ 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 khôi chọn kênh K bit thì phải thực hiện chọn hạng tử cho khối cộng giữa d và bù 2 của d cũng là một khôi chọn kênh K bit.

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



Hình 3-28. Sơ đồ khối chia không phục 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.

### 5.3. Khối chia số nguyên có dấu

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 công 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  $q_{k-1}q_{k-2}\dots q_0$ . Qi 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:

$$p_i = 1 \text{ nếu } s(i) \text{ và } d \text{ cùng dấu}$$

$$p_i = -1 \text{ nếu } s(i) \text{ và } d \text{ khác dấu.}$$

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, vẫn để 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  $p = p_k p_{k-1} \dots p_0$  với  $p_i = \{0, 1\}$  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  $q = (\bar{r}_{k-1}r_{k-2}\dots r_01)_{2's \text{ complement}}$

Có thể chứng minh như sau:

$$\begin{aligned} q &= (\bar{r}_{k-1}r_{k-2}\dots r_01)_{2's \text{ complement}} \\ &= -(1-p_{k-1}) \cdot 2^k + 1 + \sum_{i=0}^{k-2} r_i \cdot 2^{i+1} \\ &= -(2^k - 1) + 2 \cdot \sum_{i=0}^{k-1} r_i \cdot 2^i \\ &= \sum_{i=0}^{k-1} (2r_i - 1) \cdot 2^i = p \end{aligned}$$

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	$d = 6 = 0 1 1 0$
		$-d = -6 = 1 0 1 0$
<hr/>		
s(0)	1 1 1 0 1 0 1 0 1	d, s khác dấu q4 = -1
2s(0)	1 1 1 0 1 0 1 0 1	thực hiện cộng
$+ (2^4 d)$	0 0 1 1 0	
<hr/>		
s(1)	(1) 0 0 1 1 0 1 0 1	d, s cùng dấu q3 = +1
2s(1)	0 0 1 1 0 1 0 1	thực hiện trừ
$+ (-2^4 d)$	1 1 0 1 0	
<hr/>		
s(2)	(1) 0 0 0 0 1 0 1	d, s cùng dấu q2 = +1
2s(2)	0 0 0 0 1 0 1	thực hiện trừ
$+ (+2^4 d)$	1 1 0 1 0	
<hr/>		
s(3)	(0) 1 0 1 1 0 1	d, s khác dấu q1 = -1
2s(3)	1 0 1 1 0 1	thực hiện cộng
$+ (+2^4 d)$	0 0 1 1 0	
<hr/>		
S(4) =	(0) 1 1 0 0 1	s, q khác dấu q0 = -1
2S(4)	1 1 0 0 1	thực hiện cộng
$+ (-2^4 d)$	0 0 1 1 0	

S = 1 1 1 1 1 = -1 s, z cùng dấu không cần điều chỉnh số dư

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

$$q = (1 1 1 0 0 1)_2' s' complement = -7$$

- bit nằm trong dấu ngoặc là bit nhớ của khôi 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  
 $+ (2^4 d)$     0 0 1 1 0

d, s khác dấu       $q_4 = -1$   
thực hiện cộng

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

d, s cùng dấu       $q_3 = +1$   
thực hiện trừ

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

d, s cùng dấu       $q_2 = +1$   
thực hiện trừ

$s(3)$       (0) 1 1 0 0 0 1  
 $2s(3)$       1 1 0 0 0 1  
 $+ (+2^4 d)$    0 0 1 1 0

d, s khác dấu       $q_1 = -1$   
thực hiện cộng

$S(4) =$       (0) 1 1 1 0 1  
 $2S(4)$       1 1 1 0 1  
 $+ (-2^4 d)$    0 0 1 1 0

s, d khác dấu       $q_0 = -1$   
thực hiện cộng

$S =$       (1) 0 0 1 1      =      3      s, z khác dấu, điều chỉnh -d  
               1 1 0 1 0  
               (0) 1 1 0 1      =      -3

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

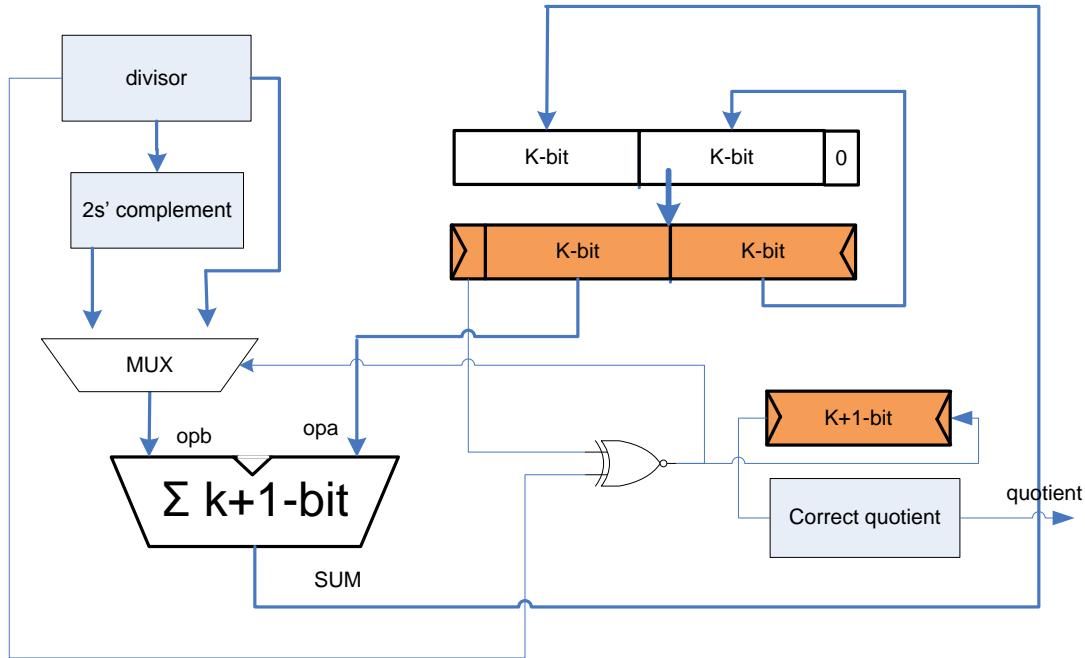
$q' = (1 1 1 0 0 1) 2^4$ 's complement = -7, do ở trên có điều chỉnh số dư bằng cách trừ d nên phải tăng q thêm 1

$q = q' + 1 = -6$ .

- 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 phần chi tiết phần cứng được thiết kế như sau:



Hình 3.29. Sơ đồ khói 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  $p_i$  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  $p_i$  bằng 1. Các giá trị này được đầy dần vào thanh ghi k-bit, giá trị  $r = r_{k-1}r_{k-2}\dots r_0$  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.

## 6. Các khối làm việc với số thực

Trong kỹ thuật tính toán có hai dạng định dạng cơ bản để biểu diễn số thực bằng chuỗi các bít nhị phân là dạng biểu diễn số thực dấu phẩy tĩnh (*fixed-point number*) và số thực dấu phẩy động (*floating-point number*).

### 6.1. Số thực dấu phẩy tĩnh

Số nguyên là một dạng đặc biệt của số thực dấu phẩy tĩnh khi mà dấu phẩy nằm ở vị trí 0, hay phần thập phân luôn bằng 0, khi đó việc thay đổi vị trí dấu phẩy sang phải k bit ta thu được giá trị của số mới bằng số cũ nhân với  $2^{-k}$  là một giá trị không nguyên. Một điểm nữa cần chú ý là tính chất trên vẫn đúng nếu như số nguyên của chúng ta là số có dấu biểu diễn dưới dạng số bù 2.

Như vậy định dạng số thực dấu phẩy tĩnh chỉ là một tổng quát của định dạng số nguyên chúng ta hay dùng. Các tham số của định dạng này gồm Fixed(s, w, f) trong đó s = 1 nếu là biểu diễn số có dấu, trong trường hợp đó số có dấu giá trị được biểu diễn dưới dạng bù 2, w là số lượng bit dùng cho biểu diễn phần nguyên và f là số lượng bit dùng để biểu diễn cho phần thập phân. Khi đó giá trị một số thực không dấu được tính toán theo công thức sau

$$\begin{aligned} r &= b_{w-1}b_{w-2}\dots b_0, b_{-1}b_{-2}\dots b_{-f} = 2^{-f} \left( \sum_{k=0}^{w+f-1} 2^k b_k \right) \\ &= \sum_{k=0}^{w-1} 2^k b_k + \sum_{k=1}^f 2^{-k} b_k \end{aligned} \quad (3.12)$$

Ví dụ số biểu diễn bằng số thực 8 bit không dấu với w = 4, f = 4 sẽ là

$$\begin{aligned} 1,5 &= 0001\ 1000 = 24 \cdot 2^{-4} \\ 3,75 &= 0011\ 1100 = 60 \cdot 2^{-4} \end{aligned}$$

Giá trị của một số thực có dấu với bit dấu là s được tính bằng

$$\begin{aligned} r &= s b_{wl-1}b_{wl-2}\dots b_0, b_{-1}b_{-2}\dots b_{-fl} = 2^{-f} \left( (-1)^s \cdot 2^{w+l} + \sum_{k=0}^{w+f-1} 2^k b_k \right) \\ &= (-1)^s \cdot 2^w + \sum_{k=0}^{w-1} 2^k b_k + \sum_{k=1}^f 2^{-k} b_k \end{aligned} \quad (3.13)$$

Ví dụ số biểu diễn bằng số thực 8 bit không dấu với w = 4, f = 4 sẽ là

$$\begin{aligned} +1,5 &= 0001\ 1000 = 24 \cdot 2^{-4} \\ -4,25 &= 1011\ 1100 = -68 \cdot 2^{-4} \end{aligned}$$

Như đã trình bày ở trên số nguyên là một trường hợp riêng của số thực dấu phẩy tĩnh, toàn bộ các khối logic số học làm việc với số nguyên như trình bày ở các phần trên đều có thể sử dụng cho các khối tính toán trên thực dấu

phẩy tĩnh. Đây là một lợi thế rất lớn của định dạng này, tuy vậy nhược điểm rất lớn của số thực dấu phẩy tĩnh là miền biểu diễn số nhỏ và độ phân giải, hay còn gọi là độ chính xác không cao. Chúng ta sẽ tìm hiểu về số thực dấu phẩy động trước khi đưa ra so sánh để thấy được nhược điểm này.

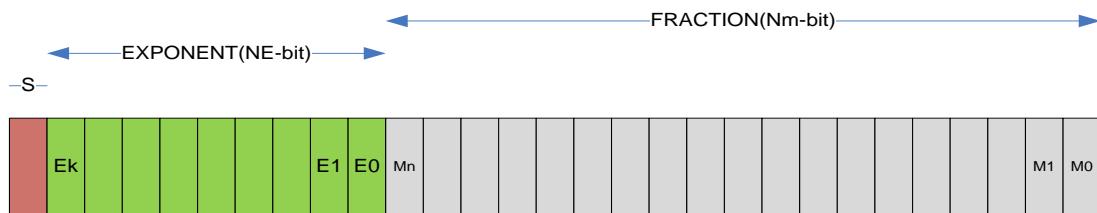
## 6.2. Số thực dấu phẩy động

Số thực dấu phẩy động được ứng dụng thực tế trong máy tính và các hệ xử lý. Để đảm bảo cho các thiết bị phần cứng cũng như các chương trình phần mềm có thể trao đổi dữ liệu và làm việc với nhau một cách chính xác khi thực hiện xử lý tính toán trên số thực dạng này, tổ chức IEEE đã nghiên cứu và xây dựng các chuẩn liên quan đến số thực dấu phẩy động (IEEE 754) cũng như quy định về các phép toán trên đó.

Trong khuôn khổ giáo trình sẽ giới thiệu về định dạng, các phép tính và một số quy tắc làm tròn cơ bản của số thực dạng này, trên cơ sở đó sẽ xây dựng thuật toán cho các khối thực hiện phép toán số thực.

### 6.2.1. Chuẩn số thực ANSI/IEEE-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:

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

K bit tiếp theo E<sub>k</sub>E<sub>k-1</sub>...E<sub>1</sub> 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  $e = \text{exponent} - (2^{k-1} - 1)$ , giá trị  $(2^{k-1} - 1)$  gọi là độ dịch của số mũ (*exponent bias*) có nghĩa là miền giá trị của số mũ từ  $-(2^{k-1} - 1)$  đến  $+(2^{k-1} - 2)$

N bít cuối cùng dùng để biểu diễn phần thập phân FRACTION với giá trị tương ứng là m = 1, m<sub>n</sub>m<sub>n-1</sub>...m<sub>1</sub>, Với số thực chuẩn thì số 1 trong công thức này

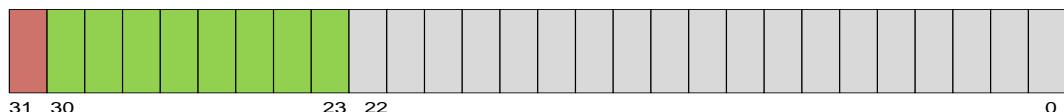
cũng không được biểu diễn trực tiếp mà ngầm định luôn luôn có nên còn gọi là bit ẩn.

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:

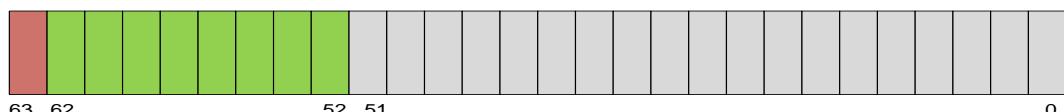
Denormalized	:	EXPONENT + BIAS = 0, FRACTION <> 0
Zero $\pm 0$	:	EXPONENT + BIAS = 0, FRACTION = 0
Infinity $\pm \infty$	:	EXPONENT + BIAS = $2^{k-1}-1$ , FRACTION = 0
Not a Number NaN	:	EXPONENT + BIAS = $2^{k-1}-1$ , FRACTION <> 0

Số không chuẩn (Denormalized) là những số có số mũ thực té bằng 0 ( $\text{EXPONENT} + \text{BIAS} = 0$ ) và số thập phân biểu diễn dưới dạng  $m = 0, m_n m_{n-1} \dots m_1$ , 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  $2^{-(2^{k-1}-1)}$

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.



IEEE - 754 Single Precision



IEEE - 754 Double Precision

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

Bảng thống kê thuộc tính của định dạng số thực dấu phẩy động theo chuẩn ANSI/IEEE 754 – 1985

Bảng 3-4

#### Các tham số của số thực dấu phẩy động chuẩn IEEE 754

Thuộc tính	Single Precision (Float)	Double Precision (Double)
Tổng số bit	32	64
Phản thập phân	$23 + 1$ ẩn	$52 + 1$ ẩn
Phản số mũ	8	11
Miền thập phân	$[1, 2 - 2^{-23}]$	$[1, 2 - 2^{-52}]$

Độ lệch số mũ (BIAS)	127	1023
Miền số mũ	[-127, +127]	[-1023, +1023]
Số không chuẩn (Denormalized)	EXPONENT = 0, FRACTION ≠ 0	EXPONENT = 0, FRACTION ≠ 0
Zero ( $\pm 0$ )	EXPONENT = 0, FRACTION = 0	EXPONENT = 0, FRACTION = 0
Vô cùng (Infinity ( $\pm \infty$ ))	EXPONENT = 255, FRACTION = 0	EXPONENT = 1023, FRACTION = 0
Không là số (Not a Number NaN)	EXPONENT = 255, FRACTION ≠ 0	EXPONENT = 1023, FRACTION ≠ 0
Số chuẩn (Normalized)	EXPONENT $\in (1 \div 254)$	EXPONENT $\in (1 \div 2046)$
Giá trị lớn nhất	$2^{-126}$	$2^{-1022}$
Giá trị nhỏ nhất	$(2 - 2^{-23}) \cdot 2^{127} \approx 2^{128}$	$(2 - 2^{-23}) \cdot 2^{1023} \approx 2^{1024}$

### 6.2.3. So sánh miền biểu diễn và độ phân giải của các dạng biểu diễn

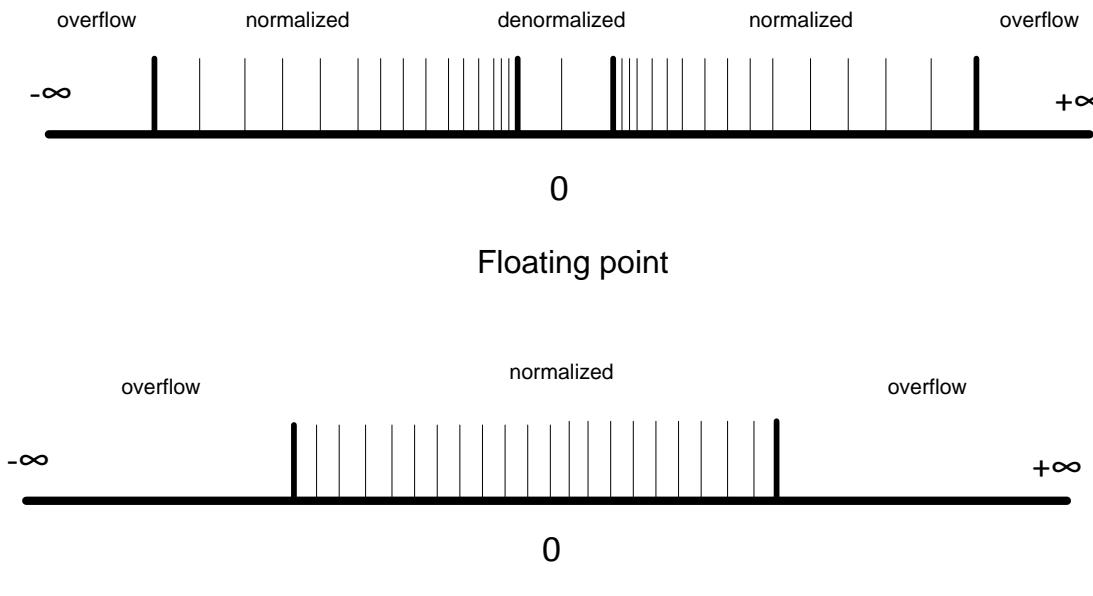
Bảng sau thể hiện các tham số của các dạng số thực:

Bảng 3-5

#### Độ phân giải và miền biểu diễn của các định dạng số thực

Dạng biểu diễn	Giá trị dương lớn nhất	Giá trị dương nhỏ nhất	Giá trị âm lớn nhất	Giá trị âm nhỏ nhất	Độ phân giải cao nhất
Fix(1, 23, 8)	$+2^{23} + 1 \cdot 2^{-8}$	$+2^{-8}$	$+2^{-8}$	$-(2^{23} + 1 \cdot 2^{-8})$	$2^{-8}$
Float32	$+2^{126} \cdot (2 - 2^{-8})$	$+2^{-127}$	$-2^{-127}$	$-2^{128} \cdot (2 - 2^{-8})$	$-2^{-127}$
Float64	$+2^{1022} \cdot (2 - 2^{-52})$	$+2^{-128}$	$-2^{-128}$	$-2^{1023} \cdot (2 - 2^{-52})$	$-2^{-1023}$

Từ bảng trên có thể thấy ưu điểm của biểu diễn số thực theo định dạng dấu phẩy động, thứ nhất là miền biểu diễn lớn hơn rất nhiều, nếu so sánh cùng biểu diễn bằng 32 bit thì miền biểu diễn của số thực dấu phẩy động lớn gấp  $\sim 2^{102}$  lần, còn về độ phân giải cao nhất thì gấp  $2^{119}$  lần. Tuy vậy lưu ý rằng số lượng tổ hợp các giá trị khác nhau biểu diễn bằng N-bit là cố định vì vậy số thực dấu phẩy động có miền biểu diễn lớn nhưng độ phân giải ở các miền giá trị không đồng đều, cao ở miền giá trị tuyệt đối nhỏ và thấp ở miền giá trị tuyệt đối lớn. Khác với số thực dấu phẩy tĩnh có phân bố đều nhưng độ phân giải thấp.



Hình 3-32. *Phân bố của các dạng biểu diễn*

Trên thực tế hầu hết các thiết bị tính toán số sử dụng định dạng dấu phẩy động được chuẩn hóa bởi IEEE (ANSI/IEEE-754), nhờ sự chuẩn hóa đó mà dữ liệu tính toán có thể được trao đổi sử dụng lẫn nhau mà không gây ra sai sót. Với các hệ thống tính toán đặc chủng phục vụ cho những bài toán hoặc lớp bài toán cụ thể thì người thiết kế thường tự định nghĩa lại dạng biểu diễn của số thực nhằm tối ưu hóa thiết kế về tất cả các mặt gồm độ chính xác, hiệu suất sử dụng phần cứng và tốc độ.

Về nhược điểm, số thực dấu phẩy động là các khối thiết kế thực hiện phép toán trên số dạng này phức tạp hơn rất nhiều với các khối làm việc với số thực dấu phẩy tĩnh. Ngoài ra việc kiểm soát sai số cho các phép toán bằng số thực dấu phẩy động thường phức tạp hơn do sự phân bố không đều trên trực số, trình tự tính toán một công thức có thể dẫn đến những sai số rất khác biệt. Trước khi đi vào thiết kế các khối tính toán ta sẽ xem xét về các chế độ làm tròn hỗ trợ bởi số thực dấu phẩy động và ảnh hưởng của nó tới sai số.

### 6.3. Chế độ làm tròn trong 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

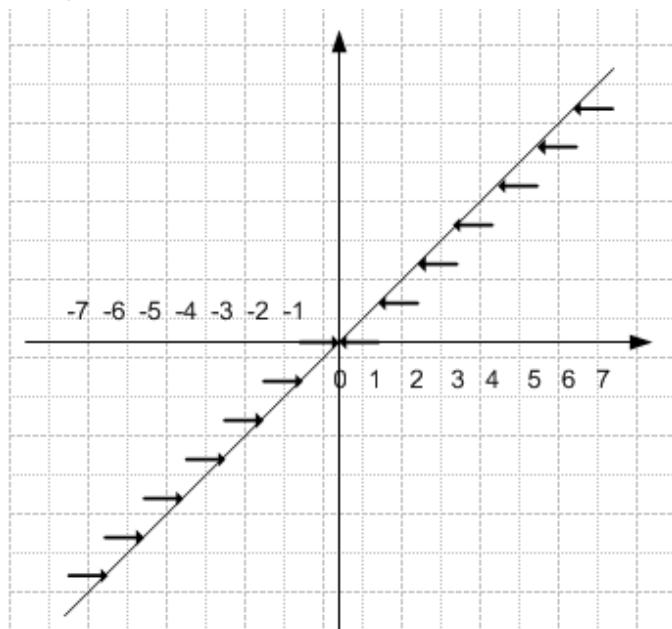
lượng bit lớn hơn so với bình thường. Làm tròn thường là động tác cuối cùng trong chuỗi các động tác thực hiện phép tính với số thực dấu phẩy động. Có nhiều phương pháp để làm tròn số, không mất tính tổng quát có thể xem chuỗi số đầu vào là một số thập phân có dạng:

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

Chuỗi số đầu ra là một số nguyên có dạng:

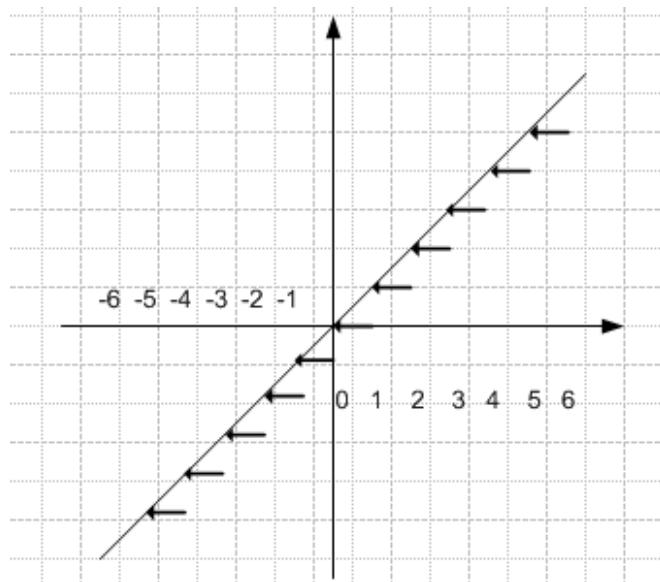
$$Z = z_{n-1}z_{n-2}\dots z_0$$

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)*.



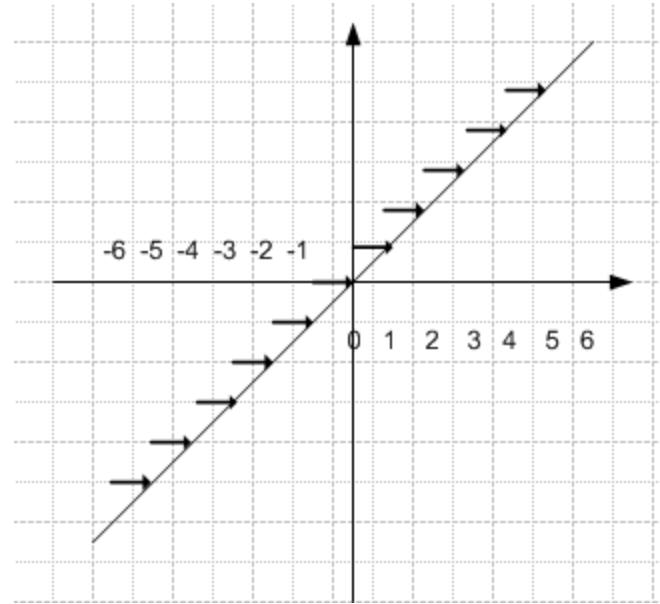
Hình 3-33.*Làm tròn hướng tới 0*

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  $-\infty$*

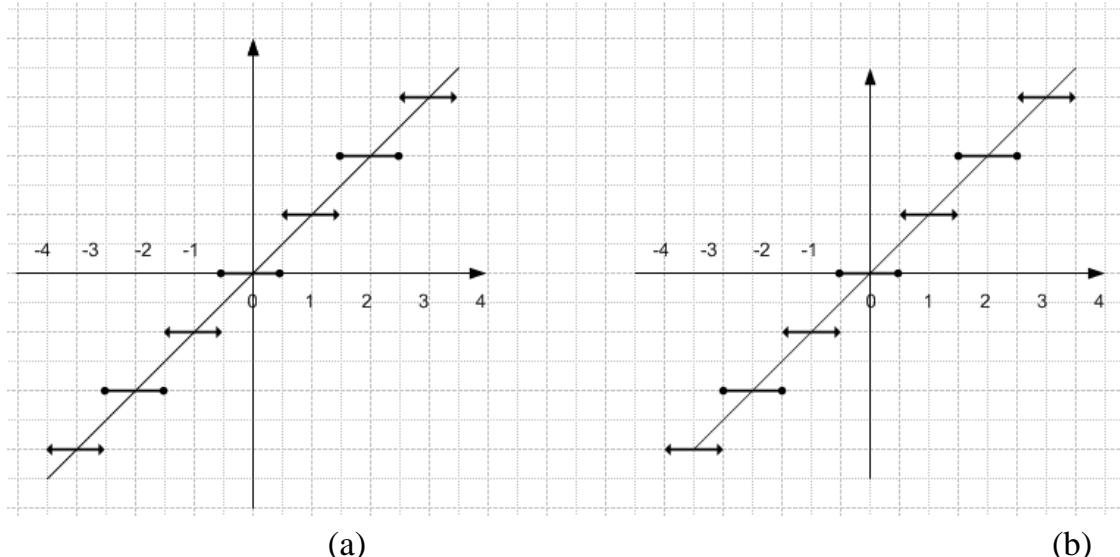
Nếu như số làm tròn được làm tròn lên giá trị cao hơn, thay vì làm tròn với giá trị thấp hơn thì ta có kiểu *làm tròn tới  $+\infty$* (round toward  $+\infty$ ). Với kiểu này thì giá trị sau làm tròn luôn lớn hơn giá trị trước làm tròn.



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 phần 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*.

#### 6.4. Phép cộng số thực dấu phẩy động

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

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

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

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

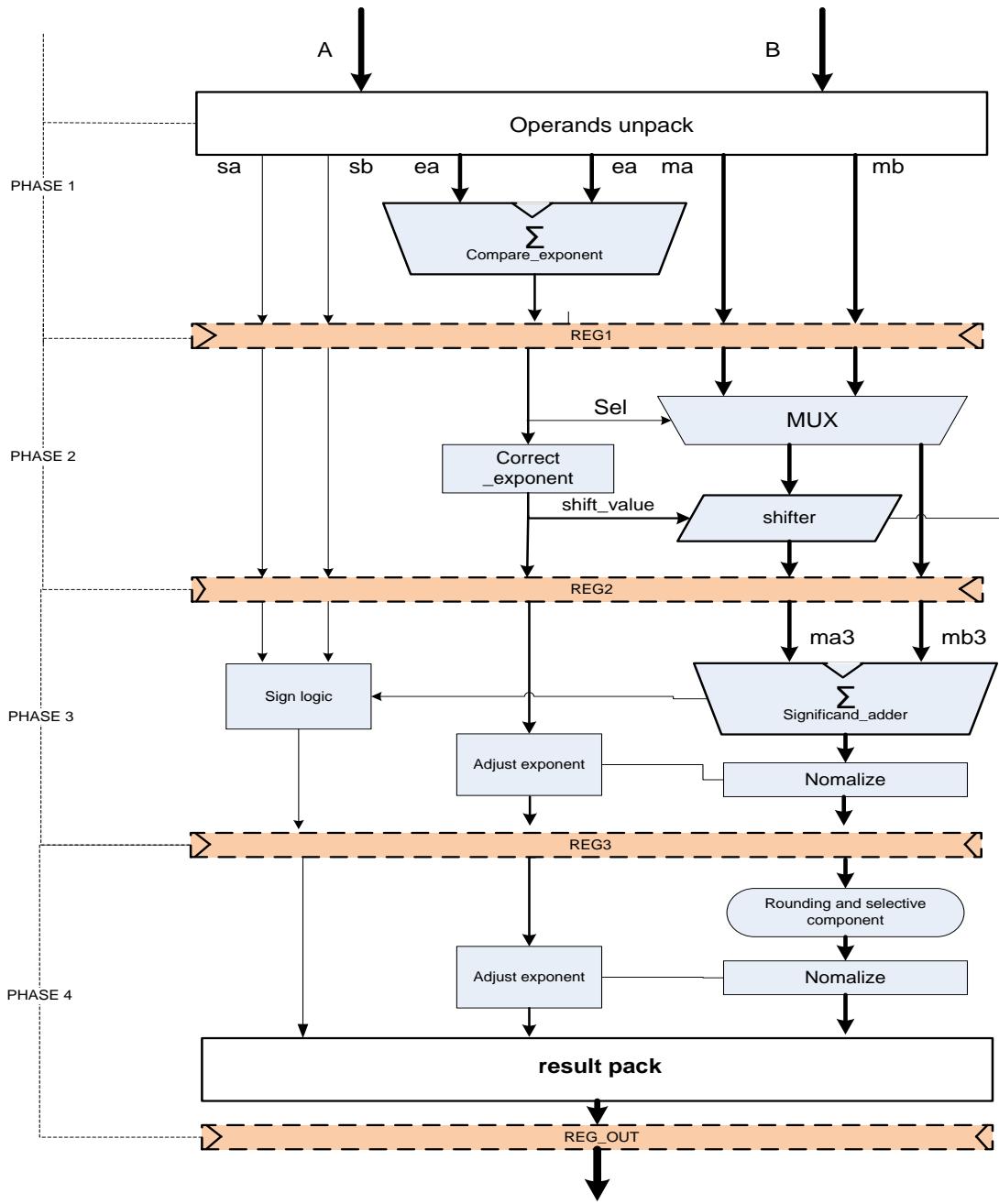
Một trong những bước đầu tiên là quy đổi hai số về cùng một số mũ chung, nếu xét về mặt toán học có thể thực hiện quy đổi về số mũ chung ea hoặc eb đều cho kết quả như nhau nếu như khôi cộng đảm bảo sự chính xác tuyệt đối. Tuy vậy trên thực tế phép cộng số thực theo chuẩn xét trên đa phần không có kết 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:

$$B = -1^{\text{sign}b} (1, b_{n-1}..b_1b_0) 2^{-(ea-eb)} 2^{ea-\text{BIAS}} \quad (3.15)$$

Tiếp theo tiến hành cộng phần thập phân của hai số

$$\begin{aligned} M &= (1, a_{n-1}..a_1a_0) + (1, b_{n-1}..b_1b_0) 2^{-(ea-eb)} \\ &= mA + mB \cdot 2^{-(ea-eb)} \end{aligned} \quad (3.16)$$

Với lý thuyết cơ bản trên khôi 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 khôi chính được biểu diễn ở hình sau:



Hình 3-37. Sơ đồ khối công 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, 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  $ea - eb \geq 0$  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  $ea - eb \leq 26$ , 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 khai rộng 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 M<sub>0</sub> 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 m<sub>a</sub>, m<sub>b</sub> khi đó số 1 đầu tiên bên trái có thể nằm ở vị trí của Cout hoặc M<sub>23</sub>. Nếu kết quả thu được không có bít nhớ Cout, và bit M<sub>23</sub> = '1', khi đó theo quy tắc làm tròn ta sẽ quan tâm tới giá trị của M<sub>0</sub> 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 M<sub>1</sub> để xét tính chẵn lẻ, M<sub>0</sub> để làm tròn, các bít G, R, S cung cấp thông tin cho phần còn lại có khác 0 hay không.

Đối với trường hợp phép trừ m<sub>a</sub> cho m<sub>b</sub> 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 M<sub>23</sub> như đã xét ở trên. Trường hợp thứ hai là số 1 rơi vào vị trí M<sub>22</sub> (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| ≤ 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 phần 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.

## 6.5. Phép nhân số thực dấu phẩy động

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:

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

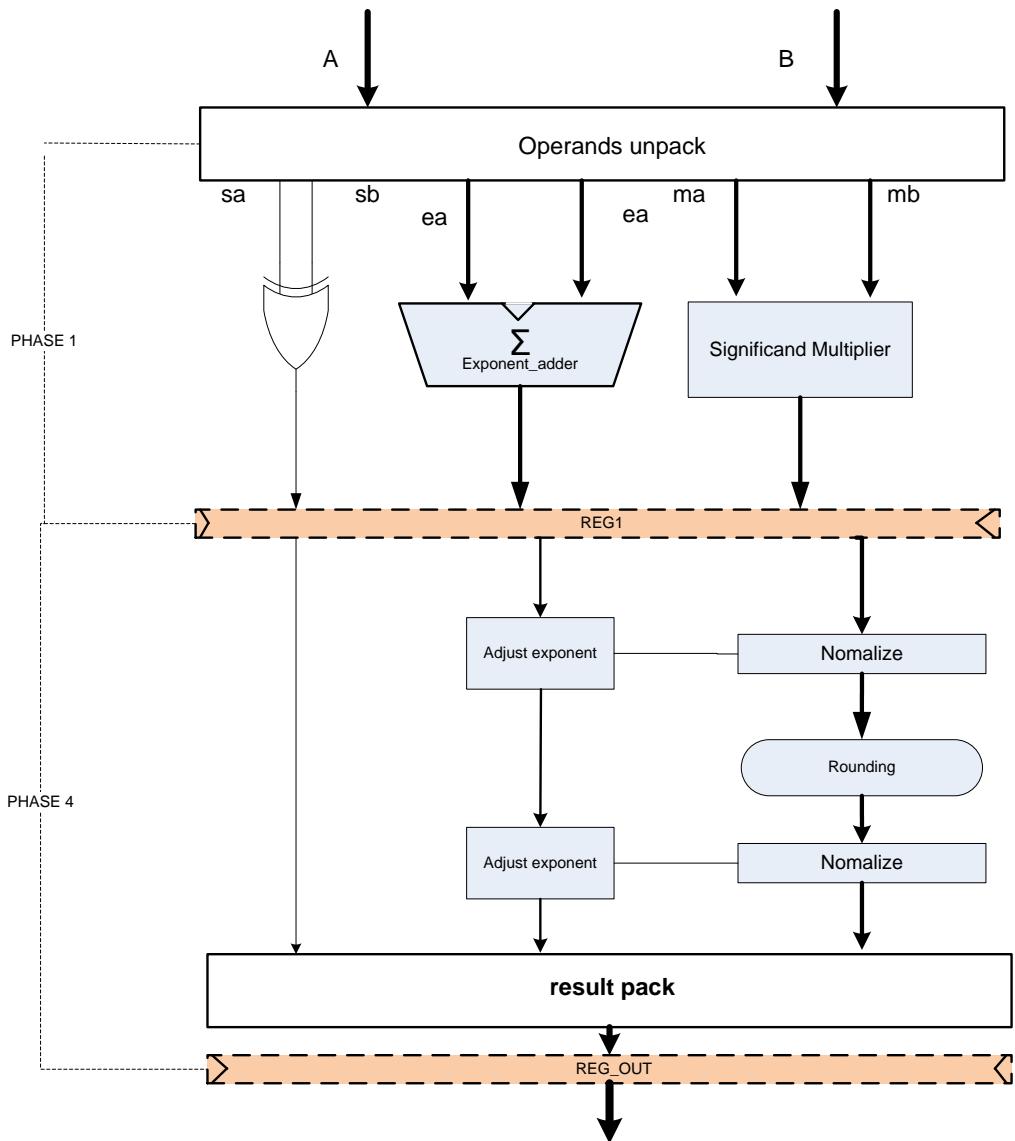
$$B = -1^{\text{signb}}(1, b_{n-1}..b_1b_0)2^{eb-BIAS}$$

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 nhân.

$$A \cdot B = -1^{\text{signa}+\text{signb}}(1, a_{n-1}..a_1a_0).(1, b_{n-1}..b_1b_0).2^{ea+eb-2 * BIAS} \quad (3.17)$$

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 =  $2^8 - 1$ , do vậy ta sẽ cho 1 vào bit Cin của bộ cộng sau đó trừ đi  $2^8 = 100000000$ , 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 M<sub>48</sub> hoặc M<sub>47</sub> 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.

## 6.6. Phép chia số thực dấu phẩy động

Xét hai số sau:

$$A = -1^{\text{sign}a} (1, a_{n-1}..a_1a_0) 2^{ea-BIAS}$$

$$B = -1^{\text{sign}b} (1, b_{n-1}..b_1b_0) 2^{eb-BIAS}$$

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:

$$A \cdot B = -1^{\text{sign}a + \text{sign}b} (1, a_{n-1}..a_1a_0) / (1, b_{n-1}..b_1b_0) \cdot 2^{ea-eb} \quad (3.18)$$

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:

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

(3.19)

Kết quả phép chia thu được thương số là một số  $q = q_{n-1} \dots q_1 q_0 q_{-1} q_{-2} r$ . Trong đó các bit  $q_i$  là kết quả chia, còn  $r$  thêm vào đại diện cho phần số dư ( $r_{n-1} \dots r_1 r_0$ ).  $r = 0$  nếu số dư bằng 0 còn  $r = 1$  nếu số dư khác 0, bit này chỉ có ý nghĩa khi thực hiện làm tròn.

<b>q</b>	<b>22</b>	<b>q</b>	<b>21</b>	<b>q</b>	<b>20</b>	.....	<b>q1</b>	<b>q0</b>	<b>q-1</b> (R)	<b>q-2</b> (S)	<b>R</b> (S)
----------	-----------	----------	-----------	----------	-----------	-------	-----------	-----------	-------------------	-------------------	-----------------

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

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

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

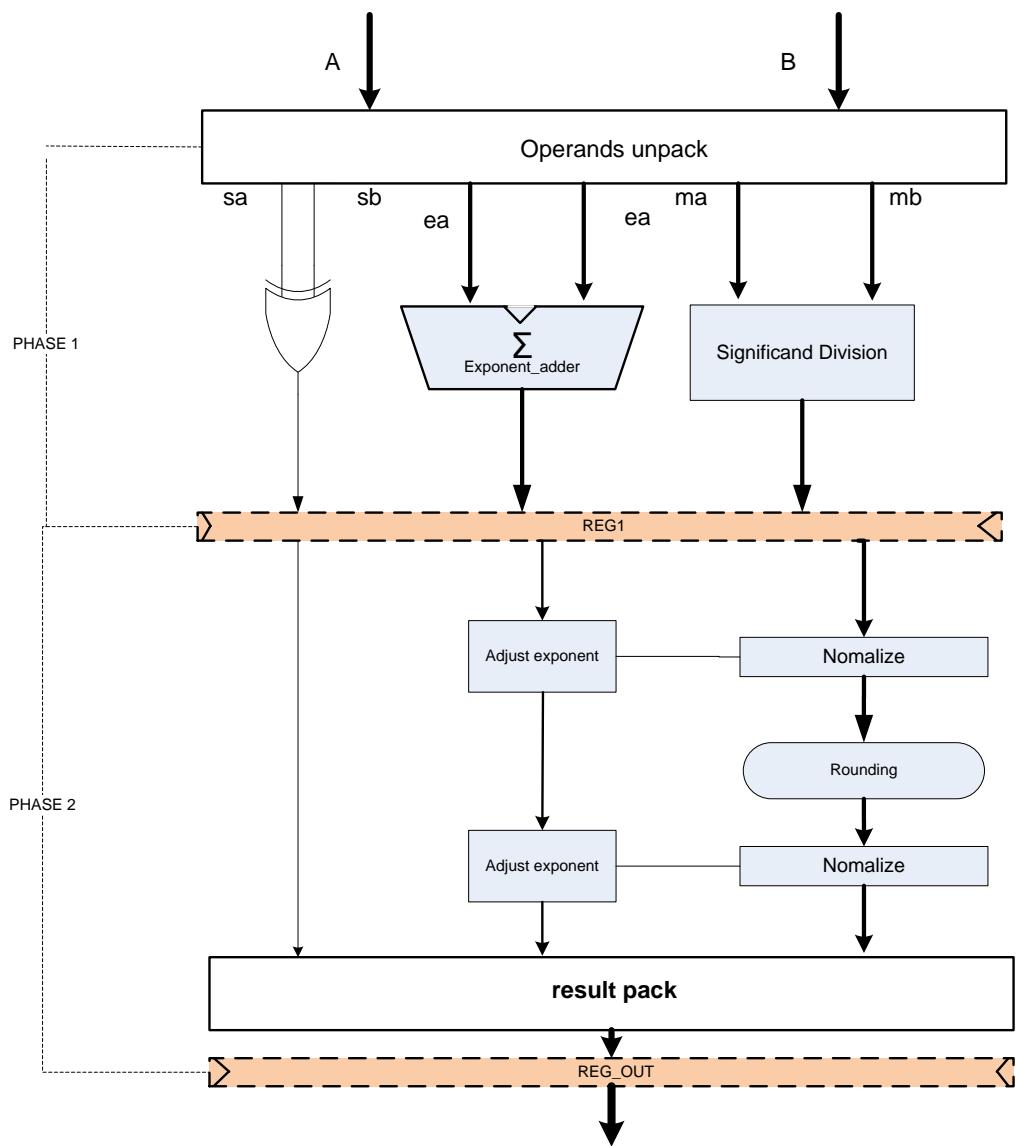
Không mất tổng quát và để dễ theo dõi xét trường hợp số thực dấu phẩy động theo chuẩn IEEE/ANSI 754 với  $n = 23$ , kết quả chia thể hiện ở hình 3.42. Có thể dễ dàng chỉ ra rằng do đặc điểm của số chia và số bị chia đều là các số 24 bit và 48 bit với bit đầu tiên bằng 1 nên trong kết quả thương  $q$  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 bít 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:



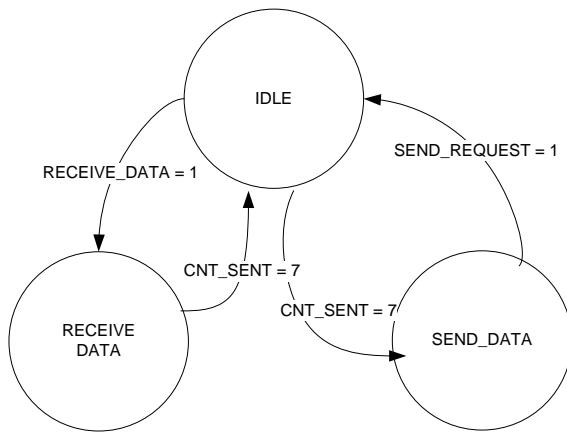
Hình 3-43. Sơ đồ khói chia số thực dấu phẩy động

## Bài tập chương 3

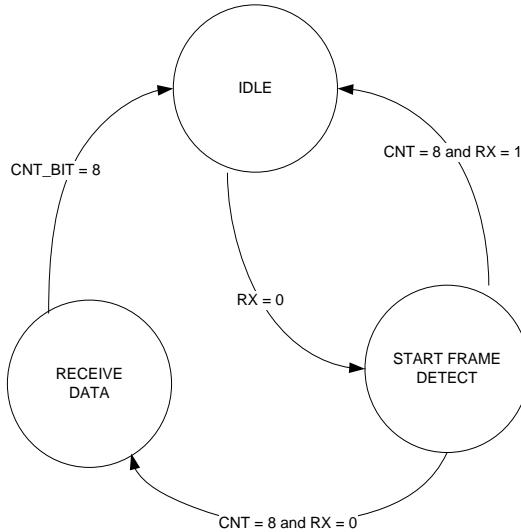
### Bài tập

1. Thiết kế và kiểm tra bộ cộng 32 bit nối tiếp dùng thanh ghi dịch và một FULL\_ADDER.
2. Thiết kế và kiểm tra bộ cộng  $2N$  bit nối tiếp dùng thanh ghi dịch và 2 FULL\_ADDER.
3. Thiết kế và kiểm tra bộ cộng 32 bit dùng thuật toán Carry look ahead adder, so sánh với các thuật toán khác.
4. Thiết kế và kiểm tra khối nhân số nguyên không dấu  $4 \times 4 = 8$ bit dùng thuật toán cộng dịch trái.
5. Thiết kế và kiểm tra khối nhân số nguyên không dấu  $4 \times 4 = 8$ bit dùng thuật toán cộng dịch phải.
6. Xây dựng sơ đồ tổng quá cho khối nhân số nguyên không dấu  $K$  bit  $\times K$  bit =  $K$ bit có kiểm soát trường hợp tràn kết quả.
7. Thiết kế và kiểm tra khối nhân số nguyên có dấu  $4 \times 4 = 8$ bit dùng mã hóa Booth cơ số 2.
8. Thiết kế và kiểm tra khối nhân số nguyên có dấu  $4 \times 4 = 8$ bit dùng thuật toán mã hóa Booth cơ số 4.
9. Xây dựng sơ đồ tổng quá cho khối nhân số nguyên có dấu  $K$  bit  $\times K$  bit =  $K$ bit có kiểm soát trường hợp tràn kết quả.
10. Thiết kế và kiểm tra khối chia số nguyên không dấu 8 bit / 4bit = 4 bit dùng thuật toán phục hồi phần dư.
11. Thiết kế và kiểm tra khối chia số nguyên không dấu 8 bit / 4bit = 4 bit dùng thuật toán không phục hồi phần dư.
12. Thiết kế và kiểm tra khối chia số nguyên có dấu 8 bit / 4bit = 4 bit trên cơ sở khối chia số không dấu không phục hồi phần dư.
13. Xây dựng thuật toán tổng quát cho phép chia số nguyên có dấu  $K$ -bit/  $K$ -bit =  $K$ -bit. Trong số số bị chia, số chia, thương số, số dư đều biểu diễn tổng quát dưới dạng  $K$ -bit nhưng.
14. Thiết kế và mô phỏng khối FIFO  $16 \times 16$ bit.
15. Thiết kế và mô phỏng khối RAM  $32 \times 8$ bit. một cổng đọc ghi đồng bộ.
16. Thiết kế và mô phỏng khối RAM  $16 \times 16$ bit. một cổng đọc không đồng bộ, ghi đồng bộ.

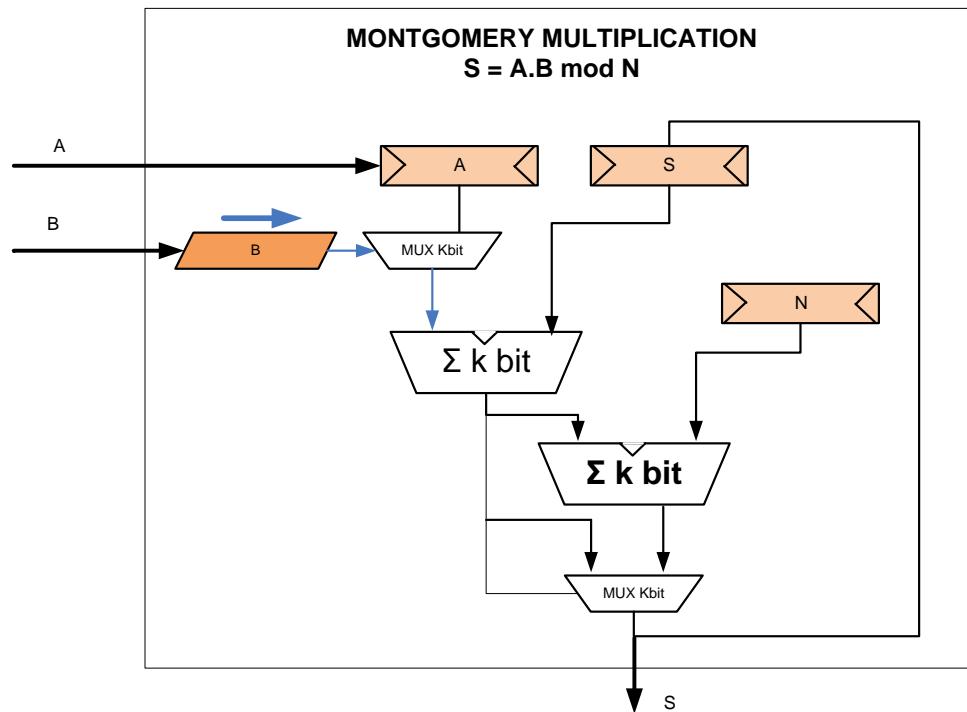
17. Thiết kế và mô phỏng khối RAM 16x16bit. hai cổng đọc ghi đồng bộ.
18. Thiết kế và mô phỏng khối RAM 16x16bit. hai cổng cổng đọc không đồng bộ, ghi đồng bộ.
19. Thiết kế khối Ram 1 cổng đọc ghi đồng bộ. 16x9 trong đó mỗi một hàng chứa 8 bit thấp là 1 byte dữ liệu còn bít cao nhất là bit parity của dữ liệu tương ứng trên hàng.
20. Thiết kế khối thanh ghi đa năng General Purpose Register trong vi xử lý, kích thước 16x16bit hỗ trợ hai cổng đọc đồng bộ và 1 cổng ghi đồng bộ.
21. Thiết kế và mô phỏng khối LIFO 16x16bit.
22. Thiết kế và mô phỏng khói ROM 64x8 bit lưu trữ giá trị rời rạc hóa của một chu kỳ hình SIN.
23. Thiết kế khói ROM đồng bộ 64x8 bit lưu trữ giá trị rời rạc hóa của một chu kỳ hình COSIN.
24. Thiết kế khói ROM CHARACTER với kích thước các ô nhớ là 24x 200 bit lưu trữ hình ảnh các chữ cái từ A tới Z, ô nhớ 200 bit được chia thành 20 hàng và 10 cột với các giá trị 0, 1 để lưu trữ hình dạng của từng chữ cái.
25. Thiết kế và kiểm tra thanh ghi dịch hỗ trợ thao tác ghi dữ liệu song song điều khiển bởi tín hiệu LOAD và các lệnh dịch logic và arithmetic trái không sử dụng toán tử dịch, giá trị dịch là một số 5 bit, dữ liệu dịch là chuỗi 32 bit. Tín hiệu điều khiển LOG = 1 nếu như phép dịch logic, LOG = 0 nếu dịch số học.
26. Thiết kế và kiểm tra thanh ghi dịch hỗ trợ thao tác ghi dữ liệu song song điều khiển bởi tín hiệu LOAD và các lệnh dịch logic và arithmetic phải không sử dụng toán tử dịch, giá trị dịch là một số 5 bit, dữ liệu dịch là chuỗi 32 bit. Tín hiệu điều khiển LOG = 1 nếu như phép dịch logic, LOG = 0 nếu dịch số học.
27. Thiết kế và kiểm tra thanh ghi dịch hỗ trợ thao tác ghi dữ liệu song song điều khiển bởi tín hiệu LOAD và phép dịch vòng trái, phải không sử dụng toán tử dịch, giá trị dịch là một số 5 bit, dữ liệu dịch là chuỗi 32 bit. Tín hiệu điều khiển LEFT = 1 nếu dịch trái và LEFT = 0 nếu dịch phải.
28. Viết mô tả VHDL cho máy trạng thái có sơ đồ sau:



29. Viết mô tả VHDL cho máy trạng thái có sơ đồ sau:



30. Thiết kế và kiểm tra bộ so sánh hai số 8 bit có dấu và không dấu.
31. Thiết kế và kiểm tra bộ cộng tích lũy 32 bit.
32. Thiết kế khói đếm lệnh chương trình cho vi xử lý (Program Counter), ngoài các cổng thông thường khói đếm hỗ trợ thêm các thao tác : tăng bộ đếm lên 1 đơn vị bằng tín hiệu  $Pcinc$ , đặt lại giá trị bộ đếm bằng giá trị  $PC\_in$  và tín hiệu điều khiển  $PC\_set$ .
33. Thiết kế khói thực hiện thuật toán MontGomery tính module của tích hai số  $A \cdot B$  cho 1 số nguyên tố  $N$  theo sơ đồ thiết kế sau, độ rộng bit  $K = 128, 256, 512$ :



34. Thiết kế hoàn chỉnh khối cộng số thực dấu phẩy động cho chuẩn ANSI/IEEE 754 32 bit, kiểu làm tròn ngầm định là làm tròn tới giá trị chẵn gần nhất.
35. Thiết kế hoàn chỉnh khối nhân số thực dấu phẩy động cho chuẩn ANSI/IEEE 754 32 bit, kiểu làm tròn ngầm định là làm tròn tới giá trị chẵn gần nhất.
36. Thiết kế hoàn chỉnh khối chia số thực dấu phẩy động cho chuẩn ANSI/IEEE 754 32 bit, kiểu làm tròn ngầm định là làm tròn tới giá trị chẵn gần nhất.
37. Dựa trên các khối số học logic đã được học, xây dựng khối thực thi lệnh đơn giản trong vi xử lý ALU, áp dụng cho các nhóm lệnh dịch, số học và logic.
38. Thiết kế khối cộng số thực dấu phẩy động 32-bit sử dụng 8 bit thấp cho phần thập phân và 24 bit cao cho phần nguyên.
39. Thiết kế khối nhân số thực dấu phẩy động 32-bit sử dụng 8 bit thấp cho phần thập phân và 24 bit cao cho phần nguyên.
40. Thiết kế khối chia số thực dấu phẩy động 32-bit sử dụng 8 bit thấp cho phần thập phân và 24 bit cao cho phần nguyên.
41. Thiết kế khối MAC (Multiplication Accumulation) cho số nguyên không dấu thực hiện thao tác cộng và nhân:  $A = B \cdot C + D$ . Nghiên cứu tối ưu hóa khối thiết kế.
42. Thiết kế khối MAC (Multiplication Accumulation) cho số nguyên có dấu thực hiện thao tác cộng và nhân:  $A = B \cdot C + D$ . Nghiên cứu tối ưu hóa khối thiết kế.

43. Thiết kế khối MAC (Multiplication Accumulation) cho số thực dấu phẩy động thực hiện thao tác cộng và nhân:  $A = B.C + D$ . Nghiên cứu tối ưu hóa khối thiết kế.

44. Thiết kế khối MAC (Multiplication Accumulation) cho thực dấu phẩy tĩnh với 8 bit thập phân phân và 24 bit phần nguyên thực hiện thao tác cộng và nhân:  $A = B.C + D$ . Nghiên cứu tối ưu hóa khối thiết kế.

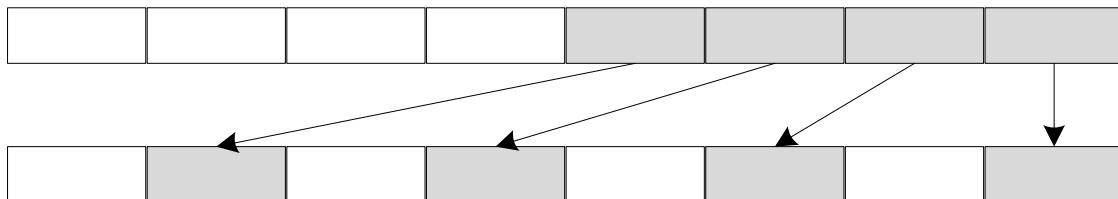
43. Thiết kế khối thu và xử lý tín hiệu. Tín hiệu đầu vào là các tổ hợp 32-bit, với yêu cầu thực hiện giải nén dữ liệu với các tham số đầu vào như sau:

N – Số lượng bit của mỗi đơn vị khói dữ liệu đầu vào N: 2, 4, 8, 16, 32.

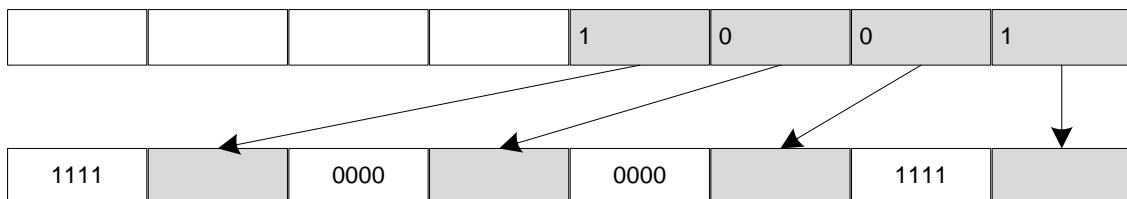
M – Số lượng bit của mỗi đơn vị khói dữ liệu đầu ra.(N <= M). M: 2, 4, 8, 16, 32)

Tín hiệu SE = 1 giải nén có dấu, SE= 0 giải nén không dấu.

Ví dụ N=4, M= 8, SE = 0



Với N=4, M= 8, SE = 1:

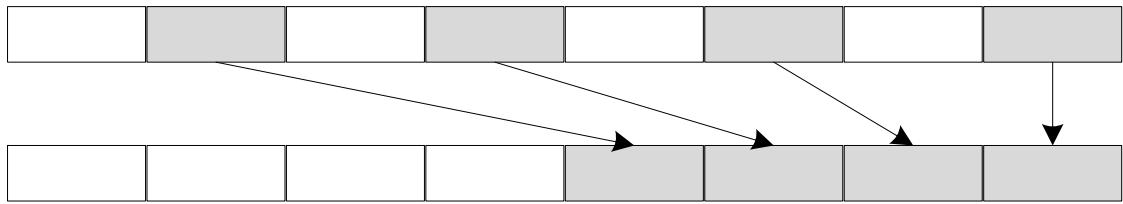


44. Thiết kế khối thu và xử lý tín hiệu. Tín hiệu đầu vào là các tổ hợp 32-bit, với yêu cầu nén dữ liệu với các tham số đầu vào như sau:

N – Số lượng bit của mỗi đơn vị khói dữ liệu đầu vào N: 2, 4, 8, 16, 32.

M – Số lượng bit của mỗi đơn vị khói dữ liệu đầu ra.(N <= M). M: 2, 4, 8, 16, 32)

Ví dụ N=4, M= 8



45. Thiết kế khối xử lý tín hiệu. Tín hiệu đầu vào là các tổ hợp 32-bit, với yêu cầu biến đổi dữ liệu như sau:

N – Số lượng bit của mỗi đơn vị khối dữ liệu đầu vào N: 2, 4, 8, 16, 32

COL, DIAG quy định kiểu ma trận đầu ra

m giá trị chỉ vị trí cột trong trường hợp COL = 1

Tổ hợp đầu vào:

3	2	1	0
---	---	---	---

Ma trận đầu ra nếu DIAG = 1, COL = 0:

0			
	1		
		2	
			3

Ma trận đầu ra nếu COL = 1

		0	
		1	
		2	
		3	

46. Thiết kế khối xử lý tín hiệu. Tín hiệu đầu vào là các tổ hợp 32-bit, với yêu cầu thực hiện các phép biến đổi ngược với bài toán 45.

47. Thiết kế khối nhân 2 vector 4 số nguyên không dấu theo các tiêu chí khác nhau về tài nguyên và về tốc độ thực thi.

48. Thiết kế khối nhân ma trận số nguyên không dấu cho 2 ma trận 4x4 theo các tiêu chí khác nhau về tài nguyên và tốc độ thực thi.

49. Thiết kế khối nhân ma trận 4x4 với vector 4 số nguyên theo các tiêu chí khác nhau về tài nguyên và tốc độ thực thi.

50. Thiết kế bộ giải mã kênh số học logic ALC1 thực hiện các lệnh sau đây trong tổ hợp lệnh của kiến trúc MIPS

## ALU instructions set I

Nº	Mnemonic	Opcode	function	Unit	description
<b>ARITHMETRIC INSTRUCTIONS</b>					
1	ADD A-28	00000 0	10000 0	IA	To add 32-bit integers.
2	ADDI	00100 0		IA	To add a constant to 32-bit integer
3	ADDU	00000 0	10000 1	IA	To add 32 bit integer without setting OV
4	ADDIU	00100 1		IA	To add a constant to 32-bit integer without setting OV
5	SUB	00000 0	10001 0	IA	To subtract 32-bit integers
6	SUBU	00000 0	10001 1	IA	To subtract 32-bit integer without setting OV
<b>LOGICAL INSTRUCTIONS</b>					
7	AND	00000 0	10010 0	LOG	To do a bitwise logical AND
8	ANDI	00110 0		LOG	To do a bitwise with a constant
9	OR	00000 0	10010 1	LOG	To do a bitwise logical OR
10	ORI	00110 1		LOG	To do a bitwise logical OR with a constant
11	NOR	00000 0	10011 1	LOG	To do a bitwise logical NOT OR
12	XOR A-172	00000 0	10011 0	LOG	To do a bitwise logical EXCLUSIVE OR
13	XORI	00111 0		LOG	To do a bitwise logical EXCLUSIVE OR with a constant
<b>SHIFT INSTRUCTIONS</b>					
14	SLL	00000 0	00000 0	SH	To logical left shift a word by a fixed number of bits

15	SLLV	00000 0	00010 0	SH	To logical left shift a word by a variable number of bits
16	SRA A-140	00000 0	00001 1	SH	To arithmetic right shift a word by a fixed number of bits
17	SRAV	00000 0	00011 1	SH	To arithmetic right shift a word by a variable number of bits
18	SLR	00000 0	00001 0	SH	To logical right shift a word by a fixed number of bits
19	SLRV	00000 0	00011 0	SH	To logical right shift a word by a variable number of bits

Chi tiết về các lệnh xem trong tài liệu [22] Mips Instruction Set Reference

Vol I

## Câu hỏi ôn tập lý thuyết

1. Trình bày thuật toán cộng Carry look ahead adder, so sánh với thuật toán cộng nối tiếp.
2. Trình bày thuật toán cộng dùng 1 full\_adder, ưu nhược điểm của thuật toán này.
3. Trình bày cấu trúc thanh ghi dịch, thuật toán dịch không dùng toán tử dịch, ví dụ ứng dụng thanh ghi dịch.
4. Trình bày cấu trúc bộ đếm.
5. Trình bày thuật toán và cấu trúc khối nhân số không dấu thông thường dùng mạch tổ hợp.
6. Trình bày thuật toán và cấu trúc khối nhân cộng dịch trái cho số không dấu.
7. Trình bày thuật toán và cấu trúc khối nhân cộng dịch phải cho số không dấu, so sánh với khối nhân cộng dịch trái.
8. Trình bày thuật toán và cấu trúc khối nhân số có dấu dùng mã hóa BOOTH cơ số 2.
9. 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.
10. Trình bày thuật toán và cấu trúc khối chia số không dấu phục hồi phần dư.
11. Trình bày thuật toán và cấu trúc khối chia số không dấu không phục hồi phần dư.
12. Trình bày thuật toán và cấu trúc khối chia số có dấu.
13. Các loại khối nhớ RAM, ROM.
14. Trình bày thuật toán xây dựng FIFO và LIFO trên cơ sở Dual-port RAM.
15. Máy trạng thái, cấu trúc máy trạng thái mô tả bằng VHDL, ứng dụng.
16. Trình bày sơ đồ khối cộng số thực dấu phẩy động theo chuẩn ANSI/IEEE 754.
17. Trình bày sơ đồ khối nhân số thực dấu phẩy động theo chuẩn ANSI/IEEE 754
17. Trình bày sơ đồ khối chia số thực dấu phẩy động theo chuẩn ANSI/IEEE 754

## Chương 4

### THIẾT KẾ MẠCH SỐ TRÊN FPGA

Kiến thức của những chương trước cung cấp kiến thức từ cơ bản tới nâng cao của các thiết kế số về mặt chức năng bằng HDL. Ở chương này chúng ta sẽ nghiên cứu về một công nghệ cho phép chuyển các thiết kế logic thành sản phẩm ứng dụng, đó là công nghệ FPGA.

FPGA là công nghệ mang lại sự thay đổi lớn lao trong kỹ thuật điện tử số hiện đại. Nếu như các IC tích hợp số trước kia được sản xuất bằng công nghệ phức tạp, sở hữu bởi số ít các quốc gia có nền tảng khoa học kỹ thuật phát triển, khi thiết kế các hệ thống số người thiết kế không có được sự tùy biến linh động cũng như những giải pháp tối ưu mà phải lệ thuộc vào các phần tử có sẵn. HDL và FPGA ra đời đã cho phép người thiết kế có khả *năng tự thiết kế IC* *chức năng* theo mục đích sử dụng một cách nhanh chóng dễ dàng, đây cũng chính là một trong những cơ sở để môn học thiết kế vi mạch tích hợp được đưa vào giảng dạy cho đối tượng đại học. Bên cạnh sự tiếp cận trực tiếp và đơn giản FPGA còn đem lại hiệu quả thiết kế cao và tính ứng dụng thực tiễn cho những bài toán số được xem rất phức tạp đối với các công nghệ cũ hơn.

Sinh viên sẽ được giới thiệu căn bản về cấu trúc của FPGA và các cách thức làm việc với FPGA. Song song đó là các bài thực hành giúp sinh viên làm quen và sử dụng thành thạo các công cụ thiết kế trên FPGA, trên cơ sở đó thực hiện các bài tập cơ bản và một số hướng nghiên cứu chuyên sâu sử dụng công nghệ FPGA.

# 1. Tổng quan về kiến trúc FPGA

## 1.2. Khái niệm FPGA

FPGA là công nghệ vi mạch tích hợp khả trình (PLD - Programmable Logic Device) 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 thông thường 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 trong những đặc điểm làm FPGA trở thành một công nghệ IC khả trình được nghiên và cứu phát triển nhiều nhất hiện nay.

Để có được khả năng đó, FPGA ra đời hoàn toàn là một công nghệ mới chứ không phải là một dạng mở rộng của các chip khả trình kiểu như PAL, PLA... Sự khác biệt đó thứ nhất nằm ở cơ chế tái cấu trúc FPGA, toàn bộ cấu hình của FPGA thường được lưu trong một bộ nhớ truy cập ngẫu nhiên (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 và chức năng logic trong IC. Có thể so sánh cơ chế đó làm việc giống như phần mềm máy tính cũng được lưu trữ trong RAM và khi thực thi sẽ được nạp lần lượt vi xử lý, nói cách khác 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 trước sau đó mới diễn ra quá trình tái cấu trúc theo nội dung thông tin chứa trong SRAM. Dữ liệu chứa trong bộ nhớ RAM phụ thuộc vào nguồn cấp, 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 FPGA được thiết kế để có thể giao tiếp với rất nhiều dạng ROM khác nhau hoặc FPGA thường được thiết kế kèm CPLD để nạp những thành phần cố định, việc tích hợp này làm FPGA nạp cấu hình nhanh hơn nhưng cơ chế nạp và lưu trữ cấu hình vẫn không thay đổi.

Ngoài khả năng đó điểm thứ hai làm FPGA khác biệt với các PLD thế hệ trước là FPGA có *khả năng tích hợp logic với mật độ cao* với số cổng logic tương đương lên tới hàng trăm nghìn, hàng triệu cổng. Khả năng đó có được 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 4-6 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 khói 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. Vì có *mật độ tích hợp lớn và tốc độ làm việc cao* nên FPGA có thể được ứng dụng cho lớp những bài toán xử lý số phức tạp đòi hỏi hiệu suất làm việc lớn mà các công nghệ trước đó không đáp ứng được.

Thiết kế trên FPGA thường được thực hiện bởi các ngôn ngữ HDL và hầu hết các dòng FPGA hiện tại hỗ trợ thiết kế theo hai ngôn ngữ chính là Verilog và VHDL, tất cả những thiết kế ở những chương trước đều có thể hiện thực hóa trên FPGA bằng một quy trình đơn giản. Ngoài HDL, thiết kế trên FPGA còn có thể được thực hiện thông qua hệ nghĩa là bằng ngôn ngữ phần mềm (thường là C/C++). Một phương pháp nữa thường dùng trong các bài toán xử lý số tín hiệu là sử dụng *System Generator* một chương trình kết hợp của Matlab với phần mềm thiết kế FPGA của Xilinx.

Hiện nay công nghệ FPGA đang được phát triển rộng rãi bởi nhiều công ty bán dẫn khác nhau. Đầu tiên là Xilinx với các dòng sản phẩm như Virtex 3, 4, 5, 6 và Spartan3, 6, Altera với Stratix, Cyclone, Arria, Bên cạnh đó còn có sản phẩm của Lattice Semiconductor Company, Actel, Achronix, Blue Silicon Technology...

Khái niệm *FPGA board, hay FPGA KIT* là khái niệm chỉ một bo mạch in trên đó có gắn chip FPGA và các phần tử khác như cổng giao tiếp, màn hình, led, nút bấm... và bao giờ cũng có phần giao tiếp với máy tính để nạp cấu hình cho FPGA. Ngoài ra board còn chứa các thiết bị ngoại vi được liên kết với các cổng vào ra của FPGA nhằm mục đích thử nghiệm.

Theo bảng so sánh 1-4 dưới đây trên có thể thấy khả năng tích hợp của FPGA là rất lớn, những FPGA mới nhất hiện nay có khả năng tích hợp lớn tương đương như các chíp chuyên dụng cho máy chủ như Xenon 6-core. Còn bản thân các chíp cỡ nhỏ như Pentium hay thậm chí Core duo nếu so sánh về mức độ tích hợp thì chúng có thể được “nạp” hoàn toàn vào một FPGA cỡ trung bình. Khả năng này của FPGA mở ra một hướng mới cho ứng dụng FPGA đó là sử dụng FPGA như một phương tiện để kiểm tra thiết kế ASIC (*ASIC prototyping with FPGA*). Kế thừa của phương pháp này là công nghệ có tên gọi “*Hard-copy*” là công nghệ cho phép sao chép toàn bộ các thiết kế đã được nạp vào FPGA thành một IC chuyên dụng (ASIC) độc lập. Tính tối ưu của thiết kế này không cao nhưng đơn giản và giảm đáng kể chi phí nếu so sánh với thiết kế ASIC chuẩn.

Tài nguyên logic của FPGA được thể hiện ở bảng so sánh sau:

Bảng 4-1

**Mật độ tích hợp của một số IC thông dụng**

IC	Transistor count	Process	Manufacture	Year
Intel 4004	2 300	10 um	Intel	1971
Zilog Z80	8 500	4 um	Zilog	1976
Intel 80286	164 000	1.5 um	Intel	1982
Pentium 2	7 500 000	0.35um	Intel	1997
Pentium 4	42 000 000	180 nm	Intel	2000
Core 2 Duo	291 000 000	65 nm	Intel	2006
Six core Xenon 7400	1 900 000 000	45 nm	Intel	2008
10-Core Xeon Westmere-EX	2 600 000 000	32 nm	Intel	2010
AMD K8	106 000 000	130 nm	AMD	2003
Spartan 3E	~40 000 000	90 nm	Xilinx	1998
Virtex 4	1 000 000 000	90 nm	Xilinx	2004
Virtex 5	1 100 000 000	65 nm	Xilinx	2006
Starix IV	2 500 000 000	40 nm	Altera	2008
Starix V	3 800 000 000	28 nm	Altera	2011
Virtex 6	~2 600 000 000	65 nm	Xilinx	2010
Virtex 7	~6 800 000 000	28nm	Xilinx	2011

Nguồn [http://en.wikipedia.org/wiki/Transistor\\_count](http://en.wikipedia.org/wiki/Transistor_count)

**1.3. Ứng dụng của FPGA trong xử lý tín hiệu số**

Do khả năng tái cấu trúc đơn giản và sở hữu một khối tài nguyên logic lớn FPGA có thể được ứng dụng cho nhiều các lớp bài toán xử lý tín hiệu số cỡ lớn mà các công nghệ trước đó không làm được hoặc làm được nhưng với tốc độ và hiệu suất thấp. Các lớp ứng dụng đó là:

- Các ứng dụng chung về xử lý số như lọc tín hiệu, tìm kiếm, phân tích, giải mã, điều chế tín hiệu, trộn tín hiệu...

- Các ứng dụng về mã hóa, giải mã giọng nói, nhận dạng giọng nói, tổng hợp giọng nói. Xử lý tín hiệu âm thanh bao gồm lọc nhiễu, trộn, mã hóa, giải mã, nén, tổng hợp âm thanh...

- Ứng dụng trong xử lý ảnh số, nén và giải nén, các thao tác biến đổi, chỉnh sửa, nhận dạng ảnh số...

- Ứng dụng trong các hệ thống bảo mật thông tin, cung cấp các khôi giải mã và mã hóa có thể thực thi với tốc độ rất cao và dễ dàng tham số hóa hoặc điều chỉnh.

- Ứng dụng trong các hệ thống thông tin như các hệ thống Voice IP, Voice mail. Modem, điện thoại di động, mã hóa và giải mã truyền thông trong mạng LAN, WIFI... trong truyền hình KTS, radio KTS...

- Ứng dụng trong điều khiển các thiết bị điện tử: ổ cứng, máy in, máy công nghiệp, dẫn đường, định vị, robots.

Các sản phẩm ứng dụng FPGA hiện tại vẫn nằm ở con số khiêm tốn nếu so sánh với các giải pháp truyền thông tuy vậy với các thế mạnh kể trên FPGA chắc chắn sẽ là một công nghệ quan trọng của tương lai. Một số những kiến trúc thích nghi Vi xử lý – FPGA với nền tảng chíp vi xử lý và FPGA được đặt trong một chip đơn mang lại hiệu quả xử lý mạnh mẽ do kết hợp được tính linh động của phần mềm và hiệu suất, tốc độ của phần cứng đang là những hướng nghiên cứu mới và có thể tạo nên sự thay đổi lớn với các thiết kế số truyền thống.

#### 1.4. Công nghệ tái cấu trúc FPGA

Trong lĩnh vực công nghệ tái cấu trúc IC hiện nay có tất cả 5 công nghệ fuse, EPROM, EEPROM, SRAM based, Antifuse trong đó SRAM-based là công nghệ phổ biến được sử dụng cho FPGA.

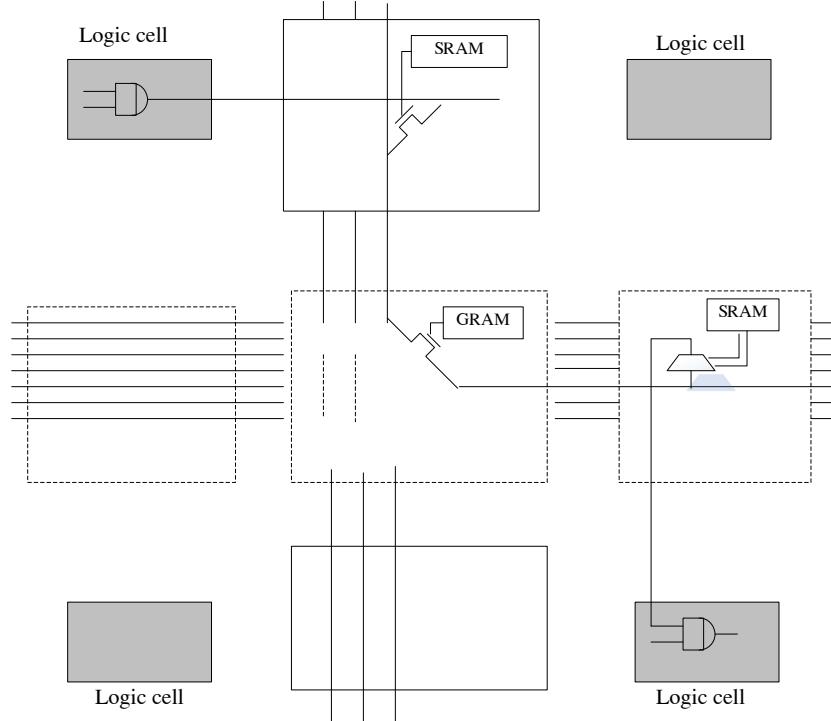
##### *SRAM-based*

Cấu hình của FPGA bản chất là mô tả các điểm kết nối giữa các thành phần có chứa trong IC, có hai dạng kết nối cơ bản là kết nối giữa các đường kết nối dẫn bằng ma trận chuyển mạch (*switch matrix*), và kết nối nội bộ trong các khôi logic. Kết nối trong ma trận chuyển là kết nối giữa hai kênh dẫn được thực hiện thông qua các *pass-transistor*, hay gọi là transistor dẫn. 1 bit thông tin từ bộ nhớ SRAM được sử dụng để đóng hoặc mở *pass-transistor* này, tương ứng sẽ ngắt hay kết nối giữa hai kênh dẫn.

Kiểu cấu trúc thứ hai phổ biến trong các khôi logic là lập trình thông qua khôi chọn kênh (*Multiplexer*). Thông tin điều khiển từ SRAM cho phép khôi chọn kênh chọn một trong số các đầu vào để đưa ra. Nếu khôi lượng đầu vào là  $2^n$ , thì yêu cầu số bit điều khiển từ SRAM là n-bit.

Kiểu cấu trúc thứ 3 được gọi là *Look-Up Table* (LUT), mỗi một LUT có thể được lập trình để thực hiện bất kỳ một hàm logic bất kỳ nào của đầu ra phụ thuộc các đầu vào. Cơ chế làm việc của LUT có thể tóm tắt như sau, giả sử cần

thực hiện một hàm  $m$  đầu vào và  $n$  đầu ra thì cần một bộ nhớ  $2^m \times n$ , chứa thông tin về  $n$  đầu ra đối với tất cả các khả năng đầu vào. Khi làm việc thì  $m$ -bit đầu vào đóng vai trò như địa chỉ để truy cập (Look-up) lên bộ nhớ (Table). Về bản chất cấu trúc này cũng giống như khối chọn kênh cỡ lớn. Trong FPGA phổ biến sử dụng các LUT có 4-6 bit đầu vào và 1 bit đầu ra.



Hình 4-1. *SRAM-based FPGA*

Như vậy tính khả trình của FPGA được thực hiện nhờ tính khả trình của các khối logic và tính khả trình của hệ thống kênh kết nối, ngoài ra là tính khả trình của các khối điều khiển cổng vào ra.

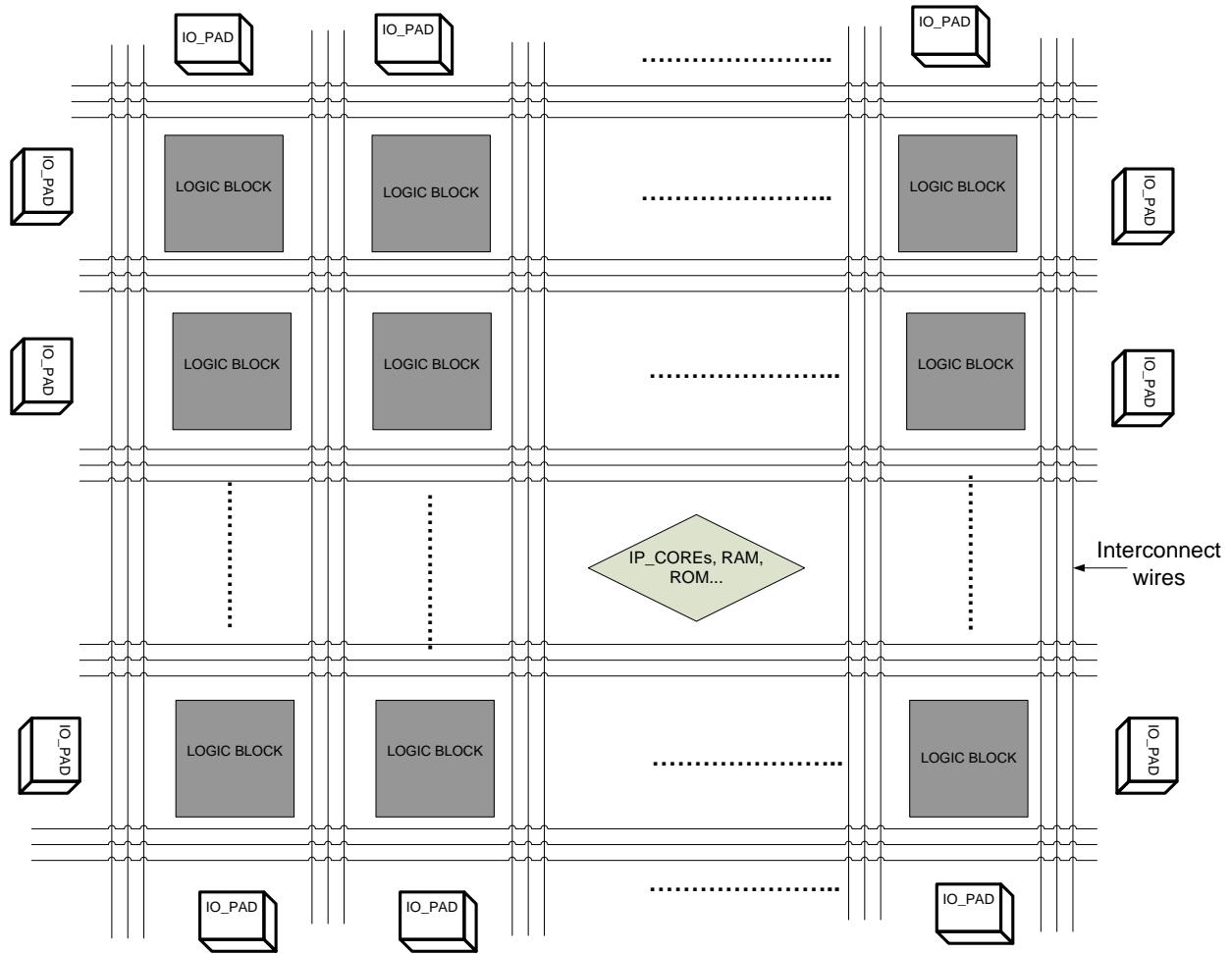
Sau đây ta sẽ đi vào nghiên cứu cấu trúc cụ thể của họ FPGA Spartan 3E của Xilinx, về cơ bản, cấu trúc của các họ Xilinx FPGA khác đều tương tự như cấu trúc này.

### 1.5. Kiến trúc tổng quan

Hình 4.2 trình bày cấu trúc tổng quan nhất cho các loại FPGA hiện nay. 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 này thay đổi từ vài trăm (Xilinx Spartan) đến vài chục nghìn (Xilinx Virtex 6, 7) 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 cientos một nghìn.

Bên cạnh các thành phần chính đó, những FPGA cỡ lớn còn được tích hợp cùng những khối thiết kế sẵn mà thuật ngữ gọi là *Hard IP cores*, các IP cores này có thể là các bộ nhớ RAM, ROM, khối thực hiện phép nhân, khối thực hiện phép nhân cộng (DSP)... bộ vi xử lý cỡ vừa và nhỏ như PowerPC hay ARM.



Hình 4-2. Kiến trúc tổng quan của FPGA

## 2. Kiến trúc chi tiết Xilinx FPGA Spartan-3E.

Để hiểu chi tiết về cấu trúc của FPGA phần dưới đây ta sẽ đi nghiên cứu một cấu trúc cụ thể của FPGA Spartan 3E, tài liệu gốc [20], [21] có thể tìm thấy trên trang chủ của Xilinx, người đọc nên tham khảo thêm để hiểu kỹ và đầy đủ hơn vấn đề.

Spartan 3E FPGA có nhiều loại khác nhau khác nhau về kích thước, tài nguyên logic, cách thức đóng gói, tốc độ, số lượng chân vào ra... bảng sau liệt kê các tham số của các dòng FPGA Spartan 3E.

Bảng 4-2

## XILINX SPARTAN 3E

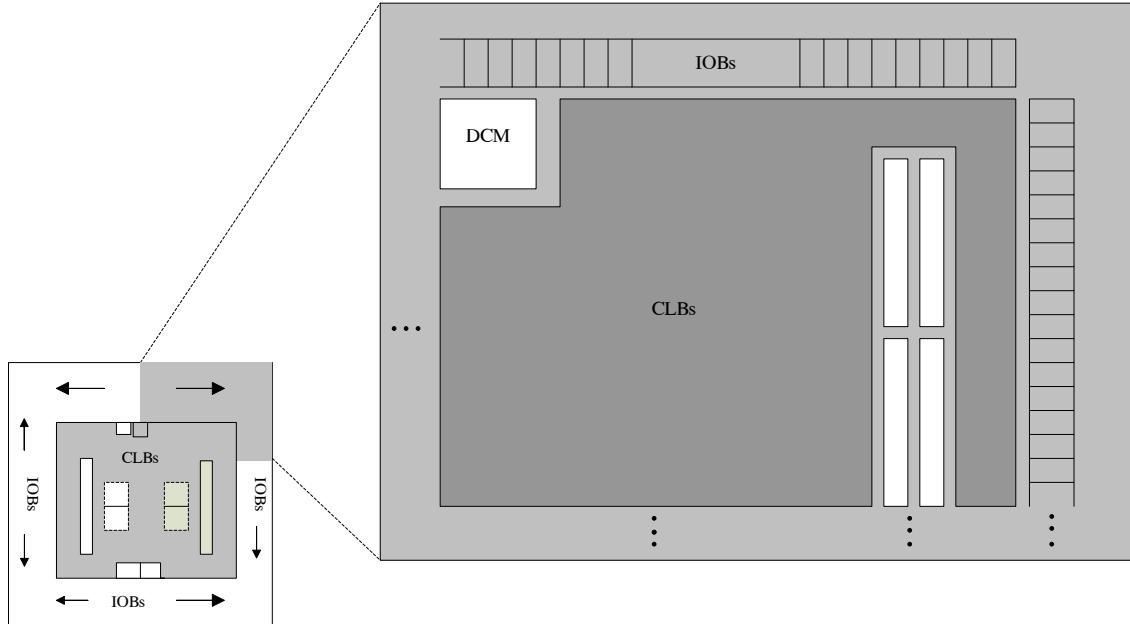
IC	Số lượng cổng (Xilinx) (*)	Cổng logic tương đương	Ma trận CLB				RAM phân tán	Khối RAM	Khối nhán chuyên dụng	DCM	Cổng vào ra	Cổng vào ra vi sai (**)
			Hàng	Cột	Tổng số CLBs	Tổng số Slices						
XC3S100E	100K	2160	22	16	240	960	15K	72K	4	2	108	40
XC3S250E	250K	5508	34	26	612	2448	38K	216K	12	4	172	68
XC3S500E	500K	10476	46	34	1164	4656	73K	360K	20	4	232	92
XC3S1200E	1200K	19512	60	46	2168	8672	136K	504K	28	8	304	124
XC3S1600E	1600K	33192	76	58	3688	14752	231K	648K	36	8	376	156

(\*) Khái niệm cổng tương đương của Xilinx được tính một cách đặc biệt, thông thường nên đánh giá tài nguyên FPGA thông qua số lượng CLBs

(\*\*) Cổng ra vào vi sai - một cặp tín hiệu vào ra được xử lý như một tín hiệu vi sai dạng số.

Ví dụ theo bảng trên XC3S500 có số Slices là 4656, tương đương 1164 CLBs (10,476 cổng tương đương) được bố trí trên 46 hàng và 24 cột. Các tài nguyên khác bao gồm 4 khối điều chỉnh/tạo xung nhịp hệ thống *Digital Clock Manager* (DCM) được bố trí 2 ở trên và 2 ở dưới. Các khối nhớ bao gồm 360K Block RAM và tối đa 73K RAM phân tán. Tích hợp 20 khối nhán 18x18 bít được bố trí sát các Block Ram. Về tài nguyên cổng vào ra XC3S500E với gói PQ208 hỗ trợ 208 chân vào ra trong đó có 8 cổng cho xung nhịp hệ thống, tối đa 232 cổng vào ra sử dụng tự do, trong đó có 158 chân Input/Output, số còn lại là chân Input. XC3S500E được thiết kế trên công nghệ 90nm và cho phép làm việc ở xung nhịp tối đa đến 300Mhz, với tốc độ như vậy XC3S500 có thể đáp ứng hầu hết những bài toán xử lý số cỡ vừa và nhỏ.

Hình vẽ dưới đây thể hiện cấu trúc tổng quan của họ FPGA này.



Hình 4-3. 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 công đọc ghi độc lập, với các FPGA họ Spartan 3 block RAM thường phân bố ở hai cột, mỗi cột chứa một vài khối RAM 18Kbit, mỗi khối RAM được nối trực tiếp với một nhân 18 bit.

**Dedicated Multiplier**: Các khối thực hiện phép nhân với đầu vào là các số nhị phân không dấu 18 bit.

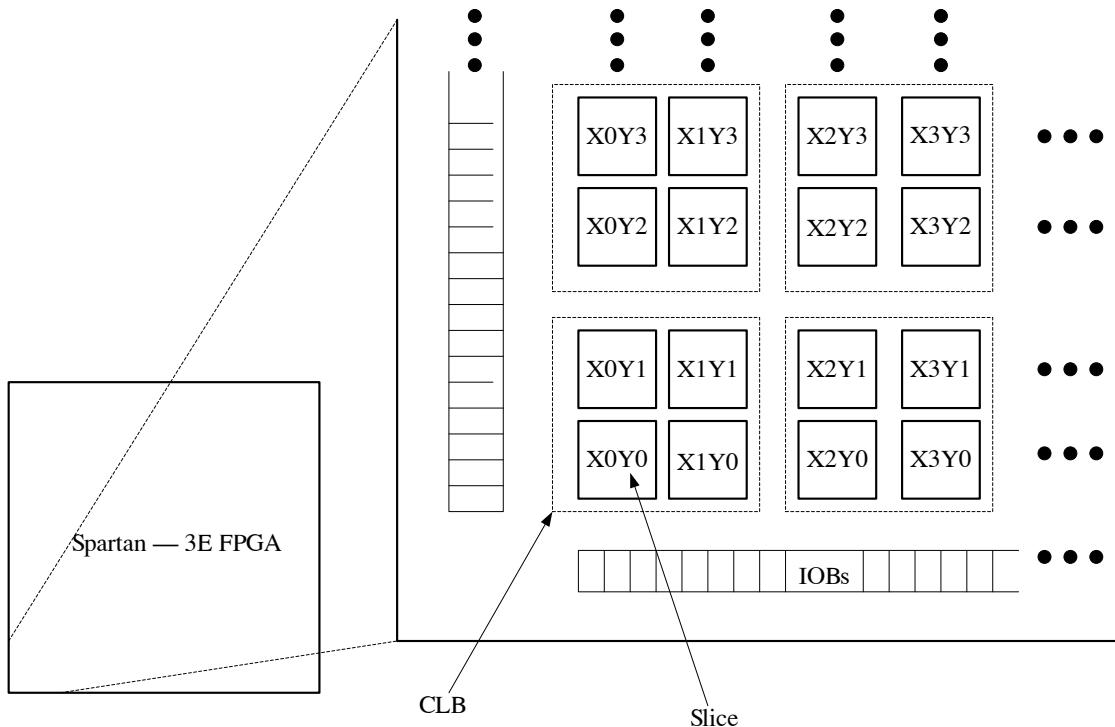
**DCM (Digital Clock Manager)** Các khối làm nhiệm vụ điều chỉnh, phân phối tín hiệu đồng bộ tới tất cả các khối khác. DCM thường được phân bố ở giữa, với hai khối ở trên và hai khối ở dưới. Ở một số đời FPGA Spartan 3E DCM còn được bố trí ở giữa.

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

## 2.1. Khối logic khả trình

Khối logic khả trình của FPGA Xilinx có tên gọi đầy đủ là *Configurable Logic Blocks (CLBs)*. CLBs là phần tử cơ bản cấu thành FPGA, là nguồn tài nguyên logic chính tạo nên các mạch logic đồng bộ lẫn không đồng bộ.

Mỗi CLB được cấu thành từ 4 Slices, mỗi Slice lại được cấu thành từ 2 LUTs (*Look Up Tables*). Phân bố của các CLB thể hiện ở Hình 4.4:



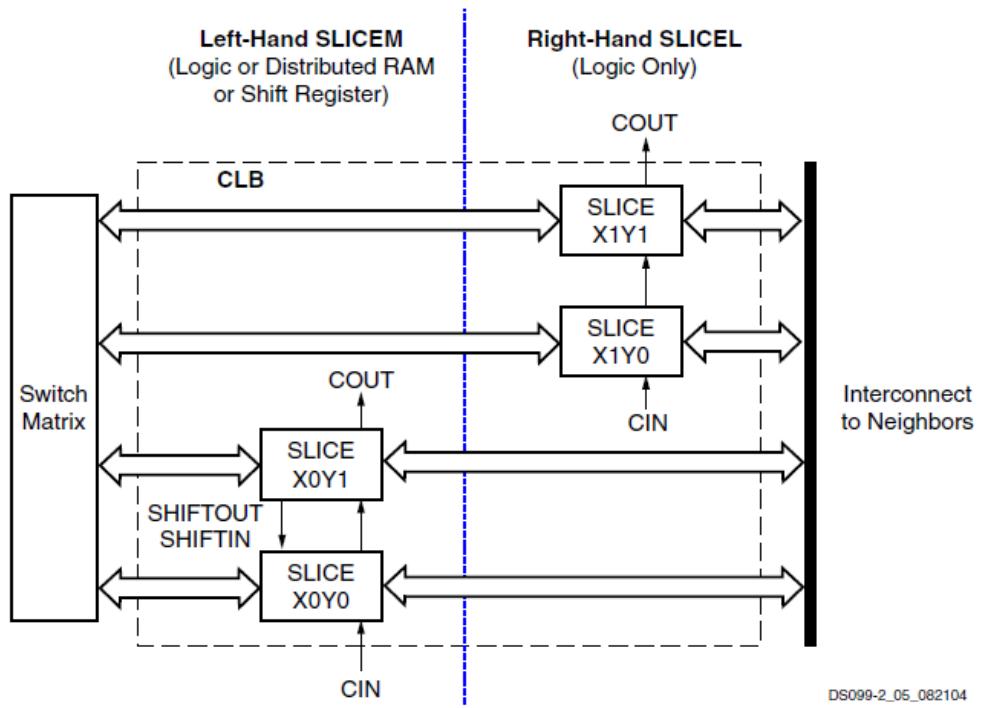
Hình 4-4. Phân bố của các CLB trong FPGA

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, đối với Spartan 3E số lượng hàng thay đổi từ 22 đến 76, số lượng cột từ 16 đến 56 tùy thuộc vào các gói cụ thể.

### 2.1.1. SLICE

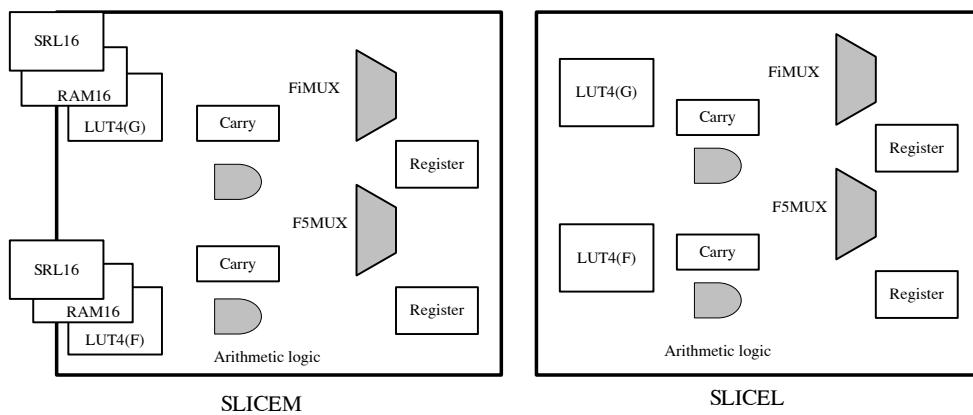
Mỗi CLB được cấu tạo thành từ 4 slices và các slices này chia làm hai nhóm trái và phải. Nhóm 2 slices bên trái có khả năng thực hiện các chức năng logic và làm việc như phần tử nhớ nên được gọi là *SLICEM (SLICE Memory)*. Nhóm 2 slices bên phải chỉ thực hiện được các chức năng logic nên được gọi là *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.



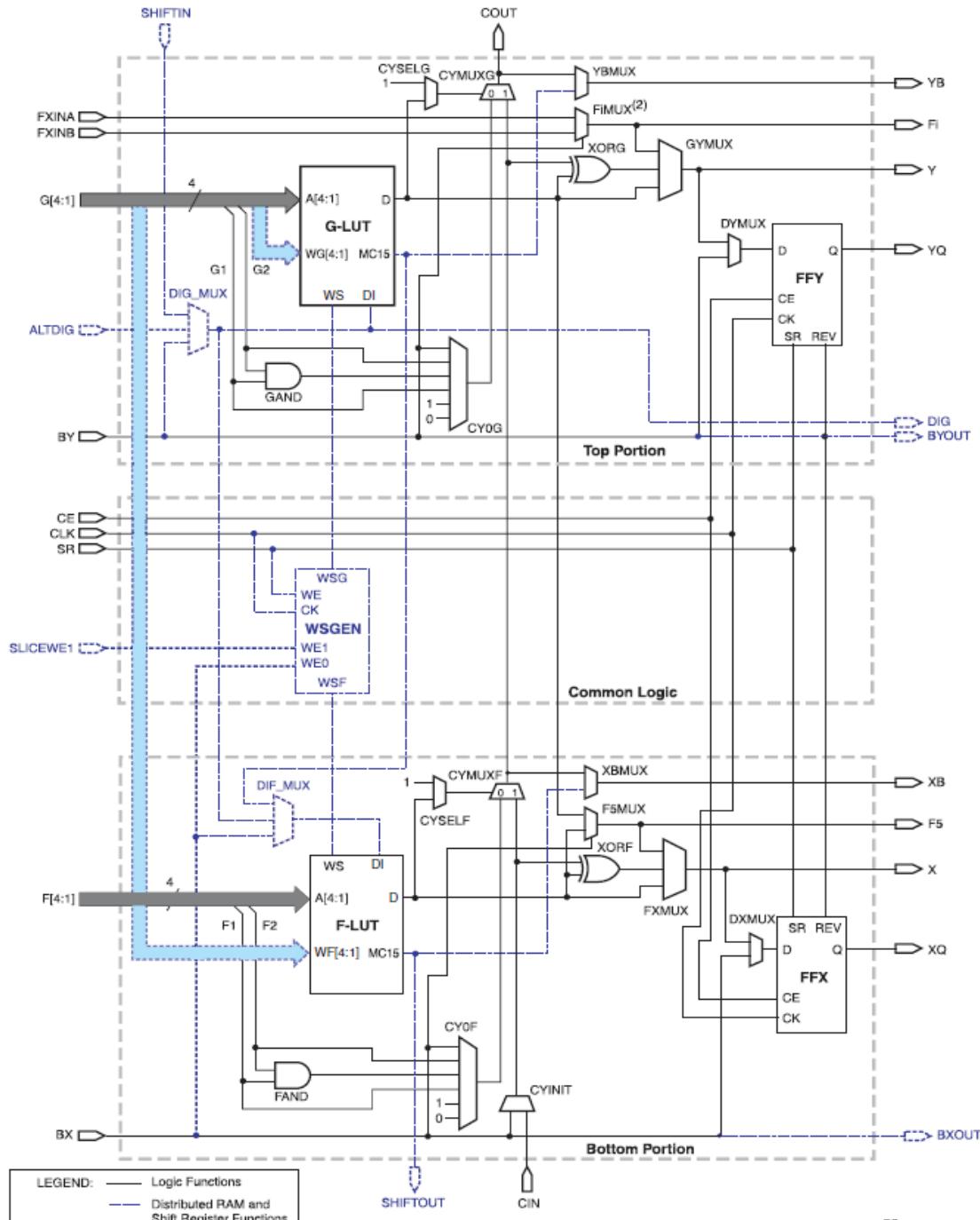
Hình 4-5. Bố trí slice bên trong một CLB

SLICEL chỉ thực hiện chức năng logic nên chỉ chứa các thành phần gồm LUT, chuỗi bít nhớ (Carry Chain), chuỗi số học (Arithmetic chain), các bộ chọn kênh mở rộng (*wide-multiplexer*) F5MUX và FiMUX, 2 Flip-flop. Còn đối với SLICEM thì ngoài các thành phần trên LUT còn có thể được cấu hình để làm việc như một thanh ghi dịch 16 bit Shift-Register (SRL16), hoặc RAM phân tán 16x1bit (*Distributed RAM*), như trình bày trên Hình 4.6.



Hình 4-6. Phân bố tài nguyên trong SLICEM và SLICEL

Cấu trúc chi tiết của một Slices được thể hiện ở hình dưới đây:



Hình 4-7. Cấu trúc chi tiết của Slice

Những đường gạch đứt thể hiện những kết nối tới các tài nguyên mà chỉ SLICEM mới có, những đường gạch liền chỉ những kết nối mà cả hai dạng SLICEs đều có.

Mỗi một slice chia làm hai phần với cấu trúc gần như nhau ở phần trên và phần dưới, mỗi phần chứa các khối chức năng giống nhau nhưng được ký hiệu khác nhau, ví dụ G-LUT chỉ LUT ở phần trên, F-LUT chỉ LUT ở phần dưới . Tín hiệu đồng bộ CLK, tín hiệu cho phép của xung nhịp CE (*Clock Enable*), tín hiệu cho phép ghi dữ liệu vào SLICEM SLICEWE1 và tín hiệu RS (*Reset/Set*) là các tín hiệu dùng chung cho cả phần trên và phần dưới của SLICE.

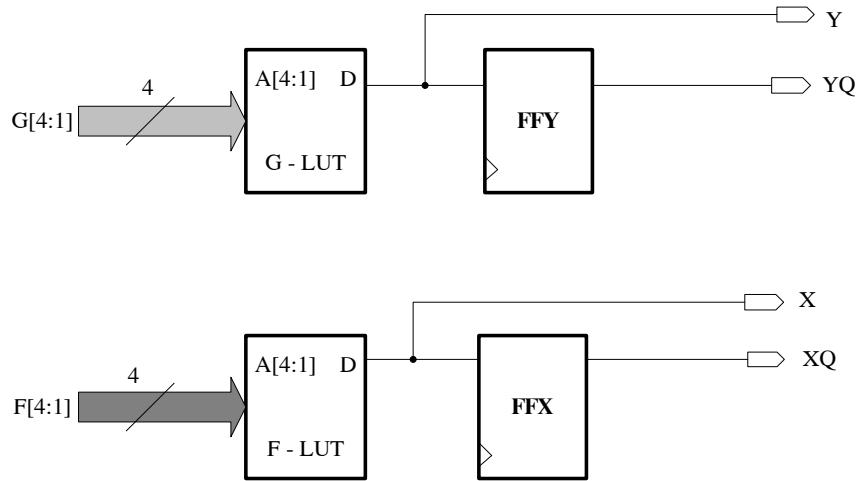
*Các đường dữ liệu cơ bản trong Slices* là các đường bắt đầu từ các đầu vào F[4:1] và G[4:1] thẳng tới F-LUT và G-LUT tương ứng, tại đây sẽ thực hiện hàm logic tổ hợp theo yêu cầu và gửi ra ở các đầu ra D. Từ đây đầu ra D được gửi ra các cổng ra của SLICE thông qua các đường sau:

- Kết thúc trực tiếp tại các đầu ra X, Y và nối ra ngoài với ma trận kết nối.
- Thông qua FMUX (GMUX) rồi DMUX làm đầu vào cho phần tử nhớ FFX (FFY) sau đó gửi ra thông qua các đầu ra QX (QY) tương ứng của các phân tử nhớ.
- Điều khiển CYMUXF (CYMUXG) của chuỗi bit nhớ (chi tiết về *Carry chain* xem mục 2.1.5).
- Gửi tới cổng XORF (XORF) để tính tổng hoặc tích riêng trong chuỗi nhớ.
- Làm đầu vào cho F5MUX (FIMUX) trong trường hợp thiết kế các khối logic, các chuỗi nhớ, thanh ghi dịch, RAM mở rộng.

Bên cạnh các đường dữ liệu cơ bản trên thì trong Slice tồn tại các đường dữ liệu “tắt” bắt đầu từ các đầu vào BX, BY và kết thúc qua một trong những đường sau:

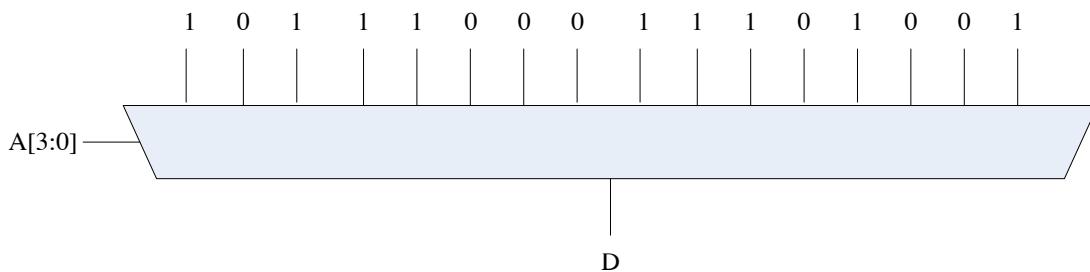
- Bỏ qua cả LUT lẫn phần tử nhớ và kết thúc ở các đầu ra BXOUT, BYOUT rồi ra ma trận kết nối.
- Bỏ qua LUT nhưng làm đầu vào cho các phân tử nhớ và kết thúc ở các đầu ra QX, QY.
- Điều khiển F5MUX hoặc FiMUX.
- Thông qua các bộ chọn kênh, tham gia như một đầu vào của chuỗi bit nhớ.
- Làm việc như đầu vào DI của LUT (khi LUT làm việc ở chế độ Distributed RAM hay Shift Register).
- BY có thể đóng vai trò của tín hiệu REV cho phân tử nhớ (xem chi tiết về REV tại mô tả về phân tử nhớ)

## 2.1.2. Bảng tham chiếu LUT



Hình 4-8. Phân bố các LUT trên một Slice

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.

2 LUTs có trong SLICEM có thể được cấu trúc để làm việc như 16x1 RAM gọi là Ram phân tán (*Distributed RAM*) hoặc được cấu trúc để làm việc như một thanh ghi dịch 16-bit SHL16. Cấu trúc của các phần tử này sẽ được nghiên cứu kỹ hơn ở phần 2.1.7 và 2.1.8

Các LUT có thể được kết hợp với nhau để thực hiện các hàm logic tùy biến có số lượng đầu vào lớn hơn 4 thông qua các bộ chọn kênh mở rộng. Ở các thế hệ FPGA về sau này, nguyên lý làm việc của LUT vẫn không thay đổi nhưng số lượng đầu vào có thể nhiều hơn, ví dụ trong Virtex-5, số lượng đầu vào là 6.

### 2.1.3. Phần tử nhớ

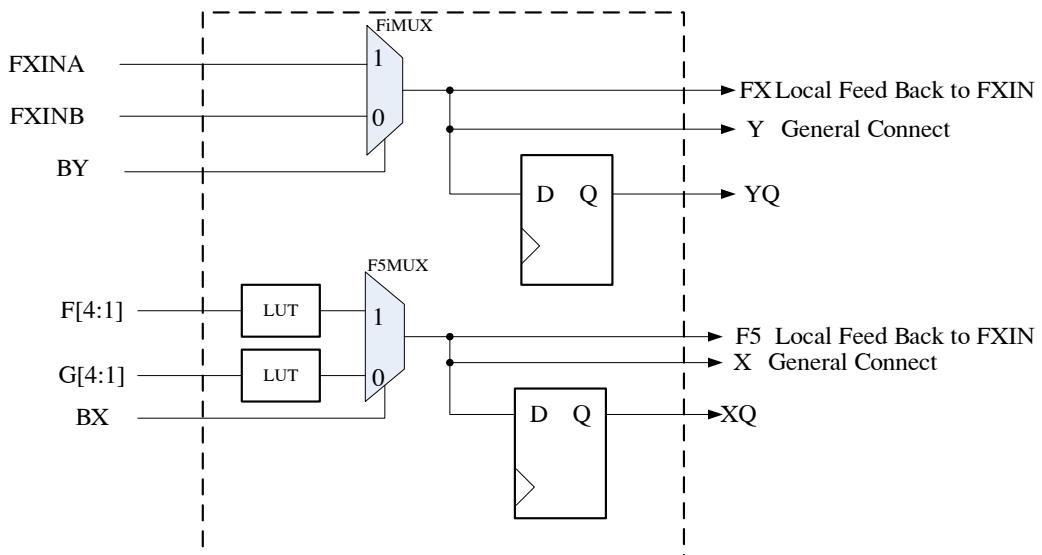
Phần tử nhớ (*Storage elements*) có trong CLBs là Flip-Flop FFX, FFY có thể được cấu hình để làm việc như D-flip-flop hoặc Latch, làm việc với các tín hiệu điều khiển đồng bộ hoặc không đồng bộ vì vậy cấu trúc của phần tử nhớ trong FPGA phức tạp hơn so với cấu trúc của D-flip-flop thông thường. Các đầu ra QX, QY của phần tử nhớ cũng là các đầu ra của Slices. Trong phần lớn các ứng dụng thường gặp phần tử nhớ được cấu trúc để làm việc như D-flipflop đồng bộ.

Các cổng giao tiếp của một phần tử nhớ bao gồm:

- D, Q là các cổng dữ liệu vào và ra tương ứng.
- C là cổng vào xung nhịp đồng bộ.
- GE (*Gate Enable*) cổng cho phép xung nhịp C khi làm việc ở chế độ latch
- CE (*Clock Enable*) cổng cho phép xung nhịp C khi làm việc ở chế độ flip-flop
- S, R là các cổng Set và Reset đồng bộ cho Flip-flop.
- PRE, CLR Cổng Set và Clear không đồng bộ
- RS Cổng vào của CLB cho S, R, PRE, hay CLR.
- REV Cổng vào pha nghịch so với RS, thường có đầu vào từ BY, có tác dụng ngược với RS. Khi cả hai cổng này kích hoạt thì giá trị đầu ra của phần tử nhớ bằng 0.

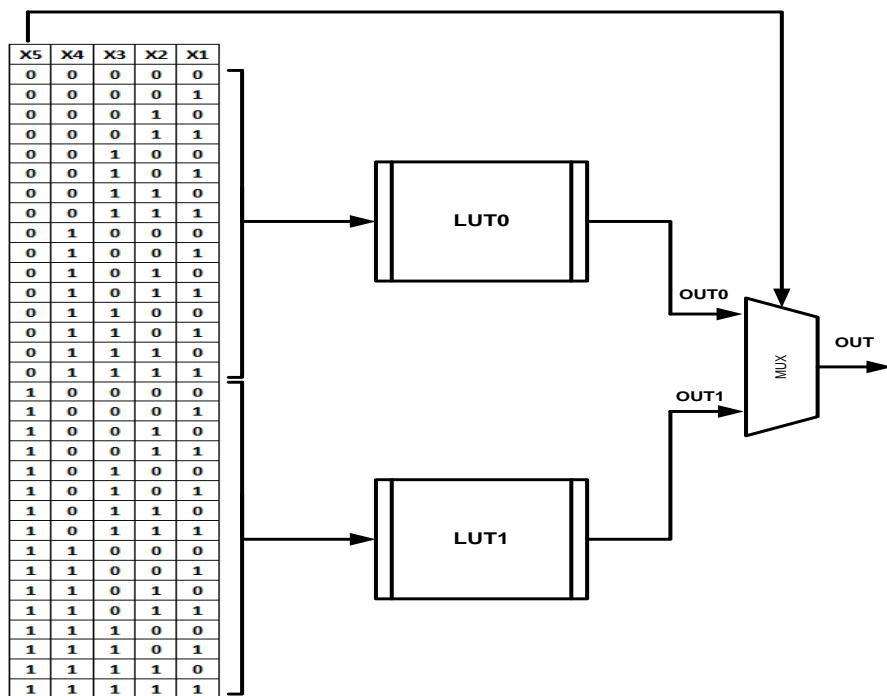
### 2.1.4. Bộ chọn kênh mở rộng

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, CLBs 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  $OUT = F(X_1, X_2, X_3, X_4, X_5)$  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  $X_5$  bằng 0, ta gọi hàm này có tên là:

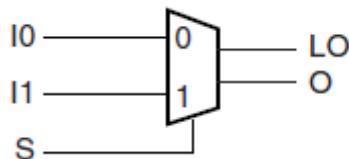
$$OUT_0 = F(X_1, X_2, X_3, X_4, 0) = F_0(X_1, X_2, X_3, X_4)$$

phần dưới với tất cả các giá trị của  $X_5$  bằng 1, ta gọi hàm này có tên là:

$$OUT_1 = F(X_1, X_2, X_3, X_4, 1) = F_1(X_1, X_2, X_3, X_4)$$

Hai hàm  $F_0$ ,  $F_1$  là các hàm 4 đầu vào được thực hiện ở tương ứng bởi LUT1, LUT2. Tín hiệu  $X_5$  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.

$$\begin{aligned} OUT &= F_0(X_1, X_2, X_3, X_4) \text{ nếu } X_5 = 0 \\ &= F_1(X_1, X_2, X_3, X_4) \text{ nếu } X_5 = 1 \end{aligned}$$



Hình 4-12. Cấu tạo của F5MUX

F5MUX được thiết kế dựa trên nguyên lý trên nhưng trên FPGA thực tế ngoài công 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.

### 2.1.5. Chuỗi bit nhớ và chuỗi số học

Trong Spartan-3E cũng như trong các FPGA thế hệ sau này đều được tích hợp các chuỗi bit nhớ (*carry chain*) và các chuỗi số học (*arithmetic chain*) đặc biệt, các chuỗi này kết hợp với các LUT được sử dụng tự động hầu hết trong các

phép toán số học thường gặp như cộng, nhân, góp phần rất lớn vào việc tăng tốc cho các phép toán này, đồng thời tiết kiệm tài nguyên logic (LUTs). Các chuỗi này được tạo thành bằng các khối chọn kênh và các cổng logic riêng biệt, các phần tử đó cũng có thể được sử dụng độc lập để thực hiện các hàm logic đơn giản khác.

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 CY nào của các Slices.

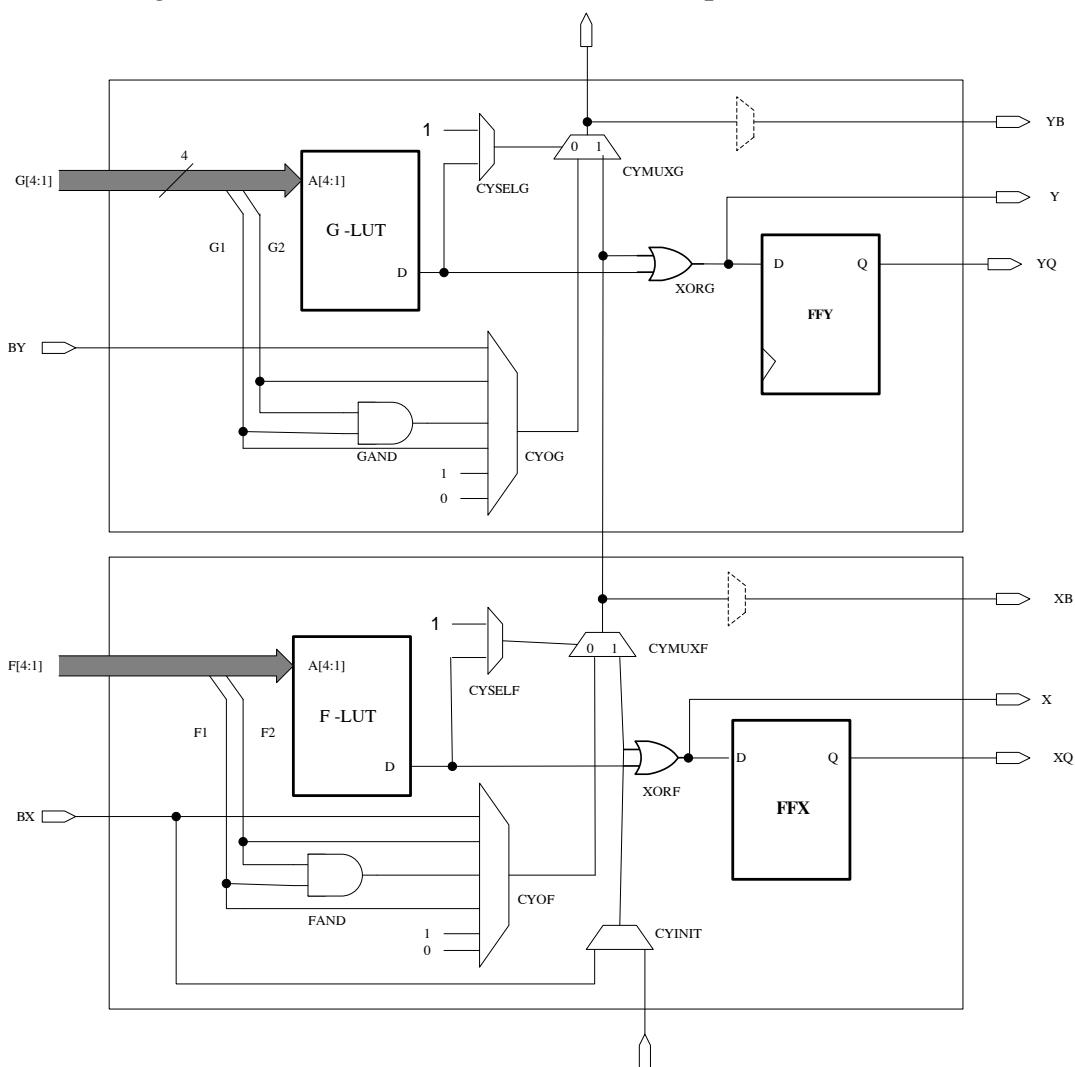
Các chuỗi số học logic bao gồm chuỗi thực hiện hàm XOR với các cổng XORG, XORF phân bố ở phần trên và phần dưới của Slice, chuỗi AND với các cổng GAND, FAND. Các chuỗi này kết hợp với các LUT để thực hiện phép nhân hoặc tạo thành các bộ đếm nhị phân.

Các thành phần cơ bản của Chuỗi bit nhớ và Chuỗi số học bao gồm:

- CYINIT: nằm ở phần dưới của Slice, chọn tín hiệu từ BX đầu vào của (Slice) nếu là điểm đầu của chuỗi hoặc CIN từ Slice kế cận trong trường hợp muốn kéo dài chuỗi nhớ.
- CYOF/CYOG: Khối chọn nhớ phát sinh tương ứng ở phần dưới và trên của Slices, có khả năng chọn một trong số các đầu vào F1,F2/ G1,G2 là đầu vào của các LUT, đầu vào từ cổng AND, đầu vào từ BX/BY, đầu vào với các giá trị cố định 0, 1 nếu sử dụng như các hàm logic thông thường
- CYMUXG lựa chọn giữa hai dạng nhớ là CYINIT nếu là nhớ lan truyền (carry propagation), CYOG nếu là nhớ phát sinh (carry generation), tương ứng với giá trị từ CYSELG là 0 và 1. Đầu ra COUT hoặc cổng YB của Slice.
- CYMUXF lựa chọn giữa hai dạng nhớ là CMUXF nếu là nhớ lan truyền (carry propagation), CYOF nếu là nhớ phát sinh (carry generation), tương ứng với giá trị từ CYSELF là 0 và 1. Đầu ra tới đầu vào của chuỗi nhớ tại phần trên của Slice hoặc cổng XB của Slice
- CYSELF/G lựa chọn giữa một trong hai tín hiệu: đầu ra của F/GLUT nếu là nhớ phát sinh, và giá trị bằng 1 cho trường hợp nhớ lan truyền.
- XORF cổng XOR cho chuỗi thực hiện hàm cộng bù 2 thuộc nửa dưới của Slice, hai đầu vào từ đầu ra của FLUT và từ đầu vào chuỗi nhớ CYINIT,

đầu ra của cổng được gửi tới cổng D của phần tử nhớ FFX hoặc trực tiếp tới đầu ra X của Slices

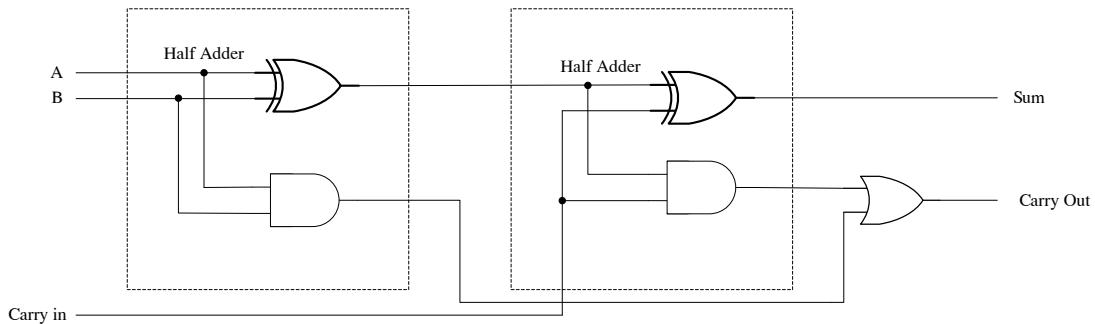
- XORG cổng XOR cho chuỗi thực hiện hàm cộng bù 2 thuộc nửa trên của Slice, hai đầu vào từ đầu ra của GLUT và từ đầu vào chuỗi nhớ CMUXF, đầu ra của cổng được gửi ra cổng D của phần tử nhớ FFY hoặc trực tiếp tới cổng Y của Slice
- FAND cổng AND cho chuỗi thực hiện hàm logic nhân thuộc nửa dưới của Slice, các đầu vào lấy trực tiếp từ đầu vào F1, F2 của các LUT, đầu ra được gửi tới CYOF để trở thành tín hiệu nhớ phát sinh cho chuỗi bit nhớ.
- FAND cổng AND cho chuỗi thực hiện hàm logic nhân thuộc nửa dưới của Slice, các đầu vào lấy trực tiếp từ đầu vào F1, F2 của các LUT, đầu ra được gửi tới CYOF để trở thành tín hiệu nhớ phát sinh cho chuỗi bit nhớ.



Hình 4-13. Chuỗi bit nhớ

Minh họa về cách sử dụng các chuỗi này để tối ưu hóa tài nguyên và tăng tốc cho FPGA như sau:

Về phép cộng, thiết kế điển hình của bộ cộng bắt đầu từ thiết kế quen thuộc của bộ cộng 1 bit đầy đủ gọi là FULL\_ADDER.



Hình 4-14. Sơ đồ logic truyền thống của FULL\_ADDER

Với cấu trúc này để thực hiện một FULL\_ADDER trên FPGA cần tối thiểu hai LUT, mỗi LUT sử dụng với 3 đầu vào A, B, CIN. Trên thực tế cấu trúc trên có thể được thiết kế khác đi nhằm tăng tốc thực hiện cũng như giảm thiểu tài nguyên bằng sơ đồ cộng thấy nhó trước.

Định nghĩa các tín hiệu sau

$$g = A \text{ and } B; \text{ generation carry - nhó phát sinh}$$

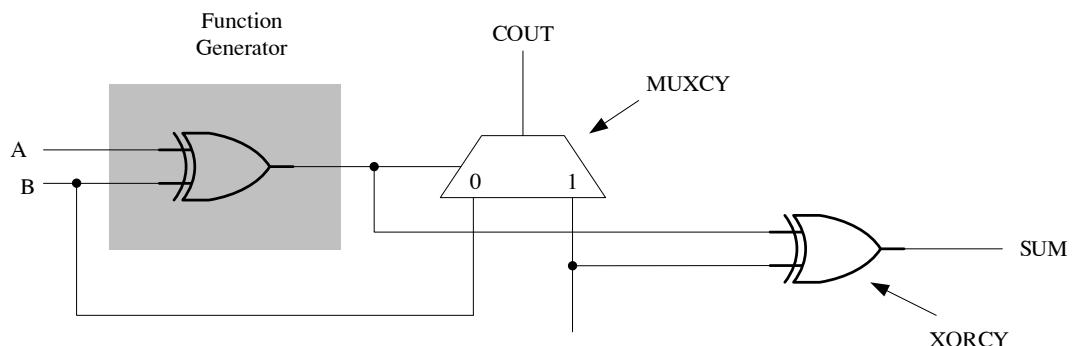
$$p = A \text{ xor } B; \text{ propagation delay - nhó lan truyền}$$

khi đó

$$\text{Sum} = p \text{ xor CIN}$$

$$\text{COUT} = (\text{CIN and (not } p)) \text{ or (a and } p);$$

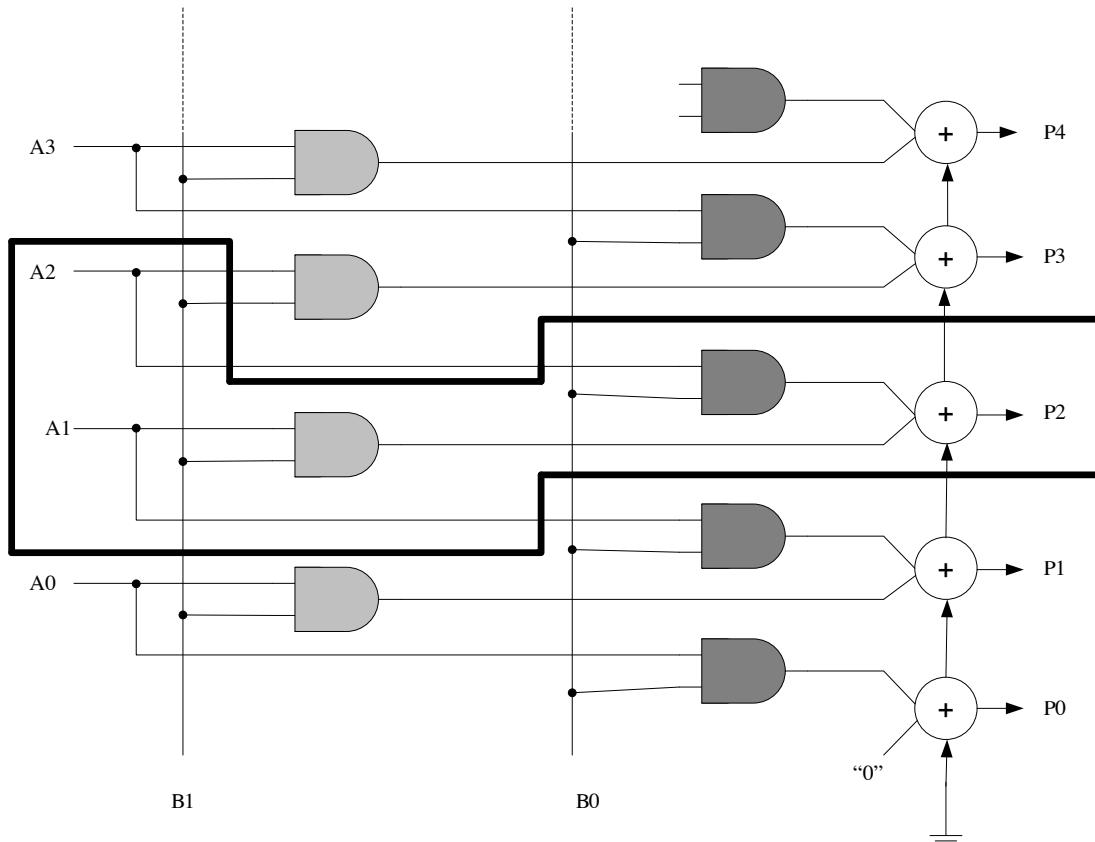
Từ hai công thức trên có thể thiết lập một sơ đồ khác của FULL\_ADDER như sau



Hình 4-15. Sơ đồ logic của FULL\_ADDER trên FPGA

Với sơ đồ trên ta có thể thấy thay vì sử dụng hai phần tử AND ta sử dụng một bộ chọn kênh 2 đầu vào. Quay lại với sơ đồ Hình 4.12 và so sánh với sơ đồ trên ta có thể thấy FULL\_ADDER có thể thực hiện với 1 LUT hai đầu vào và kết hợp với chuỗi bit nhớ có sẵn trong FPGA. Khi đó CYMUXF, CYMUXG đóng vai trò như bộ chọn kênh trên Hình 4.14. Bản thân LUT thực hiện hàm XOR. Với cách thiết kế như vậy mỗi Slice có thể tạo được chuỗi nhớ cho 2 bit. Các đầu vào CIN, COUT của các CLB được nối trực tiếp và được tối ưu hóa để trễ lan truyền gần như bằng 0 chuỗi nhớ thực hiện với một tốc độ rất nhanh.

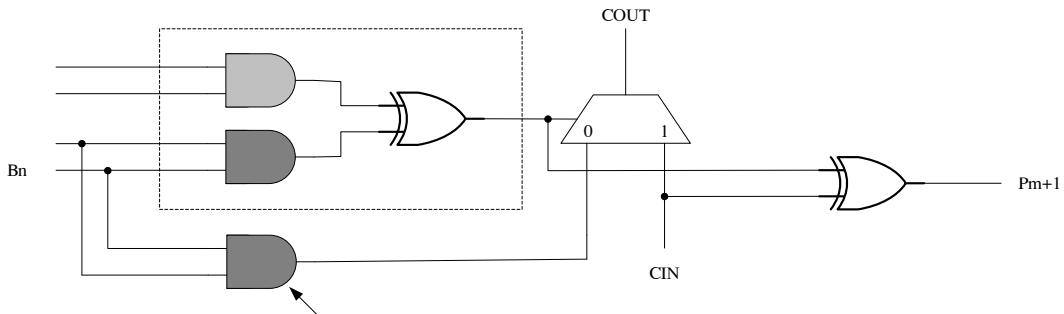
- Về phép nhân, để thực hiện phép nhân thì phải thực hiện việc tính các tích riêng (*partial products*) sau đó cộng các tích này với nhau để tạo thành kết quả đầy đủ (*full product*). Khi biểu diễn các số dưới dạng nhị phân, tích riêng hoặc bằng số bị nhân nếu như bit nhân là 1 và tích riêng bằng 0 nếu như bit nhân bằng 0. Để cộng các tích riêng thì phải sử dụng chuỗi bít nhớ tương tự như ở ví dụ trên. Còn tích riêng được tạo bởi các phần tử AND. Xét sơ đồ tính tích riêng như ở hình dưới đây:



Hình 4-16. Cách tính tổng các tích riêng trong phép nhân

Sơ đồ trên biểu diễn phép nhân  $B_0B_1$  với  $A_0A_1A_2A_3$ . Đối với mạch dùng để tính cho một bit của tích riêng  $P_2$  được khoanh vùng trên Hình 4.16 thì cần tối thiểu 1 cổng XOR 3 đầu vào (tương đương 2 cổng XOR 2 đầu vào) và hai cổng AND 2 đầu vào. Nếu thực hiện theo sơ đồ trên và chỉ dùng LUT có thể thấy mỗi đầu vào phần tử XOR xuất phát từ 1 phần tử AND, điều đó có nghĩa là mỗi phần tử AND chiếm 1 LUT. Như vậy nếu nhân hai số 8-bit ta phải cần  $8 \times 8 = 64$  LUT = 32 Slices.

Quan sát lại sơ đồ 4.7, nếu sử dụng các tài nguyên khác của chuỗi bit nhớ có thể thiết kế như sau:



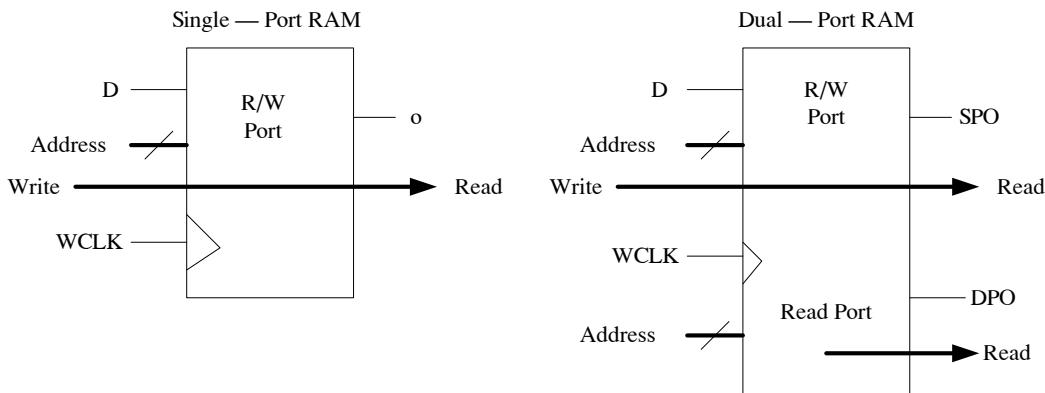
Hình 4-17. Tối ưu hóa khối MULT\_AND dùng chuỗi bit nhớ

Với sơ đồ như trên tổng  $A_m * B_n + A_{m+1} * B_n$  được thực hiện trong 1 LUT, ngoài ra tích  $A_{m+1} * B_n$  được lặp lại nhờ phần tử AND (FAND hay GAND) để gửi tới bộ chọn kênh (CYMUXF, hoặc CYMUXG). Với cách làm như vậy thì số lượng LUT đối với phép nhân 8x8 giảm xuống chỉ còn một nửa tức là 32LUTs = 16 Slices. Bên cạnh đó các phần tử MULT\_AND được tối ưu hóa về độ trễ giữa các kết nối nên có thể làm việc rất nhanh. Tuy vậy sơ đồ nhân như trên chỉ được thực hiện với các phép nhân có số bit đầu vào nhỏ. Với các phép nhân có số lượng bit lớn hơn thì FPGA sẽ sử dụng các khối nhân chuyên dụng (*dedicated multipliers*) 18-bit x 18-bit sẽ được trình bày ở dưới.

### 2.1.6. RAM phân tán

Trong mỗi CLB của Xilinx FPGA có chứa  $4 \times 16 = 64$  bit RAM tương ứng với 4 LUT nằm trong 2 SLICEM của CLB. Phần RAM có thể sử dụng như một khối 64-bit RAM một cổng (Single-port RAM) hoặc khối 32-bit RAM hai cổng(Dual-port RAM), khi đó khối RAM được tạo thành từ hai mảng nhớ 32-bit và lưu trữ dữ liệu y hệt nhau. Vì các RAM này phân bố rải rác theo CLB bên trong cấu trúc của FPGA nên chúng được gọi là các RAM phân tán (*Distributed RAM*).

*RAM*) để phân biệt với các khối RAM nằm tập trung và có kích thước lớn hơn khác là Block RAM.



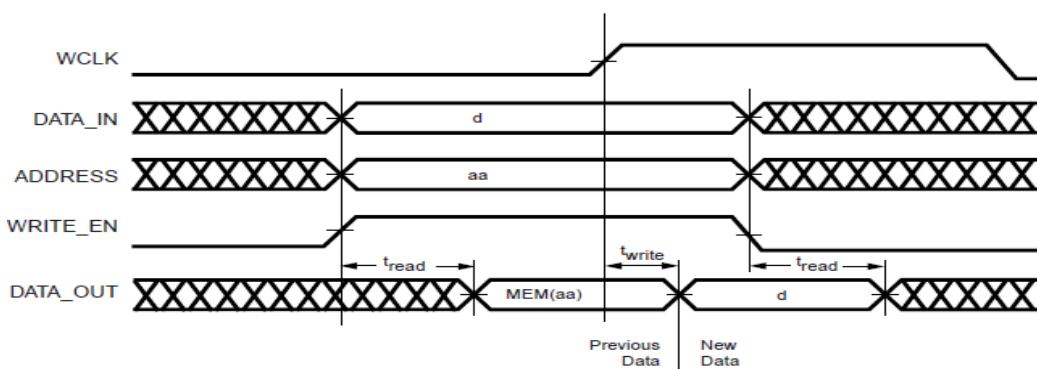
Hình 4-18. *RAM phân tán trong FPGA*

RAM phân tán trong FPGA có thể sử dụng ở một trong hai dạng như hình vẽ trên. Đôi với kiểu *single-port RAM* thì có một ghi dữ liệu 1 cổng đọc dữ liệu. Đôi với *dual-port RAM* thì có 1 cổng đọc ghi dữ liệu và một cổng chỉ thực hiện đọc dữ liệu từ RAM.

Đôi với thao tác ghi dữ liệu cho cả hai kiểu RAM được thực hiện đồng bộ trong 1 xung nhịp WCLK, tín hiệu cho phép ghi là WE (Write Enable, theo ngầm định tích cực nếu WE = '1'). Đôi với Dual-port RAM thì mỗi động tác ghi sẽ thực hiện ghi dữ liệu từ cổng D vào hai phần nhớ của RAM.

Thao tác đọc dữ liệu ở hai dạng RAM đều được thực hiện không đồng bộ và thuận túy sử dụng các khối logic tổ hợp, thời gian trễ của thao tác đọc dữ liệu bằng thời gian trễ tổ hợp và thông thường được tối ưu nhỏ hơn so với thời gian 1 xung nhịp đồng hồ.

Giản đồ sau thể hiện thao tác đọc ghi dữ liệu:

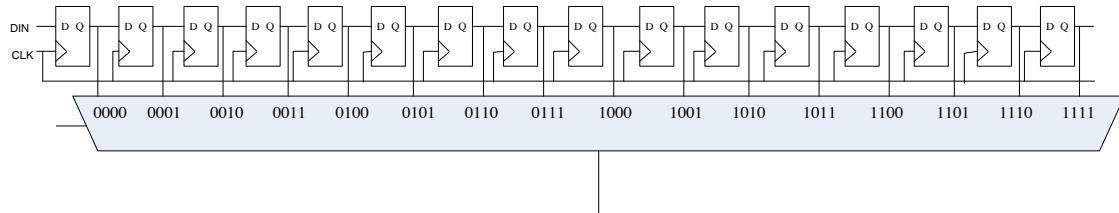


Hình 4-19. *Thao tác đọc ghi dữ liệu của Distributed RAM trong Xilinx FPGA*

Tài nguyên RAM phân tán trong FPGA được sử dụng hết sức linh động, một khối CLB đơn lẻ có thể được cấu hình để tạo thành các khối 64x1, 32x2, 16x4 Distributed RAM, các đầu vào G[4:1] và F[4:1] được dùng như các đầu vào địa chỉ. Các khối RAM lớn hơn có thể cấu tạo bằng cách ghép tài nguyên trong các CLB khác nhau lại sử dụng các bộ chọn kênh mở rộng, khi đó các cổng BX, BY được sử dụng như các bit địa chỉ bổ xung.

### 2.1.7. Thanh ghi dịch

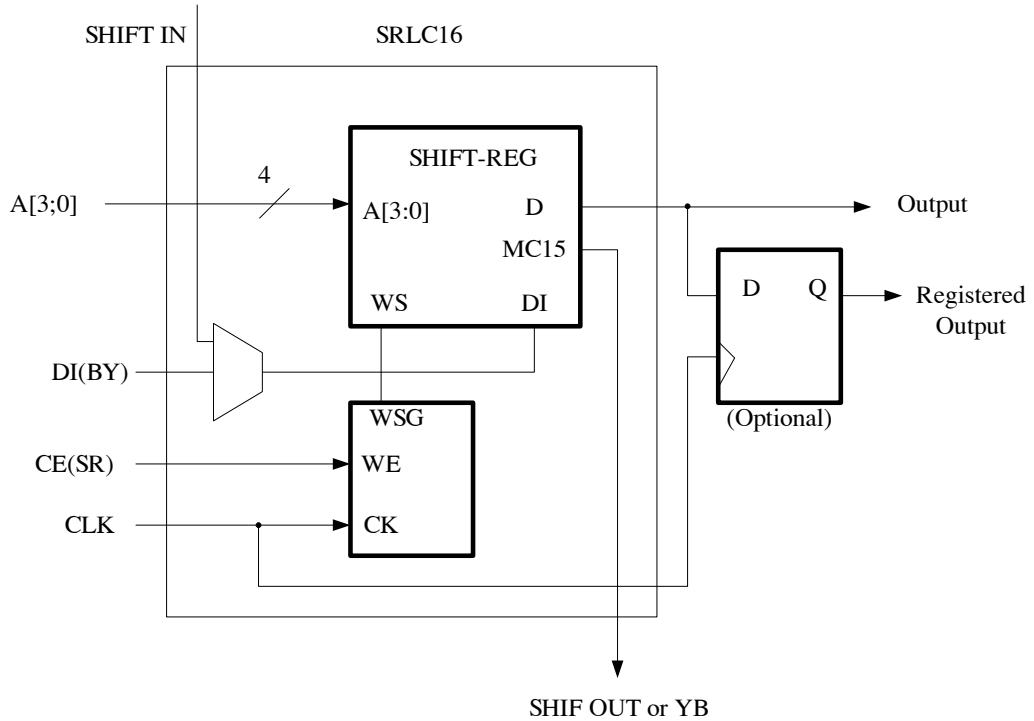
Một dạng sử dụng khác của các LUTG, và LUTF trong SLICEM là dùng như một thanh ghi dịch (*Shift Register*) 16 bit ký hiệu là SRL16.



Hình 4-20. Sử dụng LUT như thanh ghi dịch 16-bit

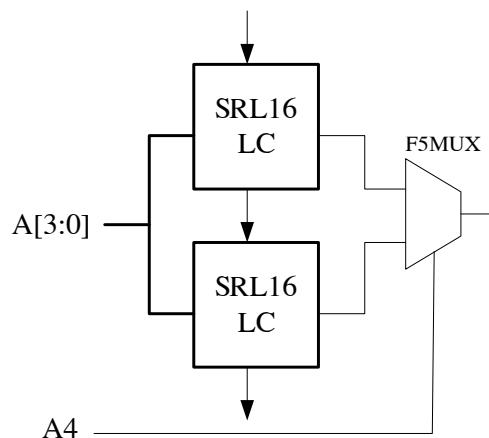
Khi sử dụng LUT như một thanh ghi dịch, cấu trúc của LUT về cơ bản giữ nguyên, các kênh chọn được nối với chuỗi các D flip-flop làm việc đồng bộ. Đầu ra D vẫn nhận giá trị tại đầu ra Q của D-flip-flop quy định bởi giá trị địa chỉ A[3:0], chính vì vậy SRL16 còn được gọi là thanh ghi dịch có địa chỉ. Ngoài đầu ra D thanh ghi dịch có đầu ra cuối cùng có tên là Q15 hoặc MC15 quy định trong thư viện các phần tử chuẩn của FPGA. Đầu vào DI có thể được bắt đầu từ cổng BY, BX hoặc đầu vào SHIFTIN từ ngoài CLB. Tín hiệu xung nhịp đồng bộ CLK và CE được lấy từ tín hiệu đồng bộ chung của Slices.

Đầu ra của MC15 của SRL16 có thể được nối tiếp với cổng SHIFTOUT của Slice hoặc YB. Đầu ra địa chỉ D có thể được gửi trực tiếp ra ngoài Slice hoặc thông qua FFX hoặc FFY, khi đó chuỗi dịch tính thêm một đơn vị, trên thực tế độ trễ của FFX, FFY thường nhỏ hơn so với độ trễ của các D-flip-flop trong thanh ghi dịch.



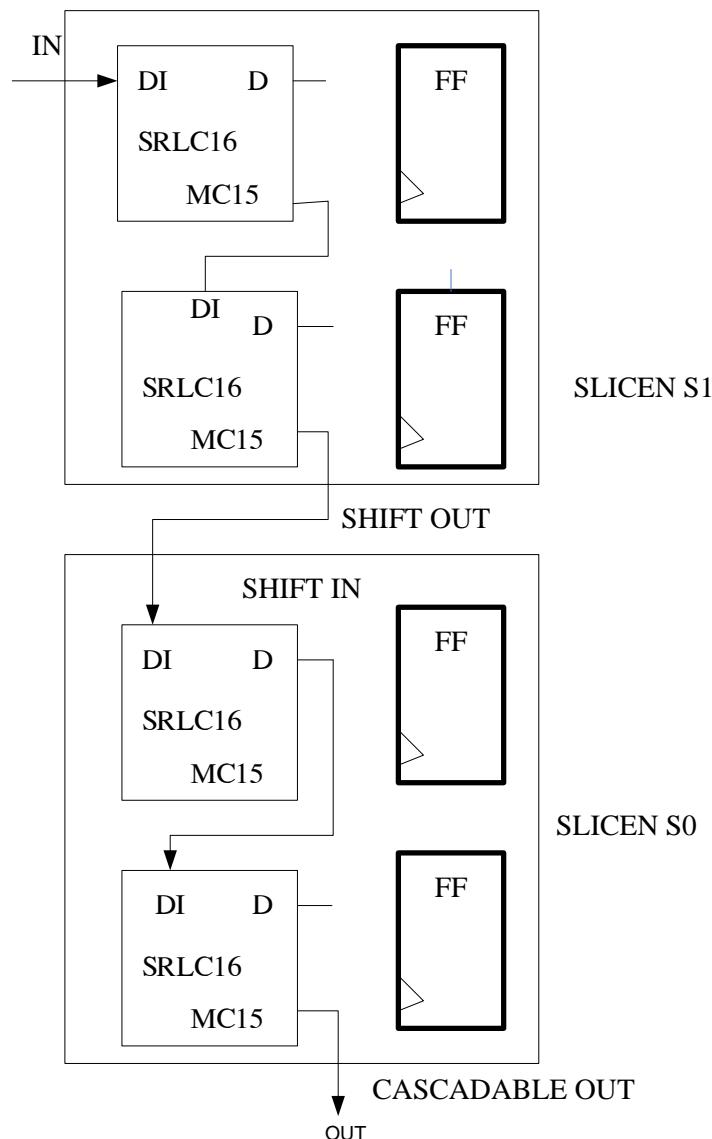
Hình 4-21. Cấu trúc của thanh ghi dịch trong FPGA

Thanh ghi dịch có thể sử dụng ở chế độ địa chỉ hoặc không. Khi muốn mở rộng thanh ghi dịch ở chế độ địa chỉ thì phải sử dụng thêm các Wide-multiplexers, ví dụ như trong một SLICEM, có thể kết hợp LUTG, LUTF ở chế độ thanh ghi dịch để tạo thành 32-bit thanh ghi dịch ở chế độ địa chỉ như ở hình sau:



Hình 4-22. Mở rộng thanh ghi dịch ở chế độ địa chỉ

Tương tự như vậy có thể sử dụng F6MUX, F7MUX... để mở rộng kích thước của thanh ghi dịch ở chế độ địa chỉ.



Hình 4-23. Mở rộng thanh ghi dịch ở chế độ không địa chỉ

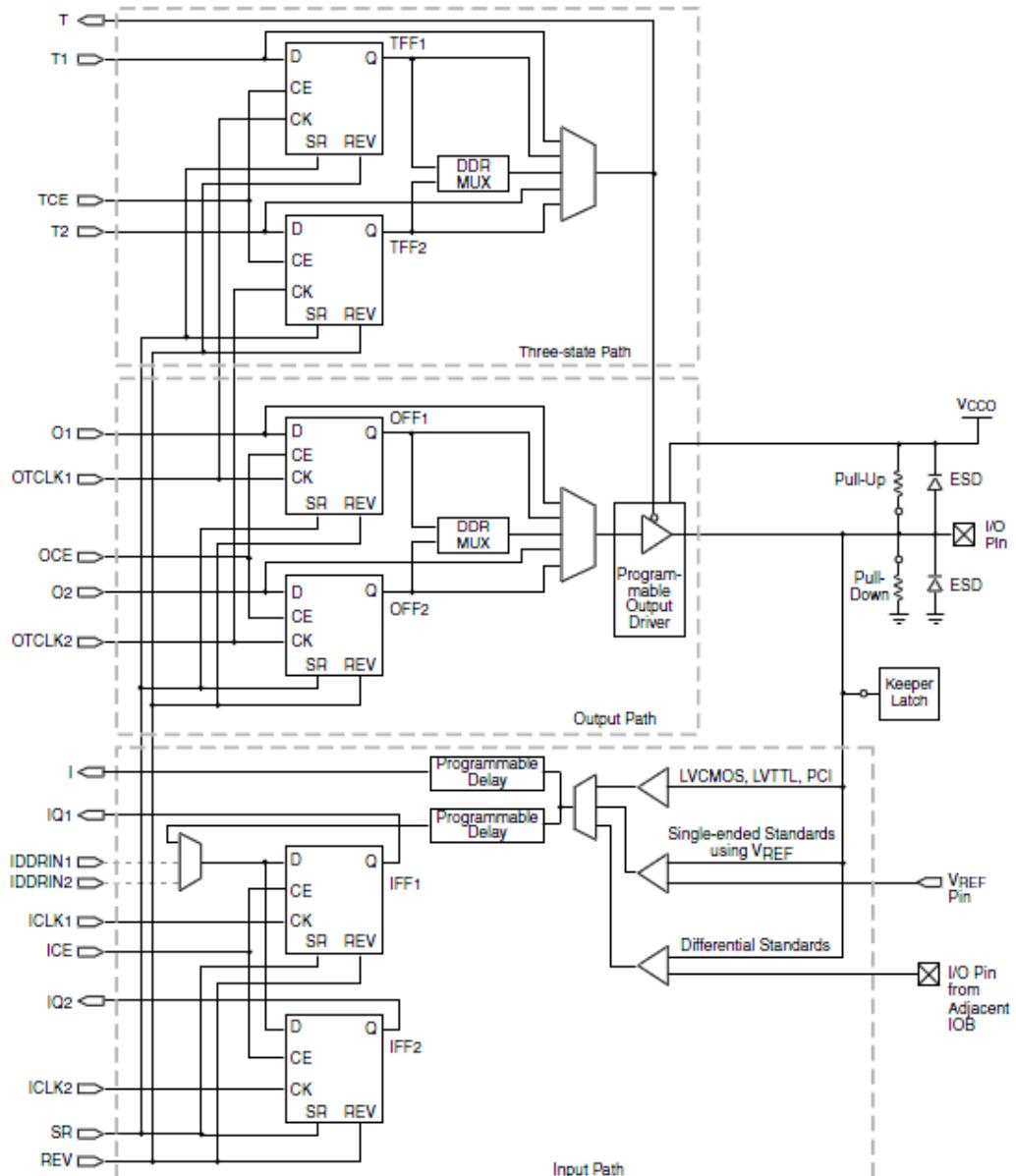
Khi sử dụng thanh ghi ở chế độ không địa chỉ, nghĩa là thực hiện dịch đủ 16 bit từ D-flip-flop thứ nhất cho đến đầu ra MC15 việc mở rộng thanh ghi dịch đơn giản là việc nối tiếp các đầu ra của thanh ghi dịch trước với đầu vào của thanh ghi dịch sau, ví dụ SHIFTOUT của CLB trên với SHIFTOUT của CLB dưới như hình vẽ dưới đây.

Cũng như đối với Ram phân tán hay chuỗi bit nhớ, thanh ghi dịch có các cổng kết nối được tối ưu hóa về mặt tốc độ làm việc, việc ghép nối các thanh ghi

liền kề không gây ra trễ đường truyền lớn. Việc sử dụng thanh ghi dịch trong Xilinx FPGA thường tự động nhưng người cũng có thể chủ động khai báo sử dụng tùy theo mục đích thiết kế.

## 2.2. Khối điều khiển vào ra

Sơ đồ nguyên lý của khối điều khiển vào ra (*Input/Output Block*) trong Spartan 3E được trình bày như ở hình dưới đây:



Hình 4-24. Sơ đồ nguyên lý của khối đệm vào ra IOB

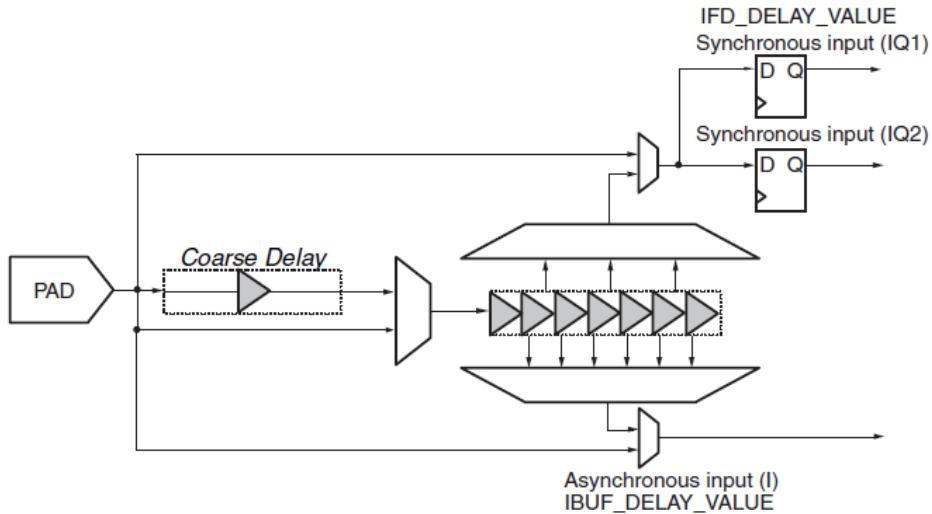
Các khối Input/Output Blocks (IOB) trong FPGA cung cấp các cổng vào ra lập trình được một chiều hoặc hai chiều giữa các chân vào ra của FPGA với các khối logic bên trong. Các khối một chiều là các khối Input-only nghĩa là chỉ đóng vai trò cổng vào, số lượng của các cổng này thường chiếm không nhiều khoảng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

Hình 4.24 mô tả sơ đồ tổng quan của một IOB, đối với các khối Input-only thì không có những phần tử liên quan đến Output. Một IOB điển hình có ba đường dữ liệu chính, đường *input*, đường *output*, đường cổng 3 trạng thái (*Three state path*), mỗi đường này đều chứa các khối làm trễ lập trình được và cặp phần tử nhớ có khả năng làm việc như Latch hoặc D-flipflop.

- Đường Input dẫn dữ liệu từ các chân vào ra của FPGA có thể qua hoặc không qua khối làm trễ khả trình vào gửi tới thằng chân dữ liệu I. Đường Input thứ hai đi qua cặp phần tử nhớ tới các chân IQ1, IQ2. Các chân I, IQ1, IQ2 dẫn trực tiếp tới phần logic bên trong của FPGA. Khi sử dụng các khối làm trễ khả trình thì thường được cấu hình để đảm bảo tối ưu cho yêu cầu về giá trị hold time của phần tử nhớ.
- Đường Output bắt đầu tại các chân O1, O2 có nhiệm vụ dẫn luồng dữ liệu từ các khối logic bên trong tới các chân vào ra của FPGA. Đường dẫn trực tiếp là đường dẫn từ O1, O2 qua khối chọn kênh tới khối dẫn 3 trạng thái tới các chân vào ra. Đường dẫn thứ hai ngoài các phần tử trên còn đi qua hai phần tử nhớ. Đầu ra còn được nối với hệ thống pull-up, pull-down resistors để đặt các giá trị cổng ra là logic 1 hoặc 0.
- Đường 3 trạng thái xác định khi nào đường dẫn ra là trạng thái trở kháng cao. Đường trực tiếp từ các chân T1, T2 tới khối điều khiển 3 trạng thái. Đường gián tiếp đi qua hai phần tử nhớ trước khi tới khối điều khiển 3 trạng thái.

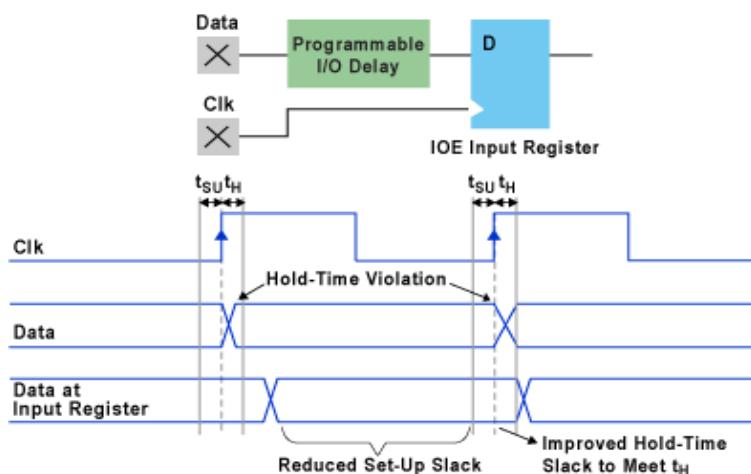
### 2.2.1. Cổng vào với độ trễ khả trình

Mỗi một đường dữ liệu vào (*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, khôi 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 phần 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 khôi 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 4-25. Khối làm trễ khả trình

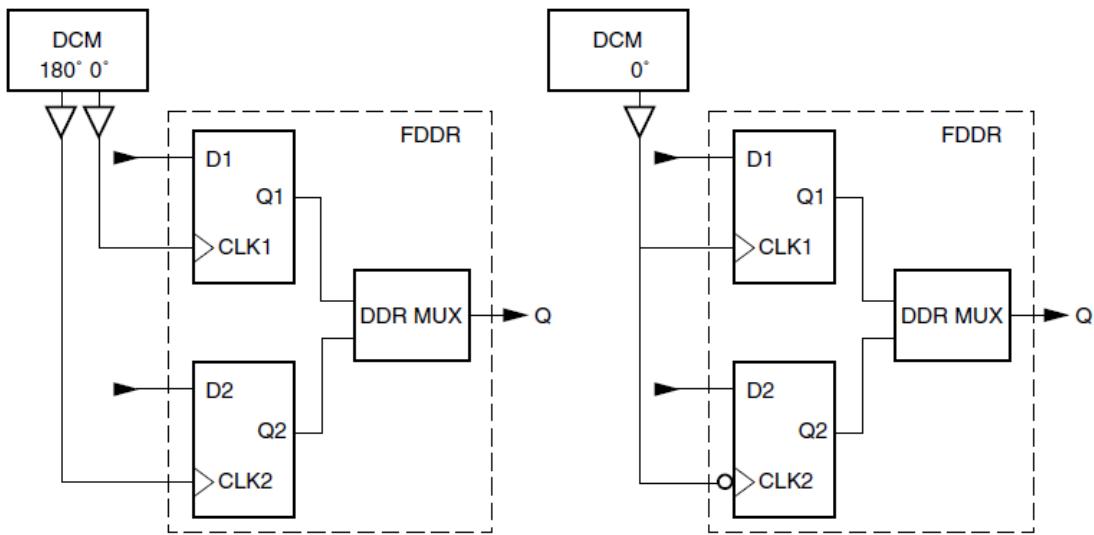
Một trong những ứng dụng của khối làm trễ là đảm bảo không vi phạm điều kiện của  $T_{hold}$  khi phần tử nhớ hoạt động ( $T_{hold}$  là thời gian tối thiểu cần giữ ổn định dữ liệu sau thời điểm kích hoạt của xung nhịp đồng bộ), ví dụ như ở hình vẽ sau:



Hình 4-26. Điều chỉnh đầu vào bằng khối làm trễ khả trình

### 2.2.2. Cổng vào ra ở chế độ DDR

Khái niệm DDR (*Double Data Rate transmission*) 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 sùn lên và sùn xuống của xung nhịp. Với cả 3 đường dữ liệu có trong IOB, mỗi đường đều có một cặp phần tử nhớ cho phép thực hiện truyền dữ liệu theo phương thức DDR.



Hình 4-27. Nguyên lý DDR

Hình 4.27 thể hiện các thức hiện thực DDR trong FPGA. Hai phần tử nhớ hoạt động ở chế độ Flip-flop. Đầu ra được nối với Multiplexer DDR MUX để điều khiển vào ra dữ liệu. Cùng một thời điểm có hai xung nhịp đồng hồ lệch pha nhau  $180^\circ$  gửi tới đầu vào xung nhịp CLK1, CLK2 với chu kỳ T của Flip-flops. Giả sử tại thời điểm sùn dương của tín hiệu CLK1 Flip-Flop 1 hoạt động, sau đó nửa chu kỳ tại thời điểm sùn âm của xung nhịp CLK1 tương ứng với sùn dương của CLK2 thì Flip-flop 2 làm việc. Như vậy chu kỳ nhận/gửi dữ liệu là  $T/2$  hay tốc độ nhận/gửi dữ liệu tăng gấp đôi.

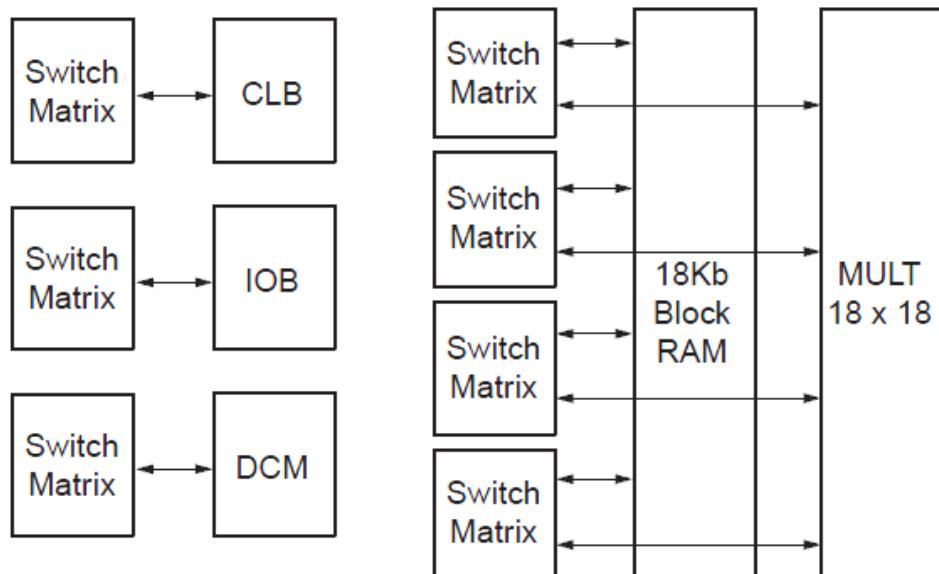
Để tạo ra hai xung nhịp lệch pha nhau có thể dùng khối DCM (Digital Clock Manager) từ một tín hiệu xung nhịp chuẩn sinh ra tín hiệu xung nhịp thứ hai bằng cách dịch pha  $180^\circ$  (hình bên trái). Phương pháp này đạt được độ trễ xung nhịp (*Clock skew*) thấp nhất. Bên cạnh đó phương pháp thứ hai như mô tả ở hình bên phải là dùng cổng đảo có trong IOB để tạo lệch pha  $180^\circ$ .

### 2.3. Hệ thống kết nối khả trìn

Hệ thống kết nối khả trìn (*Programmable Interconnects*) của FPGA dùng để liên kết các phần tử chức năng khác nhau bao gồm IOB, CLB, Block RAM, khối nhân chuyên dụng, 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 ba (*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*).

### 2.3.1. Ma trận chuyển

Ma trận chuyển (*Switch matrix*) là các khối thực hiện kết nối giữa các dạng tài nguyên kết nối của FPGA bao gồm kết nối xa, kết nối kép, kết nối ba, kết nối trực tiếp. Ô liên kết (interconnect tiles) được định nghĩa là một khối bao gồm ma trận chuyển và các phần tử chức năng của FPGA như IOB, CLB, Block RAM, Dedicated Multipliers, DCM.



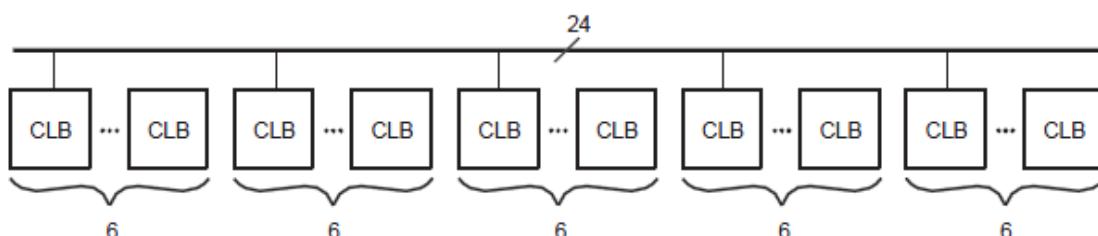
Hình 4-28. Các thành phần nối khác nhau trong Xilinx FPGA

Với CLB, IOB, DCM chỉ cần 1 ma trận chuyển để tạo thành một ô kết nối nhưng với các phần tử lớn hơn như Block RAM hay MULT18 thì cần nhiều ma trận kết nối tương ứng có số ô kết nối lớn hơn.

### 2.3.2. Các dạng kết nối

Các kiểu kết nối có trong FPGA bao gồm:

- Kết nối dài - Long lines

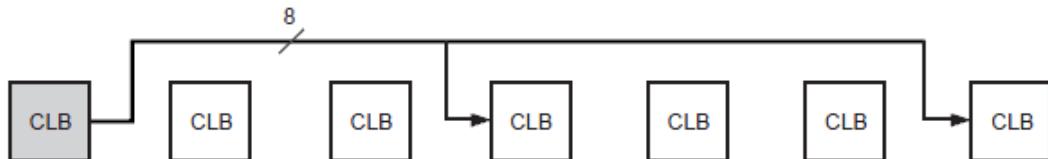


Hình 4-29. Đường kết nối dài

Đường kết nối xa gồm tổ hợp 24 đường nối 1 trong 4 CLB liên tiếp theo phương ngang hoặc phương dọc. Từ mỗi ô kết nối có 4 đường kết nối

đi qua ma trận chuyển để nối với các ô còn lại. Đường kết nối xa có trở kháng thấp do vậy thích hợp cho những tín hiệu toàn cục kiểu như CLK hay Reset.

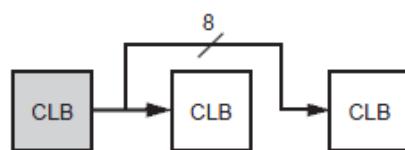
- Kết nối ba Hex lines



Hình 4-30. Đường kết nối ba

Kết nối 3 là kênh kết nối gồm 8 đường nối tới 1 trong 3 CLB liên tiếp, đối với kết nối dạng này tín hiệu chỉ có thể truyền từ một đầu xác định tới các đầu khác theo hướng mũi tên như trong hình 4.30.

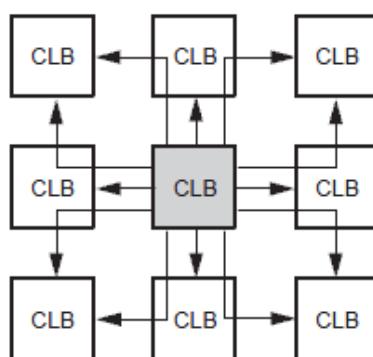
- Kết nối kép - Double lines



Hình 4-31. Đường kết nối kép

Kết nối kép là kênh kết nối gồm 8 đường nối tới 1 trong 2 CLB liên tiếp, đối với kết nối dạng này tín hiệu chỉ có thể truyền từ một đầu xác định tới các điểm khác như hex lines. Số lượng của double lines trong FPGA lớn hơn nhiều so với hai dạng long lines và hex line do khả năng kết nối linh động.

- Kết nối trực tiếp - Direct lines



Hình 4-32. Đường kết nối trực tiếp

Kết nối trực tiếp kết nối các CLB cạnh nhau theo phương ngang, dọc và chéo mà không cần thông qua ma trận kết nối.

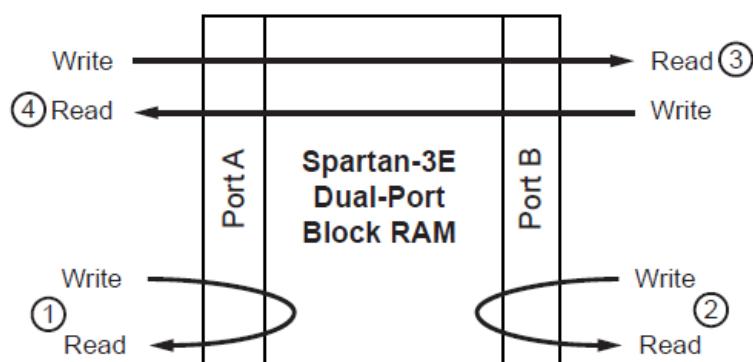
Việc phân cấp các tài nguyên kết nối trong FPGA tuy làm cho việc thiết kế bản thân FPGA phức tạp hơn cũng như tăng độ phức tạp cho thuật toán kết nối đường truyền nhưng góp phần rất lớn vào việc tiết kiệm diện tích và tối ưu hóa thiết kế trên FPGA. Trên thực tế việc sử dụng tài nguyên kết nối trong FPGA được thực hiện tự động, bản thân người thiết kế ít tham gia vào quá trình này hoặc nếu có chỉ là tạo các tín hiệu toàn cục kiểu như CLK, RS, TST để sử dụng phân bố đã được tối ưu hóa của các tín hiệu này.

## 2.4. Các phần tử khác của FPGA

Ngoài các thành phần liệt kê ở hai phần trên, trong FPGA còn được tích hợp thêm các phần tử chức năng đặc biệt khác, đối với Spartan 3E là Khối RAM (*Block RAM*), và khối nhân chuyên dụng (*Dedicated multiplier*) 18 bit MULT18. Một số những dòng FPGA khác được tích hợp thêm DSP là khối nhân-cộng 18 bit ứng dụng cho các bài toán xử lý tín hiệu số, ở một số FPGA còn được nhúng cứng vi xử lý như PowerPC 405, ARM... để có thể lập trình phần mềm trực tiếp.

### 2.4.1. Khối RAM

Bên cạnh nguồn tài nguyên lưu trữ dữ liệu như trình bày ở trên là RAM phân tán (*Distributed RAM*) với bản chất là một hình thức sử dụng khác của LUT thì trong Xilinx FPGA còn được tích hợp các RAM (*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. Giao diện của một *Khối RAM* như sau:

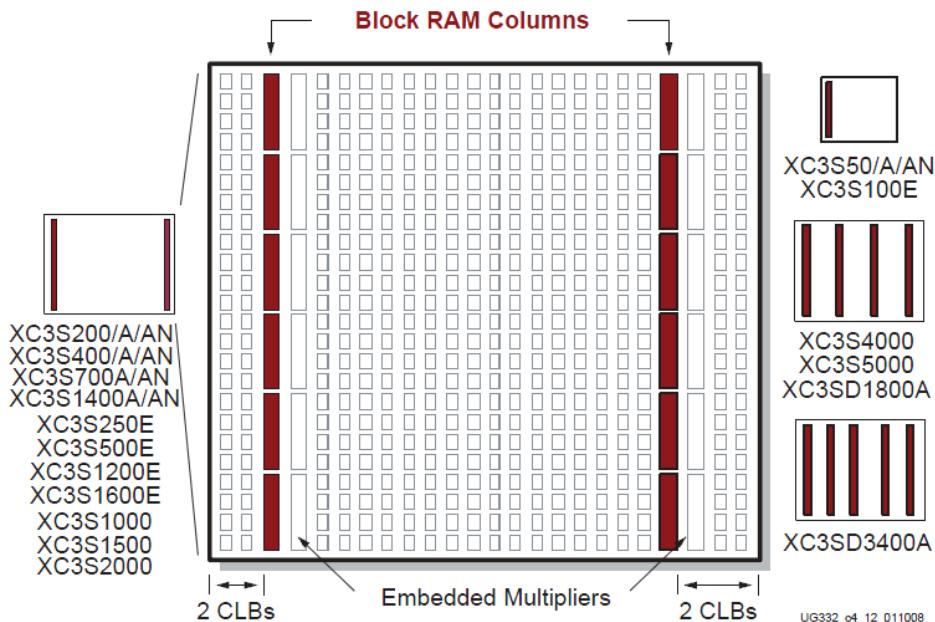


Hình 4-33. Giao diện khối RAM

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, kênh 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.

Vị trí của các Block RAM này trong FPGA thường được bố trí như ở hình sau:



Hình 4-34. Phân bố của các khối RAM trong Spartan 3E FPGA

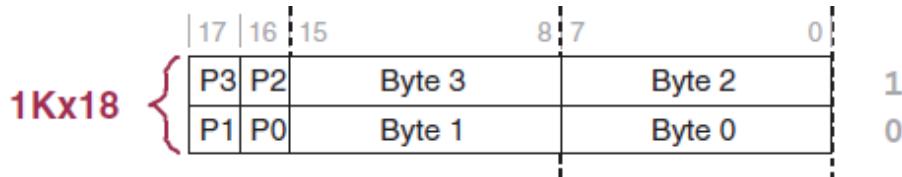
Tùy theo từng FPGA cụ thể mà có thể có từ một đến 5 cột bố trí Block RAM, các cột này thường được bố trí bên cạnh cùng các khối nhân 18-bit. 16-bit cổng A phần thuộc khối nhớ bên trên dùng chung với 16 bit cổng A, tương tự như vậy với 16 bit cổng B của Block RAM được chia sẻ với 16 bit cổng B của khối nhân.

Về kích cỡ của các khối RAM có thể được cấu hình một trong các dạng sau, nếu ký hiệu M là số hàng, W là số bít dữ liệu, P là số bit kiểm tra chẵn lẻ (Parity) trên một hàng Size =  $M \times (W+P)$  bit

- Cấu hình 16K x 1 không có bit kiểm tra chẵn lẻ
- Cấu hình 8K x 2 không có bit kiểm tra chẵn lẻ
- Cấu hình 4K x 4 không có bít kiểm tra chẵn lẻ
- Cấu hình 2K x (8+1), có 1 bit kiểm tra chẵn lẻ
- Cấu hình 1K x (16+2) với hai bit kiểm tra chẵn lẻ
- Cấu hình 512 x (32+4) với 4 bit kiểm tra chẵn lẻ.

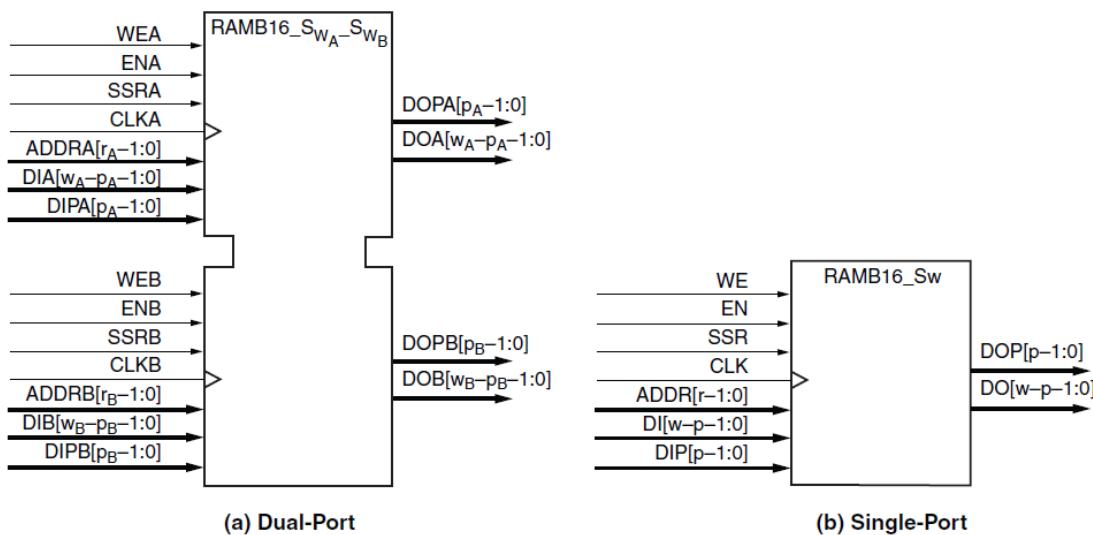
Cách kiểm tra chẵn lẻ như sau, mỗi bit kiểm tra tương ứng với 1 byte hay 8 bit dữ liệu, tính chẵn lẻ xác định bằng số lần xuất hiện bit 1 trong chuỗi 8 bit.

Ví dụ nếu cấu hình của RAM là 1K x(16+2) có nghĩa là bit Pi là bit kiểm tra chẵn lẻ của Bytei như ở hình vẽ dưới đây..



Hình 4-35. Parity bit calculation

*Block RAM* trên thực tế đều là các khối RAM hai cổng nhưng các phần tử tương ứng được mô tả trong thư viện chuẩn của Xilinx và có thể được khởi tạo để hoạt động như RAM 2 cổng (*Dual-port RAM*) hoặc RAM 1 cổng (*Single-port RAM*). Các cổng vào ra của Block RAM được mô tả ở hình vẽ sau:



Hình 4-36. Chi tiết về khối RAM

Tên gọi của các RAM được đặt theo cú pháp RAMB16\_S(Wa)\_S(Wb), trong đó Wa = W + P là tổng độ rộng kênh dữ liệu và số bit kiểm tra. RAM một cổng có tên tương ứng RAM16\_SW lược bỏ đi phần tên của một cổng. Ví dụ RAM16B\_S18\_S9 nghĩa là RAM hai cổng với tổng độ rộng kênh dữ liệu ở cổng A là 18 bit và cổng B là 9 bit, RAM16B\_S34 là khối RAM một cổng với độ rộng kênh dữ liệu là 34 bit.

Đối với Block RAM hai cổng các tín hiệu vào ra được mô tả như sau:

- CLKA, CLKB là xung nhịp đồng bộ cho các cổng A, B tương ứng
- WEA, WEB là tín hiệu cho phép ghi vào cổng A, B tương ứng,
- ENA, ENB là tín hiệu cho phép các cổng A, B hoạt động,
- SSRA, SSRB là các tín hiệu Set và Reset đồng bộ cho các đầu ra DOA, DOB,

- ADDRA, ADDR[B-1] là kênh địa chỉ của các cổng A, B tương ứng trong đó giá trị R tính bằng công thức:  $r = 14 - \log_2(W)$  trong đó W là số bit của kênh dữ liệu.
- DIA, DIB[W-1:0] là các kênh dữ liệu vào của cổng A, B.
- DIPA, DIPB[p-1:0] là kênh dữ liệu kiểm tra chẵn lẻ vào của cổng A, B.
- DOA, DOB[W-1:0] kênh dữ liệu ra của cổng A, B.
- DOPA, DOPB[P-1] kênh dữ liệu kiểm tra chẵn lẻ ra của cổng A, B.

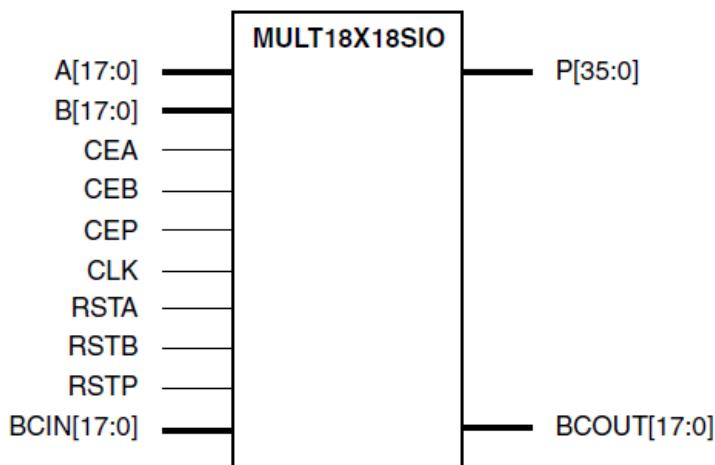
Đối với Block RAM một cổng tên các tín hiệu giữ nguyên như trên nhưng bỏ bớt hậu tố A hoặc B vì chỉ có một cổng duy nhất.

#### 2.4.2. Khối nhân chuyên dụng 18x18

Các khối nhân chuyên dụng 18bitx18bit (*Dedicated Multiplier*) được thiết kế riêng, thường được ứng dụng trong các bài toán xử lý tín hiệu số, ký hiệu là MULT18X18SIO trong thư viện chuẩn của Xilinx.

Các khối nhân 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 các khối này bằng với số lượng của các khôi RAM trong FPGA, ngoài ra hai thành phần này còn chia sẻ với nhau các cổng A, B 16 bit dùng chung..

Khối nhân trong Spartan 3E 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 (bit dấu luôn bằng 0). Mô tả các cổng vào ra của phần tử nhân MULT18X18SIO thể hiện ở hình sau:



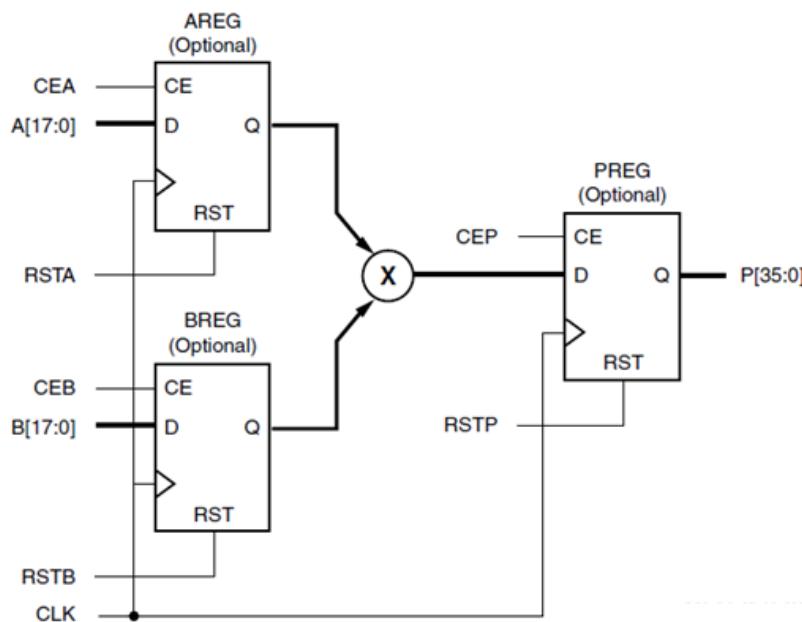
Hình 4-37. Cổng vào ra của khối nhân 18 bit

Khối nhân có tất cả 13 cổng vào ra với các chức năng như sau:

- A, B[17:0] là cổng vào 18 bit số nhân và số bị nhân.

- P[35:0] là 36 bit kết quả nhân (Product)
- CEA, CEB là tín hiệu cho phép xung nhịp ở các đầu vào A, B
- RSTA, RSTB, RSTP là các cổng Set/Reset đồng bộ tương ứng cho các giá trị A, B, P.
- CLK là tín hiệu xung nhịp đồng bộ cho các Flip-flop trong khối nhân
- BCIN, BCOUT[17:0] là các cổng vào ra tương ứng nhằm chia sẻ giá trị số bị nhân giữa các khối nhân với nhau nhằm mục đích tạo thành các khối nhân nhiều bit hơn. BCOUT = BCIN.

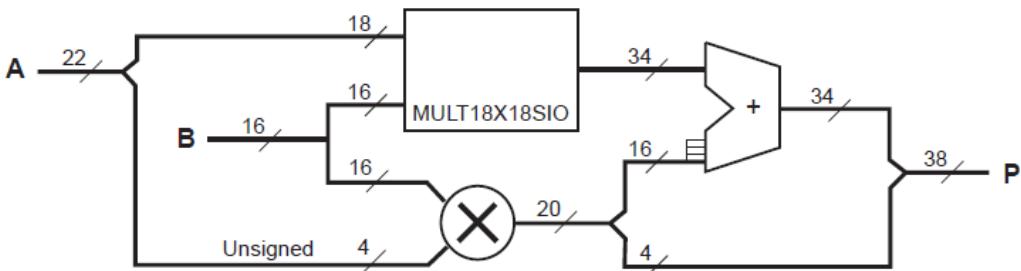
Pipelined option: Khối nhân có thể được thực hiện như một khối tổ hợp thuần túy hoặc có thể chia nhỏ bởi các thanh ghi để đạt hiệu suất làm việc cao hơn. Cấu trúc pipelined của khối nhân thể hiện ở hình sau:



Hình 4-38. Cấu trúc pipelined của khối nhân

Các nhân tử A, B và kết quả P có thể được lưu trong các thanh ghi trung gian gồm AREG, BREG, PREG, mỗi thanh ghi là một chuỗi các Flip-flop. Trong cấu trúc pipelined đó thì REGA, REGB có cùng mức.

Bên cạnh khả năng cấu trúc dạng pipelined để tăng tốc cho phép toán, các khối nhân có thể được kết hợp với nhau thông qua cổng BCIN và BCOUT để thực hiện phép nhân với các nhân tử lớn hơn. Ví dụ khi tiến hành phép nhân hai số 22 bit x 16 bit có thể được thực hiện theo sơ đồ sau:



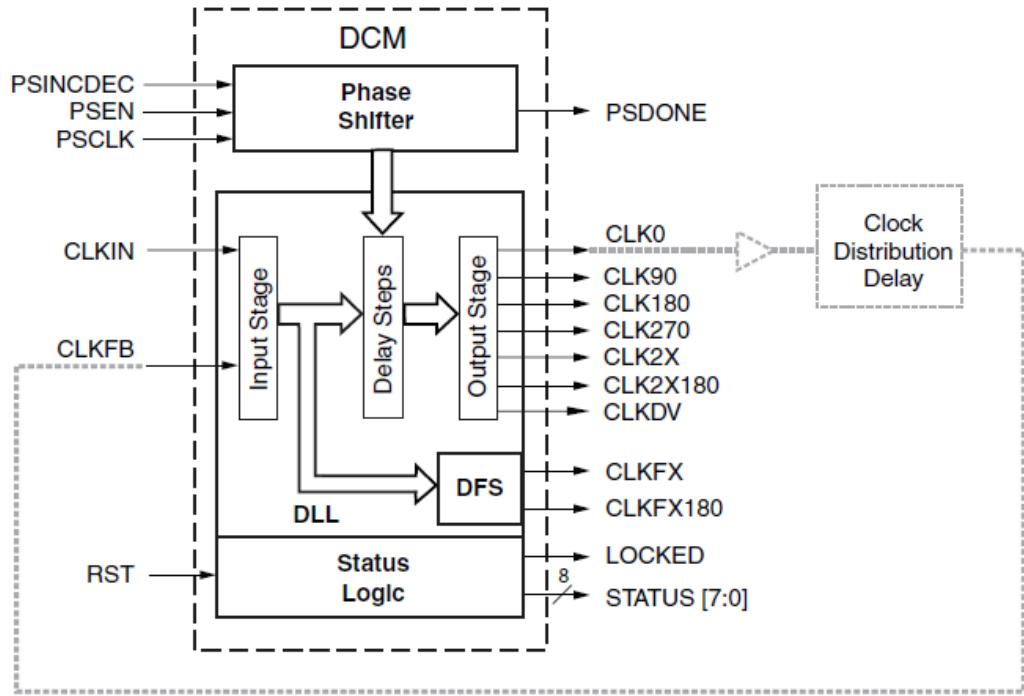
Hình 4-39. Hiện thực phép nhân  $22 \times 16$  bằng khối nhân 18 bit

Để thực hiện phép nhân trên số nhân 22 bit được phân tách thành 4 bit thấp và 18 bit cao, các phần này lần lượt được gửi tới một khối nhân thường 16 bit x 4 bit và một khối nhân 18bit x 18bit MULT18X18SIO. Kết quả hai phép nhân ở khối nhân thường là một số 20 bit trong đó 16 bit cao được gửi tới bộ cộng để cộng với 34-bit kết quả từ khối nhân MULT18X18SIO. Kết quả thu được bằng cách ghép 4 bit thấp với 34 bit tổng để tạo thành số 38-bit

#### 2.4.3. Khối điều chỉnh xung nhịp đồng bộ

*Digital Clock Manager* (DCM) là một khối đặc biệt trong FPGA có nhiệm vụ điều chỉnh và tạo ra xung nhịp đồng bộ (*Clock*) theo những yêu cầu cụ thể của bài toán. DCM có cấu tạo không đơn giản và có số lượng hạn chế (2-4 DCM trong Spartan 3E). 3 thao tác chính mà khối DCM có thể thực hiện là:

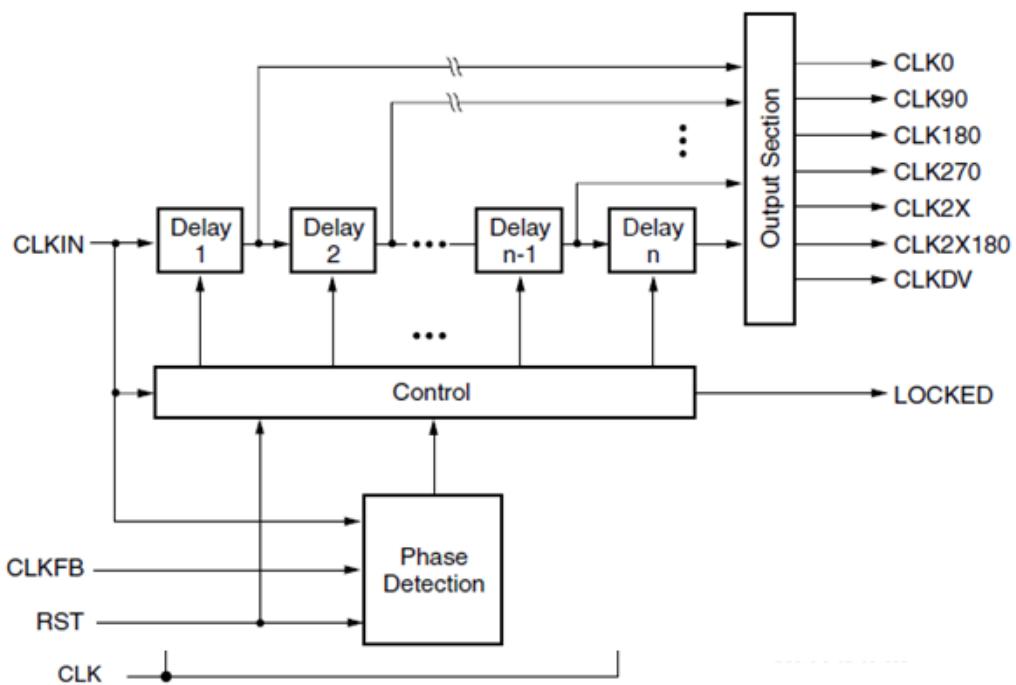
- Loại bỏ độ trễ giữa các xung Clock ở các vị trí khác nhau (*Clock Skew Elimination*). Xung đồng bộ gửi tới các thành phần khác nhau trong FPGA có thể không đến đồng thời do sự khác biệt về tải đường truyền. DCM có khả năng tăng các giá trị Thold, Tsetup của xung đồng bộ và thời gian từ điểm kích hoạt cho tới khi đầu ra ổn định Tclk\_q để “đồng nhất” các xung đồng bộ. Trong các bài toán đòi hỏi làm việc với tần số cao thì đây là một trong những thao tác không thể bỏ qua.
- Tổng hợp tần số (*Frequency Synthesis*): Tổng hợp tần số ở đây bao gồm nhân và chia tần số, với tần số cố định đầu vào DCM có thể thực hiện thao tác nhân tần số với 1 số M, chia cho một số D hoặc đồng thời nhân và chia M/D. Đây là một khả năng đặc biệt quan trọng cho những bài toán yêu cầu tần số làm việc là cố định như điều khiển VGA, DAC, ADC, LCD...
- Dịch pha (*Phase shifting*) Dịch pha của xung nhịp đồng bộ đi 0, 90, 180, hoặc 270 độ.



Hình 4-40. Sơ đồ khái DCM

Khối DCM được cấu tạo từ 4 khối chính, khối dịch pha PhS (*Phase shifter*), khối lặp khóa pha DLL (*Delay Locked Loop*), khối tổng hợp tần số DFS (*digital Frequency Synthesis*) và khối Trạng thái logic của DCM.

**Khối DLL:** Khối DLL được cấu tạo bởi một chuỗi các phần tử làm trễ mà thực chất các khối đếm. DLL có đầu vào xung nhịp CLKIN, đầu vào CLKFB (*CLOCK FEEDBACK*), xung nhịp này sẽ được sử dụng để so sánh pha ở khối *Phase detection* để thực hiện điều chỉnh pha thông qua một vòng lặp hồi tiếp dương, tùy thuộc vào cấu hình cài đặt mà CLKFB có thể là đầu ra CLK2X hoặc CLK0 (đây cũng là đặc điểm khác biệt giữa DLL và PLL).



Hình 4-41. Sơ đồ khói DLL

DLL có 7 đầu ra xung nhịp CLK0, CLK90, CLK180, CLK270, CLK2X (nhân đôi tần số), CLKDV (chia tần) và một đầu ra LOCKED báo hiệu khi pha giữa CLKFB và CLKIN đã trùng nhau. Khối DLL có các tham số như sau:

Bảng 4-3

### Khối DLL

Tham số	Ý nghĩa	
CLOCK_FEEDBACK	Lựa chọn xung nhịp phản hồi để so sánh pha	NONE : không 1X: CLK0 2X: CLK2X
CLKIN_DIVIDE_BY_2	Chia đôi tần số xung CLKIN	TRUE : có chia FALSE : không chia
CLKDV_DIVIDE	Giá trị chia tần	Nhận các giá trị 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7, 7.5, 8, 9, 10, 11, 12, 13, 14, 15
CLKIN_PERIOD	Thông tin bộ xung về Chu kỳ CLKIN đầu vào theo [ns]	Số thực biểu diễn tần số đầu vào

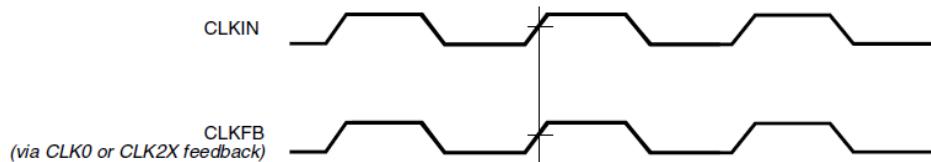
*Khối tổng hợp tần số DFS* có thể được thực hiện độc lập hoặc trên nền tảng của DLL, chi tiết xem thêm trong tài liệu của Xilinx. DFS có đầu vào là CLKIN và hai đầu ra là CLKFX và CLKFX180, trong đó

$$\text{CLKFX} = \text{CLKIN} * M/D$$

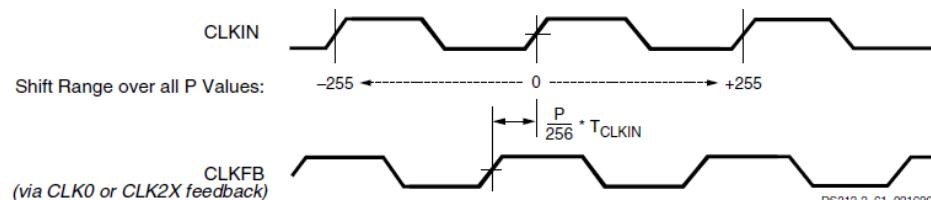
Còn CLKFX180 là CLKFX bị dịch pha 180 độ. M là hệ số nhân, D là hệ số chia được định nghĩa khi cài đặt DCM bởi các tham số tương ứng CLKFX\_MULTIPLY (giá trị nguyên từ 2-32) và CLKFX\_DIVIDE, (giá trị nguyên từ 1 – 32).

*Khối dịch pha PS:* Bản thân DLL cung cấp các đầu ra với độ lệch pha chuẩn lần lượt là 0, 90, 180, 270, ngoài ra trong DCM còn có khối PS cho phép “tinh chỉnh” độ lệch pha của tín hiệu CLKIN và CLKFB với độ phân giải (-255 + 255) trên miền (-360 + 360) độ như ở hình sau.

a. CLKOUT\_PHASE\_SHIFT = NONE



b. CLKOUT\_PHASE\_SHIFT = FIXED



Hình 4-42. Sơ đồ khối dịch pha

Trong chế độ không dịch pha thì CLKIN và CLKFB đồng pha, trong chế độ dịch pha thì CLKFB có thể lệch so với CLKIN một đại lượng pha tương ứng  $t_{PS} = P/256 * T_{CLKIN}$ .

Khi CLKFB và CLKIN lệch pha thì sẽ dẫn tới sự lệch pha tương ứng của tất cả các đầu ra kể trên.

Các tham số và tín hiệu của khối dịch pha liệt kê ở hai bảng sau:

Bảng 4-4

**Tham số của khối dịch pha**

Tham số	Ý nghĩa	Ghi chú
CLOCK_OUT_PHASESHIFT	Chọn chế độ dịch pha	NONE : không FIX: dịch pha
PHASE_SHIFT	Giá trị dịch pha	Số nguyên từ -255 đến +255

Các tín hiệu của khối dịch pha bao gồm:

Bảng 4-5

**Tín hiệu của khối dịch pha**

Tham số	Ý nghĩa	Ghi chú
PSEN	Tín hiệu cho phép dịch pha	Tín hiệu không đồng bộ
PSCLOCK	Tín hiệu đồng bộ cho khối PS	Xung đồng bộ
PSINCDEC	Khi bằng 1 thì tăng giá trị dịch pha, khi bằng 0 thì giảm giá trị dịch pha	Tín hiệu thay đổi đồng bộ bởi PSCLOCK
PSDONE	Báo hiệu quá trình đồng bộ pha kết thúc.	

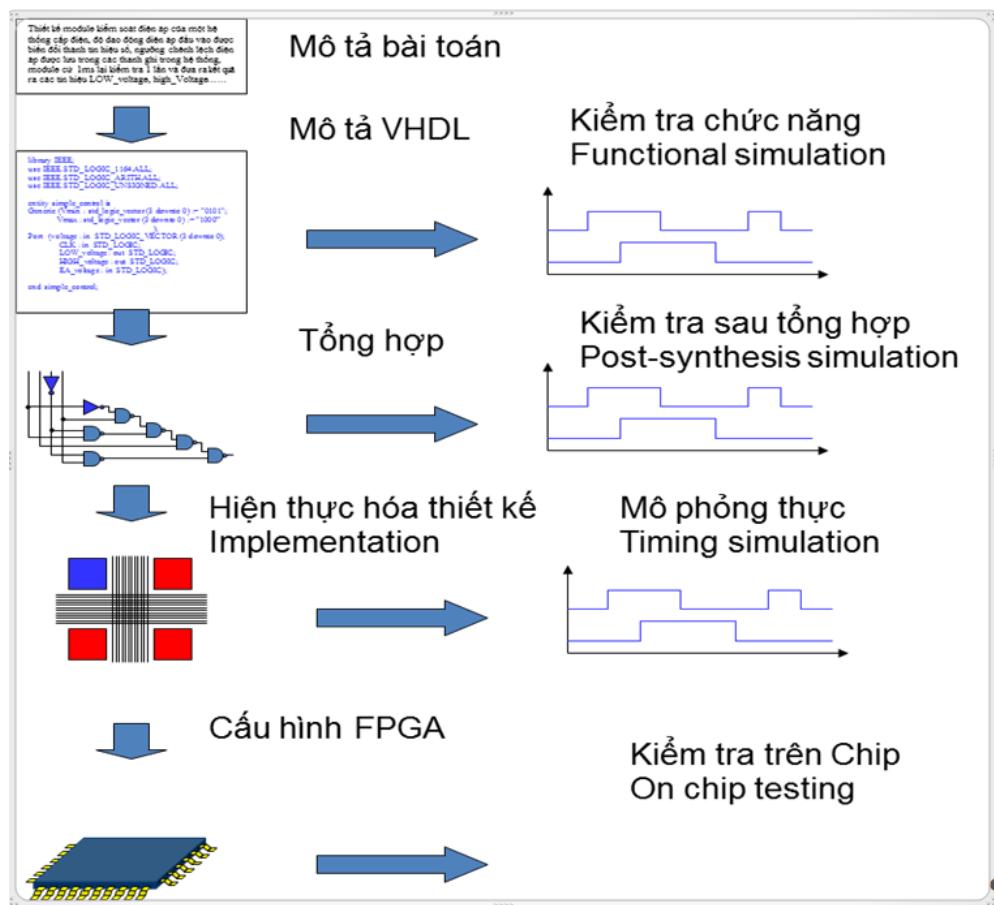
### **3. Quy trình thiết kế FPGA bằng ISE**

Trong chương trình của Điện tử số các mạch số sau khi đã xác định được chức năng thì sẽ tiến hành tổng hợp mạch trên các công logic cơ bản kết thúc bằng xây dựng sơ đồ logic cho mạch. Với kích thước vừa phải thì bước tổng hợp mạch có thể làm bằng tay. Tuy vậy quá trình tổng hợp mạch số trong chương trình nghiên cứu ở đây ở đây rất phức tạp do kích thước thiết kế lớn (cỡ hàng trăm nghìn cổng) nên thực hiện trên giấy bút là không thể mà phải thực hiện nhờ sự trợ giúp của phần mềm máy tính. Các phần mềm này bản thân được tích hợp những thuật toán tổng hợp từ đơn giản tới phức tạp và trên lý thuyết có thể tổng hợp các vi mạch số với kích thước bất kỳ. Nói một cách khác khi đã có các bản thiết kế VHDL cho chức năng của mạch như ở các chương trước thì quy trình biến thiết kế đó thành mạch làm việc gần như tự động hoàn toàn.

Một trong những yếu tố góp phần vào sự thành công của FPGA phải kể đến là có một quy trình thiết kế đơn giản, hoàn thiện được thực hiện bằng các bộ phần mềm chuyên dụng. Các phần mềm này được tích hợp nhiều các thuật toán xử lý tối ưu khác nhau nhằm tăng tính tự động hóa cho quy trình thiết kế.

Nội dung của những phần dưới đây được viết dựa trên cơ sở các bước thiết kế FPGA bằng tổ hợp phần mềm *Xilinx ISE (Integrated Software Environments)*. Để tìm hiểu kỹ hơn người đọc có thể tham khảo thêm các tài liệu gốc có trên trang chủ của Xilinx.

Có một số điểm khác nhau cho từng loại FPGA hay cho FPGA của từng hãng nhưng quy trình thiết kế IC số sử dụng FPGA chung đều có thể chia thành năm bước thể hiện ở sơ đồ dưới đây:



Hình 4-43. Quy trình thiết kế trên FPGA

### 3.1. Mô tả thiết kế

Sau khi đã có sơ đồ thuật toán chi tiết, để tiến hành mô tả chức năng vi mạch người thiết kế có thể sử dụng một số phương pháp khác nhau như trình bày dưới đây.

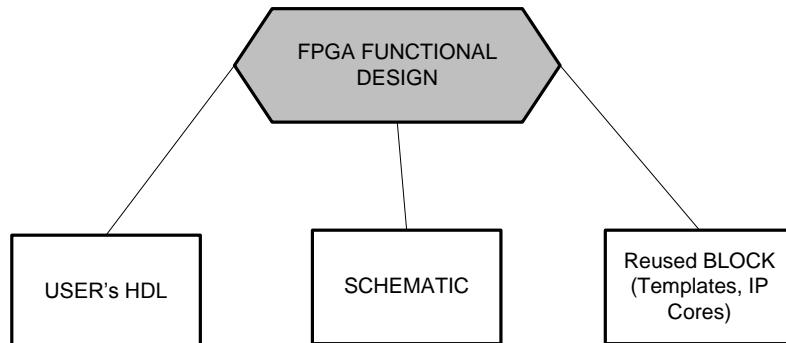
Khi chọn mô tả bằng sơ đồ (*schematic*) thì ISE cung cấp một thư viện các phần tử dưới dạng đồ họa (*graphic symbols*) như các khối cộng, trừ, buffer, thanh ghi, khói nhân, RAM, cổng vào ra... Người thiết kế sẽ sử dụng các phần tử này và thực hiện kết nối trên sơ đồ để tạo thành mạch hoàn chỉnh.

Mô tả theo sơ đồ tuy rõ ràng nhưng không phù hợp cho những thiết kế phức tạp. Đối với những thiết kế lớn thì nên sử dụng phương pháp mô tả bằng ngôn ngữ HDL, có hai dạng HDL là HDL do người dùng thiết kế, đó là dạng mô tả “trung tính” có thể sử dụng cho bất kỳ đối tượng phần cứng nào mà chúng ta từng thực hiện ở những chương trước. Rõ ràng với ưu điểm như vậy thì trong thiết kế ta nên tận dụng tối đa dạng thiết kế này.

Bên cạnh dạng mô tả HDL của người dùng thì có dạng mô tả HDL thứ hai sử dụng các khối thiết kế sẵn. Loại mô tả này có hai dạng, thứ nhất là các khối thiết kế được định nghĩa trong thư viện UNISIM của Xilinx. Khi muốn cài đặt các khối này thì phải khai báo thêm thư viện Unisim ở đầu thiết kế:

```
library UNISIM;
use UNISIM.VCOMPONENTS.all;
```

Một số khối thiết kế điển hình là các LUT, thanh ghi dịch, Block RAM, ROM, DCM... có thể tìm thấy trong Language template của ISE, các khối này gọi chung là các phần tử cơ bản của FPGA (*FPGA primitives*), đặc điểm của các khối này là phụ thuộc vào đối tượng FPGA cụ thể.

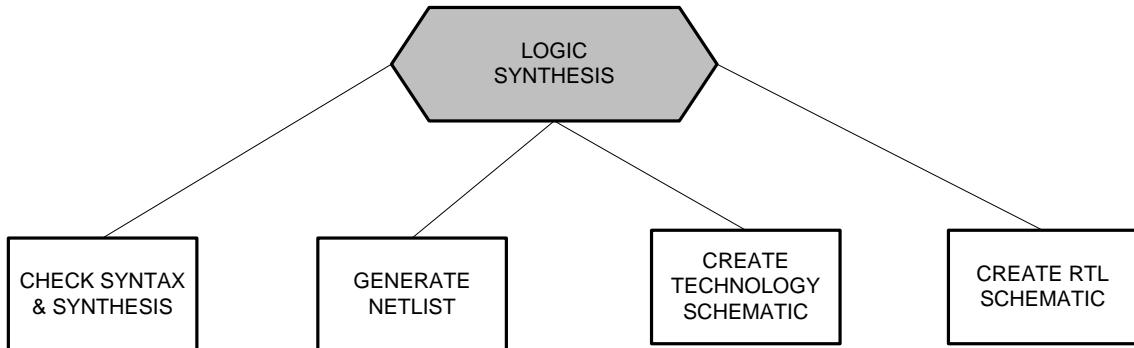


Hình 4-44. Các dạng mô tả thiết kế trên FPGA

Ngoài *FPGA primitives* thì ISE cho phép người dùng sử dụng một số khối thiết kế sẵn ở dạng IP Core (*Intellectual Property core*). IP core là các khối thiết kế sẵn có đăng ký sở hữu trí tuệ và thường là các thiết kế khá phức tạp ví dụ như các khối FIFO, khối làm việc với số thực (Floating Point Unit), khối chia, các khối CORDIC, các khối giao tiếp Ethernet, PCI EXPRESS, SPI, các khối xử lý số tín hiệu... Trong khuôn khổ chương trình học thì việc sử dụng này là được phép tuy vậy nếu muốn sử dụng các khối này với mục đích tạo ra sản phẩm ứng dụng thì cần xem xét kỹ vấn đề bản quyền. Với sự hỗ trợ phong phú của các IP Cores này cho phép thực hiện những thiết kế lớn và hữu dụng trên FPGA. Lưu ý là khi sử dụng các khối này thì phần thiết kế HDL thực sự cũng bị “giấu đi” mà chương trình chỉ cung cấp các mã đã biên dịch và mô tả giao diện (*wrapper file*) của IP Core được sử dụng.

### 3.2. Tổng hợp thiết kế

Quá trình tổng hợp FPGA (*FPGA Synthesis*) bằng chương trình ISE bao gồm các bước như sau.



Hình 4-45. *Tổng hợp thiết kế FPGA trên Xilinx ISE*

**Check Syntax & Synthesis:** Trước khi thiết kế được tổng hợp thì mã nguồn VHDL được biên dịch và kiểm tra trước. Nếu xuất hiện lỗi cú pháp ở mã nguồn thì qua trình tổng hợp sẽ dừng lại. Nếu mô tả VHDL không có lỗi thì chuyển sang bước thứ hai là tổng hợp (*synthesis*). *Tổng hợp thiết kế là chuyển mô tả từ mức trừu tượng cao (con người có thể đọc hiểu) xuống mức trừu tượng thấp hơn (máy tính mới có khả năng đọc hiểu)*. Đối với FPGA quá trình tổng hợp logic là quá trình biên dịch từ mô tả chức năng sang mô tả công (*netlist*). Mô tả công bao gồm vẫn là các mô tả VHDL nhưng sử dụng các phần tử của FPGA, hiểu một cách khác nếu mô tả chức năng là sơ đồ nguyên lý thì mô tả *netlist* là sự chi tiết hóa sơ đồ nguyên lý. Có thể so sánh mã dưới dạng Netlist như mã Assembly của chương trình gốc mô tả bằng các ngôn ngữ lập trình bậc cao C/C++, Pascal, Basic...

Các mã nguồn VHDL được chia thành hai dạng là tổng hợp được (*Synthesizable code*) và không tổng hợp được (*Simulation-only code*), việc phân biệt hai dạng mã nguồn này được ISE làm tự động. Khi cố tình tổng hợp một cấu trúc chỉ dùng cho mô phỏng thì sẽ gây ra lỗi. Người thiết kế vì vậy ngoài việc đảm bảo chức năng làm việc đúng cho mạch còn luôn phải đảm bảo rằng những cấu trúc viết ra là những cấu trúc có thể tạo thành mạch thật nghĩa là tổng hợp được.

Điểm đáng lưu ý tại thời điểm này là quá trình tổng hợp bao gồm cả quá trình tối ưu logic cho thiết kế, ví dụ nếu chúng ta sử dụng bộ trừ để làm phép so sánh nhưng về thực chất để so sánh hai số chỉ cần thiết lập bít nhớ cuối cùng của phép trừ mà không cần quan tâm đến bản thân giá trị thu được của phép tính. Chính vì vậy sau khi tổng hợp ta nhận được cảnh báo (*Warning*) từ chương trình.

Xst:646 - Signal <sub<3:0>> is assigned but never used. This unconnected signal will be trimmed during the optimization process.

Tất cả những tín hiệu thừa sẽ bị bỏ đi sau khi tổng hợp nhằm tiết kiệm tài nguyên FPGA, nếu muốn bỏ cảnh báo này chúng ta có thể thay đổi thiết kế của bộ so sánh bằng cách mô tả chi tiết chuỗi nhớ thay vì dùng cả bộ cộng/trừ 4 bit. Những cảnh báo dạng này thuộc dạng vô hại vì nó không ảnh hưởng tới chức năng của mạch. Tuy vậy kinh nghiệm thực tế cho thấy trong mọi trường hợp việc kiểm tra các cảnh báo là cực kỳ cần thiết vì một thiết kế được biên dịch và mô phỏng đúng rất có thể chức năng của mạch bị thay đổi do việc tối ưu (cắt bỏ các tín hiệu bị coi là thừa), hoặc nhờ những cảnh báo này có thể tìm ra những sai sót trong thiết kế nguyên lý, nhất là với những thiết kế lớn và phức tạp.

*Synthesis report:* Kết quả tổng hợp được ghi dưới dạng một tệp văn bản, trong đó thông kê về các phần tử logic, các phần tử nhớ, số LUT, số cổng vào ra (IO\_BUF) và tham số về mặt thời gian, ví dụ kết quả tổng hợp như sau:

```
=====
* Final Report *
=====
Final Results
RTL Top Level Output File Name : sp3_led.ngr
Top Level Output File Name : sp3_led
Output Format : NGC
Optimization Goal : Speed
Keep Hierarchy : NO
Device utilization summary:
-----
Selected Device : 3s500epq208-4
Number of Slices: 4 out of 4656 0%
Number of 4 input LUTs: 8 out of 9312 0%
Number of IOs: 16
Number of bonded IOBs: 16 out of 158 10%
=====
TIMING REPORT
Timing Summary:
-----
Speed Grade: -4
Minimum period: No path found
Minimum input arrival time before clock: No path found
Maximum output required time after clock: No path found
Maximum combinational path delay: 6.320ns
```

Khi phân tích kết quả tổng hợp cần lưu ý hai thông tin cơ bản. Thứ nhất là thông tin về tài nguyên, trong báo cáo sẽ liệt kê các dạng tài nguyên và số lượng của từng loại được sử dụng cho toàn khối thiết kế (*Device utilization summary*). Thông tin này trong một số trường hợp còn giúp người thiết kế kiểm định lại sơ

đồ thuật toán ban đầu bằng cách so sánh giữa tài nguyên thực tế sau tổng hợp với tài nguyên ước tính sơ bộ ở ban đầu.

Thông tin thứ hai là thông tin về mặt thời gian, theo ngầm định thiết kế sẽ được tối ưu với tiêu chí đầu tiên là giảm tối đa thời gian trễ (*Optimization Goal : Speed*). Các thông tin thời gian được liệt kê bao gồm thời gian trễ tổ hợp (*combinational delay*) và chu kỳ của xung nhịp (*Clock informations*) nếu có. Trong báo cáo cũng sẽ liệt các đường gây trễ cực đại (*Critical paths*), các thông tin này có thể giúp người thiết kế tối ưu hóa lại mô tả VHDL hoặc thuật toán để đạt được độ trễ thấp hơn một cách hiệu quả.

*Kết xuất mô tả netlist:* mô tả netlist là mô tả VHDL của thiết kế nhưng được ánh xạ lên thư viện phần tử logic của FPGA. Mô tả netlist là dạng mô tả ở mức công vì vậy không mô tả trực quan được chức năng của vi mạch mà chỉ thể hiện được cấu trúc của mạch, trong đó các khối con (*components*) là các phần tử cơ bản được mô tả trong thư viện UNISIM của FPGA. Ví dụ một mô tả netlist có dạng như sau:

```
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library UNISIM;
use UNISIM.VCOMPONENTS.ALL;
use UNISIM.VPKG.ALL;

entity sp3_led is
  port (
    LED1 : out STD_LOGIC;
    LED2 : out STD_LOGIC;
    ...
    SW7 : in STD_LOGIC := 'X';
    SW8 : in STD_LOGIC := 'X');
end sp3_led;

architecture Structure of sp3_led is
  signal LED1_OBUF_1 : STD_LOGIC;
  signal LED2_OBUF_3 : STD_LOGIC;
  ...
  signal SW7_IBUF_29 : STD_LOGIC;
  signal SW8_IBUF_31 : STD_LOGIC;
begin
  LED81 : LUT2
    generic map(
      INIT => X"1"
    )

```

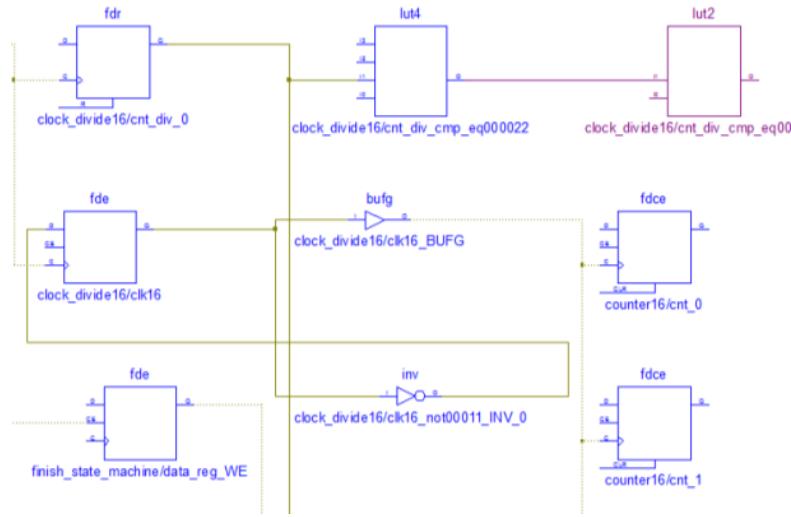
```

port map (
    I0 => SW8_IBUF_31,
    I1 => SW7_IBUF_29,
    O => LED8_OBUF_15);

...
LED8_OBUF : OBUF
port map (
    I => LED8_OBUF_15,
    O => LED8);
LED51_INV_0 : INV
port map (
    I => SW5_IBUF_25,
    O => LED5_OBUF_9);
end Structure;
-----
```

*Netlist* có thể không phản ánh đúng bản chất thực tế của mạch mà mô tả này chỉ sử dụng để kiểm tra lại chức năng của vi mạch sau khi ánh xạ lên thư viện phần tử FPGA. Việc kiểm tra này được thực hiện giống như kiểm tra mô tả VHDL ban đầu, tức là có thể dùng bất kỳ chương trình mô phỏng logic nào mà hỗ trợ thư viện UNISIM.

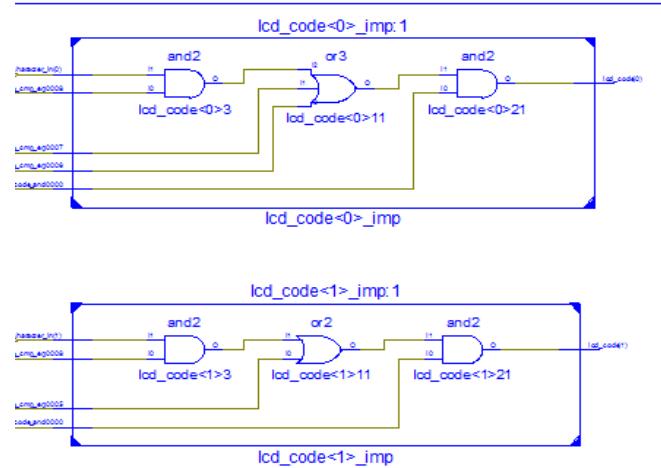
*Create Technology schematic (Sơ đồ công nghệ chi tiết)* Sau khi tổng hợp chương trình cũng cho phép kết xuất sơ đồ công nghệ chi tiết của thiết kế mà bản chất là mô tả trực quan bằng hình ảnh của *netlist*, ví dụ một sơ đồ công nghệ chi tiết ở hình sau:



Hình 4-46. Sơ đồ công nghệ

Việc so sánh sơ đồ này với sơ đồ nguyên lý ở bước ban đầu cho phép kiểm tra trực quan sơ bộ việc thực hiện đúng sơ đồ nguyên lý của mô tả VHDL.

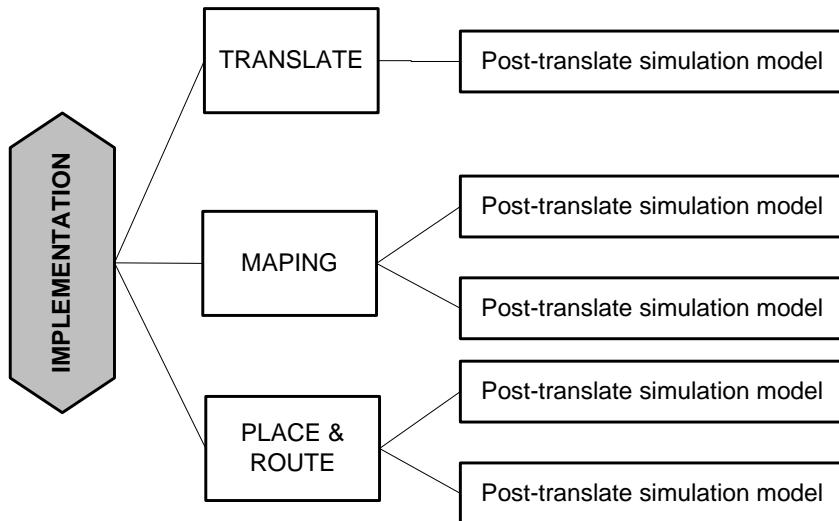
*Create RTL schematic (Sơ đồ logic chi tiết)* Sơ đồ logic chi tiết là sơ đồ thể hiện chức năng của thiết kế sử dụng các công logic chuẩn như AND, OR, NOT, FFD thay vì sử dụng các phần tử chuẩn của FPGA, sơ đồ này vì thế không phụ thuộc vào đối tượng công nghệ cụ thể.



Hình 4-47. Sơ đồ logic

### 3.3. Hiện thực hóa thiết kế

Hiện thực hóa thiết kế (*Implementation*) kế FPGA là quá trình chuẩn bị dữ liệu cho việc cấu hình FPGA từ thông tin đầu vào là mô tả netlist. Quá trình này bắt đầu bằng quá trình biên dịch và ánh xạ thiết kế lên đối tượng FPGA cho tới khi thiết kế vật lý được phân bố cụ thể và kết nối với nhau. Quá trình đó gồm 3 bước như sau:



Hình 4-48. Quá trình hiện thực hóa FPGA

### 3.3.1. Translate

*Netlist* của chương trình được dịch thành định dạng EDIF (*Electronic Device Interchangeable Format*) hoặc NGC format (một định dạng netlist riêng của Xilinx) sau đó kết hợp với hai file quy định điều kiện ràng buộc của thiết kế.

- NCF (*Native Constraint File*) chứa những thông tin vật lý về thời gian, tốc độ, các tham số tải, tham số vật lý kỵ sinh... của chip vật lý FPGA là đối tượng sẽ tiến hành cấu hình.

- UCF (*User Constraint File*) chứa những ràng buộc yêu cầu từ phía người thiết kế với vi mạch của mình. UCF được xem là một phần quan trọng trong thiết kế, nếu như mô tả chức năng chỉ quy định vi mạch định làm gì thì trong file UCF sẽ chứa những yêu cầu đòi hỏi về tốc làm việc (timing constraint) cũng như mức độ sử dụng tài nguyên. Các yêu cầu này là cơ sở cho các trình biên dịch trong ISE tối ưu hóa thiết kế. Để có thể viết được những yêu cầu này thì người thiết kế trước hết phải hiểu rất rõ thiết kế của mình, chẳng hạn trong thiết kế có dùng những tín hiệu xung nhịp như thế nào, khu vực tổ hợp nào có khả năng gây ra thời gian trễ lớn nhất và ước tính được giá trị của độ trễ. Phân tích nội dung một file UCF có nội dung như dưới đây.

```
# IO location defination
NET "HIGH_voltage" LOC = P102;
NET "LOW_voltage" LOC = P100;
NET "voltage[0]" LOC = P160;
NET "voltage[1]" LOC = P161;
NET "voltage[2]" LOC = P162;
NET "voltage[3]" LOC = P163;

# Timing constraint
INST "LOW_voltage" TNM = "OUT_REG";
INST "HIGH_voltage" TNM = "OUT_REG";
NET "voltage[0]" OFFSET = IN 2 ns VALID 0.5 ns BEFORE
"CLK" TIMEGRP "OUT_REG" RISING;
NET "voltage[1]" OFFSET = IN 2 ns VALID 0.5 ns BEFORE
"CLK" TIMEGRP "OUT_REG" RISING;
NET "voltage[2]" OFFSET = IN 2 ns VALID 0.5 ns BEFORE
"CLK" TIMEGRP "OUT_REG" RISING;
NET "voltage[3]" OFFSET = IN 2 ns VALID 0.5 ns BEFORE
"CLK" TIMEGRP "OUT_REG" RISING;
```

Nhóm mô tả thứ nhất quy định cách gán chân vào ra của FPGA, cách gán chân này phụ thuộc thứ nhất vào từng loại FPGA, thứ hai vào bản mạch ứng dụng FPGA cụ thể, chẳng hạn như ở trên các chân voltage được gán cho các

cổng từ P160 đến P163, còn hai tín hiệu LOW\_voltage và HIGH\_voltage được gắn cho chân P102 và P100.

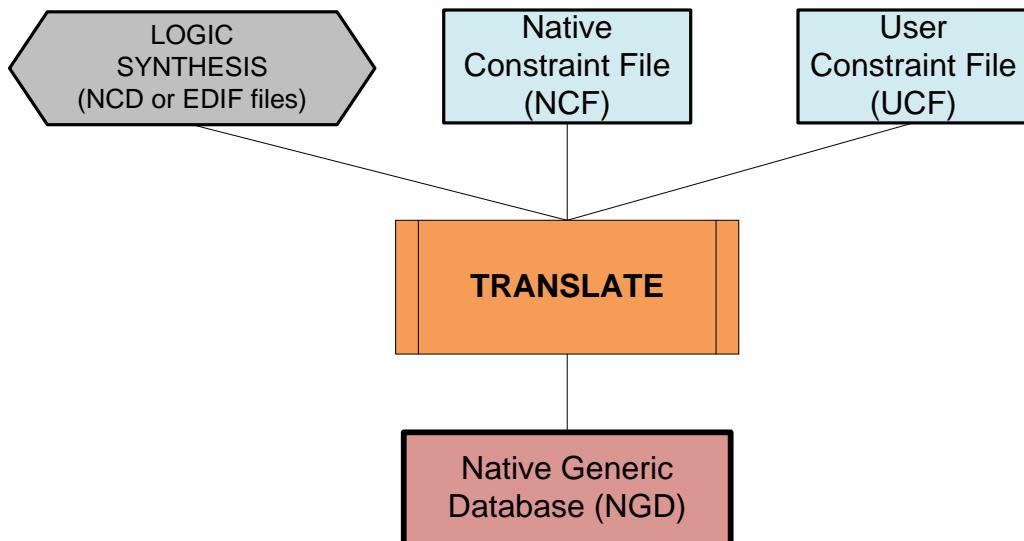
Nhóm thứ hai mô tả điều kiện ràng buộc về mặt thời gian: hai lệnh đầu ghép hai Flip-flop đầu ra của mạch thành một nhóm có tên là OUT\_REG, 4 lệnh sau quy định thời gian trễ của 4 tín hiệu đầu vào có thời gian:

```
OFFSET = IN 2 ns VALID 0.5 ns BEFORE "CLK" TIMEGRP  
"OUT_REG" RISING
```

nghĩa là thời gian trễ của các tín hiệu này trước khi đến các Flip-Flop của nhóm OUT\_REG không quá 2 ns và thời gian giữ tín hiệu ổn định cho Flip-Flop (Setup time) không quá 0,5 ns.

Xilinx ISE hỗ trợ trình soạn thảo với giao diện đồ họa cho UCF tuy vậy có thể sử dụng trình soạn thảo Text bất kỳ để soạn file UCF, yêu cầu duy nhất là đặt tên file UCF trùng với tên thiết kế.

(\*) Chi tiết hướng dẫn về UCF có thể xem trong tài liệu hướng dẫn của Xilinx <http://www.xilinx.com/itp/xilinx10/books/docs/cgd/cgd.pdf>



Hình 4-49. Quá trình biên dịch

Quá trình translate sẽ đọc các thông tin từ 3 file trên và chuyển về định dạng NGD (*Native Generic Database*) của Xilinx để phục vụ cho hai bước kế tiếp là Mapping và Routing.

NGD bản chất là một mô tả cấu trúc của mạch trên cơ sở các phần tử chức năng được mô tả trong thư viện có tên SIMPRIM (simulation primitives) của Xilinx. Các phần tử này không thuộc vào FPGA cụ thể, nó có thể được sử dụng cho tất cả các dạng FPGA và CPLD của Xilinx.

Người thiết kế có thể tạo ra file netlist để phục vụ mô phỏng kiểm tra sau Translate (*Post-translate simulation model*). Ví dụ file netlist như sau:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library SIMPRIM;
use SIMPRIM.VCOMPONENTS.ALL;
use SIMPRIM.VPACKAGE.ALL;

entity sp3_led is
    port (
        LED1 : out STD_LOGIC;
        LED2 : out STD_LOGIC;
        ...
        SW7 : in STD_LOGIC := 'X';
        SW8 : in STD_LOGIC := 'X'
    );
end sp3_led;

architecture Structure of sp3_led is
    signal LED1_OBUF_1 : STD_LOGIC;
    signal SW8_IBUF_31 : STD_LOGIC;
begin
    LED81 : X_LUT2
        generic map(
            INIT => X"1"
        )
        port map (
            ADR0 => SW8_IBUF_31,
            ADR1 => SW7_IBUF_29,
            O => LED8_OBUF_15);
    ...
    SW8_IBUF : X_BUF
        port map (
            I => SW8,
            O => SW8_IBUF_31);
    LED51_INV_0 : X_INV
        port map (
            I => SW5_IBUF_25,
            O => LED5_OBUF_9);
    LED8_OBUF : X_OBUF
        port map (
            I => LED8_OBUF_15,
            O => LED8);
    NlwBlockROC : X_ROC
        generic map (ROC_WIDTH => 100 ns)
        port map (O => GSR);
```

```

NlwBlockTOC : X_TOC
port map (O => GTS) ;

end Structure;

```

### 3.3.2. Maping

*Mapping* là động tác gán các khối sơ đồ logic vào các khối cơ sở của một FPGA cụ thể, đầu vào của quá trình là dữ liệu được lưu trong file NGD bao gồm mô tả logic sử dụng các phần tử chức năng độc lập với công nghệ có trong thư viện SIMPRIM và các thông tin về các khối, đường kết nối cố định. Đầu ra của quá trình này là một file dạng NCD (*Native Circuit Database*), file này chứa mô tả chức năng mạch thiết kế trên đối tượng FPGA cụ thể.

Quá trình maping trải qua các bước cơ sở như sau.

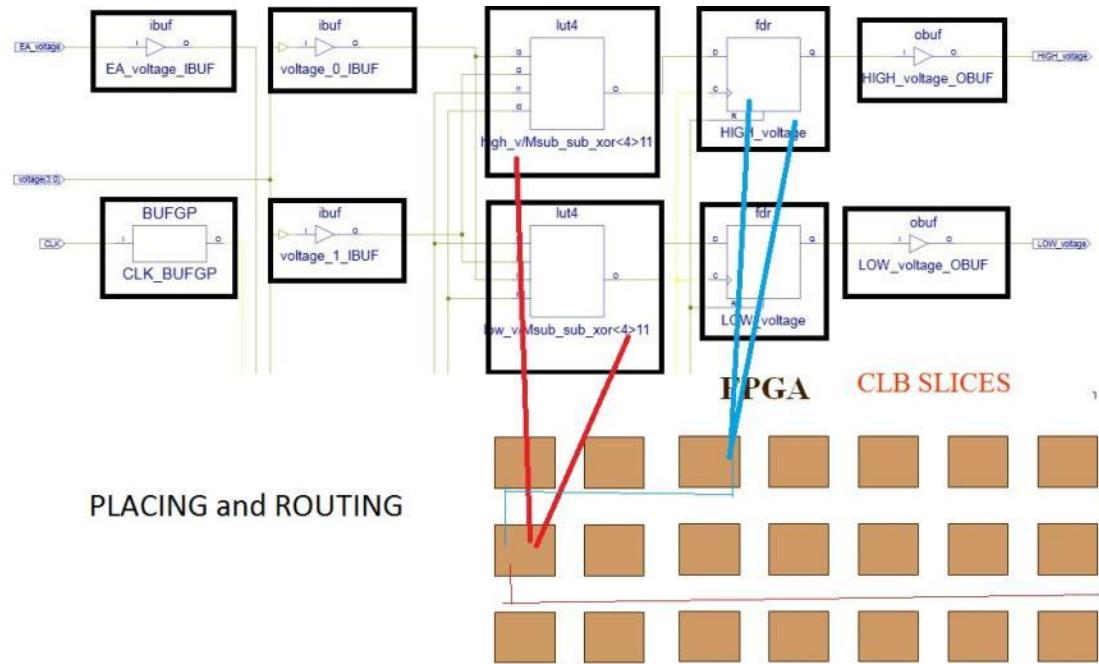
- Đọc thông tin của đối tượng FPGA.
- Đọc thông tin thiết kế từ file NGD.
- Thực hiện DRC (*Design Rule Check*) Kiểm tra thiết kế bao gồm: kiểm tra khói (Block check) là kiểm tra sơ đồ kết nối của các khói, kiểm tra sự tồn tại của các mô tả khói con. Kiểm tra các đường nối logic (*Net check*) bao gồm kiểm tra các khói đầu vào đầu ra và việc tuân thủ các quy tắc nối các đầu vào đầu ra đó. Ngoài ra còn có các kiểm tra khác như kiểm tra các vị trí vào ra (PAD check), kiểm tra các khói đệm cho tín hiệu đồng bộ (Clock buffer check), kiểm tra sự trùng lặp tên gọi(Name check).
- Nếu như bước DRC không phát sinh ra lỗi thì quá trình maping được thực hiện tiếp tục, ở bước này sẽ tiến hành lược bỏ các phần tử thừa trong thiết kế và ánh xạ các khói thiết kế lên các khói chức năng của đối tượng FPGA cụ thể...
- Tạo ra một file chứa các điều kiện ràng buộc của mô tả vật lý của mạch sinh ra bởi bước trên PCF (*Physical Constraint File*), với nội dung là tập hợp tất cả các điều kiện ràng buộc của thiết kế ở cấp độ cổng (NCF, UCF) và đối tượng FPGA cụ thể.
- Thực hiện DRC với thiết kế đã ánh xạ, nếu DRC ở bước này không gây ra lỗi thì sẽ thực hiện bước cuối cùng là tạo ra file NCD.

### 3.3.3. Place and Routing

*Placing & Routing* (PAR) là quá trình ánh xạ những khối logic đã được phân chia ở phần Maping sang những khối logic (LUT, IOBUF...) có vị trí cụ cù

thể trên FPGA và kết nối chúng lại với nhau thông qua khối tài nguyên kết nối . người thiết kế có thể can thiệp vào quá trình này bằng FPGA editor, một công cụ giao diện đồ họa tích hợp trong ISE, nhưng trên thực tế thì quá trình này thường thực hiện hoàn toàn tự động bằng công cụ PAR (Place and Route).

Hình dưới đây minh họa cho quá hai quá trình trên.



Hình 4-50. *Phân bố và kết nối*

*Placing:* Ở bước này PAR lựa chọn các khối logic chức năng cụ thể phân bố trên FPGA để gán cho các khối chức năng trên mô tả thiết kế, việc lựa chọn dựa trên các tiêu chí như nguồn tài nguyên, độ dài kết nối, điều kiện ràng buộc trong PCF file... Quá trình này thực hiện thông qua một số pha, kết thúc mỗi pha thì thiết kế được tối ưu thêm một mức, kết thúc Placing một kết quả NCD mới được tạo ra.

*Routing:* là quá trình tiến hành sử dụng các tài nguyên kết nối (interconnects), các kết nối được thực hiện nhằm đạt thời gian trễ thấp nhất có thể, khi kết nối PAR sẽ phải quan tâm tới thông tin trong PCF file. Quá trình này cũng được thực hiện thành nhiều pha, ở mỗi pha một file NCD mới sẽ được lưu lại nếu như có được sự tối ưu về thời gian so với phương án trước đó.

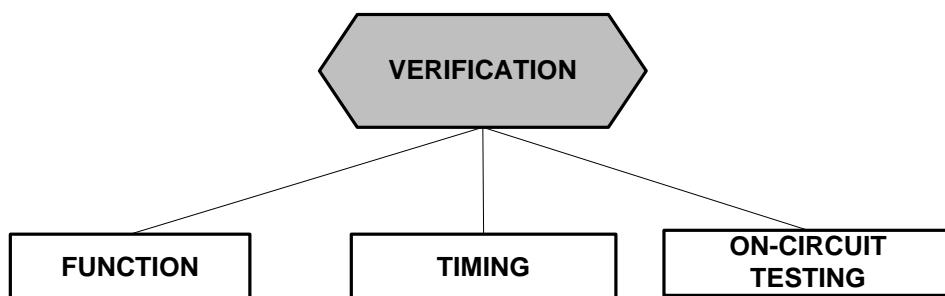
*Floorplaning:* Là quá trình cho phép người thiết kế sử dụng FPGA editor để can thiệp vào quá trình Placing và Routing, bước này có thể làm trước hoặc sau các bước của PAR.

### 3.4. Cấu hình FPGA

Sau khi thiết kế đã được hiện thực hóa ở bước 3.3 thì có thể thực hiện cấu trúc FPGA (*FPGA Configuration*), quá trình cấu trúc như đã trình bày ở phần 2 là việc ghi dữ liệu vào SRAM, dữ liệu này sẽ quy định cách thức làm việc, kết nối của các phần tử trong FPGA. Thiết kế được dịch sang 1 file cấu hình (BIT file) và nạp vào trong FPGA thông qua giao thức JTAG. File BIT này cũng có thể được biến đổi thành các định dạng PROM khác nhau để nạp vào trong ROM, khi cần sẽ được đọc để tái cấu trúc FPGA mà không cần phải nạp lại từ máy tính.

### 3.5. Kiểm tra thiết kế trên FPGA

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 4-51. *Kiểm tra thiết kế FPGA*

#### 3.5.1. 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.

### **3.5.2. 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.

### **3.5.3. Kiểm tra trực tiếp trên mạch**

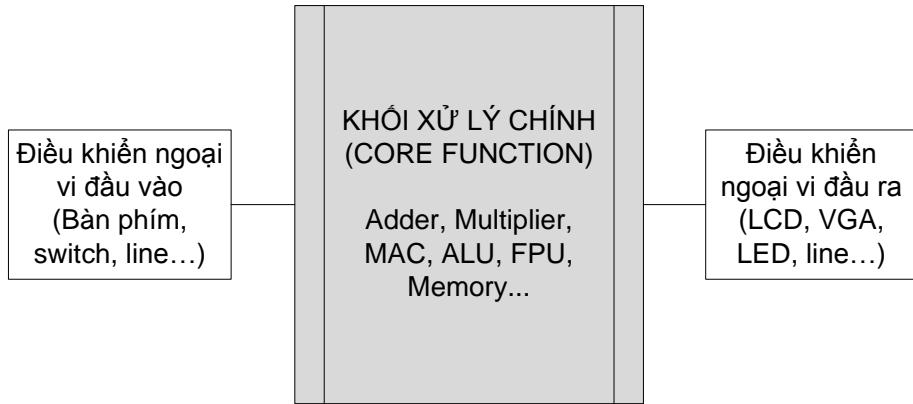
Kiểm tra trực tiếp trên mạch (On-circuit Testing) là quá trình thực hiện sau khi cấu hình FPGA đã được nạp vào IC, đối với những thiết kế đơn giản thì mạch được nạp có thể được kiểm tra một cách trực quan bằng các đối tượng như màn hình, LED, switch, cổng COM.

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.

[http://www.xilinx.com/support/documentation/sw\\_manuals/xilinx11/chipscope\\_pro\\_sw\\_cores\\_11\\_1\\_ug029.pdf](http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/chipscope_pro_sw_cores_11_1_ug029.pdf)

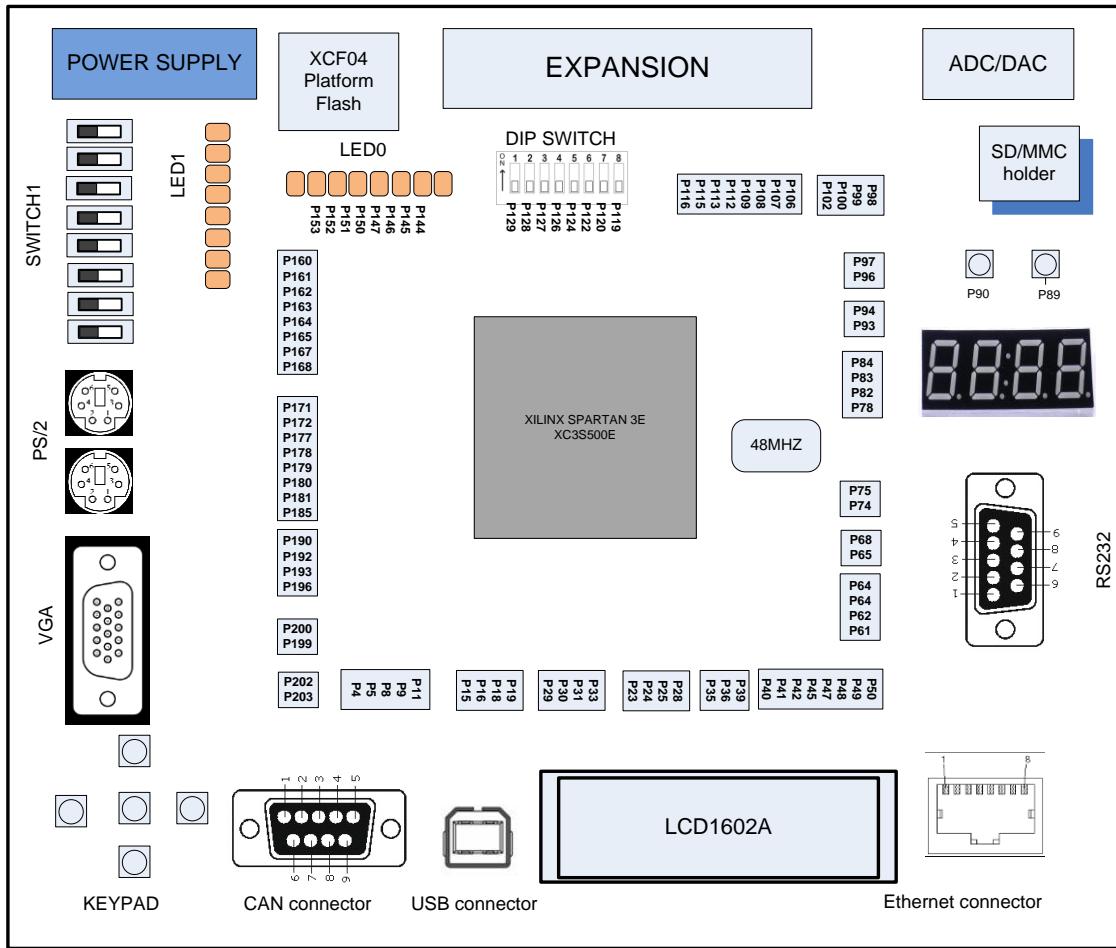
## **4. Một số ví dụ thiết kế trên FPGA bằng ISE**

Sơ đồ một khối thiết kế chuẩn trên FPGA có thể chia thành các khối chính như hình vẽ sau:



Hình 4.52. Sơ đồ khói thiết kế đầy đủ trên FPGA

Khối xử lý chính có thể là bất kỳ khói thiết kế logic chức năng nào chúng ta đã thực hiện ở những chương trước, lưu ý rằng những thiết kế được đề cập trong giáo trình hầu hết chỉ chiếm rất ít trên tổng số tài nguyên của FPGA. Khối điều khiển ngoại vi chia thành khói ngoại vi đầu vào và đầu ra. Tùy đặc tính từng loại mà độ phức tạp khi thiết kế sẽ khác nhau. Các ví dụ trình bày ở dưới đây là những ví dụ thiên về các giao tiếp cơ bản trên mạch FPGA. Tất cả đều được tổng hợp và kiểm tra trên mạch phát triển ứng dụng FPGA của bộ môn kỹ thuật Xung, số, vi xử lý (có hình vẽ kèm). Trên cơ sở các ví dụ này người học có thể thực hiện nạp các thiết kế chức năng nghiên cứu từ những chương trước với tùy biến đầu vào đầu ra bằng các ngoại vi có sẵn để tạo nên những thiết kế hoàn chỉnh và có thể kiểm tra các thiết kế đó trên mạch thực tế.



Hình 4-53. Sơ đồ mạch thí nghiệm FPGA

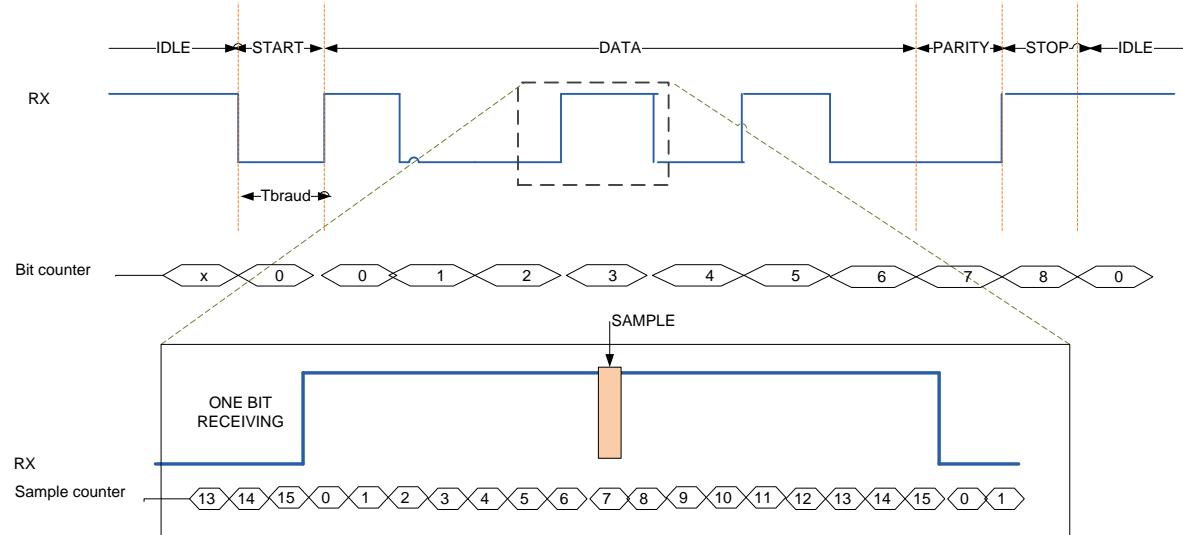
Trong mạch sử dụng IC FPGA Spartan 3E XCS500K, Flash ROM XFC04 với 4M cho lưu trữ cố định cấu hình. Mạch được nạp thông qua giao thức chuẩn JTAG. Các ngoại vi hỗ trợ bao gồm: Khối tạo xung nhịp tần số 48Mhz, Hệ thống các 7 phím ấn đa chức năng, 2 khối Switch 8-bit, 2 khối Led 8-bit, Led 7 đoạn với khả năng hiển thị 4 ký tự số, 2 cổng giao tiếp PS/2, cổng giao tiếp RS232, cổng giao tiếp USB-RS232, cổng giao tiếp VGA, màn hình LCD1602A hiển thị các ký tự văn bản, cổng giao tiếp CAN, cổng giao tiếp Ethernet, khối AD/DA sử dụng IC PCF8591. Mạch dùng một nguồn ngoài duy nhất 5V với dòng tối thiểu 1A. Chi tiết về mạch phát triển ứng dụng FPGA xem trong phụ lục 3.

#### 4.1. Thiết kế khối nhận thông tin UART

UART là khối thực hiện giao thức truyền tin dị bộ nối tiếp (*Universal Asynchronous Receiver/Transmitter*), đặc điểm giao thức này là đơn giản, phổ biến, nhược điểm là tốc độ trao đổi thông tin hạn chế. Trong ví dụ dưới đây ta sẽ

minh họa một thiết kế sử dụng FPGA để cấu hình một khối nhận thông tin nối tiếp UART.

Để hiểu về thiết kế này trước hết ta tìm hiểu qua về giao thức truyền nhận thông tin nối tiếp. Thông tin nối tiếp được truyền theo một dây dẫn duy nhất và các bit thông tin được mã hóa theo mức điện áp trên dây dẫn.



Hình 4-54. Giao thức truyền tin dị bộ nối tiếp

Hình trên mô tả tín hiệu đầu vào cho một khối nhận tín hiệu từ đường truyền nối tiếp, ở trạng thái nghỉ (IDLE), không có dữ liệu thì tín hiệu được giữ ở mức cao, để bắt đầu truyền thông tin tín hiệu Rx sẽ chuyển về mức thấp trong một khoảng thời gian đủ lớn, thời gian này bằng thời gian nhận 1 bit thông tin tương ứng với tốc độ truyền của cổng gọi là  $T_{\text{braud}} = 1/F_{\text{braud}}$ . Sau bit START này thì các bit dữ liệu được truyền nối tiếp trên Rx, tương ứng trạng thái DATA trên hình vẽ, cổng COM có thể được cấu hình để truyền nhận 6, 7 hay 8 bit thông tin. Sau khi kết thúc truyền các tin này có thể có thêm một bit kiểm tra chẵn lẻ của khối tin PARITY, một bit STOP (mức logic 1) được giữ với thời gian bằng 1, 1.5 hay 2  $T_{\text{braud}}$ . Khi kết thúc quá trình truyền tin thì Rx trở về trạng thái nghỉ ở mức điện áp cao.

Đường truyền nối tiếp đơn giản và tốc độ không cao, thông thường  $F_{\text{braud}}$  thay đổi từ 1200 đến 115200 Bit/s. Đường truyền này sử dụng các thiết bị độc lập với nhau và bị ảnh hưởng của nhiễu không nhỏ, mặt khác bộ nhận và bộ chia hoạt động không đồng bộ (không cùng xung nhịp hệ thống) do đó cần phải có cơ chế thu nhận nhằm tránh lỗi phát sinh. Cơ chế đó cũng được minh họa ở trên hình 4.49. Một bit thông tin được chia thành 16 điểm lấy mẫu (Samples), các điểm này được xác định bằng một bộ đếm mẫu (sample counter). Vì xác suất lỗi

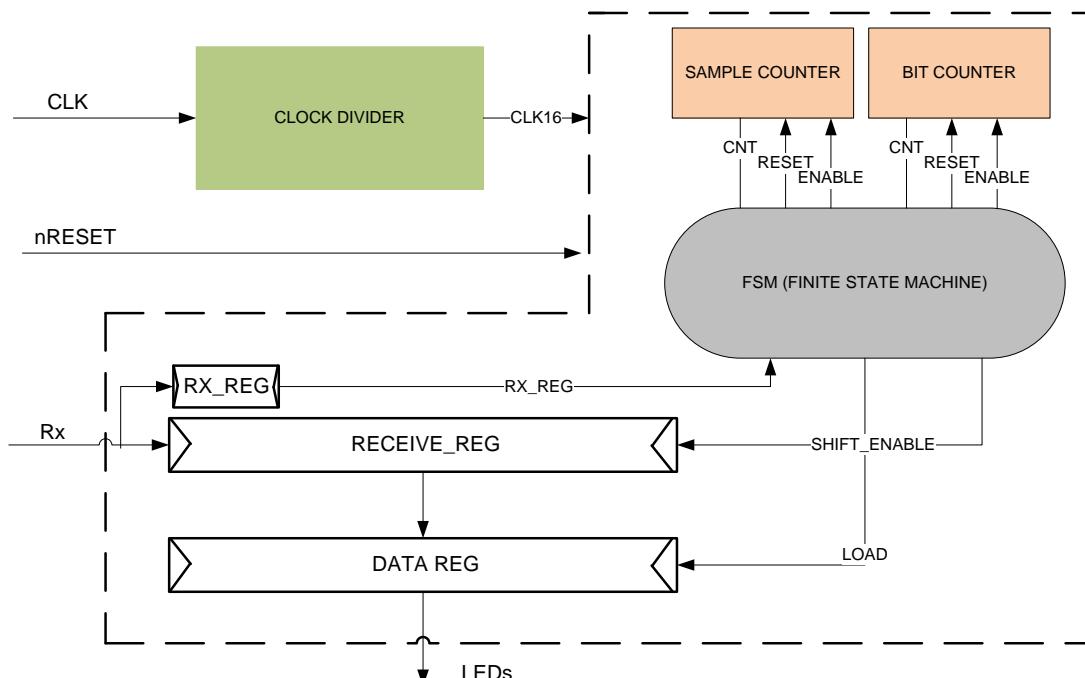
ở các vị trí mẫu đầu và cuối là cao nhất còn xác suất lỗi ở vị trí giữa là thấp nhất do đó ta chọn điểm lấy mẫu ở giữa, nghĩa là bit thông tin đang nhận bằng Rx ở trạng thái counter = 8.

Với tốc độ cao nhất có thể là 115200 Bit/s thì tần số của bộ đếm lấy mẫu bằng  $115200 \times 16 = 1\ 843\ 200$  Hz  $< 2\text{Mhz}$  vẫn là một tốc độ không cao đối với các thiết kế số trên FPGA. Trên thực tế cũng không phải lúc nào cũng phải chia bit thông tin thành 16 mẫu mà có thể chia lớn hơn hoặc nhỏ hơn.

Trên cơ sở giao thức như trên có thể thiết kế một khối nhận thông tin từ cổng COM với cấu hình:

- Tốc độ truyền Braud rate = 9600
- 7 bit dữ liệu (truyền ký tự ASCII) Data bit = 7,
- Hỗ trợ Bit Parity,
- 1 bit STOP.

Sơ đồ khái niệm của khái niệm nhận như ở hình dưới đây:

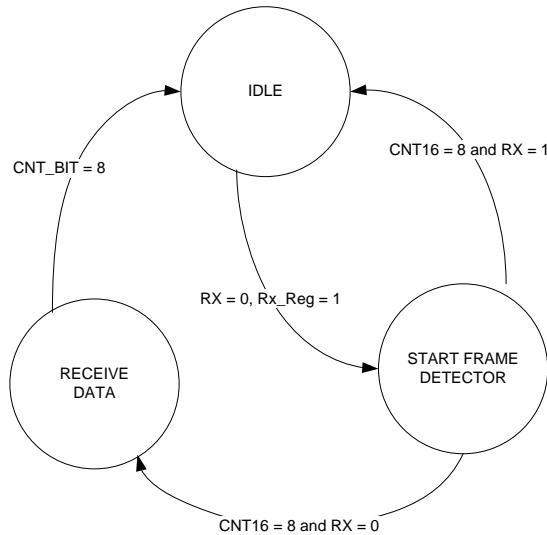


Hình 4-55. Sơ đồ khái niệm nhận thông tin nối tiếp

Tín hiệu Rx là đầu vào của cổng COM. Dao động cơ sở của mạch lấy từ bộ dao động thạch anh có tần số CLK = 27Mhz, tần số này được chia nhỏ một bộ chia tần với hệ số chia là  $(9600 \times 16)$  trước khi sử dụng làm xung nhịp hệ thống CLK16 cho toàn khái niệm, trong đó 9600 là tốc độ Braud. Tín hiệu nRESET cũng lấy từ 1 chân điều khiển của FPGA. Hai tín hiệu CLK16 và nRESET dùng chung cho

tất cả các khối khác trong bộ nhận. Để kiểm tra dữ liệu vào thì đơn giản các bit thông tin sẽ được gán cho các LEDs tương ứng trên mạch.

Bộ nhận được xây dựng dưới dạng máy trạng thái hữu hạn (Finite State Machine), ở ví dụ minh họa này ta chỉ sử dụng 3 trạng thái là trạng thái nhận biết bit START từ Rx - STATE FRAME DETECTOR, trạng thái nhận dữ liệu (kể cả PARITY bit và STOP bit) - DATA RECEIVING. Trạng thái nghỉ IDLE. Sơ đồ chuyển trạng thái như sau:



Hình 4-56. Máy trạng thái khói nhận thông tin nối tiếp

Từ trạng thái IDLE sẽ chuyển sang trạng thái STATE FRAME DETECT nếu như đường truyền chuyển từ mức 1 xuống mức 0, để bắt được bước chuyển này ta sử dụng một thanh ghi giữ chậm tín hiệu Rx có tên là Rx\_Reg, máy trạng thái ghi nhận Rx chuyển xuống mức 0 nếu Rx = 0 và Rx\_Reg = 1.

Khi đã chuyển sang trạng thái STATE FRAME DETECT máy trạng thái sẽ khởi động bộ đếm mẫu, tại vị trí lấy mẫu nếu Rx vẫn bằng 0 thì kết luận đây chính là tín hiệu START và sẽ chuyển sang trạng thái nhận dữ liệu RECEIVE DATA.

Theo lý thuyết ở trạng thái này chúng ta sẽ đếm đèn 8 và lấy mẫu ở điểm chính giữa nhưng vì khi xác nhận tín hiệu START ta xác nhận ở điểm giữa CNT = 8 và RESET ngay bộ đếm mẫu tại điểm này nên điểm lấy mẫu của ta không phải ở vị trí CNT = 8 nữa mà đối với các bit dữ liệu ta sẽ lấy tại các điểm CNT = 15.

Trong trạng thái nhận dữ liệu thì bộ đếm bit cũng làm việc, khi có 1 bit thông tin được nhận bộ đếm này cộng thêm 1, đồng thời máy trạng thái cũng sẽ

điều khiển thanh ghi dịch nhận dữ liệu RECEIVE\_REG thông qua tín hiệu SHIFT\_ENABLE để thanh ghi này dịch qua trái mỗi lần 1 bit, các bit thông tin được đẩy dần vào thanh ghi này cho tới đây.

Khi đã nhận đủ thông tin giá trị CNT\_BIT = 9 thì toàn bộ thông tin từ RECEIVE\_REG được ghi song song sang thanh ghi dữ liệu DATA\_REG, máy trạng thái chuyển về trạng thái nghỉ và chờ để nhận dữ liệu tiếp theo.

Mã thiết kế VHDL của khối này được ghép từ 6 khối nhỏ bao gồm khối chia tần, 2 khối đếm, thanh ghi dịch, thanh ghi song song, máy trạng thái fsm.vhd, và khối tổng receiver.vhd. Nội dung của các khối được liệt kê dưới đây:

#### **Khối chia tần clk\_div.vhd:**

```
-----
LIBRARY ieee;
USE ieee.Std_logic_1164.ALL;
USE ieee.Std_logic_unsigned.ALL;
-----
ENTITY clk_div IS
    GENERIC(baudDivide : std_logic_vector(7 downto 0) := "10101110");
    PORT(
        clk : in std_logic;
        clk16 : inout Std_logic);
END clk_div;
-----
ARCHITECTURE rtl OF clk_div IS
    SIGNAL cnt_div : Std_logic_vector(7 DOWNTO 0);
BEGIN
    --Clock Dividing Functions--
    process (CLK, cnt_div)
        begin
            if (Clk = '1' and Clk'event) then
                if (cnt_div = baudDivide) then
                    cnt_div <= "00000000";
                else
                    cnt_div <= cnt_div + 1;
                end if;
            end if;
        end process;

    process (cnt_div, clk16, CLK)
    begin
        if CLK = '1' and CLK'Event then
            if cnt_div = baudDivide then
                clk16 <= not clk16;
            end if;
        end if;
    end process;
```

```

        else
            clk16 <= clk16;
        end if;
    end if;
end process;
END rtl;
-----
```

Bộ đếm counter.vhd sử dụng tham số tĩnh để thiết lập số bit cho giá trị đếm:

```

LIBRARY ieee;
USE ieee.Std_logic_1164.ALL;
USE ieee.Std_logic_unsigned.ALL;
-----
```

```

ENTITY counter IS
    GENERIC(n : Positive := 3);
    PORT(
        clk          : in  std_logic;
        reset        : in  std_logic;
        clock_enable : in  std_logic;
        count_out    : out std_logic_vector((n-1) DOWNTO 0)
    );
END counter;
-----
```

```

ARCHITECTURE rtl OF counter IS
    SIGNAL cnt : std_logic_vector((n-1) DOWNTO 0);
BEGIN
    PROCESS (clk, reset)
    BEGIN
        if reset = '1' then
            cnt <= (others =>'0');
        elsif rising_edge(clk) then
            if clock_enable = '1' then
                cnt <= cnt + 1;
            end if;
        end if;
    END PROCESS;
    count_out <= cnt;
END rtl;
```

Thanh ghi dịch shifter.vhd:

```

library ieee;
use IEEE.STD_LOGIC_1164.ALL;
-----
```

```

entity shifter is
    generic (n : positive := 8);
    port (
        clk           : in std_logic;
        reset         : in std_logic;
        Rx            : in std_logic;
        shift_enable : in std_logic;
        shift_value  : inout std_logic_vector(n-1 downto
0)
    );
end entity;
-----
architecture rtl of shifter is

begin
    shifting: process (clk, reset)
    begin
        if reset = '1' then
            shift_value <= (others => '0');
        elsif clk = '1' and clk'event then
            if shift_enable = '1' then
                shift_value <= Rx & shift_value(n-1 downto 1);
            end if;
        end if;
    end process shifting;
end architecture;
-----
```

Thanh ghi song song reg.vhd:

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity reg is
generic (n: positive := 8);
port(
    WE      : in std_logic; -- write enable
    D       : in std_logic_vector(n-1 downto 0);
    Q       : out std_logic_vector(n-1 downto 0);
    CLK     : in std_logic;
    RESET   : in std_logic
);
end reg;
-----
architecture behavioral of reg is
signal Q_sig : std_logic_vector(n-1 downto 0);
begin
```

```

reg_p: process (CLK, RESET)
begin
    if RESET = '1' then
        Q_sig <= (others => '0');
    elsif CLK = '1' and CLK'event then
        if WE = '1' then
            Q_sig <= D;
        end if;
    end if;
end process reg_p;
Q <= Q_sig;
end behavioral;
-----
```

Máy trạng thái hữu hạn fsm.vhd:

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
-----
ENTITY fsm IS
    PORT(
        RESET          : in  std_logic;
        Rx             : in  std_logic;
        cnt16         : in  std_logic_vector (3 downto 0);
        cnt_bit        : in  std_logic_vector (3 downto 0);
        clk16          : in  std_logic;
        cnt_bit_reset : out std_logic;
        cnt_bit_enable: out std_logic;
        cnt16_reset   : out std_logic;
        cnt16_enable   : out std_logic;
        shift_enable   : out std_logic;
        data_reg_WE   : out std_logic);
END fsm;
-----
architecture rtl of fsm is

    signal Rx_reg           : std_logic;
    signal state             : std_logic_vector (1 downto 0);
    constant Idle_state     : std_logic_vector (1 downto 0) := "00";
    constant start_frame_state : std_logic_vector (1 downto 0)
        := "01";
    constant receive_state   : std_logic_vector (1 downto 0)
        := "10";

    Begin
        reg_RX: process (clk16)
        begin
```

```

if clk16 = '1' and clk16'event then
    Rx_reg <= Rx;
end if;
end process reg_RX;
receiving: process (clk16, Rx, RESET, Rx_reg)
begin
    if RESET = '1' then
        state <= Idle_state;
    elsif clk16 = '1' and clk16'event then
        case state is
            when Idle_state =>
                cnt16_reset <= '0';
                data_reg_WE <= '0';
                cnt_bit_reset <= '0';
                if Rx = '0' and Rx_reg = '1' then
                    state <= start_frame_state;
                    cnt16_enable <= '1';
                    cnt16_reset <= '1';
                end if;
            when start_frame_state =>
                cnt16_reset <= '0';
                if cnt16 = "0101" then
                    if Rx = '0' then
                        cnt16_enable <= '1';
                        cnt16_reset <= '1';
                        cnt_bit_reset <= '1';
                        state <= receive_state;
                    else
                        cnt16_enable <= '0';
                        state <= Idle_state;
                    end if;
                end if;
            when receive_state =>
                cnt_bit_reset <= '0';
                cnt16_reset <= '0';
                if cnt16 = "1110" then --luu data Rx
                    cnt_bit_enable <= '1';
                    shift_enable <= '1';
                else
                    cnt_bit_enable <= '0';
                    shift_enable <= '0';
                end if;
                if cnt_bit = "1000" then --nhan den bit thu 8
                    data_reg_WE <= '1';
                    cnt16_enable <= '0';
                    cnt16_reset <= '1';
                end if;
        end case;
    end if;
end process receiving;

```

```

        cnt_bit_enable <= '0';
        cnt_bit_reset <= '1';
        shift_enable <= '0';
        state <= Idle_state;
    end if;
    when others =>
        cnt16_enable <= '0';
        cnt_bit_enable <= '0';
        shift_enable <= '0';
        state <= Idle_state;
    end case;
end if;
end process;
end architecture rtl;
-----
```

**Khối tổng (receiver) receiver.vhd:**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity Receiver is
generic (n: positive := 8);
port(
    Rx : in std_logic;
    data_out : out std_logic_vector(7 downto 0);
    CLK : in std_logic;
    nRESET : in std_logic
);
end Receiver;
-----
architecture behavioral of Receiver is

signal RESET          : std_logic;
signal clk16           : std_logic;
signal shift_enable    : std_logic;
signal shift_value     : std_logic_vector(9 downto 0);
signal cnt_bit_reset   : std_logic;
signal cnt_bit_enable  : std_logic;
signal cnt_bit         : std_logic_vector (3 downto 0);
signal cnt16_reset     : std_logic;
signal cnt16_enable    : std_logic;
signal cnt16           : std_logic_vector (3 downto 0);
signal data_reg_WE     : std_logic;
-----
component clk_div is
  GENERIC (

```

```

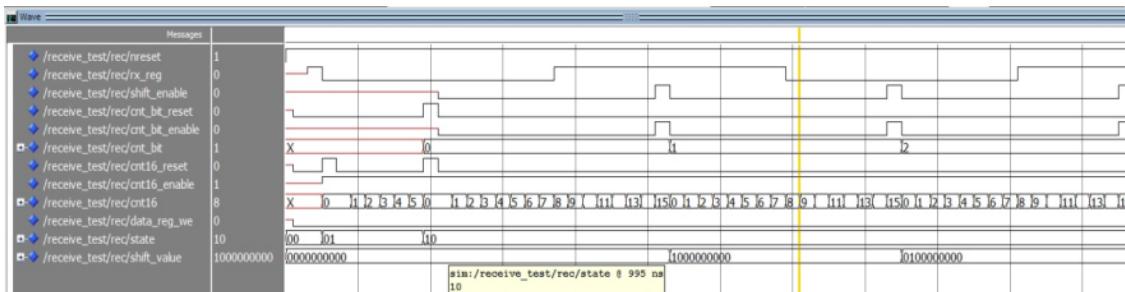
baudDivide : std_logic_vector(7 downto 0) :=
"10101110");
PORT(
    clk      : in std_logic;
    clk16   : inout Std_logic);
end component;
-----
component reg is
generic (n: positive := 8);
port(
    WE   : in std_logic;
    D    : in std_logic_vector(n-1 downto 0);
    Q    : out std_logic_vector(n-1 downto 0);
    CLK  : in std_logic;
    RESET : in std_logic);
end component;
-----
component counter IS
    GENERIC(n : Positive := 3);
    PORT(
        clk, reset, clock_enable : IN Std_logic;
        count_out: out Std_logic_vector((n-1) DOWNTO 0));
END component;
-----
component shifter is
generic (n : positive := 8);
port(
    clk      : in std_logic;
    RESET   : in std_logic;
    Rx      : in std_logic;
    shift_enable : in std_logic;
    shift_value  : inout std_logic_vector(n-1 downto 0));
end component;
-----
component fsm is
PORT(
    RESET          : in  std_logic;
    Rx             : in  std_logic;
    cnt16         : in  std_logic_vector (3 downto 0);
    cnt_bit       : in  std_logic_vector (3 downto 0);
    clk16         : in  std_logic;
    cnt_bit_reset : out std_logic;
    cnt_bit_enable: out std_logic;
    cnt16_reset   : out std_logic;
    cnt16_enable  : out std_logic;
    shift_enable   : out std_logic;

```

```

        data_reg_WE      : out std_logic);
end component;
-----
begin
    RESET <= not nRESET;
    finish_state_machine: component fsm
        port map (
            RESET, Rx, cnt16, cnt_bit, clk16,
            cnt_bit_reset, cnt_bit_enable, cnt16_reset,
            cnt16_enable, shift_enable, data_reg_WE);
    clock_divide16: component clk_div
        generic map ("01011001") -- = 27Mhz/9600/16/2
        port map (clk, clk16);
    receive_reg: component shifter
        generic map (10)
        port map (clk16, RESET, Rx, shift_enable, shift_value);
    counter16: component counter
        generic map (4)
        port map (clk16, cnt16_reset, cnt16_enable, cnt16);
    counter_bit: component counter
        generic map (4)
        port map (clk16, cnt_bit_reset, cnt_bit_enable,
cnt_bit);
    data_reg: component reg
        generic map (8)
        port map (data_reg_WE, shift_value(9 downto 2),
data_out, clk16, RESET);
end behavioral;
-----
```

### Kết quả mô phỏng trên Modelsim



Hình 4-57. Kết quả mô phỏng khôi nhận UART

Sau khi khởi động hoặc sau khi tín hiệu nRESET tích cực khôi nhận đều chuyển về sang thái IDLE (state = 00). Ở trạng thái này khi Rx = 0 và Rx\_reg = 1 bộ đếm mẫu cnt16 bắt đầu đếm và trạng thái chuyển sang START\_FRAME\_DETECT (state = 01), khi cnt16 đếm đến giá trị bằng 5 tức là 6 mẫu, nếu kể cả 1 mẫu trễ do Rx\_reg thì vị trí mẫu 7/16, nếu đến vị trí này mà

Rx bằng 0 thì khôi nhận xem tín hiệu trên Rx là tín hiệu START và bắt đầu chuyển sang trạng thái nhận dữ liệu RECEIVE DATA(state = 10), ở trạng thái này bộ đếm mẫu cnt16 đếm liên tục từ 0 đến 15, vị trí lấy mẫu là vị trí khi cnt16 = 15. Ở ví dụ mô phỏng trên tín hiệu Rx cứ 16 xung nhịp của clk16 thì thay đổi từ 0 lên 1 hoặc 1 xuống 0 do đó ta sẽ thu được tuần tự các bit 0101... tương ứng đầy dần vào thanh ghi dịch shift\_value. Quá trình thu kết thúc khi bộ đếm bit counter\_bit = 8, tương ứng đã nhận đủ 7 bit dữ liệu, 1 bit PARITY và 1 bit STOP.

Trước khi tổng hợp và hiện thực thiết kế trên FPGA cần thực hiện gán các chân tín hiệu, thiết lập ràng buộc cho mạch trên Kit FPGA. Để gán các chân tín hiệu trên FPGA ta sử dụng giao diện đồ họa trong chương trình PlanAhead hoặc tạo một tệp có đuôi mở rộng receiver.ucf với nội dung như sau:

```

INST "CLK_BUFGP" LOC = BUFGMUX_X2Y10;
NET "CLK" LOC = P184;
NET "nRESET" LOC = P29;
NET "Rx" LOC = P109;
NET "data_out[0]" LOC = P102;
NET "data_out[1]" LOC = P100;
NET "data_out[2]" LOC = P99;
NET "data_out[3]" LOC = P98;
NET "data_out[4]" LOC = P97;
NET "data_out[5]" LOC = P96;
NET "data_out[6]" LOC = P94;
NET "data_out[7]" LOC = P93;
NET "clock_divide16/clk161" TNM_NET =
"clock_divide16/clk161";
TIMESPEC TS_clock_divide16_clk161 = PERIOD
"clock_divide16/clk161" 20 ns HIGH 50 %;
NET "CLK" TNM_NET = "CLK";
TIMESPEC TS_CLK = PERIOD "CLK" 10 ns HIGH 50 %;
(*) Lưu ý rằng vị trí các cổng có thể khác nhau cho các
mạch FPGA khác nhau.

```

Các dòng bắt đầu bằng từ khóa NET thiết lập cài đặt cho các cổng vào ra của thiết kế tương ứng với các vị trí trên mạch thật, mỗi vị trí có một ký hiệu và số thứ tự riêng. Cổng CLK được lấy từ chân tạo dao động thạch anh P184 trên mạch, tương ứng trên sơ đồ FPGA dưới đây chân CLK nằm gần hai khối DCM ở trên. Chân nRESET được lấy từ vị trí P29 là một nút nhấn để tạo xung RESET, nút nhấn trên mạch có mức tích cực âm, vị trí của nRESET nằm ở giữa cạnh trái. Các chân dữ liệu ra được gán cho vị trí của 8 đèn LED trên mạch, vị trí của các cổng này tại góc dưới phải của mạch. Và cuối cùng là tín hiệu Rx lấy từ chân Rx mạch

tương ứng P109 của FPGA, vị trí cổng này nằm gần vị trí chân các đèn LED trên cạnh phải.

Ngoài thiết lập về vị trí, trong tệp này còn thiết lập điều kiện ràng buộc cho các tín hiệu CLK và CLK16, các tín hiệu này được định nghĩa là tín hiệu đồng bộ với yêu cầu về chu kỳ tương ứng không lớn hơn 10 ns và 20 ns. Trên thực tế những yêu cầu về thời gian không quan trọng vì khối nhận của chúng ta làm việc ở xung nhịp nhỏ hơn 2Mhz hay với  $T_{min} = 1/2\text{Mhz} = 500\text{ ns}$ . Các lệnh này chỉ có ý nghĩa báo cho trình biên dịch biết khi kết nối cần phải “xem” các tín hiệu này là tín hiệu đồng bộ.

### Kết quả tổng hợp trên FPGA

Device utilization summary:

```
-----
Selected Device : 3s500epq208-4
Number of Slices: 30 out of 4656 0%
Number of Slice Flip Flops: 41 out of 9312 0%
Number of 4 input LUTs: 44 out of 9312 0%
Number of IOs: 11
Number of bonded IOBs: 11 out of 158 6%
Number of GCLKs: 2 out of 24 8%
=====
```

### TIMING REPORT

Speed Grade: -4

Minimum period: 4.851ns (Maximum Frequency: 206.143MHz)  
Minimum input arrival time before clock: 5.216ns  
Maximum output required time after clock: 4.283ns  
Maximum combinational path delay: No path found

Timing Detail:

```
-----
All values displayed in nanoseconds (ns)
```

Kết quả sau tổng hợp bao gồm kết quả về sử dụng tài nguyên và kết quả về thời gian. Về sử dụng tài nguyên, khối nhận cổng COM sử dụng một lượng tài nguyên rất nhỏ 30/4656 SLICEs. Về mặt thời gian, thời gian trễ lớn nhất trước xung nhịp là 5,2 ns, tần số cực đại của mạch là ~ 200Mhz, trên thực tế mạch của chúng ta hoạt động ở xung nhịp nhỏ hơn 2Mhz nên những kết quả trên hoàn toàn đáp ứng yêu cầu thiết kế.

Kết quả về thời gian tĩnh sau khi thực hiện kết nối và phân bố (Post place & route static timing) là kết quả chính xác thu được trên mạch nạp, như quan sát sau kết quả này thay đổi không đáng kể so với kết quả sau tổng hợp.

All constraints were met.

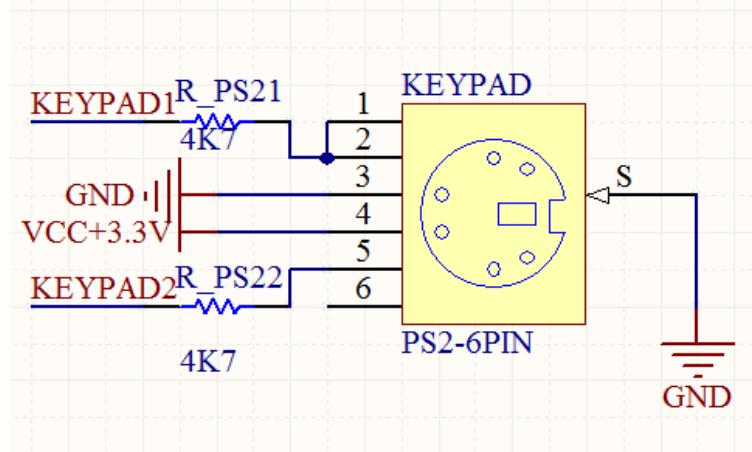
Data Sheet report:

```

-----
All values displayed in nanoseconds (ns)
Clock to Setup on destination clock CLK
Source Clock|Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
-----+-----+-----+-----+-----+
CLK      | 5.004 |           |           |           |
-----+-----+-----+-----+-----+

```

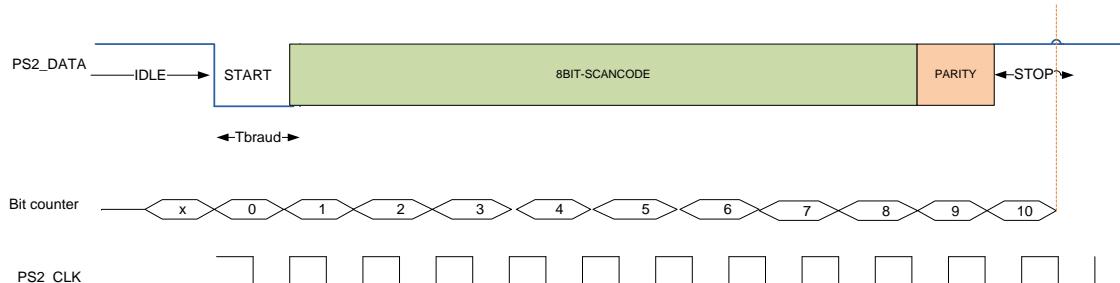
#### 4.2. Thiết kế khối điều khiển PS/2 cho Keyboard, Mouse



Hình 4-58. Mạch giao tiếp PS/2

Cổng giao tiếp PS/2 xuất phát từ tên gọi sản phẩm máy tính thứ hai (Personal System/2) của IBM. Cổng này dùng để nối máy tính với các thiết bị ngoại vi như chuột và bàn phím. Cổng giao tiếp này có tất cả 6 chân đánh số từ 1 đến 6. Chân số 3 và 4 tương ứng là GND và nguồn. Nguồn sử dụng là 5V nhưng thực tế với thiết kế trên FPGA có thể sử dụng mức điện áp 3.3V. Chân 5 là chân tín hiệu đồng bộ CLK. Còn chân 1, 2 là các chân dữ liệu DATA1 và DATA2, ngầm định bàn phím giao tiếp qua DATA1, bàn phím qua chân DATA2 và có thể dùng một cổng PS/2 với 1 thiết bị chia để kết nối với cả bàn phím và chuột, tuy vậy để đơn giản ta dùng hai cổng riêng biệt và nối cứng DATA1 và DATA2 với nhau như trên hình vẽ, có nghĩa là cổng này có thể kết nối với cả hai loại ngoại vi.

Giao thức điều khiển cho các thiết bị này rất giống với giao thức của cổng COM như trình bày ở trên. Điểm khác biệt duy nhất đó là các thiết bị này làm việc đồng bộ với thiết bị tiếp nhận thông tin (chung một tín hiệu CLK), giao thức truyền dữ liệu được thể hiện ở hình sau:



Hình 4-59. Giản đồ sóng của giao tiếp PS/2

Tín hiệu CLK thường được tạo bởi khối giao tiếp trong trường hợp này là FPGA, còn DATA là tín hiệu hai chiều. Yêu cầu về mặt thời gian cho các tín hiệu này thể hiện ở bảng sau:

Bảng 4-6

**Yêu cầu về mặt thời gian của giao tiếp PS/2**

Ký hiệu	Tên gọi	Tối đa	Tối thiểu
$T_{PS2\_CLK}$	Chu kỳ xung Clock	30us	50us
$T_{SU}$	Thời gian Setup của PS2_CLK	5us	25us
$T_{Hold}$	Thời gian Hold của PS2_CLK	5us	25us

Từ bảng trên có thể tính được tần số của tín hiệu CLK vào khoảng 20kHz -30Khz, với tần số thấp như vậy có thể yên tâm về đòi hỏi đối với  $T_{su}$  và  $T_{hold}$  nếu thiết kế trên FPGA.

Khi mỗi phím được ấn, bàn phím với giao tiếp PS/2 chuẩn sẽ gửi một mã tương ứng gồm 11-bit trong đó có 1 bit 0 là START BIT, sau là 8 bit mã quét phím (*Scan code*), bit tiếp theo là bit parity, 1 STOP bit (bảng 1). Mã *Scan code* là mã tương ứng duy nhất 1 ký tự trên bàn phím chuẩn với 8 bit nhị phân, bảng mã cho các ký tự thông dụng được liệt kê ở hình vẽ dưới đây (*nguồn Xilinx.com*):

<b>ESC</b> 76	<b>F1</b> 05	<b>F2</b> 06	<b>F3</b> 04	<b>F4</b> 0C	<b>F5</b> 03	<b>F6</b> 0B	<b>F7</b> 83	<b>F8</b> 0A	<b>F9</b> 01	<b>F10</b> 09	<b>F11</b> 78	<b>F12</b> 07	<b>↑</b> E0 75
' ~ OE 16	1! 1E	2@ 26	4\$ 25	5% 2E	6^ 36	7& 3D	8* 3E	9( 46	0) 45	- 4E	=+ 55	Back Space ← 66	→ E0 74
<b>TAB</b> 0D	<b>Q</b> 15	<b>W</b> 1D	<b>E</b> 24	<b>R</b> 2D	<b>T</b> 2C	<b>Y</b> 35	<b>U</b> 3C	<b>I</b> 43	<b>O</b> 44	<b>P</b> 4D	[{ 54	}] 5B	\ \br/>5D
<b>Caps Lock</b> 58	<b>A</b> 1C	<b>S</b> 1B	<b>D</b> 23	<b>F</b> 2B	<b>G</b> 34	<b>H</b> 33	<b>J</b> 3B	<b>K</b> 42	<b>L</b> 4B	:: 4C	.. 52	<b>Enter</b> ← 5A	← E0 6B
<b>Shift</b> ↑ 12	<b>Z</b> 1Z	<b>X</b> 22	<b>C</b> 21	<b>V</b> 2A	<b>B</b> 32	<b>N</b> 31	<b>M</b> 3A	,< 41	>. 49	/? 4A	Shift 59	↓ E0 72	
<b>Ctrl</b> 14	<b>Alt</b> 11	<b>Space</b> 29				<b>Alt</b> E0 11	<b>Ctrl</b> E0 14						

Hình 4-60. Bảng mã SCAN CODE của bàn phím chuẩn

Nếu một phím được ấn và giữ thì bàn phím sẽ liên tục gửi chuỗi giá trị tương ứng trong với tần suất 100ms/mã. Khi phím được thả thì bàn phím sẽ gửi mã có giá trị scan code là F0 theo sau là mã phím được ấn.

Khi một phím chức năng như Alt, Control phải, các phím mũi tên gọi chung là các phím mở rộng thì bàn phím gửi mã là E0 kèm theo mã phím ấn, khi phím dạng này thả thì tổ hợp E0, F0 được gửi và sau đó là mã phím ấn.

Không có các mã khác nhau để phân biệt chữ cái hoa và thường mà để phân biệt phải sử dụng thêm cờ trạng thái phím SHIFT có đang được giữ không, bản thân phím SHIFT được coi như một phím thông thường.

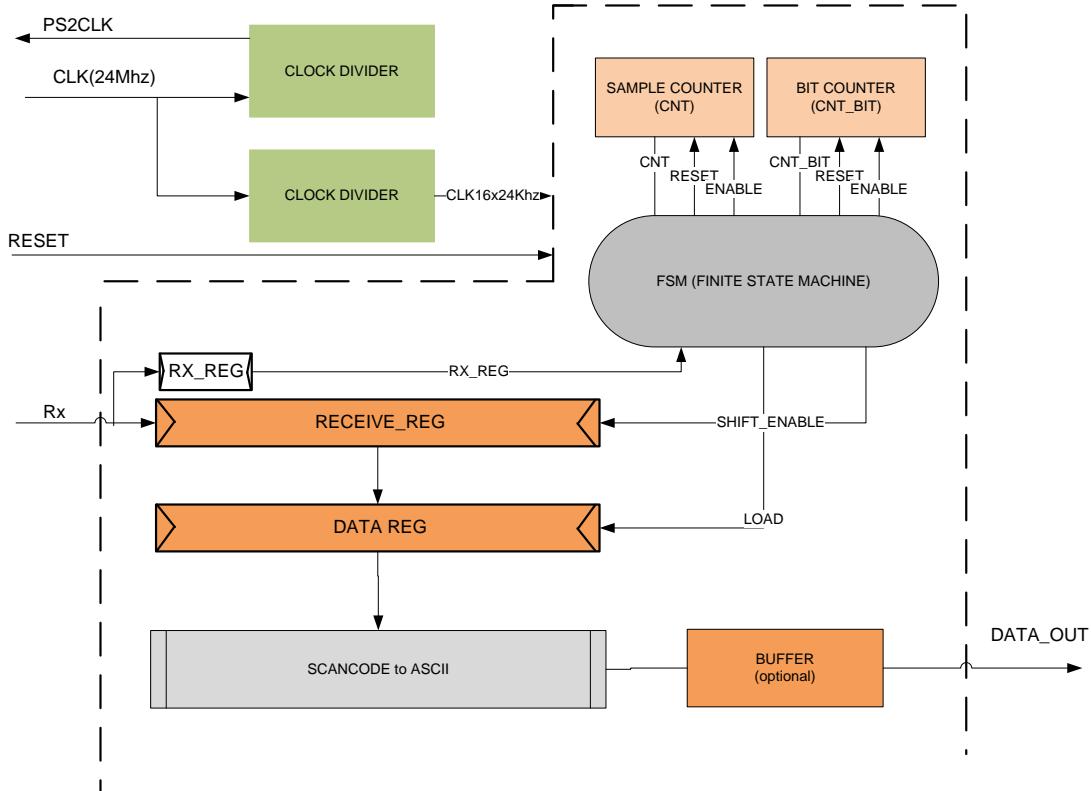
Ngoài việc nhận mã phím ấn thì thiết bị tiếp nhận đầu vào từ bàn phím có thể sử dụng giao thức đường truyền trên để gửi một số lệnh điều khiển đơn giản tới bàn phím, các lệnh đó liệt kê ở bảng sau:

Bảng 4-7

#### Các lệnh giao tiếp với bàn phím chuẩn

Lệnh	Mô tả
ED	Bật/ tắt các đèn điều khiển. Khi nhận lệnh này bàn phím trả lời bằng giá trị FA. Ké tiếp một giá trị được gửi tới bàn phím với nội dung như sau: Bit từ 7 đến 3: Không quan tâm Bít 2: CapLock Led Bit 1: NumLock Led Bit 0: Scroll Lock Led
EE	Echo: bàn phím nhận lệnh này sẽ trả lời bằng giá trị EE, sử dụng để kiểm tra bàn phím.
F3	Đặt thời gian lặp phím hay độ nhạy phím, sau lệnh này bàn phím gửi giá trị FA, 8bit ké tiếp gửi tới bàn phím để xác lập thời gian lặp.
FE	Khi gặp lệnh này bàn phím gửi lại mã phím được ấn gần nhất.
FF	Lệnh Reset lại bàn phím.

Lưu ý rằng bàn phím có thể hoạt động mà không nhất thiết phải có một quá trình khởi tạo vì vậy đơn giản có thể thiết kế khôi giao tiếp đơn thuần chỉ nhận tín hiệu từ bàn phím. Sơ đồ sau thể hiện một thiết kế đơn giản để giao tiếp với bàn phím.



Hình 4-61. Sơ đồ khối giao tiếp PS/2

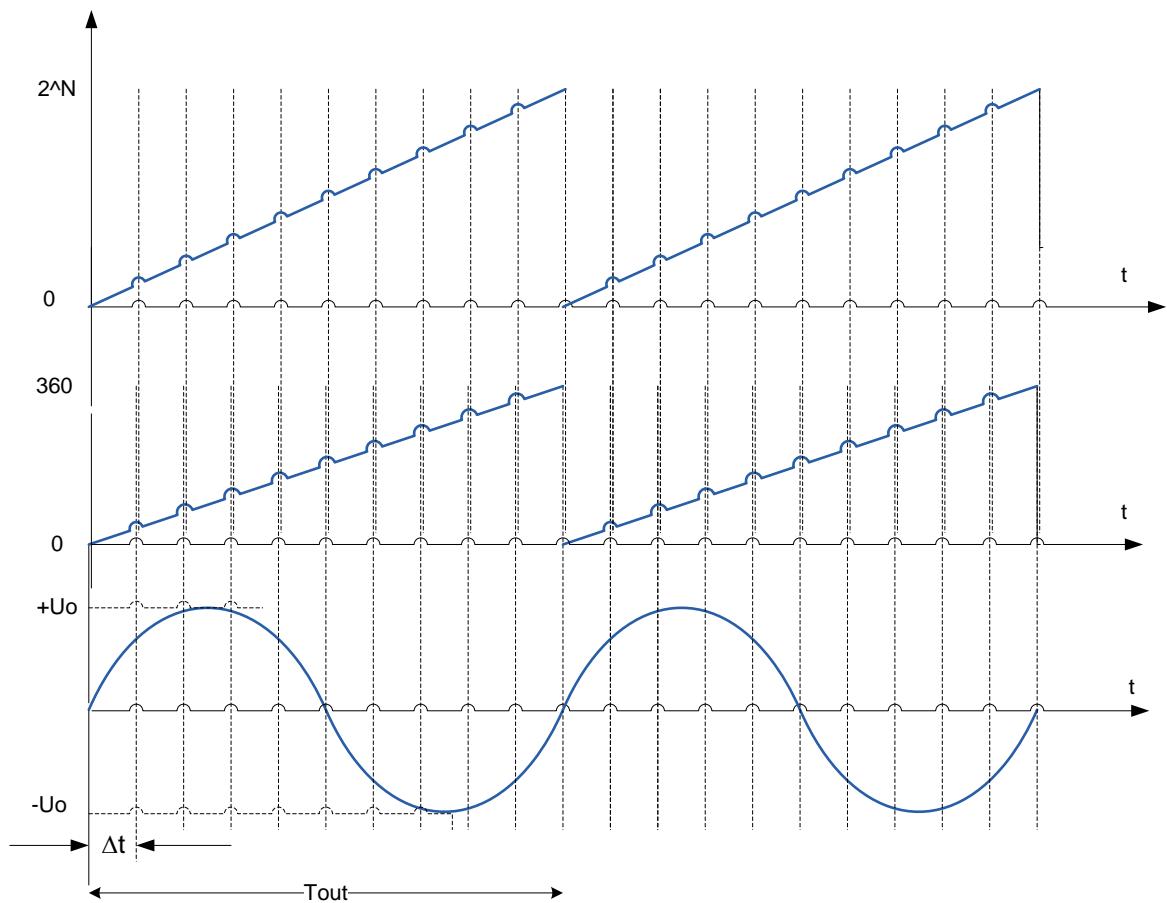
Sơ đồ này về cơ bản giữ nguyên phần nhận thông tin với giao thức UART của phần 4.1. Ta thực hiện điều chỉnh ở khối chia tần số, với giả sử ban đầu tần số thạch anh có sẵn là 24Mhz, ta chia 1000 để được tần số PS2\_CLK = 24kHz. Ngoài ra toàn bộ khối nhận hoạt động dưới sự điều khiển của 1 xung Clock có tần số bằng PS2\_CLK x 16 = 16 x24 = 384kHz, tần số này được tạo bởi khối chia tần thứ 2.

Vì mã thu được là mã SCancode do vậy cần một khối ROM để chuyển giá trị này thành mã ASCII (khối SCancode to ASCII). Kết quả sau khối ROM được lưu trữ trong BUFFER trước khi đưa đến các khối xử lý cấp cao hơn, tùy theo mục đích mà BUFFER này có thể được thiết kế hay không.

#### 4.3. Thiết kế khối tổng hợp dao động số NCO

NCO viết tắt của Numerically Controler Oscillator là khối tổng hợp dao động bằng vi mạch số. Khối này có khả năng tổng tạo ra dao động với tần số mong muốn một cách trực tiếp bằng một vi mạch số tích hợp.

Cơ sở toán học của NCO thể hiện như ở hình sau:



Hình 4-62. Cơ sở toán học của NCO

Sóng mà ta muốn tổng hợp có hàm số phụ thuộc thời gian và tần số như sau

$$Y(t) = U_o \sin(\Delta\varphi t) = U_o \sin(2\pi f t).$$

Nếu đặt  $\varphi(t) = \Delta\varphi t = 2\pi f t$  thì  $\varphi(t)$  là một hàm phụ thuộc tuyến tính theo thời gian, nếu biểu diễn giá trị pha theo thang 0-360° thì đồ thị theo thời gian tương ứng là đồ thị thứ hai từ trên xuống dưới ở hình trên.

Nếu chia nhỏ trục thời gian thành các điểm cách nhau liên tiếp một khoảng  $\Delta t$ , gọi các điểm chia trên trục thời gian lần lượt là 0,  $\Delta t, 2\Delta t, \dots, K\Delta t$ , Ta có

$$\varphi_k(t) = \varphi(k\Delta t) = \Delta\varphi k\Delta t = 2\pi f k\Delta t$$

Giá trị biên độ tương ứng là

$$y_k = U_o \sin(\varphi_k(t)) = U_o \sin(\Delta\varphi k\Delta t)$$

Áp dụng định lý Kachenhikov về rời rạc hóa và khôi phục tín hiệu có phô giới hạn ta có thể tính toán giá trị  $\Delta t$  để thu được tổ hợp các giá trị  $u_k$  đủ để khôi phục hoàn toàn sóng ban đầu giá trị đó phải thỏa mãn

$$\Delta t < 1/2f$$

Trong đó  $f$  là tần số cần tổng hợp.

Chuyển sang bài toán trên mạch số, các giá trị tích lũy pha và giá trị biên độ thứ nhất phải được số hóa, tức là biểu diễn dưới dạng số. Giá trị tích lũy pha không nhất thiết thay đổi từ  $0-360^\circ$  mà trên thực tế ta chọn một miền giá trị  $0-2^N$ , trong đó  $2^N > 360$ , để thu được độ chính xác và độ phân giải cao hơn.

Gọi  $p_0, p_1, \dots$  là giá trị rời rạc pha đã được số hóa các giá trị này được tính bằng một khối tích lũy pha  $N$  bit, bản chất là một khối cộng tích lũy. Đầu vào của khối cộng này là giá trị tích lũy  $N$  bit  $m$ . Sau một khoảng thời gian  $\Delta t = 1/f_0$  trong đó  $f_0$  là tần số cơ sở của mạch (tần số làm việc của bộ cộng tích lũy)

$$p_k = k \cdot m$$

giá trị lớn nhất mà khối cộng tích lũy biểu diễn được

$$p_L = \left[ \frac{2^N}{m} \right] \cdot m = L \cdot m$$

khi đó có thể tính được giá trị biên độ  $u_k$  được số hóa tương ứng trong bảng SIN là:

$$u_k = [U_0 \sin(\frac{p_k \cdot 2\pi}{2^N})] = [U_0 \sin(\frac{k \cdot m \cdot 2\pi}{2^N})]$$

Bảng SIN tương ứng 1 chu kỳ của sóng sin, để thực hiện hết một chu kỳ này thì cần một khoảng thời gian bằng

$$T_{out} = L \cdot \Delta t = \left[ \frac{2^N}{m} \right] \cdot \frac{1}{f_0}$$

Suy ra tần số của sóng thu được tính gần đúng theo công thức:

$$f_{out} = \frac{1}{T_{out}} = \frac{F_{clk}}{\left[ \frac{m}{2^N} \right]} \sim \frac{F_{clk} \cdot m}{2^N}$$

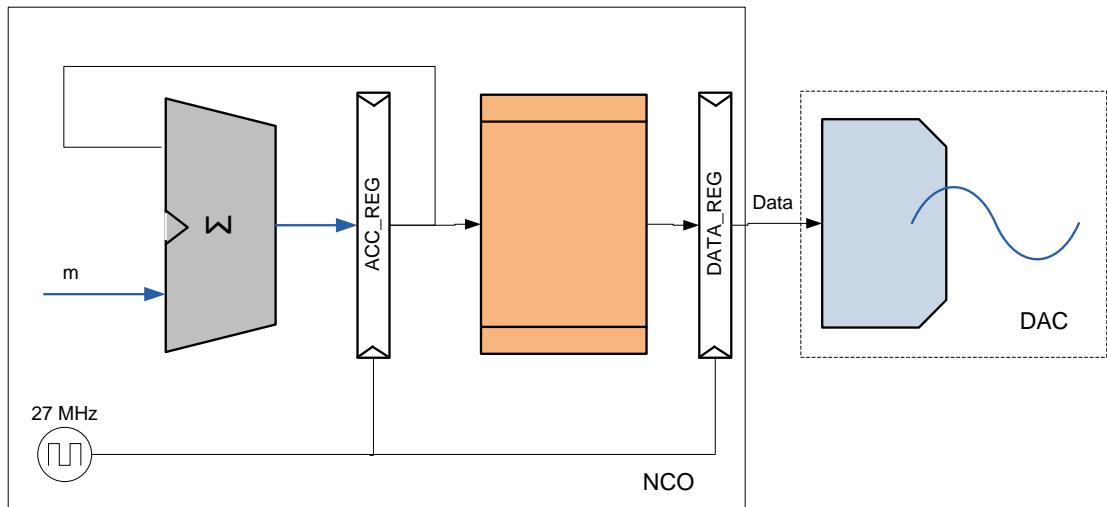
Như vậy tần số đầu ra phụ thuộc trực tiếp vào 3 tham số

- $N$  là số bit của dùng cho thanh ghi của bộ tích lũy pha.
- $m$  là từ điều khiển tần số đầu vào.
- $F_{clk}$  là tần số xung nhịp cơ sở của mạch.

Cũng từ công thức trên có thể giới hạn được miền thay đổi của tần số ra của NCO

$$\frac{F_{clk}}{2^N} < f_{out} < F_{clk}$$

Hình dưới đây là mô tả sơ đồ khối của NCO:



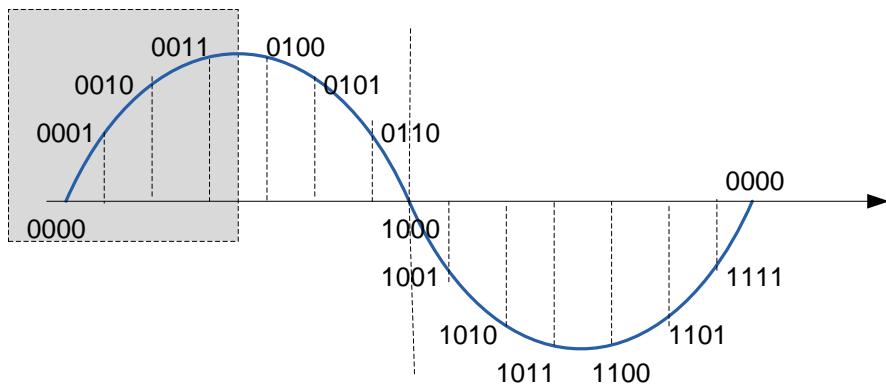
Hình 4-63 Sơ đồ khối của NCO

- Khối tích lũy pha là bộ cộng tích lũy có nhiệm vụ tạo ra tổ hợp các giá trị  $p_k$  theo thời gian, khối cộng tích lũy có đầu vào xung nhịp là xung nhịp cơ sở của mạch, giá trị tích lũy  $m$  hay còn gọi là từ điều khiển tần số.

- Khối tham chiếu bảng SIN : từ giá trị pha tích lũy  $p_k$  để tham chiếu giá trị biên độ sóng SIN  $u_k$  tương ứng, khối này bản chất là một khối ROM hoặc RAM lưu trữ các giá trị của chu kỳ sóng được lượng tử và số hóa.

Đầu ra của NCO là các giá trị số rời rạc theo thời gian, để thu được sóng SIN cần thêm một khối biến đổi tín hiệu số-tương tự DAC.

Khi thiết kế khối SIN ROM cần lưu ý một đặc điểm của sóng SIN là hàm lẻ nên trong một chu kỳ giá trị biên độ ở hai nửa chu kỳ là đối nhau, còn trong một nửa chu kỳ thì sóng SIN là một hàm chẵn, đối xứng qua trực  $k\pi/2$ . Do đó để lưu trữ các giá trị trong một chu kỳ sóng chỉ cần lưu trữ giá trị của  $1/4$  chu kỳ. Để hiểu cụ thể hơn quan sát hình sau:



Hình 4-64. Mã hóa bảng SIN ROM cho N = 4

Với trường hợp N=4 ta có mã hóa cho bảng SIN ROM như hình trên, các giá trị 4-bit  $x_3x_2x_1x_0$  trên hình tương ứng là địa chỉ đầu vào của khối ROM. Với nhận xét như ở trên ta thấy thay vì sử dụng bảng ROM 16 giá trị có thể sử dụng bảng ROM 4 giá trị (miền đánh dấu đậm), bảng ROM này có địa chỉ biểu diễn bằng 2 bit  $x_1x_0$ . Ta giả sử giá trị biên độ SIN cũng biểu diễn dưới dạng 4-bit  $d_3d_2d_1d_0$ . Khi đó:

Nếu  $x_3x_2 = 00$  thì giá trị tương ứng giữ nguyên.

$$d = (d)_{00x_1x_0}$$

Nếu  $x_3x_2 = 01$  thì giá trị tương của biên độ

$$d = (d)_{01x_1x_0} = (d)_{00\bar{x}_1\bar{x}_0}$$

Nếu  $x_3x_2 = 10$  thì giá trị tương của biên độ

$$d = (d)_{10x_1x_0} = -(d)_{00x_1x_0}$$

Nếu  $x_3x_2 = 11$  thì giá trị tương của biên độ

$$d = (d)_{11x_1x_0} = -(d)_{00\bar{x}_1\bar{x}_0}$$

Toàn bộ thao tác trên có thể được thực hiện bằng một khối logic không phức tạp, bù lại ta sẽ tiết kiệm được  $\frac{3}{4}$  số lượng ô nhớ cần cho bảng ROM. Việc mã hóa chỉ  $\frac{1}{4}$  bảng ROM cũng giúp tiết kiệm 1 bit dấu dùng để biểu diễn giá trị vì miền giá trị của bảng ROM luôn là giá trị dương.

Dưới đây là mã nguồn của một khối NCO dùng 6 bit cho giá trị tích lũy pha, tương ứng bảng ROM sẽ có  $2^6/4 = 16$  địa chỉ mã hóa bằng 4 bit, giá trị trong bảng ROM mã hóa bằng 4 bit với giá trị thay đổi từ 0 đến 15.

#### Mã nguồn khối SIN ROM sin\_rom.vhd

```
-----
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_signed.ALL;
use IEEE.STD_LOGIC_arith.ALL;
-----
entity sin_rom16 is
    port (
        clk      : in std_logic;
        cs       : in std_logic;
        rst      : in std_logic;
        address  : in std_logic_vector(5 downto 0);
        dataout  : out std_logic_vector (4 downto 0)
    );
end sin_rom16;
-----
architecture behavioral of sin_rom16 is
    signal addr :std_logic_vector (3 downto 0);
    signal data :std_logic_vector (3 downto 0);
    signal data1, data2 :std_logic_vector (4 downto 0);
----- Du lieu trong ROM duoc nap cac gia tri co dinh
begin
    sinrom: process (rst, cs, addr)
    begin
        if rst='0' and cs='1' then
            case addr is
                when x"0"  => data <= x"0";
                when x"1"  => data <= x"1";
                when x"2"  => data <= x"3";
                when x"3"  => data <= x"4";
                when x"4"  => data <= x"6";
                when x"5"  => data <= x"7";
                when x"6"  => data <= x"8";
                when x"7"  => data <= x"A";
                when x"8"  => data <= x"B";
                when x"9"  => data <= x"C";
                when x"A"  => data <= x"C";
                when x"B"  => data <= x"E";
                when x"C"  => data <= x"E";
                when x"D"  => data <= x"F";
                when x"E"  => data <= x"F";
            end case;
        end if;
    end process;
end;
```

```

        when x"F"    => data <= x"F";
        when others => null;
      end case;
      end if;
    end process sinrom;

data1 <= '0' & data;
data2 <= not (data1) + 1;
mod_address: process(address)
begin
  case address (4) is
    when '0' => addr <= address (3 downto 0);
    when '1' => addr <= not address (3 downto 0);
    when others => null;
  end case;
end process mod_address;
get_data: process (clk, data)
begin
  if clk = '1' and clk'event then
    case address (5) is
      when '0' => dataout <= data1;
      when '1' => dataout <= data2;
      when others => null;
    end case;
  end if;
end process get_data;
end behavioral;
-----
```

Bộ cộng đơn giản chỉ có cổng a, b và sum adder.vhd:

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-----
entity adder is
generic (N :  natural := 32);
port(
  A    : in  std_logic_vector(N-1 downto 0);
  B    : in  std_logic_vector(N-1 downto 0);
  SUM : out std_logic_vector(N-1 downto 0)
);
end adder;
-----
architecture behavioral of adder is
```

```

begin

    plus: process (A, B)
    begin
        sum <= a + b;
    end process plus;
end behavioral;
-----
```

**Thanh ghi reg.vhd:**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
```

---

```

entity reg is
generic (N : natural := 32);
port(
    D      : in std_logic_vector(N-1 downto 0);
    Q      : out std_logic_vector(N-1 downto 0);
    CLK    : in std_logic;
    RESET  : in std_logic
);
end reg;
-----
```

---

```

architecture behavioral of reg is
begin
    reg_p: process (CLK, RESET)
    begin
        if RESET = '1' then
            Q <= (others => '0');
        elsif CLK = '1' and CLK'event then
            Q <= D;
        end if;
    end process reg_p;
end behavioral;
```

---

**Khối cộng tích lũy accumulator.vhd:**

```

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

---

```

entity accumulator is
generic (N : natural := 32);
port(
    A      : in std_logic_vector(N-1 downto 0);
```

```

        Q      : out std_logic_vector(N-1 downto 0);
        CLK    : in  std_logic;
        RESET : in  std_logic
    );
end accumulator;
-----
architecture structure of accumulator is
signal sum : std_logic_vector(N-1 downto 0);
signal Q_sig : std_logic_vector(N-1 downto 0);
----COMPONENT ADDER----
component adder is
generic (N : natural := 32);
port(
    A    : in  std_logic_vector(N-1 downto 0);
    B    : in  std_logic_vector(N-1 downto 0);
    SUM : out std_logic_vector(N-1 downto 0)
);
end component;
----COMPONENT REG----
component reg is
generic (N : natural := 32);
port(
    D    : in  std_logic_vector(N-1 downto 0);
    Q    : out std_logic_vector(N-1 downto 0);
    CLK   : in  std_logic;
    RESET : in  std_logic
);
end component;
begin
add: component adder
    generic map (6)
    port map (A, Q_sig, sum);
regg: component reg
    generic map (6)
    port map (sum, Q_sig, CLK, RESET);
    Q <= Q_sig;
end structure;
-----
```

### **Khối nco nco.vhd:**

```

library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-----
entity nco is
```

```

port (
    m      : in std_logic_vector(5 downto 0);
    clk    : in std_logic;
    nRESET : in std_logic;
    cs     : in std_logic;
    dataout : out std_logic_vector(4 downto 0)
);
end entity;
-----
architecture behavioral of nco is
signal RESET   : std_logic;
signal m_reg   : std_logic_vector(5 downto 0);
signal address : std_logic_vector(5 downto 0);
-----
component accumulator is
generic (N : natural := 32);
port(
    A      : in std_logic_vector(N-1 downto 0);
    Q      : out std_logic_vector(N-1 downto 0);
    CLK    : in std_logic;
    RESET : in std_logic
);
-----
end component;
component sin_rom16 is
port (
    clk      : in std_logic;
    cs       : in std_logic;
    rst      : in std_logic;
    address : in std_logic_vector(5 downto 0);
    dataout : out std_logic_vector(4 downto 0)
);
end component;
-----
begin
    RESET <= not nRESET;
    process (clk, RESET)
    begin
        if RESET = '1' then
            m_reg <= (others => '0');
        elsif clk = '1' and clk'event then
            m_reg <= m;
        end if;

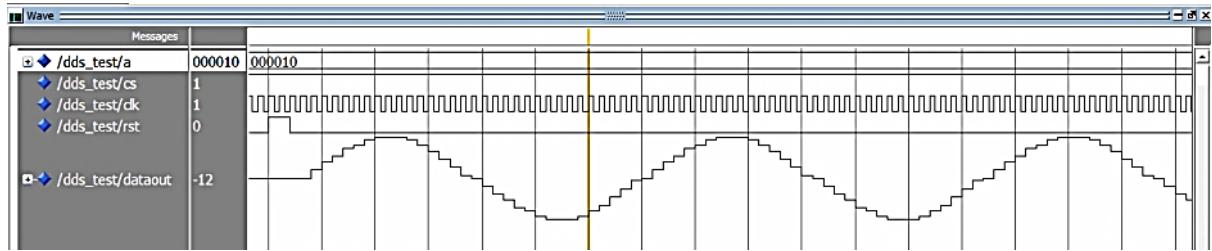
        end process;
    u1: accumulator

```

```

generic map (6)
port map (m_reg, address , clk , RESET);
u2: sin_rom16
port map (clk, cs, RESET, address, dataout);
end architecture behavioral;
-----
```

### Kết quả mô phỏng trên Modelsim



Hình 4-65. Kết quả mô phỏng khối NCO

Như quan sát trên giản đồ sóng giá trị tín hiệu ở đầu ra có dạng sóng Sin đúng như mong muốn, để tăng độ phân giải và thu được dạng sóng đầu ra tốt hơn thì tăng số bit dành cho thanh ghi tích lũy và số bit sử dụng cho biểu diễn giá trị biên độ sóng.

Nội dung nco.ucf:

```

NET "dataout[0]" LOC = "P102";
NET "dataout[1]" LOC = "p100";
NET "dataout[2]" LOC = "P99";
NET "dataout[3]" LOC = "p98";
NET "dataout[4]" LOC = "P97";
NET "m[0]" LOC = "P161";
NET "m[1]" LOC = "p162" ;
NET "m[2]" LOC = "P163";
NET "m[3]" LOC = "p164";
NET "m[4]" LOC = "P165";
NET "m[5]" LOC = "P167";
NET "CLK" LOC = P184;
NET "nRESET" LOC = P29;
NET "CLK" TNM_NET = "CLK";
TIMESPEC TS_CLK = PERIOD "CLK" 5.2 ns HIGH 50 %;
```

Tù điều khiển tần số được đặt giá trị tương ứng bằng 6 switch trên mạch, tín hiệu ra được gửi đến 5 LEDs. Về xung nhịp làm việc ta hạn chế xung CLK là 3.2 ns, con số này có được sau tổng hợp sơ bộ.

Kết quả tổng hợp trên FPGA

Device utilization summary:

```

Selected Device : 3s500epq208-4
Number of Slices: 11 out of 4656 0%
Number of Slice Flip Flops: 20 out of 9312 0%
Number of 4 input LUTs: 20 out of 9312 0%
Number of IOs: 14
Number of bonded IOBs: 14 out of 158 8%
IOB Flip Flops: 1
Number of GCLKs: 1 out of 24 4%
=====
Timing Summary:
-----
Speed Grade: -4
Minimum period: 3.702ns (Maximum Frequency: 270.124MHz)
Minimum input arrival time before clock: 2.360ns
Maximum output required time after clock: 4.283ns
Maximum combinational path delay: No path found
-----
All values displayed in nanoseconds (ns)

```

Kết quả sau tổng hợp bao gồm kết quả về sử dụng tài nguyên và kết quả về thời gian. Về sử dụng tài nguyên, NCO sử dụng 11/4656 SLICEs. Về mặt thời gian, xung nhịp cực đại là 270Mhz, con số này cũng là giới hạn cho tần số sê tổng hợp được ở đầu ra vì đây là giới hạn của tần số cơ sở.

Kết quả về thời gian tĩnh sau khi thực hiện kết nối và phân bổ thu được xung nhịp cực đại chính xác là 302Mhz.

```

All values displayed in nanoseconds (ns)
Clock to Setup on destination clock clk
Source Clock |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
-----+-----+-----+-----+
clk | 3.303 | | |
-----+-----+-----+-----+
Timing summary:
-----
```

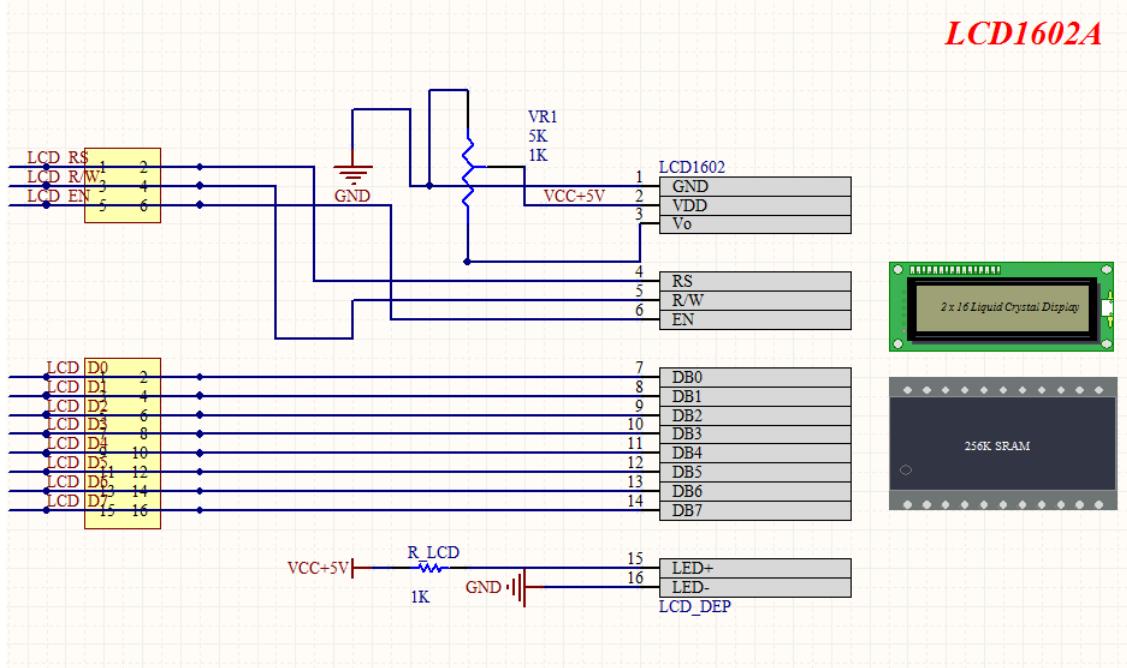
Timing errors: 1 Score: 103 (Setup/Max: 103, Hold: 0)  
Constraints cover 61 paths, 0 nets, and 28 connections  
Design statistics:

Minimum period: 3.303ns{1} (Maximum frequency: 302.755MHz)

*Lưu ý do DAC nhận các bit đầu vào dưới dạng số nguyên không dấu còn giá trị biểu diễn trên sóng SIN khi mô phỏng là số có dấu dưới dạng bù 2. Để DAC làm việc đúng và quan sát chính xác tín hiệu cần điều chỉnh lại mã nguồn để xuất các giá trị đầu ra dạng số nguyên dương.*

#### 4.4. Thiết kế khối điều khiển LCD1602A

Màn hình LCD1602A đơn sắc hiển thị 2x16 ký tự chuẩn được sử dụng khá rộng rãi trong các ứng dụng vừa và nhỏ vì tính đơn giản trong giao tiếp cũng như trong điều khiển. Tài liệu chi tiết về màn hình loại này có thể xem thêm ở [24], trong ví dụ này chỉ trình bày thiết kế giao tiếp LCD trong chế độ 8-bit với phần khởi tạo tối thiểu. Mạch giao tiếp LCD được thể hiện ở hình sau.



Hình 4-66. Mạch giao tiếp với LCD1602A

##### 4.3.1. Các chân giao tiếp của LCD1602A

- VDD, GND và Vo: Cáp nguồn - 5v và đất, chân Vo được dùng để điều chỉnh độ tương phản trên màn hình LCD, thông thường ta mắc một biến trở cỡ 5-10K để điều chỉnh mức điện áp vào chân này. Mặc dù điện áp nguồn của LCD là 5V nhưng LCD có thể giao tiếp với FPGA bằng mức điện áp 3.3V.

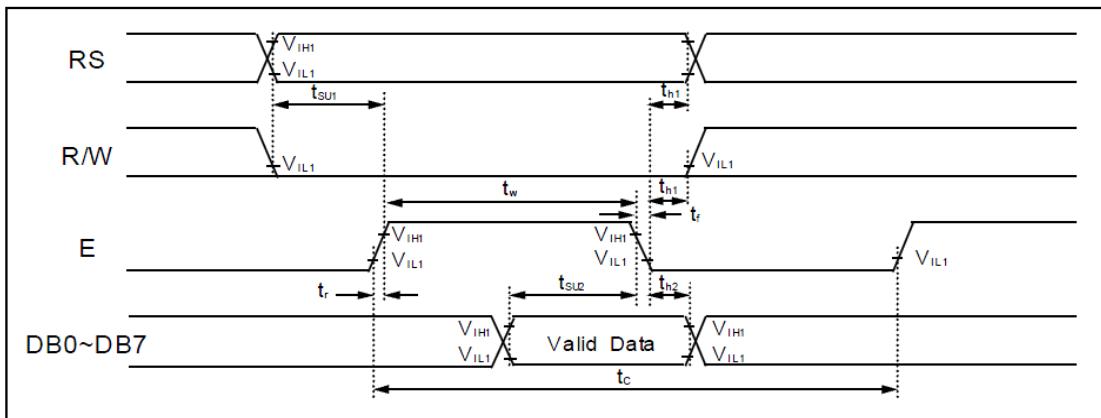
- Hai chân LED+, LED- dùng để cấp nguồn cho đèn Back Light tích hợp phía sau LCD để tăng độ sáng cho màn hình, có thể nối hoặc không.

- LCD\_RS (Register Select): Sử dụng để lựa chọn truy cập vào một trong hai dạng thanh ghi được tích hợp trong LCD: thanh ghi dữ liệu và thanh ghi lệnh. Nếu RS = 1 thì thanh ghi mã lệnh được chọn còn RS = 0 thì thanh ghi dữ liệu được chọn.

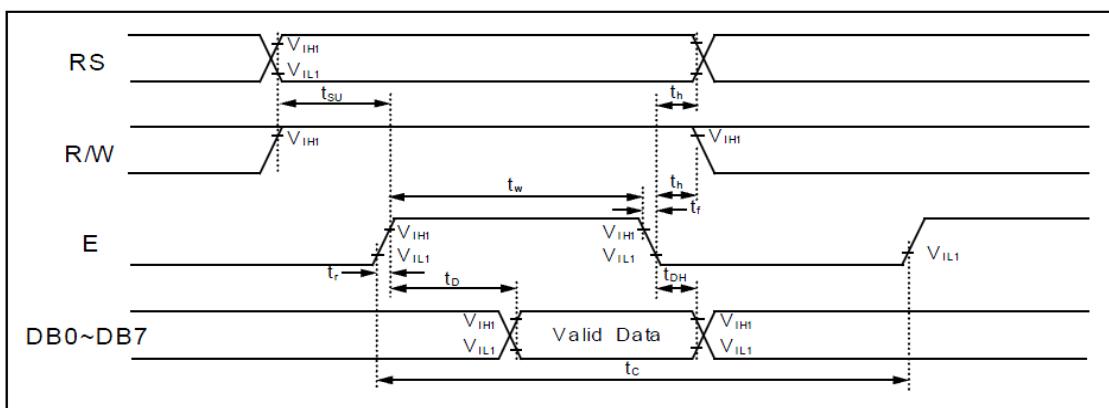
- LCD\_R/W: Tín hiệu quy định chiều trao đổi thông tin trên kênh dữ liệu DB[7:0], nếu R/W = 1 thì thiết bị điều khiển đọc thông tin từ LCD, nếu R/W = 0

thì thiết bị điều khiển ghi thông tin lên LCD. Thông thường thông tin được ghi lên LCD là chính nên R/W = 0.

- LCD\_E (Enable): Chân cho phép E dùng để chốt dữ liệu trên kênh dữ liệu. Để chốt dữ liệu xung này phải giữ tích cực trong khoảng thời gian tối thiểu  $T_W \geq 450\text{ns}$ . Hình vẽ dưới đây thể hiện giản đồ sóng cho quá trình đọc và ghi dữ liệu trên LCD, với các tham số thời gian khác xem thêm trong tài liệu tham khảo về LCD1602A.



Hình 4-67. Chu trình ghi dữ liệu lên LCD1602A



Hình 4-68. Chu trình đọc dữ liệu lên LCD1602A

- Chân DB0 - DB7: Các chân dữ liệu của LCD
  - Nếu R/W = 1, RS = 0 khi D7 = 1 nghĩa là LCD đang bận thực thi các tác vụ bên trong và ở thời điểm đó LCD không nhận thêm bất cứ dữ liệu nào, thông thường ta dùng bit D7 trong trường hợp này để kiểm tra trạng thái LCD mỗi khi muốn gửi tiếp dữ liệu vào LCD.

### 4.3.2. Các lệnh cơ bản của LCD1602A

Bảng sau liệt kê các lệnh cơ bản của LCD1602A:

Bảng 4-8

**Bảng các lệnh cơ bản của LCD1602**

Lệnh	<b>LCD_RS</b>	<b>LCD_R/W</b>	<b>DB[7]</b>	<b>DB[6]</b>	<b>DB[5]</b>	<b>DB[4]</b>	<b>DB[3]</b>	<b>DB[2]</b>	<b>DB[1]</b>	<b>DB[0]</b>	Thời gian thực hiện
Clear Display	0	0	0	0	0	0	0	0	0	0	82us-1.64ms
Return Cursor Home	0	0	0	0	0	0	0	0	-	-	40us-1.6ms
Entry Mode Set	0	0	0	0	0	0	0	0	I/D	S	40us
Display On/Off	0	0	0	0	0	0	1	D	C	B	40us
Cursor and Display Shift	0	0	0	0	0	1	S/C	R/L	-	-	40us
Function Set	0	0	0	0	1	0	1	0	-	-	40us
Set CG RAM Address	0	0	0	1	A5	A4	A3	A2	A1	A0	40us
Set DD RAM Address	0	0	1	A6	A5	A4	A3	A2	A1	A0	40us
Read Busy Flag and Address	0	1	BF	A6	A5	A4	A3	A2	A1	A0	1us
Write to CG RAM or DDRAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	40us
Read to CG RAM or DDRAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	40us

**Clear Display:** Xóa hết màn hình bằng cách ghi các ký tự trống 0x20 lên DDRAM và trả con trỏ DDRAM về vị trí 0.

**Return Cursor Home:** Đưa con trỏ về vị trí ban đầu nhưng không xóa dữ liệu trong DDRAM.

**Entry mode set:** Đặt chế độ cho con trỏ tăng hay giảm bằng bít I/D, I/D = 1 con trỏ tăng lên 1 mỗi khi có một ký tự được ghi vào RAM, I/D = 0 con trỏ giảm 1. Chế độ chuẩn ta đặt I/D = 1. Toàn màn hình sẽ dịch khi đây nếu S = 1, giữ nguyên nếu S = 0.

**Display on/off:** Bật hay tắt màn hình, con trỏ và trạng thái nhấp nháy của con trỏ tương ứng bằng các bit D, C, B.

**Cursor and Display Shift:** Di chuyển con trỏ và dịch toàn bộ màn hình mà không thay đổi nội dung trong DD RAM.

**Function Set:** Thiết lập chế độ làm việc/8bit hay 4 bit, 1 hay hai dòng hiển thị, chọn bộ ký tự.

**Set CG RAM Adress:** đặt địa chỉ truy cập tới bộ nhớ tạo ký tự Character Generation RAM (trong trường hợp muốn tạo và sử dụng các ký tự theo mong muốn)

**Set DD RAM Adress:** đặt địa chỉ cho bộ nhớ dữ liệu, nơi lưu trữ các ký tự để hiển thị lên màn hình.

**Read Busy Flag and Address:** Kiểm tra trạng thái bận của LCD, trạng thái bận được trả về bít DB[7], các bit còn lại DB[6:0] là giá trị địa chỉ DD RAM hiện hành.

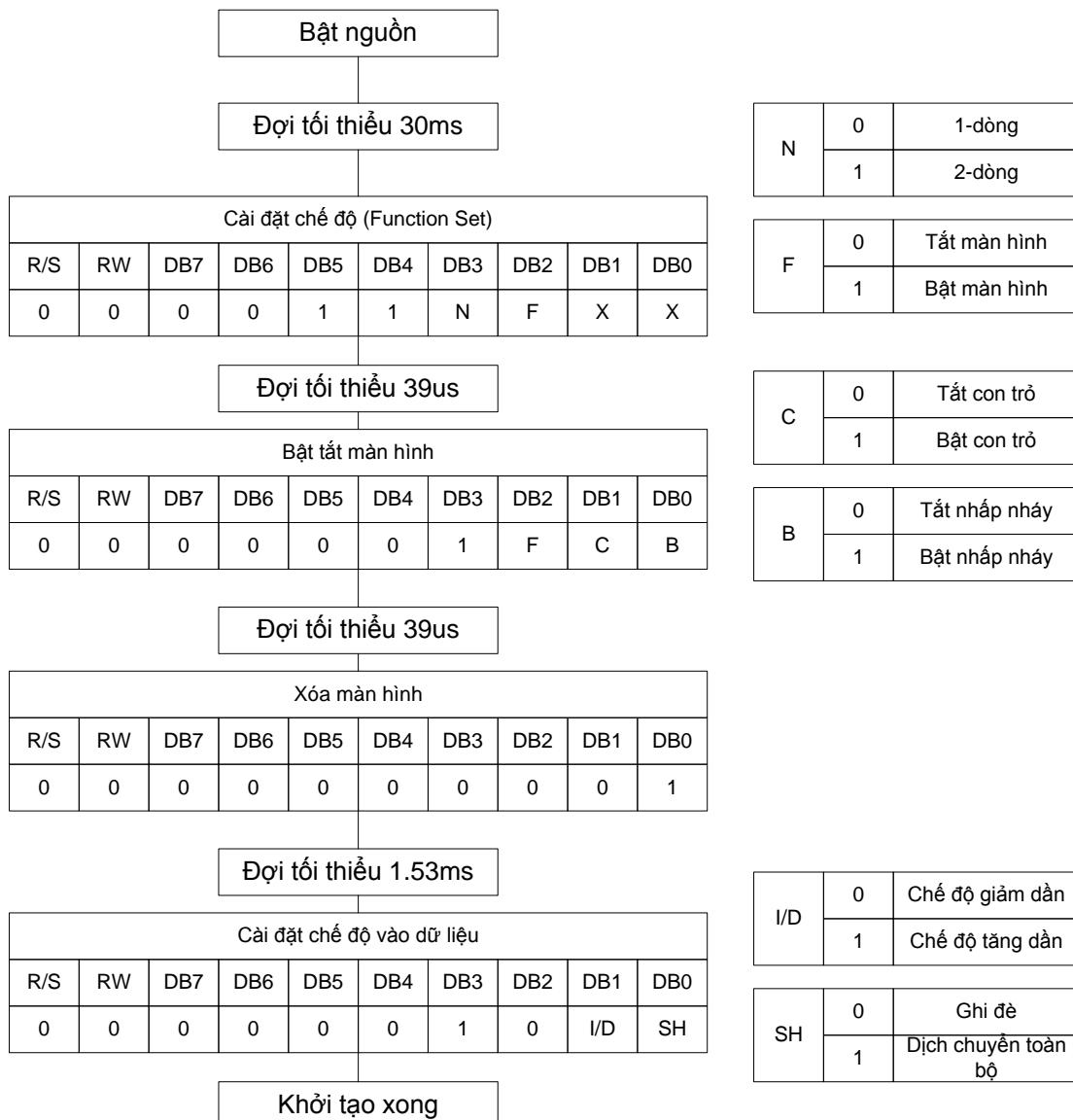
**Write Data to CGRAM or DDRAM:** Ghi dữ liệu vào các RAM tương ứng trong LCD. Lệnh này sử dụng để in các ký tự ra màn hình, khi đó con trỏ địa chỉ RAM tương ứng tự động dịch chuyển lên 1 hoặc xuống 1 tùy thiết lập bởi Entry mode set.

**Read Data from CGRAM or DDRAM:** Đọc vào các RAM tương ứng trong LCD. Con trỏ địa chỉ RAM tương ứng cũng tự động tăng hay giảm 1 đơn vị tùy thuộc thiết lập bởi Entry mode set.

#### 4.3.3. Quá trình khởi tạo LCD cho chế độ làm việc 8 bit.

Việc khởi tạo cho LCD1602A phải tuân thủ chính xác các yêu cầu về thời gian nghỉ tối thiểu giữa các lệnh khởi tạo, sơ đồ dưới đây thể hiện quy trình khởi tạo cho LCD1602A với chế độ làm việc 8-bit. Quá trình này gồm 5 bước, khi mới cấp nguồn cần phải đợi tối thiểu 30ms cho điện áp nguồn ổn định ở mức cần thiết. Sau đó LCD phải được nạp tương ứng 4 lệnh khởi tạo với giá trị ở sơ đồ dưới đây và tuân thủ đúng khoảng thời gian nghỉ giữa các lệnh. Sau khi trải qua

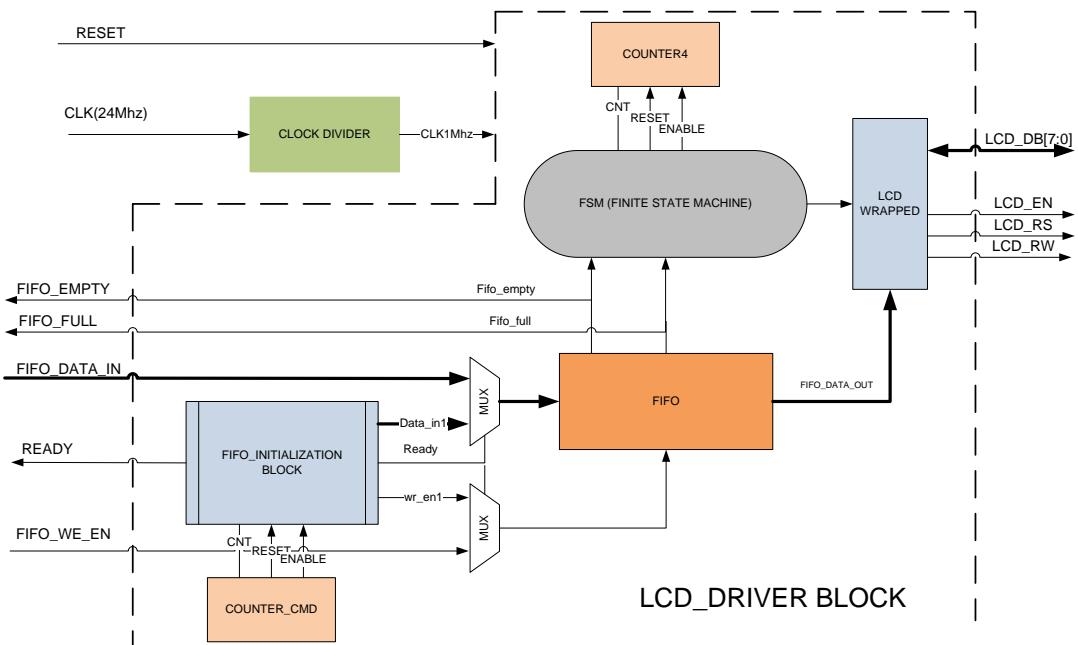
chính xác 6 bước này LCD trở về trạng thái chờ dữ liệu/lệnh mới, mỗi lệnh kế tiếp thực hiện trong với thời gian tối thiểu là 40us.



Hình 4-69. *Khởi tạo LCD1602A cho chế độ 8-bit*

#### 4.3.4. Sơ đồ thiết kế khói điều khiển LCD1602A bằng FPGA

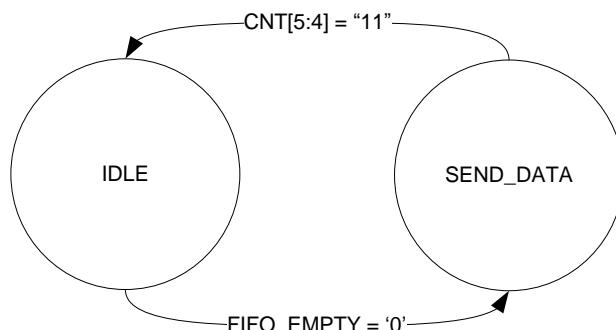
Sơ đồ khói điều khiển LCD được thể hiện ở hình sau:



Hình 4-70. Sơ đồ khối LCD DRIVER trên FPGA

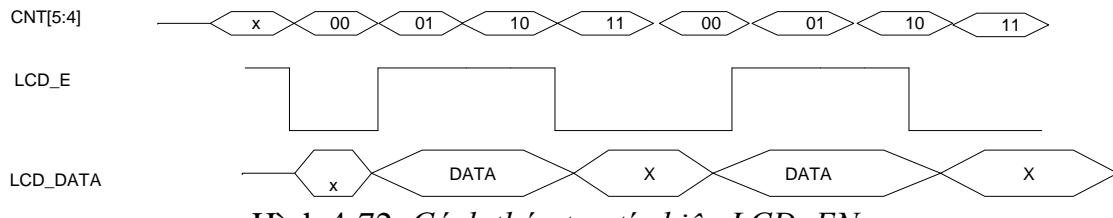
Với mục đích có thể dễ dàng ghép nối với các khối thiết kế khác như một khối kết xuất hiển thị thông tin, khối LCD được thiết kế trong một tệp VHDL duy nhất bao gồm ba khối chức năng chính. Tín hiệu đồng bộ cơ sở là tín hiệu xung nhịp 1Mhz, có thể dùng bộ đếm để chia tần mà không cần DCM trong trường hợp này.

**Khối máy trạng thái FSM:** Khối FSM có nhiệm vụ đọc các lệnh được đệm trong FIFO và tạo các tín hiệu điều khiển cần thiết để gửi tới LCD. Khi ở trạng thái IDLE FSM sẽ luôn luôn kiểm tra trạng thái của khối FIFO, nếu phát hiện trong FIFO có dữ liệu ( $\text{FIFO\_EMPTY} \neq 0$ ) FSM chuyển sang trạng thái gửi dữ liệu SEND\_DATA. Khi ở trạng thái này FSM khởi tạo bộ đếm 64 và đọc dữ liệu từ FIFO để gửi tới LCD.



Hình 4-71. Máy trạng thái của khối điều khiển LCD

Khi thực hiện gửi lệnh tới LCD, nhiệm chính là tạo ra tín hiệu điều khiển LCD\_EN với yêu cầu về mặt thời gian như ở dưới đây:



Hình 4-72. Cách thực tạo tín hiệu LCD\_EN

Tín hiệu LCD\_EN phải tích cực khi dữ liệu trên LCD\_DATA ổn định ít nhất 450ns và thời gian thực hiện lệnh không ít hơn 40us với khoảng nghỉ giữa các lệnh cỡ 10us nên trong thiết kế thực ta để khoảng thời tích cực gian này là 32us, và khoảng nghỉ là 16us, nghĩa là 1 lệnh thực thi trong  $16+32+16 = 64\text{us}$ . Để làm được như thế ta dùng một bộ đếm CNT 6 bit và gán:

$$\text{LCD\_EN} = \text{CNT}[4] \text{ xor } \text{CNT}[5];$$

Khi gán như vậy LCD\_EN sẽ tích cực (bằng 1) khi CNT[5:4] nhận các giá trị giữa 01, 10 và không tích cực ở các giá trị đầu và cuối 00, 11. Khi LCD\_EN tích cực khối FSM đồng thời đọc dữ liệu từ FIFO và gửi tới LCD.

**Khối đệm dữ liệu FIFO:** hoạt động ở tần số 1Mhz có nhiệm vụ lưu các giá trị đếm trước khi thực sự gửi tới LCD bằng khối FSM. FIFO có kích thước 16 hàng và mỗi hàng 10 bit tương ứng các giá trị [LCD\_RS, LCD\_RW, LCD\_DB[7:0]], Tín hiệu điều khiển LCD\_EN được tạo bởi khối FSM. Số lượng hàng của FIFO có thể thay đổi tùy theo đặc điểm ứng dụng. Khi muốn kết xuất kết quả ra LCD, các khối bên ngoài chỉ cần thực hiện ghi mã lệnh tương ứng vào FIFO, tất cả các công việc còn lại do FSM đảm nhận. Thiết kế như vậy làm đơn giản hóa việc truy xuất LCD vì sẽ không cần phải quan tâm đến các tham số thời gian.

(\*) Người thiết kế có thể lựa chọn sử dụng FIFO có sẵn như một IP Core có trong ISE hỗ trợ bởi Xilinx hoặc tự thiết kế FIFO bằng VHDL từ đầu giới thiệu trọng chương trước. Cách tạo và sử dụng IP Core xem thêm trong phần phụ lục thực hành thiết kế trên FPGA.

**Khối khởi tạo LCD:** khối khởi tạo LCD được thực hiện tự động ngay sau khi hệ thống bắt đầu làm việc (sau RESET), nó bao gồm các bước được nêu ở 4.3.3. Cách thực hiện thực hóa quy trình khởi tạo là sử dụng một bộ đếm 18 bit và tương ứng với các giá trị của bộ đếm để gửi các lệnh cần thiết vào FIFO. Sở dĩ kích thước bộ đếm lớn là do tại bước đầu tiên của quá trình khởi tạo LCD cần khoảng 30ms để ổn định điện áp đầu vào,  $30\text{ms} = 30000\text{us} \sim 2^{15}$ . Khi quy trình

khởi tạo này được kích hoạt thì FIFO chỉ được phép ghi dữ liệu bởi khối khởi tạo LCD, tín hiệu READY khi đó bằng 0 báo cho các khối bên ngoài không được phép truy cập vào FIFO. Khi quá trình khởi tạo hoàn tất READY = 1 và FIFO trở về trạng thái chờ dữ liệu từ bên ngoài. Trên sơ đồ READY điều khiển hai khối MUX để lựa chọn các tín hiệu FIFO\_DATA\_IN và FIFO\_WR\_EN từ bên trong hoặc bên ngoài.

Mã nguồn khôi lcd\_driver được liệt kê dưới đây:

```
-----
-- lcd_driver.vhd
-- Company: BMKTVXL
-- Engineer: Trinh Quang Kien
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library UNISIM;
use UNISIM.VComponents.all;
Library XilinxCoreLib;
-----
entity lcd_driver is
port (
    clk1Mhz : in std_logic;
    reset   : in std_logic;
    lcd_rs  : out std_logic;
    lcd_rw  : out std_logic;
    lcd_e   : out std_logic;
    data_in : in std_logic_vector (9 downto 0);
    wr_en   : in std_logic; -- Write Enable FIFO
    full    : out std_logic; -- FIFO full
    ready   : out std_logic; -- LCD ready
    lcd_data: out std_logic_vector (7 downto 0)
);
end lcd_driver;
-----
architecture Behavioral of lcd_driver is
signal cnt           : std_logic_vector(5 downto 0);
signal cnt_reset     : std_logic;
signal cnt_enable    : std_logic;
signal lcd_state     : std_logic;
signal cnt_cmd_enable: std_logic;
signal rd_en         : std_logic; -- Read enable
signal data_out      : std_logic_vector (9 downto 0);
signal lcd_code      : std_logic_vector (9 downto 0);
```

```

signal empty           : std_logic; -- FIFO empty

constant LCD_IDLE      : std_logic := '0';
constant lcd_SEND_DATA : std_logic := '1';

signal data_in1, data_in2 : std_logic_vector(9 downto 0);
signal wr_en1, wr_en2   : std_logic;
signal cnt_cmd          : std_logic_vector (17 downto
0) := "000000000000000000000000";

-----
component fifo_16x10
    port (
        clk: IN std_logic;
        rst: IN std_logic;
        din: IN std_logic_VECTOR(9 downto 0);
        wr_en: IN std_logic;
        rd_en: IN std_logic;
        dout: OUT std_logic_VECTOR(9 downto 0);
        full: OUT std_logic;
        empty: OUT std_logic);
end component;
-----

begin
    ff : fifo_16x10
    port map (
        clk => clk1Mhz,
        rst => reset,
        din => data_in2,
        wr_en => wr_en2,
        rd_en => rd_en,
        dout => data_out,
        full => full,
        empty => empty);
-- INST_TAG_END -- End INSTANTIATION Template -----
    counter:
    process (clk1Mhz, reset)
    begin
        if cnt_reset = '1' then
            cnt <= (others => '0');
        elsif clk1Mhz = '1' and clk1Mhz'event then
            if cnt_enable = '1' then
                cnt <= cnt +1;
            end if;
        end if ;
    end process counter;

```

```

process (clk1Mhz, reset, lcd_state, cnt)
begin
    if reset = '1' then
        lcd_state <= lcd_idle;
        cnt_enable <= '0';
    elsif clk1Mhz = '1' and clk1Mhz'event then
        case lcd_state is
            when lcd_idle =>
                cnt_reset <= '0';
                lcd_code <= "01" & x"00";
                if empty = '0' then -- existing data in fifo
                    cnt_reset <= '1';
                    cnt_enable <= '1';
                    lcd_state <= lcd_SEND_DATA;
                    lcd_e <= '0';
                    rd_en <= '1';
                end if;
            when lcd_SEND_DATA =>
                cnt_reset <= '0';
                lcd_code <= data_out;
                lcd_e <= cnt(5) xor cnt(4) ;
                rd_en <= '0';
                if cnt(5 downto 4) = "11" then
                    cnt_reset <= '1';
                    cnt_enable <= '0';
                    lcd_e <= '0';
                    lcd_state <= lcd_idle;
                end if;
            when others =>
                lcd_state <= lcd_idle;
        end case;
    end if;
end process;
lcd_data <= lcd_code(7 downto 0);
lcd_rs <= lcd_code(9);
lcd_rw <= lcd_code(8);
counter_cmd:
process (clk1Mhz, reset, cnt_cmd_enable)
begin
    if reset = '1' then
        cnt_cmd <= (others => '0');
    elsif clk1Mhz = '1' and clk1Mhz'event then
        if cnt_cmd_enable = '1' then
            cnt_cmd <= cnt_cmd +1;
        end if;
    end if ;
end if ;

```

```

end process counter_cmd;

process (cnt_cmd)
begin
case cnt_cmd(16 downto 0) is
when "1000000000000001" => data_in1 <= "00"& x"38";
                                wr_en1    <= '1';
-- 8bit 2line mode/display on
when "1000000000000010" => data_in1 <= "00"& x"38";
                                wr_en1    <= '1';
-- 8bit 2line mode/display on
when "1000000000000011" => data_in1 <= "00"& x"38";
                                wr_en1    <= '1';
-- 8bit 2line mode/display on/off
-- delay minimum 39us
when "1000000001000000" => data_in1 <= "00"& x"0c";
                                wr_en1    <= '1';
-- display on
-- delay minimum 39us
when "1000000010000000" => data_in1 <= "00"& x"01";
                                wr_en1    <= '1';
-- clear display
-- delay minimum 1530us
when "1000010000000000" => data_in1 <= "00"& x"06";
                                wr_en1    <= '1';
-- entry mode set incrementer/shiff off
-- delay 1000us --
when "1000100000000001" => data_in1 <= "10"& x"31";
                                wr_en1    <= '1';
-- write character "1" for testing only;
when "10001000000011100" => data_in1 <= "00"& x"00";
                                wr_en1    <= '1';
-- do nothing finished initialization process
when others      => data_in1 <= "00"& x"00";
                                wr_en1    <= '0';
end case;

if cnt_cmd(17) = '1' then
    cnt_cmd_enable <= '0';
else
    cnt_cmd_enable <= '1';
end if;
end process;

-- finished initilization process, set ready for new data
ready <= not cnt_cmd_enable;

```

```

--when ready, the fifo receives data from higher module
--when not ready, the fifo receives data from this module
for initialization process
    process (data_in,data_in1,wr_en,wr_en2,cnt_cmd_enable)
    begin
        if cnt_cmd_enable = '0' then
            data_in2 <= data_in;
            wr_en2    <= wr_en;
        else
            data_in2 <= data_in1;
            wr_en2    <= wr_en1;
        end if;
    end process;

end Behavioral;

```

Kết quả tổng hợp thiết kế trên FPGA Spartan 3E 3s500-pq208 với tần số xung nhịp cơ sở 24Mhz thu được như sau:

```

Selected Device          : 3s500epq208-5
Number of Slices         : 47 out of 4656 1%
Number of Slice Flip Flops : 48 out of 9312 0%
Number of 4 input LUTs    : 90 out of 9312 0%
Number of Ios             : 14
Number of bonded IOBs     : 14 out of 158 8%
Number of GCLKs           : 2 out of 24 8%
-----+-----+-----+
Clock Signal      | Clock buffer(FF name) | Load |
-----+-----+-----+
cd/clk161 | BUFG | 39 |
clk | BUFGP | 9 |
-----+-----+-----+
Timing Summary:
-----
Speed Grade: -5
Minimum period: 4.188ns (Maximum Frequency: 238.780MHz)
Minimum input arrival time before clock: 3.569ns
Maximum output required time after clock: 7.739ns
Maximum combinational path delay: 3.476ns
-----
```

Kết quả cho thấy khối LCD\_Driver chiếm khá ít tài nguyên với chỉ 90 LUT, hoàn toàn phù hợp cho việc gắn vào các thiết kế lớn hơn với vai trò khối kết xuất hiện thị thông tin. Về tốc độ thì mạch điều khiển theo kết quả có thể chạy được ở tốc độ 200Mhz trong khi yêu cầu thực tế là 1Mhz.

Kết quả thực tế về thời gian sau kết nối và phân bố như sau:

Data Sheet report:

All values displayed in nanoseconds (ns)

Clock to Setup on destination clock clk

clk		3.744			

Kết quả trên mạch FPGA thể hiện ở hình sau:

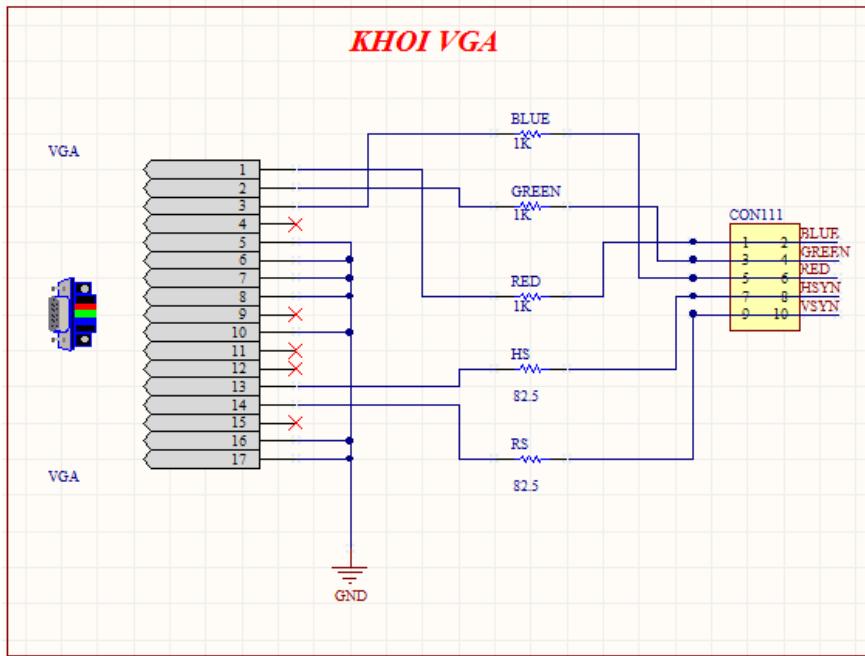


Hình 4-73. Kết quả trên mạch FPGA của khối điều khiển LCD

## 4.5. Thiết kế điều khiển VGA trên FPGA

### 4.5.1 Yêu cầu giao tiếp VGA đơn giản

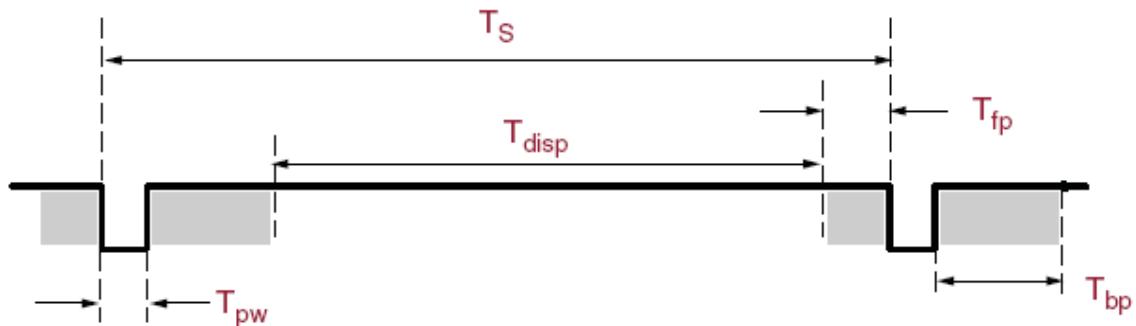
Giao tiếp VGA trong chế độ đơn giản nhất gồm 5 tín hiệu điều khiển, các tín hiệu này được nối với châm cắm 15-PIN theo sơ đồ sau:



Hình 4-74. Mạch giao tiếp VGA đơn giản

Tín hiệu RED, GREEN, BLUE tương ứng để thể hiện màu sắc, với 3 bit tín hiệu này thì tối đa có 8 màu hiển thị. Để có nhiều màu hiển thị hơn sử dụng một hệ thống các điện trở mắc song song tương tự như hệ thống DAC, tức là điện trở sau có giá trị lớn gấp hai lần điện trở trước đó. Với cách mắc như thế với một tổ hợp n đầu vào sẽ sinh ra  $2^n$  mức điện tương ứng hay tương ứng có  $2^n$  màu sắc khác nhau ở đầu ra.

Tín hiệu VS (*Vertical Synchronous*) và HS (*Horizontal Synchronous*) là các tín hiệu quét màn hình theo phương đứng và phương ngang tương ứng, dạng tín hiệu và yêu cầu về mặt thời gian của các tín hiệu này như sau.



Hình 4.75. Giản đồ sóng tín hiệu quét ngang và đọc cho màn hình VGA

Màn hình sẽ làm việc theo cơ chế quét theo từng hàng từ trên xuống dưới, chu kỳ làm tươi màn hình  $T_{refresh}$  được tính bằng tổng thời gian để quét hết toàn

bộ một lượt màn hình, tần số này tùy thuộc vào chế độ phân giải màn hình sẽ hiển thị và phụ thuộc tần số quét hỗ trợ được bởi màn hình. Các màn hình hiện đại khác nhau và thường có giá trị tần số quét khoảng từ 50Hz - 100Hz. Thông tin chi tiết yêu cầu về mặt thời gian quét của màn hình có thể tham khảo từ nguồn.

[http://www.epanorama.net/documents/pc/vga\\_timing.html](http://www.epanorama.net/documents/pc/vga_timing.html)

Chú ý phân cực của xung quét có thể dương hoặc âm, tham số xung quét trên hình vẽ theo phân cực dương, xung đồng bộ mức thấp, các xung còn lại có mức cao. Ví dụ cho chế độ 800x600 – 60Hz thì các tham số cụ thể được tính như sau:

Tham số với xung quét ngang, xung nhịp quét điểm ảnh  $F_0 = 40\text{Mhz}$ ,  $T_0 = 1/40\text{Mhz} = 25\text{ns}$ .

Bảng 4-9

#### Tham số quét ngang VGA

Ký hiệu	Tên gọi	Giá trị ( $xT_0$ )	Số lượng đếm	Tổng thời gian (us)
$T_{pw}$	Thời gian đồng bộ	$128 \times T_0$	128	3,20
$T_{display}$	Thời gian hiển thị	$800 \times T_0$	800	20,00
$T_{hs}$	Tổng thời gian quét	$1056 \times T_0$	1056	26,40
$T_{bp}$	Thời gian vòm sau	$88 \times T_0$	88	2,20
$T_{fp}$	Thời gian vòm trước	$40 \times T_0$	40	1,00

Tổng thời gian để quét hết 1 hàng là  $T_{hs} = 1056 \times T_0 = 26400 \text{ ns}$ , từ đó tính được tần số quét hàng là  $F_h = 1/26400 \text{ ns} = 37,87\text{Khz}$

Tham số với xung quét dọc:

Bảng 4-10

#### Tham số quét dọc VGA

Ký hiệu	Tên gọi	Giá trị ( $xT_{hs}$ )	Số lượng đếm	Tổng thời gian (us)
$T_{pw}$	Thời gian đồng bộ	$4 \times T_{hs}$	4	105,60
$T_{display}$	Thời gian hiển thị	$600 \times T_{hs}$	600	15840,00
$T_{vs}$	Tổng thời gian quét	$628 \times T_{hs}$	628	1657,92
$T_{bp}$	Thời gian vòm sau	$23 \times T_{hs}$	23	607,20
$T_{fp}$	Thời gian vòm trước	$1 \times T_{hs}$	1	26,40

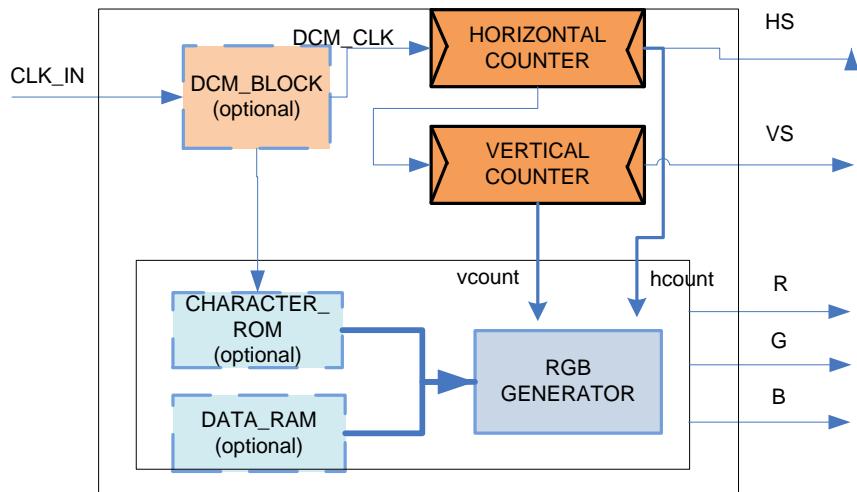
Từ đó tính được chu kỳ làm tươi và tần số làm tươi màn hình

$$T_{\text{refresh}} = T_{\text{vs}} = 628 \times T_{\text{hs}} = 628 \times 26400 = 16579200 \text{ ns} = 16,5792 \text{ us}$$

$$T_{\text{refresh}} = 1/T_{\text{refresh}} = 1/16,5792 \text{ us} = 60 \text{ Hz.}$$

#### 4.5.2. 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 khái thiết kế như sau:



Hình 4-76 Sơ đồ khái thiết kế VGA

#### 4.5.3. Khối DCM

Với tần số quét 60Hz như ví dụ trên thì xung nhịp đầu vào là 40Mhz, nếu dao động thạch anh trên mạch FPGA có tần số đúng bằng tần số này thì không cần thiết phải có khối DCM, trong các trường hợp còn lại thì buộc phải có khối DCM.

DCM là khái có sẵn trong FPGA có khả năng điều chỉnh về pha tần số và dạng của xung nhịp đồng bộ. Trong ví dụ này ta dùng DCM để điều chỉnh tần số từ 25Mhz lên tần số 40Mhz bằng cách đặt các tham số nhân và chia tần lìa lượt là 8 và 5 vì:

$$40 \text{ Mhz} = 25 \text{ Mhz} * 8 / 5$$

Dạng mô tả chuẩn của DCM có thể tìm trong menu *Edit/Language Templates*, mô tả dưới đây được chỉnh sửa cho DCM của FPGA SPARTAN-3E (dcm.vhd)

---

-- Engineer: Trinh Quang Kiên

```

-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library UNISIM;
use UNISIM.VComponents.all;
-----
entity dcm_block is
    Port (
        CLK_IN      : in  STD_LOGIC;
        DCM_CLK     : out STD_LOGIC
    );
end dcm_block;
-----
architecture Behavioral of dcm_block is
Begin
    DCM_SP_inst : DCM_SP generic map (
        CLKDV_DIVIDE => 2.0,
        CLKFX_DIVIDE => 5,
        CLKFX_MULTIPLY => 8,
        CLKIN_DIVIDE_BY_2 => FALSE,
        CLKIN_PERIOD => 40.0,
        CLKOUT_PHASE_SHIFT => "NONE",
        CLK_FEEDBACK => "1X",
        DESKEW_ADJUST => "SYSTEM_SYNCHRONOUS",
        DLL_FREQUENCY_MODE => "LOW",
        DUTY_CYCLE_CORRECTION => TRUE,
        PHASE_SHIFT => 0,
        STARTUP_WAIT => FALSE)
    port map (
        CLKFX => DCM_CLK, -- DCM CLK synthesis out (M/D)
        CLKFB => CLK_IN,   -- DCM clock feedback
        CLKIN => CLK_IN    -- Clock input);
end Behavioral;
-----
```

#### 4.5.4. Khối tạo xung quét ngang và dọc

Đây là khối hạt nhân của điều khiển VGA, nhiệm vụ của khối này là tạo các xung HS và VS bằng hai bộ đếm được ghép nối tiếp, bộ đếm cơ sở là bộ đếm cho xung quét ngang (HORIZONTAL COUNTER) với xung đầu vào đếm là xung nhịp DCM\_CLK = 40 Mhz lấy từ DCM. Bộ đếm thứ hai là cho xung quét ngang hay còn gọi là bộ đếm hàng (VERTICAL COUNTER) được tăng lên 1 sau

khi mỗi hàng được đếm xong. Tham số cho các bộ đếm được đặt trong một gói mô tả có tên vga\_pkg.vhd với nội dung như sau:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
package vga_pkg is
    -- horizontal timing (in pixels count )
    constant H_DISPLAY      : natural := 800;
    constant H_BACKPORCH    : natural := 88;
    constant H_SYNCTIME     : natural := 128;
    constant H_FRONTPORCH   : natural := 40;
    constant H_PERIOD        : natural := 1056;
    constant H_ONDISPLAY    : natural := H_SYNCTIME +
        H_BACKPORCH;
    constant H_OFFDISPLAY   : natural := H_ONDISPLAY + H_DISPLAY;
    constant H_COUNT_W      : natural := 11;
    -- vertical timing (in lines count)
    constant V_DISPLAY       : natural := 600;
    constant V_BACKPORCH    : natural := 23;
    constant V_SYNCTIME     : natural := 4;
    constant V_FRONTPORCH   : natural := 1;
    constant V_PERIOD        : natural := 628;
    constant V_ONDISPLAY    : natural := V_SYNCTIME +
        V_BACKPORCH;
    constant V_OFFDISPLAY   : natural := V_ONDISPLAY +
        V_DISPLAY;
    constant V_COUNT_W      : natural := 10;

end vga_pkg;
package body vga_pkg is
end vga_pkg;
```

Các tham số này phải khớp với các yêu cầu về mặt thời gian của các tín hiệu HS và VS ở bảng trên. Mô tả của khối tạo xung quét như sau (vga\_800x600x60Hz.vhd)

```
-- VGA controller for 800x600x60Hz
-- the dcm_clk must around 40Mhz (generate by DCM if
require)
-- All timing information is get from
http://www.epanorama.net/documents/pc/vga\_timing.html
-- Based on reference VGA project from Digilent
-- Recreated by TQ KIEN
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```

library work;
use work.vga_pkg.all;
library UNISIM;
use UNISIM.VComponents.all;
-----
entity vga_800_600_60 is
port(
    rst      : in std_logic;
    dcm_clk : in std_logic;
    HS       : out std_logic;
    VS       : out std_logic;
    hcount   : out std_logic_vector(H_COUNT_W-1 downto 0);
    vcount   : out std_logic_vector(V_COUNT_W-1 downto 0);
    video_ena : out std_logic
);
end vga_800_600_60;
-----
architecture Behavioral of vga_800_600_60 is
-- horizontal and vertical counters
signal hcmt : std_logic_vector(H_COUNT_W-1 downto 0) := 
(others => '0');
signal vcmt : std_logic_vector(V_COUNT_W-1 downto 0) := 
(others => '0');
signal SIG_POL : std_logic := '0';
-----
begin
    hcmt <= hcmt;
    vcmt <= vcmt;
-- increment horizontal counter at dcm_clk rate to H_PERIOD
    SIG_POL <= '0';
    h_counter: process(dcm_clk)
    begin
        if(rising_edge(dcm_clk)) then
            if(rst = '1') then
                hcmt <= (others => '0');
            elsif(hcmt = H_PERIOD) then
                hcmt <= (others => '0');
            else
                hcmt <= hcmt + 1;
            end if;
        end if;
    end process h_counter;
-- Horizontal timing detail
-----
-- _____ | _____ VIDEO _____ | _____ | VIDEO(next line)

```

```

--      | -C- | -----D----- | -E- |
--      |_ | _____ |_|
--      | B |
--      | -----A----- |
--A (1056)      - Scanline time
--B (128)       - Sync pulse lenght
--C (88)        - Back porch
--D (800)       - Active video time
--E (40)        - Front porch
-- SIG_POL is polarity of SYN signals, the Horizontal SYN is
active by SIG_POL during sync time (B)
-----
hs_generate: process(dcm_clk)
begin
    if(rising_edge(dcm_clk)) then
        if(hcnt >= H_SYNCTIME) then
            HS <= SIG_POL;
        else
            HS <= not SIG_POL;
        end if;
    end if;
end process hs_generate;

-- verital timming detail
-----
--      |-----VIDEO-----|-----VIDEO(next line)
--      | -C- |-----D----- | -E- |
--      |_ | _____ |_|
--      | B |
--      | -----A----- |
--A (628)      - Scanline time
--B (4)        - Sync pulse lenght
--C (23)       - Back porch
--D (600)      - Active video time
--E (1)        - Front porch
-- SIG_POL is polarity of SYN signals, the Vertiacal SYN is
active by SIG_POL during sync time (B)
-----
v_counter: process(dcm_clk)
begin
    if(rising_edge(dcm_clk)) then
        if(rst = '1') then
            vcnt <= (others => '0');
        elsif(hcnt = H_PERIOD) then

```

```

        if(vcnt = V_PERIOD) then
            vcnt <= (others => '0');
        else
            vcnt <= vcnt + 1;
        end if;
    end if;
end process v_counter;

vs_generate: process(dcm_clk)
begin
    if(rising_edge(dcm_clk)) then
        if (vcnt >= V_SYNCTIME) then
            VS <= SIG_POL;
        else
            VS <= not SIG_POL;
        end if;
    end if;
end process vs_generate;

-- enable video output when pixel is in visible area
video_ena <= '1' when
    (hcnt > H_ONDISPLAY and
     hcnt < H_OFFSETDISPLAY and
     vcnt > V_ONDISPLAY and
     vcnt < V_OFFSETDISPLAY) else '0';

end Behavioral;
-----
```

#### 4.5.5. Khối tạo điểm ảnh (RGB\_Generator)

Khối tạo điểm ảnh có đầu vào là các giá trị tọa độ hcount và vcount của điểm ảnh và đầu ra là màu sắc tương ứng của điểm ảnh đó, khối này có thể chứa các khối CHARACTER\_ROM lưu trữ dạng của font chữ trong chế độ TEXT hoặc lưu trữ dữ liệu (hình ảnh) trong khối RAM. Trong ví dụ này ta sẽ tạm thời bỏ qua các khối trên và chỉ tạo một khối đơn giản tạo sự thay đổi màu sắc theo một số bit của giá trị tọa độ nhằm mục đích quan sát và kiểm tra, với từng ứng dụng cụ thể người sử dụng sẽ phải thiết kế lại các khôi ROM, RAM cho phù hợp với yêu cầu.

```

-----  

-- Company: BM KTVXL  

-- Engineer: Trinh Quang Kien  

-----  

library IEEE;
```

```

use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library work;
use work.vga_pkg.all;
-----
entity RGB_gen is
port (
    dcm_clk : in std_logic;
    video_ena: in std_logic;
    hcount   : in std_logic_vector(H_COUNT_W-1 downto 0);
    vcount   : in std_logic_vector(V_COUNT_W-1 downto 0);
    RED      : out std_logic;
    BLUE     : out std_logic;
    GREEN    : out std_logic);
end RGB_gen;
-----
architecture Behavioral of RGB_gen is
begin
    process (dcm_clk)
    begin
        if rising_edge(DCM_CLK) then
            if video_ena = '1' then
                RED    <= vcount(3);
                BLUE   <= vcount(5);
                GREEN  <= hcount(6);
            end if;
        end if;
    end process;
end Behavioral;
-----
```

#### 4.5.6. Khối tổng quát

Khối tổng quát có tên VGACOMP chứa mô tả thiết kế được dùng để nạp vào FPGA ghép bởi các khối trên. Nội dung của khối này như sau

```

-- vgacomp.vhd
-- Trinh Quang Kien - BMKTVXL
-----
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
library UNISIM;
use UNISIM.Vcomponents.ALL;
library work;
use work.vga_pkg.all;
```

```

-----
entity VgaComp is
  port (
    CLK_25MHz : in  std_logic;
    RST        : in  std_logic;
    RED        : out std_logic;
    BLUE       : out std_logic;
    GREEN      : out std_logic;
    HS         : out std_logic;
    VS         : out std_logic);
end VgaComp;

architecture Structural of VgaComp is
signal nRST      : std_logic;
signal video_ena : std_logic;
signal dcm_clk   : std_logic;
signal CLK_IN    : std_logic;
signal hcount    : std_logic_vector(H_COUNT_W-1 downto 0);
signal vcount    : std_logic_vector(V_COUNT_W-1 downto 0);
-----
component dcm_block is
  Port (
    CLK_IN  : in STD_LOGIC;
    DCM_CLK : out STD_LOGIC);
end component;
-----
component vga_800_600_60 is
  port(
    rst      : in std_logic;
    dcm_clk : in std_logic;
    HS      : out std_logic;
    VS      : out std_logic;
    hcount  : out std_logic_vector(H_COUNT_W-1 downto 0);
    vcount  : out std_logic_vector(V_COUNT_W-1 downto 0);
    video_ena : out std_logic);
end component;
-----
component RGB_gen is
  port(
    dcm_clk  : in std_logic;
    video_ena : in std_logic;
    hcount :in std_logic_vector(H_COUNT_W-1 downto 0);
    vcount :in std_logic_vector(V_COUNT_W-1 downto 0);
    RED      : out std_logic;
    BLUE     : out std_logic;
    GREEN    : out std_logic);

```

```

end component;
-----
begin
    CLK_IN <= CLK_25MHz;
    nRST    <= not RST;

    dcm_gen: component dcm_block
        port map (
            CLK_IN  => CLK_IN,
            DCM_CLK => DCM_CLK);
    VgaCtrl800_600 : vga_800_600_60
        port map (
            dcm_clk    => DCM_CLK,
            rst        => nRST,
            video_ena => video_ena,
            HS         => HS,
            hcount     => hcount,
            vcount     => vcount,
            VS=>VS) ;

    p_RGB : component RGB_gen
        port map (
            dcm_clk    => dcm_clk,
            video_ena => video_ena,
            hcount     => hcount,
            vcount     => vcount,
            RED        => RED,
            BLUE       => BLUE,
            GREEN      => GREEN);
end Structural;

```

Thiết lập cài đặt cho đầu vào đầu ra của thiết kế như sau: (vgacompc.ucf).

Thiết lập này có thể khác nhau cho các mạch thực tế khác nhau:

```

NET "BLUE" LOC = P83;
NET "GREEN" LOC = P89;
NET "RED" LOC = P90;
NET "RST" LOC = P29;
NET "CLK_25MHz" LOC = P184;
NET "HS" LOC = P78;
NET "VS" LOC = P82;
NET "CLK_25MHz" TNM_NET = "CLK_25MHz";
TIMESPEC TS_CLK_25MHz = PERIOD "CLK_25MHz" 35 ns HIGH 50 %;
NET "BLUE" SLEW = FAST;
NET "CLK_25MHz" SLEW = FAST;
NET "GREEN" SLEW = FAST;
NET "HS" SLEW = FAST;

```

```

NET "RED" SLEW = FAST;
NET "VS" SLEW = FAST;
OFFSET = OUT 35 ns AFTER "CLK_25MHz";
OFFSET = IN 35 ns VALID 35 ns BEFORE "CLK_25MHz" RISING;

```

Ví dụ trên minh họa cho quá trình điều khiển VGA bằng thiết kế VHDL, người học có thể trên cơ sở đó thiết kế các khối điều khiển VGA hoàn chỉnh có khả năng hiển thị ký tự văn bản, đối tượng đồ họa theo yêu cầu. Kết quả tổng hợp cho thấy khối thiết kế chiếm một lượng tài nguyên Logic rất nhỏ và có thể hoạt động với tốc độ lên tới cỡ 200Mhz nghĩa là có thể đáp ứng được những màn hình có độ phân giải lớn và tốc độ quét cao.

Device utilization summary:

```

Selected Device : 3s500epq208-5
Number of Slices: 28 out of 4656 0%
Number of Slice Flip Flops: 26 out of 9312 0%
Number of 4 input LUTs: 49 out of 9312 0%
Number of IOs: 7
Number of bonded IOBs: 7 out of 158 4%
Number of GCLKs: 1 out of 24 4%
Number of DCMs: 1 out of 4 25%

```

TIMING REPORT

Clock Information:

Clock Signal	Clock buffer(FF name)	Load
CLK	dcm_gen/DCM_SP_inst:CLKFX	26

Timing Summary:

Speed Grade: -5

Minimum period: 4.770ns (Maximum Frequency: 209.651MHz)

Minimum input arrival time before clock: 3.838ns

Maximum output required time after clock: 4.040ns

Maximum combinational path delay: No path found

Kết quả về mặt thời gian tĩnh của mạch VGA sau khi kết nối và sắp đặt như sau:

Data Sheet report:

All values displayed in nanoseconds (ns)

Clock to Setup on destination clock clk

Src:Rise	Src:Fall	Src:Rise	Src:Fall
----------	----------	----------	----------

Source	Clock	Dest:Rise	Dest:Rise	Dest:Fall	Dest:Fall
clk		3.744			

Với mã nguồn trên quan sát trên thực tế sẽ thu được hình ảnh có dạng sau sau trên màn hình:



Hình 4.77. Kết quả trên mạch FPGA của khối điều khiển VGA

## Bài tập chương 4

### 1. Bài tập cơ sở

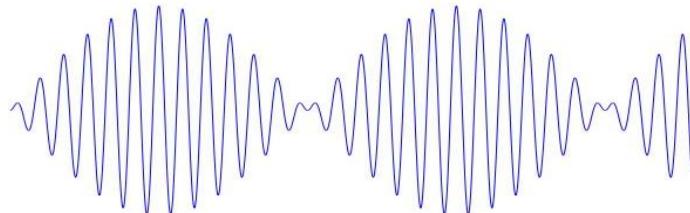
1. Thiết kế, tổng hợp các cổng logic cơ bản trên FPGA kiểm tra trên mạch thí nghiệm.
2. Thiết kế, tổng hợp các flip-flop D, JK, T, RS trên FPGA. Kiểm tra hoạt động trên mạch thí nghiệm.
3. Tổng hợp các khối đếm để chia tần số từ tần số của bộ tạo dao động ra tần số 1HZ, quan sát kết quả bằng Led Diod.
4. Thiết kế khối đếm nhị phân 4 bit, tổng hợp và hiển thị trên Led 7 đoạn.
5. Thiết kế, tổng hợp đồng hồ số trên FPGA hiển thị giờ phú thông qua 4 ký tự số của led 7 đoạn. Sử dụng phím ấn để đặt lại giờ phút, giây.
6. Thiết kế, tổng hợp bộ cộng NBCD cho 2 số có 1 chữ số trên FPGA hiển thị đầu vào và đầu ra trên led 7 đoạn, trong đó đầu vào được lấy từ switch.
7. Thiết kế, tổng hợp bộ trừ NBCD 1 số có 2 số cho 1 số có 1 chữ số trên FPGA hiển thị kết quả và đầu ra trên led 7 đoạn, trong đó đầu vào được lấy từ các switch.
8. Sử dụng các giao tiếp cơ bản (Switch, Led, 7-Seg...) tổng hợp các khối dịch theo các cách khác nhau: sử dụng toán tử, không dùng toán tử trên FPGA. So sánh kết quả thu được về mặt tài nguyên và về mặt diện tích.
9. Sử dụng các giao tiếp cơ bản (Switch, Led, 7-Seg...) tổng hợp các bộ cộng theo các cách khác nhau: sử dụng toán tử, nối tiếp, nối tiếp bit, thấy nhớ trước trên FPGA. So sánh kết quả thu được về mặt tài nguyên và về mặt diện tích.
10. Sử dụng các giao tiếp cơ bản (Switch, Led, 7-Seg...) tổng hợp các bộ nhân số nguyên không dấu theo các cách khác nhau: sử dụng toán tử, cộng dịch trái, cộng dịch phải trên FPGA. So sánh kết quả thu được về mặt tài nguyên và về mặt diện tích.
11. Sử dụng các giao tiếp cơ bản (Switch, Led, 7-Seg...) tổng hợp các bộ nhân số nguyên không dấu dùng thuật toán: sử dụng toán tử, Booth2, Booth4 trên FPGA. So sánh kết quả thu được về mặt tài nguyên và về mặt diện tích.
12. Sử dụng các giao tiếp cơ bản (Switch, Led, 7-Seg...) tổng hợp các bộ chia số nguyên không dấu theo các cách khác nhau: sử dụng toán tử, Booth2, Booth4 trên FPGA. So sánh kết quả thu được về mặt tài nguyên và về mặt diện tích.

13. Sử dụng các giao tiếp cơ bản (Switch, Led, 7-Seg...) tổng hợp các bộ chia số nguyên có dấu theo các cách khác nhau: sử dụng toán tử, Booth2, Booth4 trên FPGA. So sánh kết quả thu được về mặt tài nguyên và về mặt diện tích.
14. Sử dụng các giao tiếp cơ bản (Switch, Led, 7-Seg...) tổng hợp khối cộng số thực dấu phẩy động theo sơ đồ thuật toán ở chương III và theo cách sử dụng IP Core FPU trên FPGA. So sánh kết quả thu được về mặt tài nguyên và về mặt diện tích.
15. Sử dụng các giao tiếp cơ bản (Switch, Led, 7-Seg...) tổng hợp khối nhân số thực dấu phẩy động theo sơ đồ thuật toán ở chương III và theo cách sử dụng IP Core FPU trên FPGA. So sánh kết quả thu được về mặt tài nguyên và về mặt diện tích.
16. Sử dụng các giao tiếp cơ bản (Switch, Led, 7-Seg...) tổng hợp khối chia số thực dấu phẩy động theo sơ đồ thuật toán ở chương III và theo cách sử dụng IP Core FPU trên FPGA. So sánh kết quả thu được về mặt tài nguyên và về mặt diện tích.
17. Thiết kế tổng hợp khối FIFO trên FPGA bằng cách sử dụng thuật toán khôi FIFO ở chương III và dùng IP Core có sẵn, so sánh kết quả tổng hợp theo từng cách.
18. Xây dựng khôi nhân sử dụng Dedicated Multiplier, so sánh kết quả tổng hợp với các bộ nhân số nguyên đã làm ở các bài ở chương III.

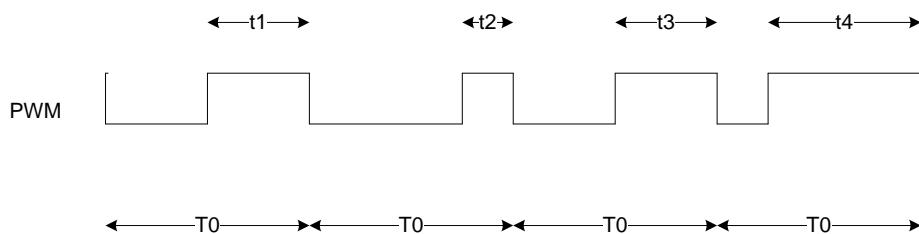
## 2. Bài tập nâng cao

1. Thiết kế khôi truyền nhận thông tin dị bộ nối tiếp (UART) hiện thực hóa trên FPGA thực hiện truyền và nhận ký tự chuẩn thông qua Hyper Terminal.
2. Thiết kế khôi truyền nhận thông tin qua chuẩn I2C, hiện thực hóa và kiểm tra trên FPGA với IC AD/DA PCF8591.
3. Thiết kế khôi truyền nhận thông tin qua giao thức chuẩn SPI bằng VHDL, hiện thực hóa và kiểm tra trên FPGA.
4. Thiết kế hoàn chỉnh khôi truyền nhận chuẩn PS/2 để giao tiếp với bàn phím chuẩn.
5. Thiết kế hoàn chỉnh khôi truyền nhận chuẩn PS/2 để giao tiếp với chuột máy tính.
6. Thiết kế hoàn chỉnh khôi giao tiếp với màn hình LCD 1602A ở các chế độ làm việc 4 bit và 8 bit.

7. Thiết kế khối nhập liệu từ bàn phím chuẩn PS/2, dữ liệu nhập vào được hiển thị lên màn hình LCD1602A.
8. Thiết kế khối nhập liệu từ bàn phím chuẩn PS/2, dữ liệu nhập vào được truyền thông qua cổng giao tiếp RS232.
9. Bộ tổng hợp tần số NCO, xuất ra dạng sóng hình sin với tần số có thể thay đổi được.
10. Bộ điều chế, thu và biến đổi tín hiệu AM đơn giản sử dụng khối NCO kết hợp với biến điều biên độ  $U_0$  theo quy luật có giải tần thấp hơn nhiều so với giải tần của sóng điều chế theo hình vẽ sau:

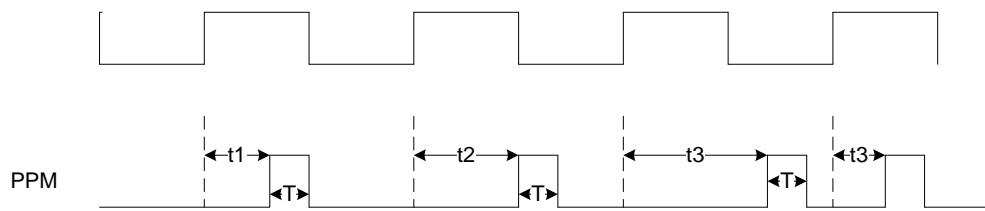


11. Thiết kế khối kết xuất tín hiệu điều chế xung dài rộng PWM (pulse width modulation) như hình vẽ sau



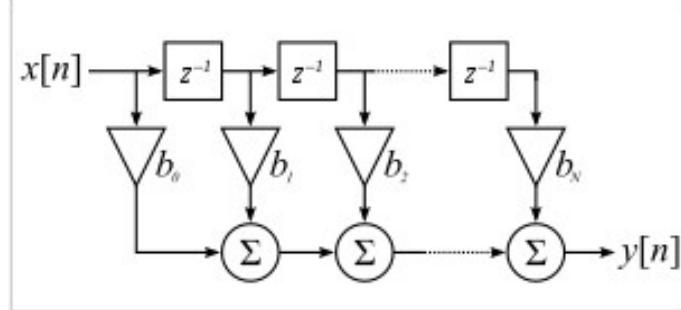
Tín hiệu đầu ra là tín hiệu xung vuông có chu kỳ không đổi là  $T_0$  nhưng có độ rộng xung (mức 1) thay đổi theo thời gian  $t_1, t_2, t_3, t_4 \dots$  theo một quy luật tùy ý (phụ thuộc thông tin điều chế).

12. Thiết kế khối kết xuất tín hiệu điều chế xung dài rộng PPM (pulse phase modulation) như hình vẽ sau



Tín hiệu đầu ra là tín hiệu xung vuông có có độ rộng xung (mức 1) không đổi T nhưng có độ lệch pha so với xung chuẩn lần lượt các giá trị t1, t2, t3, t4...theo một quy luật tùy ý (phụ thuộc thông tin điều chế). Quan sát kết quả trên Osiloscope.

13. Thiết kế và kiểm tra khôi đếm thời gian và định thời với xung vào chuẩn 1Mhz (chia từ DCM) có chức năng làm việc tương tự như Timer0 và Timer1 trong vi điều khiển 89c51. Cấu tạo của bộ đếm/định thời gồm có thanh ghi cấu hình TCON, hai thanh ghi đếm THLx, THx (với x = 0, 1) Timers có thể hoạt động ở chế độ 8 bit tự động khởi tạo lại hoặc ở chế độ 16-bit. Các Timers phải sinh ra tín hiệu báo ngắt mỗi khi đếm xong. Chi tiết xem thêm trong tài liệu hướng dẫn của 89c51
14. Thiết kế và kiểm tra khôi đếm thời gian và định thời với xung vào chuẩn 1Mhz (chia từ DCM) có chức năng làm việc tương tự như Timer2 trong vi điều khiển 89c52, ngoài những chức năng như Timer1 và timer 2 còn có hỗ trợ cồng vào ra tốc độ cao. Chi tiết xem thêm trong tài liệu hướng dẫn của 89C52.
15. Nghiên cứu xây dựng khôi mã hóa theo thuật toán AES, mô tả bằng VHDL, tổng hợp và hiện thực hóa trên FPGA. Xem thêm trong tài liệu [36].
16. Nghiên cứu xây dựng khôi mã hóa theo thuật toán DES, mô tả bằng VHDL và hiện thực hóa trên FPGA. Xem thêm trong tài liệu giới thiệu trong [37]
17. Nghiên cứu xây dựng khôi mã hóa theo thuật toán RSA-128bit với yêu cầu tính cơ bản là thực hiện phép toán tính module của lũy thừa  $A^B$  theo số N, Chi tiết về RSA xem trong tài liệu [38]. Trong thiết kế sử dụng khôi nhân MontGomery ở phần bài tập chương III. Hiện thực hóa, kiểm tra trên FPGA.
18. Nghiên cứu thuật toán CORDIC (Coordinate Rotation Digital Computer) ứng dụng để tính toán các hàm SIN, COSIN. Xem thêm tài liệu giới thiệu trong [35].
19. Nghiên cứu thuật toán CORDIC (Coordinate Rotation Digital Computer) ứng dụng để tính toán các hàm ARCTAN. Xem thêm tài liệu giới thiệu trong [35].
20. Nghiên cứu xây dựng sơ đồ hiện thực hóa cho biến đổi Fourier DFT (*Discret Fourier Transform*) và sơ đồ hiện thực hóa trên FPGA với N= 4, 8, 16.
21. Nghiên cứu xây dựng sơ đồ hiện thực hóa thiết kế biến đổi Fourier nhanh cho dãy giá trị rời rạc FFT (*Fast Fourier Transform*) cho N = 16 và phân chia theo cơ số 2, cơ số 4 theo thời gian.
22. Thiết kế mạch lọc số theo sơ đồ dưới đây:



Ở sơ đồ trên ký hiệu Z tương ứng là các Flip-flop giữ chậm, ký hiệu tam giác là các khối nhân, ký hiệu sig-ma là các khối cộng, toàn bộ khối hoạt động đồng bộ.  $b_i$  là các hằng số của bộ lọc,  $x[n]$ ,  $y[n]$  là chuỗi tín hiệu rời rạc vào và ra từ bộ lọc.

23. Hiện thực giao thức VGA trên mạch FPGA có khả năng truy xuất hình ảnh và văn bản. Trong thiết kế sử dụng các khối cơ bản trình bày trong mục 4.5 và bổ xung đầy đủ khái ROM cho ký tự và khái RAM để lưu nhớ đối tượng hiển thị.

### 3. Câu hỏi ôn tập lý thuyết

1. Định nghĩa FPGA, ưu điểm của FPGA với các chip khả trinh khác.
2. Nguyên lý làm việc của FPGA, khả năng tái cấu trúc, tài nguyên FPGA.
3. Trình bày kiến trúc tổng quan của FPGA, các dạng tài nguyên của FPGA.
4. Trình bày kiến trúc tổng quan của Spartan 3E FPGA, các tài nguyên của FPGA này.
5. Trình bày cấu trúc chi tiết của CLB, SLICE, LUT.
6. Trình bày cấu trúc và nguyên lý làm việc của Arithmetic chain, Carry Chain, vai trò của các chuỗi này trong FPGA
7. Trình bày cấu trúc của Programmable Interconnects trong FPGA
8. Trình bày cấu trúc của IOB trong FPGA.
9. Trình bày đặc điểm, cấu trúc và cách sử dụng của Distributed RAM và Shift Register trong FPGA.
10. Trình bày đặc điểm, cấu trúc và cách sử dụng của Block RAM và Multiplier 18x18 trong Spartan 3E FPGA.
11. Quy trình thiết kế trên FPGA.
12. Khái niệm tổng hợp thiết kế. Cách thiết lập các điều kiện ràng buộc cho thiết kế.
13. Các bước hiện thực thiết kế (Translate, Mapping, Place & Routing)
14. Các dạng kiểm tra thiết kế trên FPGA

## **PHỤ LỤC**

**Phụ lục 1: THỐNG KÊ CÁC HÀM, THỦ TỤC, KIỂU DỮ LIỆU  
CỦA VHDL TRONG CÁC THƯ VIỆN CHUẨN IEEE.**

**1. Các kiểu dữ liệu hỗ trợ trong các thư viện chuẩn IEEE**

Tên kiểu	Giải thích
<b>Thư viện IEEE.STD_LOGIC_1164</b>	
<b>BIT</b>	STD_ULOGIC
<b>BITVECTOR</b>	STD_LOGIC_VECTOR
<b>STD_ULOGIC</b>	9 mức logic chuẩn gồm X, 0, 1, L, H, Z, W, -, U
<b>STD_LOGIC</b>	Giống STD_ULOGIC nhưng được định nghĩa cách thức các giá trị hợp với nhau
<b>STD_ULOGIC_VECTORTOR</b>	Chuỗi STD_ULOGIC
<b>STD_LOGIC_VECTOR</b>	Chuỗi STD_LOGIC
<b>X01</b>	Kiểu con của STD_LOGIC với chỉ các giá trị (0, 1, X)
<b>X01Z</b>	Kiểu con của STD_LOGIC với chỉ các giá trị (0, 1, X, Z)
<b>UX01</b>	Kiểu con của STD_LOGIC với chỉ các giá trị (0, 1, U, X)
<b>UX01Z</b>	Kiểu con của STD_LOGIC với chỉ các giá trị (0, 1, U, X, Z)
<b>Thư viện IEEE.STD_LOGIC_ARITH</b>	
<b>UNSIGNED</b>	Chuỗi STD_LOGIC được xem như số không dấu
<b>SIGNED</b>	Chuỗi STD_LOGIC được xem như số có dấu
<b>SMALL_INT</b>	Kiểu INTEGER với chỉ các giá trị 0, 1
<b>Thư viện IEEE.STD_LOGIC_UNSIGNED</b>	
<b>CONV_INTEGER</b>	STD_LOGIC_VECTOR
<b>Thư viện IEEE.STD_LOGIC_SIGNED</b>	
<b>CONV_INTEGER</b>	STD_LOGIC_VECTOR
<b>Thư viện IEEE.NUMERIC_BIT</b>	
<b>SIGNED</b>	Chuỗi BIT được xem như số có dấu
<b>UNSIGNED</b>	Chuỗi BIT được xem như số không dấu
<b>Thư viện IEEE.NUMERIC_STD</b>	
<b>SIGNED</b>	Chuỗi STD_LOGIC được xem như số có dấu
<b>UNSIGNED</b>	Chuỗi STD_LOGIC được xem như số không dấu

## 2. Các hàm thông dụng hỗ trợ trong các thư viện chuẩn IEEE

Tên hàm (Đối biến)	Giá trị trả về	Ghi chú
<b>Thư viện IEEE.STD_LOGIC_1164</b>		
<b>AND( l : std_ulogic; r : std_ulogic )</b>	UX01	
<b>NAND( l : std_ulogic; r : std_ulogic )</b>	UX01	
<b>OR( l : std_ulogic; r : std_ulogic )</b>	UX01	
<b>NOR( l : std_ulogic; r : std_ulogic )</b>	UX01	
<b>XOR( l : std_ulogic; r : std_ulogic )</b>	UX01	
<b>XNOR( l : std_ulogic; r : std_ulogic )</b>	UX01	
<b>NOT( l : std_ulogic; r : std_ulogic )</b>	UX01	
<b>AND( l, r : std_logic_vector )</b>	std_logic_vector	
<b>NAND( l, r : std_logic_vector )</b>	std_logic_vector	
<b>OR( l, r : std_logic_vector )</b>	std_logic_vector	
<b>NOR( l, r : std_logic_vector )</b>	std_logic_vector	
<b>XOR( l, r : std_logic_vector )</b>	std_logic_vector	
<b>XNOR( l, r : std_logic_vector )</b>	std_logic_vector	
<b>NOT((l, r : std_ulogic_vector))</b>	std_logic_vector	
<b>AND( l, r : std_ulogic_vector )</b>	std_ulogic_vector	
<b>NAND( l, r : std_ulogic_vector )</b>	std_ulogic_vector	
<b>OR( l, r : std_ulogic_vector )</b>	std_ulogic_vector	
<b>NOR( l, r : std_ulogic_vector )</b>	std_ulogic_vector	
<b>XOR( l, r : std_ulogic_vector )</b>	std_ulogic_vector	
<b>XNOR( l, r : std_ulogic_vector )</b>	std_ulogic_vector	
<b>NOT(( l, r : std_ulogic_vector ))</b>	std_ulogic_vector	
<b>rising_edge (SIGNAL s : std_ulogic)</b>	BOOLEAN	
<b>falling_edge (SIGNAL s : std_ulogic)</b>	BOOLEAN	
<b>Is_X ( s : std_ulogic_vector )</b>	BOOLEAN	
<b>Is_X ( s : std_ulogic_vector )</b>	BOOLEAN	
<b>Is_X ( s : std_ulogic )</b>	BOOLEAN	
<b>Thư viện IEEE.STD_LOGIC_ARITH</b>		

+, - (L, R: SIGNED, SIGNED)	SIGNED	
+, - (L, R: UNSIGNED, UNSIGNED)	UNSIGNED	
+, - (L, R: UNSIGNED, SIGNED)	SIGNED	
+, - (L: SIGNED, R: INTEGER)	SIGNED	
+, - (L: UNSIGNED, R: INTEGER)	UNSIGNED	
+, - (L: STD_ULOGIC, R: SIGNED)	SIGNED	
+, - (L : STD_ULOGIC, R: UNSIGNED)	UNSIGNED	
+, - (L: SIGNED, R: UNSIGNED)	STD_LOGIC_VECTOR	
+, - (L: INTEGER, R: SIGNED, UNSIGNED)	STD_LOGIC_VECTOR	
+, - (L: STD_ULOGIC, R: SIGNED, UNSIGNED)	STD_LOGIC_VECTOR	
* (L, R: SIGNED, SIGNED)	SIGNED	
* (L, R: UNSIGNED, UNSIGNED)	UNSIGNED	
* (L, R: UNSIGNED, SIGNED)	SIGNED	
* (L: SIGNED, UNSIGNED, R: SIGNED, UNSIGNED)	STD_LOGIC_VECTOR	
<, <=, >, >=, = (L: SIGNED, UNSIGNED, R: SIGNED, UNSIGNED)	BOOLEAN	
<, <=, >, >=, = (L: INTEGER, R: SIGNED, UNSIGNED)	BOOLEAN	
<b>SHL</b> (ARG: SIGNED; COUNT: UNSIGNED)	SIGNED	
<b>SHL</b> (ARG: UNSIGNED; COUNT: UNSIGNED)	UNSIGNED	
<b>SHR</b> (ARG: SIGNED; COUNT: UNSIGNED)	SIGNED	
<b>SHR</b> (ARG: UNSIGNED; COUNT: UNSIGNED)	UNSIGNED	

#### Thư viện IEEE.STD\_LOGIC\_UNSIGNED

+, - (L: STD_LOGIC_VECTOR, R: STD_LOGIC_VECTOR)	STD_LOGIC_VECTOR	
+, - (L: STD_LOGIC_VECTOR, R: INTEGER)	STD_LOGIC_VECTOR	
+, - (L: STD_LOGIC_VECTOR, R: STD_LOGIC)	STD_LOGIC_VECTOR	
* (L: STD_LOGIC_VECTOR, R: STD_LOGIC_VECTOR)	STD_LOGIC_VECTOR	
<, <=, >, >=, = (L: STD_LOGIC_VECTOR, R: INTEGER)	BOOLEAN	
<, <=, >, >=, = (L: STD_LOGIC_VECTOR, R:	BOOLEAN	

STD_LOGIC_VECTOR)		
<b>SHL(ARG: STD_LOGIC_VECTOR; COUNT: STD_LOGIC_VECTOR)</b>	STD_LOGIC_VECTOR	
<b>SHR(ARG: STD_LOGIC_VECTOR; COUNT: STD_LOGIC_VECTOR)</b>	STD_LOGIC_VECTOR	
<b>Thư viện IEEE.STD_LOGIC_SIGNED</b>		
+,- (L: STD_LOGIC_VECTOR, R: STD_LOGIC_VECTOR)	STD_LOGIC_VECTOR	
+,- (L: STD_LOGIC_VECTOR, R: INTEGER)	STD_LOGIC_VECTOR	
+,- (L: STD_LOGIC_VECTOR, R: STD_LOGIC)	STD_LOGIC_VECTOR	
* (L: STD_LOGIC_VECTOR, R: STD_LOGIC_VECTOR)	STD_LOGIC_VECTOR	
<, <=, >, >=, = (L: STD_LOGIC_VECTOR, R: STD_LOGIC_VECTOR)	BOOLEAN	
<, <=, >, >=, = (L: STD_LOGIC_VECTOR, R: INTEGER)	BOOLEAN	
<b>SHL(ARG: STD_LOGIC_VECTOR; COUNT: STD_LOGIC_VECTOR)</b>	STD_LOGIC_VECTOR	
<b>SHR(ARG: STD_LOGIC_VECTOR; COUNT: STD_LOGIC_VECTOR)</b>	STD_LOGIC_VECTOR	
<b>Thư viện IEEE.NUMERIC_BIT</b>		
+,- (L, R: UNSIGNED)	UNSIGNED	
+,- (L, R: SIGNED)	SIGNED	
+,- (L: NATURAL, R: SIGNED)	SIGNED	
+,- (L: NATURAL, R: UNSIGNED)	UNSIGNED	
+,- (L: INTEGER, R: SIGNED)	SIGNED	
*, /, mod, rem (L, R: UNSIGNED)	UNSIGNED	
*, /, mod, rem (L, R: SIGNED)	SIGNED	
*, /, mod, rem (L: NATURAL, R: UNSIGNED)	UNSIGNED	
*, /, mod, rem (L: INTEGER, R: SIGNED)	SIGNED	
<, <=, >, >=, = (L: UNSIGNED, R: UNSIGNED)	BOOLEAN	
<, <=, >, >=, = (L: SIGNED, R: SIGNED)	BOOLEAN	
<, <=, >, >=, = (L: INTEGER, R: SIGNED)	BOOLEAN	
<, <=, >, >=, = (L: NATURAL, R: UNSIGNED)	BOOLEAN	

UNSIGNED <b>sll, sla, srl, sra, ror, rol</b> INTEGER	UNSIGNED	
SIGNED <b>sll, sla, srl, sra, ror, rol</b> INTEGER	SIGNED	
<b>SHIFT_LEFT(L: UNSIGNED, R: NATURAL)</b>	UNSIGNED	
<b>SHIFT_LEFT(L: SIGNED, R: NATURAL)</b>	SIGNED	
<b>SHIFT_RIGHT(L: UNSIGNED, R: NATURAL)</b>	UNSIGNED	
<b>SHIFT_RIGHT(L: SIGNED, R: NATURAL)</b>	SIGNED	
<b>ROTATE_RIGHT(L: UNSIGNED, R: NATURAL)</b>	UNSIGNED	
<b>ROTATE_RIGHT(L: SIGNED, R: NATURAL)</b>	SIGNED	
<b>RESIZE(L: UNSIGNED, R: NATURAL)</b>	UNSIGNED	
<b>RESIZE(L: SIGNED, R: NATURAL)</b>	SIGNED	
<b>Thư viện IEEEEE.NUMERIC_STD</b>		
+, - (L, R: UNSIGNED)	UNSIGNED	
+, - (L, R: SIGNED)	SIGNED	
+, - (L: NATURAL, R: SIGNED)	SIGNED	
+, - (L: NATURAL, R: UNSIGNED)	UNSIGNED	
+, - (L: INTEGER, R: SIGNED)	SIGNED	
*, /, mod, rem (L, R: UNSIGNED)	UNSIGNED	
*, /, mod, rem (L, R: SIGNED)	SIGNED	
*, /, mod, rem (L: NATURAL, R: UNSIGNED)	UNSIGNED	
*, /, mod, rem (L: INTEGER, R: SIGNED)	SIGNED	
<, <=, >, >=, = (L: UNSIGNED, R: UNSIGNED)	BOOLEAN	
<, <=, >, >=, = (L: SIGNED, R: SIGNED)	BOOLEAN	
<, <=, >, >=, = (L: INTEGER, R: SIGNED)	BOOLEAN	
<, <=, >, >=, = (L: NATURAL, R: UNSIGNED)	BOOLEAN	
UNSIGNED <b>sll, sla, srl, sra, ror, rol</b> INTEGER	UNSIGNED	
SIGNED <b>sll, sla, srl, sra, ror, rol</b> INTEGER	SIGNED	
<b>SHIFT_LEFT(L: UNSIGNED, R: NATURAL)</b>	UNSIGNED	
<b>SHIFT_LEFT(L: SIGNED, R: NATURAL)</b>	SIGNED	
<b>SHIFT_RIGHT(L: UNSIGNED, R: NATURAL)</b>	UNSIGNED	
<b>SHIFT_RIGHT(L: SIGNED, R: NATURAL)</b>	SIGNED	
<b>ROTATE_RIGHT(L: UNSIGNED, R: NATURAL)</b>	UNSIGNED	
<b>ROTATE_RIGHT(L: SIGNED, R: NATURAL)</b>	SIGNED	

<b>RESIZE(L: UNSIGNED, R: NATURAL)</b>	UNSIGNED	
<b>RESIZE(L: SIGNED, R: NATURAL)</b>	SIGNED	

### 3. Các hàm phục vụ cho quá trình mô phỏng kiểm tra thiết kế

Tên hàm (Đối biến)	Giá trị trả về	Ghi chú
<b>Thư viện IEEE.STD_LOGIC_TEXTIO</b>		
<b>READ(l : inout LINE, R: out std_ulogic )</b>	std_ulogic trong R	
<b>READ(l : inout LINE, R: out std_ulogic, Good: Boolean )</b>	std_ulogic trong R	
<b>READ(l : inout LINE, R: out std_ulogic_vector )</b>	std_ulogic_vector trong R	
<b>READ(l : inout LINE, R: out std_ulogic_vector, Good: Boolean )</b>	std_ulogic_vector trong R	
<b>WRITE(l : inout LINE, R: in std_ulogic_vector )</b>	LINE	
<b>WRITE(l : inout LINE, R: in std_ulogic_vector, Good: Boolean )</b>	LINE	
<b>READ(l : inout LINE, R: out std_logic_vector)</b>	std_logic trong R	
<b>READ(l : inout LINE, R: out std_logic_vector, Good: Boolean )</b>	std_logic trong R	
<b>WRITE(l : inout LINE, R: in std_logic_vector, Good: Boolean )</b>	LINE	
<b>HREAD(l : inout LINE, R: out std_ulogic_vector)</b>	std_logic trong R	
<b>HREAD(l : inout LINE, R: out std_ulogic_vector, Good: Boolean )</b>	std_logic trong R	
<b>HWRITE(l : inout LINE, R: in std_ulogic_vector, Good: Boolean )</b>	LINE	
<b>HREAD(l : inout LINE, R: out std_logic_vector)</b>	std_logic trong R	
<b>HREAD(l : inout LINE, R: out std_logic_vector, Good: Boolean )</b>	std_logic trong R	
<b>HWRITE(l : inout LINE, R: in std_logic_vector, Good: Boolean )</b>	LINE	
<b>OREAD(l : inout LINE, R: out std_ulogic_vector)</b>	std_logic trong R	
<b>OREAD(l : inout LINE, R: out std_ulogic_vector, Good: Boolean )</b>	std_logic trong R	

<b>OWRITE</b> (l : inout LINE, R: in std_ulogic_vector, Good: Boolean)	LINE	
<b>OREAD</b> (l : inout LINE, R: out std_logic_vector)	std_logic trong R	
<b>OREAD</b> (l : inout LINE, R: out std_logic_vector, Good: Boolean)	std_logic trong R	
<b>OWRITE</b> (l : inout LINE, R: in std_logic_vector, Good: Boolean)	LINE	
<b>Thư viện STD.ENV</b>		
<b>STOP</b> (STATUS: INTEGER)	PROCEDURE	
<b>FINISH</b> (STATUS: INTEGER)	UNSIGNED	
<b>RESOLUTION_LIMIT ()</b>	Delay_length (Bước thời gian cơ sở của quá trình mô phỏng)	
<b>Thư viện IEEE.STD_TEXTIO</b>		
<b>READLINE</b> (file f: TEXT; L: out LINE)	String trong LINE	
<b>READ</b> (L:inout LINE; VALUE: out bit; GOOD : out BOOLEAN)	BIT trong VALUE	
<b>READ</b> (L:inout LINE; VALUE: out bit)	BIT trong VALUE	
<b>READ</b> (L:inout LINE; VALUE: out bit_vector; GOOD : out BOOLEAN)	bit_vector trong VALUE	
<b>READ</b> (L:inout LINE; VALUE: out bit_vector)	bit_vector trong VALUE	
<b>READ</b> (L:inout LINE; VALUE: out BOOLEAN; GOOD : out BOOLEAN)	BOOLEAN trong VALUE	
<b>READ</b> (L:inout LINE; VALUE: out BOOLEAN)	BOOLEAN trong VALUE	
<b>READ</b> (L:inout LINE; VALUE: out Charater; GOOD : out BOOLEAN)	Charater trong VALUE	
<b>READ</b> (L:inout LINE; VALUE: out Charater)	Charater trong VALUE	
<b>READ</b> (L:inout LINE; VALUE: out INTEGER; GOOD : out BOOLEAN)	INTEGER trong VALUE	
<b>READ</b> (L:inout LINE; VALUE: out INTEGER)	INTEGER trong	

	VALUE	
<b>READ(L:inout LINE; VALUE: out REAL; GOOD : out BOOLEAN)</b>	REAL trong VALUE	
<b>READ(L:inout LINE; VALUE: out REAL)</b>	REAL trong VALUE	
<b>READ(L:inout LINE; VALUE: out STRING; GOOD : out BOOLEAN)</b>	STRING trong VALUE	
<b>READ(L:inout LINE; VALUE: out STRING)</b>	STRING trong VALUE	
<b>READ(L:inout LINE; VALUE: out TIME; GOOD : out BOOLEAN)</b>	TIME trong VALUE	
<b>READ(L:inout LINE; VALUE: out TIME)</b>	TIME trong VALUE	
<b>SREAD (L : inout LINE; VALUE : out STRING; STRLEN : out NATURAL);</b>	STRING trong VALUE	
<b>OREAD(L:inout LINE; VALUE: out bit_vector; GOOD : out BOOLEAN)</b>	bit_vector trong VALUE	
<b>OREAD(L:inout LINE; VALUE: out bit_vector)</b>	bit_vector trong VALUE	
<b>HREAD(l : inout LINE, R: out bit_vector)</b>	Bit_vector trong R	
<b>HREAD(l : inout LINE, R: out bit_vector, Good: Boolean)</b>	Bit_vector trong R	
<b>WRITELINE (file f : TEXT; L : inout LINE)</b>	Ghi LINE ra file	
<b>WRITE(L : inout LINE; VALUE : in bit)</b>		
<b>WRITE(L : inout LINE; VALUE : in bit_vector)</b>		
<b>WRITE(L : inout LINE; VALUE : in BOOLEAN)</b>		
<b>WRITE(L : inout LINE; VALUE : in CHARACTER)</b>		
<b>WRITE(L : inout LINE; VALUE : in INTEGER)</b>		
<b>WRITE(L : inout LINE; VALUE : in REAL)</b>		
<b>WRITE(L : inout LINE; VALUE : in TIME)</b>		
<b>SWRITE(L : inout LINE; VALUE : in STRING)</b>		
<b>OWRITE(l : inout LINE, R: in BIT_VECTOR)</b>		

<b>HWRITE</b> (l : inout LINE, R: in BIT_VECTOR)		
--	--	--

#### 4. Các hàm biến đổi kiểu dữ liệu dùng trong VHDL

Tên hàm (Đổi biến)	Giá trị trả về	Ghi chú
<b>Thư viện IEEE.STD_LOGIC_1164</b>		
<b>TO_BIT(Arg: STD_ULOGIC)</b>	BIT	
<b>TO_BITVECTOR (Arg: STD_LOGIC_VECTOR)</b>	BIT_VECTOR	
<b>TO_BITVECTOR</b> (Arg:STD_ULOGIC_VECTOR)	BIT_VECTOR	
<b>TO_STD_ULOGIC (Arg: BIT)</b>	STD_ULOGIC	
<b>TO_STD_LOGICVECTOR (Arg: BIT_VECTOR)</b>	STD_LOGIC_VEC TOR	
<b>TO_STD_LOGICVECTOR</b> (Arg: STD_ULOGIC_VECTOR)	STD_LOGIC_VEC TOR	
<b>TO_STD_ULOGICVECTOR</b> (Arg: BIT_VECTOR)	STD_ULOGIC_VE CTOR	
<b>TO_STD_ULOGICVECTOR(Arg: STD_LOGIC_VECTOR);</b>	STD_ULOGIC_VE CTOR	
<b>TO_X01(Arg: STD_LOGIC_VECTOR)</b>	STD_LOGIC_VEC TOR	
<b>TO_X01 (Arg: STD_ULOGIC_VECTOR)</b>	STD_ULOGIC_VE CTOR	
<b>TO_X01 (Arg: STD_ULOGIC)</b>	X01	
<b>TO_X01(Arg: BIT_VECTOR)</b>	STD_LOGIC_VEC TOR	
<b>TO_X01 (Arg: BIT_VECTOR)</b>	STD_ULOGIC_VE CTOR	
<b>TO_X01 (Arg: BIT)</b>	X01	
<b>TO_X01Z(Arg: STD_LOGIC_VECTOR)</b>	STD_LOGIC_VEC TOR	
<b>TO_X01Z(Arg: STD_ULOGIC_VECTOR)</b>	STD_ULOGIC_VE CTOR	
<b>TO_X01Z(Arg: STD_ULOGIC)</b>	X01Z	

<b>TO_X01Z</b> (Arg: BIT_VECTOR)	STD_ULOGIC_VECTTOR	
<b>TO_X01Z</b> (Arg: BIT_VECTOR)	STD_LOGIC_VECTOR	
<b>TO_X01Z</b> (Arg: BIT)	X01Z	
<b>TO_UX01</b> (Arg: STD_LOGIC_VECTOR)	STD_LOGIC_VECTOR	
<b>TO_UX01</b> (Arg: STD_ULOGIC_VECTOR)	STD_ULOGIC_VECTTOR	
<b>TO_UX01</b> (Arg: STD_ULOGIC)	X01Z	
<b>TO_UX01</b> (Arg: BIT_VECTOR)	STD_ULOGIC_VECTTOR	
<b>TO_UX01</b> (Arg: BIT_VECTOR)	STD_LOGIC_VECTOR	
<b>TO_UX01</b> (Arg: BIT)	X01Z	
<b>Thư viện IEEE.STD_LOGIC_ARITH</b>		
<b>CONV_INTEGER</b> (Arg: INTEGER)	INTEGER	
<b>CONV_INTEGER</b> (Arg: UNSIGNED)	INTEGER	
<b>CONV_INTEGER</b> (Arg: SIGNED)	INTEGER	
<b>CONV_INTEGER</b> (Arg: STD_ULOGIC)	SMALL_INT	
<b>CONV_UNSIGNED</b> (Arg: INTEGER, Size INTEGER)	UNSIGNED	
<b>CONV_UNSIGNED</b> (Arg: UNSIGNED, Size INTEGER)	UNSIGNED	
<b>CONV_UNSIGNED</b> (Arg: SIGNED, Size INTEGER)	UNSIGNED	
<b>CONV_UNSIGNED</b> (Arg: STD_ULOGIC, Size INTEGER)	UNSIGNED	
<b>CONV_SIGNED</b> (Arg: INTEGER, Size INTEGER)	SIGNED	
<b>CONV_SIGNED</b> (Arg: UNSIGNED, Size INTEGER)	SIGNED	
<b>CONV_SIGNED</b> (Arg: SIGNED, Size INTEGER)	SIGNED	
<b>CONV_SIGNED</b> (Arg: STD_ULOGIC, Size INTEGER)	SIGNED	

<b>CONV_STD_LOGIC_VECTOR</b> (Arg: INTEGER, Size INTEGER)	STD_LOGIC_VEC TOR	
<b>CONV_STD_LOGIC_VECTOR</b> (Arg: UNSIGNED, Size INTEGER)	STD_LOGIC_VEC TOR	
<b>CONV_STD_LOGIC_VECTOR</b> (Arg: SIGNED, Size INTEGER)	STD_LOGIC_VEC TOR	
<b>CONV_STD_LOGIC_VECTOR</b> (Arg: STD_ULOGIC, Size INTEGER)	STD_LOGIC_VEC TOR	
<b>EXT</b> (Arg: STD_LOGIC_VECTOR, Size INTEGER)	STD_LOGIC_VEC TOR	
<b>SXT</b> (Arg: STD_LOGIC_VECTOR, Size INTEGER)	STD_LOGIC_VEC TOR	
<b>Thư viện IEEE.STD_LOGIC_UNSIGNED</b>		
<b>CONV_INTEGER</b> (STD_LOGIC_VECTOR)	INTEGER	
<b>Thư viện IEEE.STD_LOGIC_SIGNED</b>		
<b>CONV_INTEGER</b> (STD_LOGIC_VECTOR)	INTEGER	
<b>Thư viện IEEE.NUMERIC_BIT</b>		
<b>TO_INTEGER</b> (Arg: UNSIGNED)	INTEGER	
<b>TO_INTEGER</b> (Arg: UNSIGNED)	INTEGER	
<b>TO_SIGNED</b> (Arg: INTEGER, Size NATURAL)	SIGNED	
<b>TO_UNSIGNED</b> (Arg: INTEGER, Size NATURAL)	UNSIGNED	
<b>Thư viện IEEE.NUMERIC_STD</b>		
<b>TO_INTEGER</b> (UNSIGNED)	INTEGER	
<b>TO_INTEGER</b> (UNSIGNED)	INTEGER	
<b>TO_SIGNED</b> (Arg: INTEGER, Size NATURAL)	SIGNED	
<b>TO_UNSIGNED</b> (Arg: INTEGER, Size NATURAL)	UNSIGNED	

## **Phụ lục 2: THỰC HÀNH THIẾT KẾ VHDL**

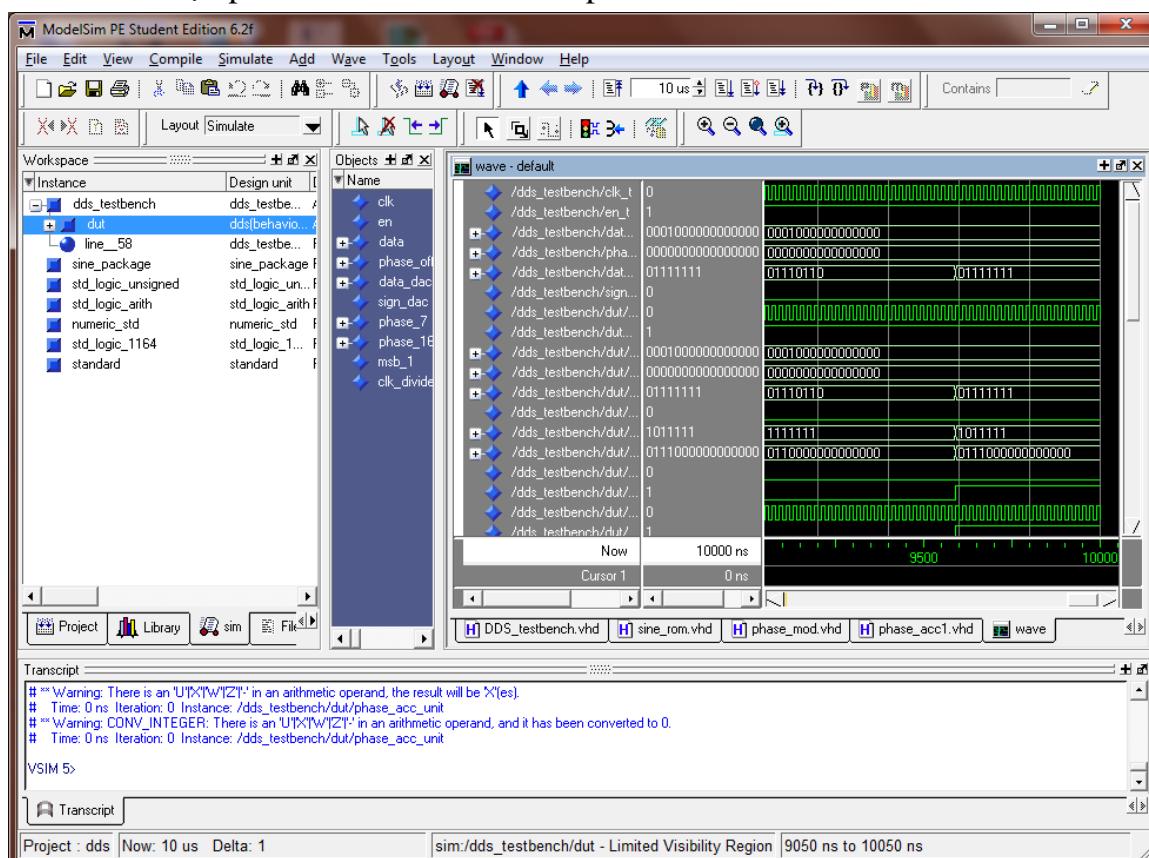
## **Bài 1: Mô phỏng VHDL trên ModelSim**

## 1. Mục đích

Giúp sinh viên làm quen với chương trình mô phỏng Modelsim, làm quen với cấu trúc chương trình VHDL và cách kiểm tra nhanh một thiết kế trên VHDL. Yêu cầu sinh viên phải có kiến thức cơ sở về VHDL.

## 2. Giới thiệu về chương trình mô phỏng Modelsim.

Do các ngôn ngữ mô tả phần cứng như VHDL được chuẩn hóa bởi IEEE và được công bố rộng rãi nên có rất nhiều các phần mềm mô phỏng mạch số được nhiều công ty khác nhau phát triển. Điểm chung của các chương trình này là đều phải có một trình biên dịch và có khả năng mô phỏng mạch theo thời gian thực, kết xuất kết quả một số dạng nhất định như File text, file định kiểu, hay phổ biến và trực quan nhất là dưới dạng giản đồ sóng. Dưới đây sẽ giới thiệu chương trình mô phỏng là ModelSim, đây là một chương trình mô phỏng khá mạnh và chính xác được phát triển bởi Mentor Graphics.



ModelSim là một chương trình phần mềm thương mại, tuy vậy bên cạnh các phiên bản phải trả tiền license, có phiên bản miễn phí dành cho sinh viên và người nghiên cứu không sử dụng với mục đích thương mại. Phiên bản này có tên

là ModelSim Student Edition có thể được tải trực tiếp từ trang chủ của Mentor Graphics theo địa chỉ

<http://model.com/content/modelsim-pe-student-edition-hdl-simulation>

Sau khi cài chương trình sẽ đòi hỏi cài đặt cấp phép sử dụng (license). Để có được license cần phải điền đủ vào bản khai báo các thông tin cá nhân như hòm thư, địa chỉ vv... Mentor Graphic sẽ gửi vào hòm thư của bạn một file license có tên là student\_license.dat, file này cho phép sử dụng phần mềm trong vòng 180 ngày, để kích hoạt license chỉ việc copy vào thư mục gốc của modelSim (thường là C:\Modeltech\_pe\_edu\_6.5 trong đó “6.5” là số hiệu phiên bản của chương trình)

*Chú ý:* Hướng dẫn mô phỏng một thiết kế và sử dụng chương trình có trong thư mục “C:\Modeltech\_pe\_edu\_6.2f\docs\pdfdocs”, đối với các phiên bản khác nhau thì có thể đường dẫn sẽ khác nhau.

Sau đây chúng ta sẽ lần lượt học cách sử dụng chương trình thông qua một ví dụ cụ thể.

### 3. Viết mã nguồn VHDL

Trong Modelsim cũng tích hợp một trình soạn thảo file nguồn tuy vậy cũng như các ngôn ngữ lập trình khác mã nguồn VHDL của thiết kế có thể được soạn thảo bằng bất kỳ một chương trình soạn thảo nào. Một trong những chương trình soạn thảo khá tốt và tiện dụng là Notepad++ (<http://notepad-plus-plus.org/download>), chương trình này hỗ trợ hiện thị nhiều ngôn ngữ lập trình khác nhau trong đó có VHDL và Verilog. File nguồn của mã VHDL có đuôi là .vhd. Khi soạn thảo file có đuôi dạng này bằng Notepad thì toàn bộ các từ khóa, cấu trúc ngôn ngữ được làm đậm hoặc đổi màu cho dễ quan sát và sửa lỗi.

```

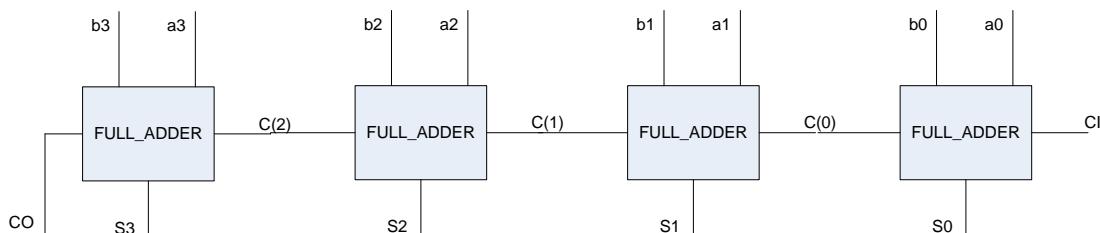
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_UNSIGNED.ALL;
4 use IEEE.STD_LOGIC_ARITH.ALL;
5
6
7 entity adder16 is
8 port(
9     Clk: in std_logic;
10    A : in std_logic_vector(15 downto 0);
11    B : in std_logic_vector(15 downto 0);
12    CI : in std_logic;
13    SUM : out std_logic_vector(15 downto 0);
14    CO : out std_logic);
15 end adder16;
16
17 architecture Behavioral of adder16 is
18 signal tmp: std_logic_vector(16 downto 0);
19 begin
20
21 tmp <= conv_std_logic_vector((conv_integer(A)+conv_integer(B)+conv_integer(CI)),17);
22
23 process(clk)
24 begin
25 if clk'event and clk='1' then
26     SUM <= tmp(15 downto 0);
27     CO <= tmp(16);
28 end if;
29 end process;

```

VHSIC Hardware Description | 698 chars 729 bytes 32 lines | Ln:15 Col:13 Sel:0 (0 bytes) in 0 ranges | UNIX | ANSI | INS | ...

### Chương trình Notepad++

Để đơn giản và dễ hiểu phần này ta sẽ minh họa bằng ví dụ quen thuộc về bộ cộng 4 bit. Bộ cộng được thiết kế đơn giản nhất bằng cách ghép nối tiếp 4 khối full\_adder 1 bit.

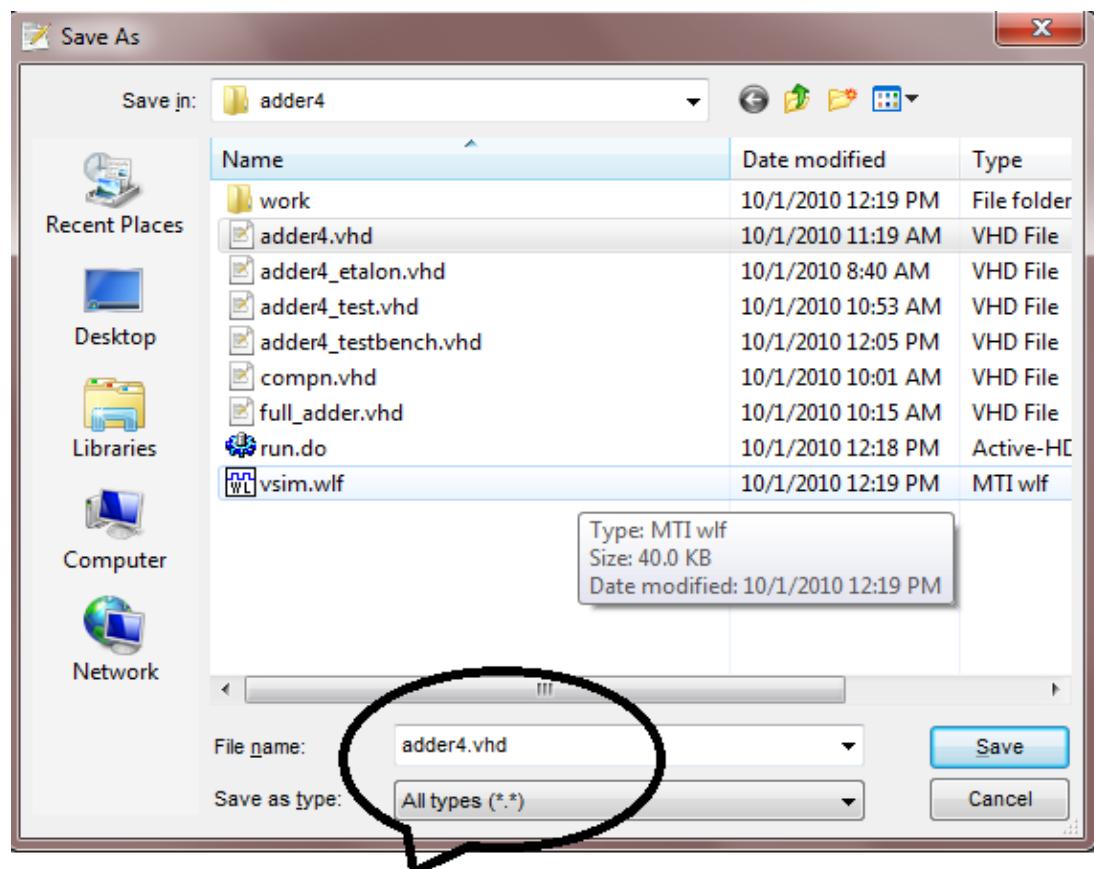


Cấu trúc của 4 bit - adder

Khối full\_adder như trên lý thuyết có thể mô tả theo các kiến trúc khác nhau, ở đây để đơn giản ta chọn kiểu mô tả theo luồng dữ liệu (dataflow)

**Bước 1:** Tạo trong thư mục D:\Student một thư mục có tên adder4. Thư mục chúng ta làm việc sẽ là D:\Student\adder4

**Bước 2:** Trong Notepad++ tạo mới một file bằng cách chọn menu File/new, soạn thảo file với nội dung sau, soạn thảo xong đó chọn File/Save, và lưu file dưới tên full\_adder.vhd trong thư mục làm việc D:\Student\adder4, lưu ý để lưu dưới dạng vhd ở ô chọn File types phải chọn là All files(\*)



```

Nội dung full_adder.vhd
----- full_adder -----
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;
-----
```

**Bước 3:** Tạo mã nguồn của bộ cộng 4-bit, lưu thành file adder4.vhd với nội dung như sau:

```

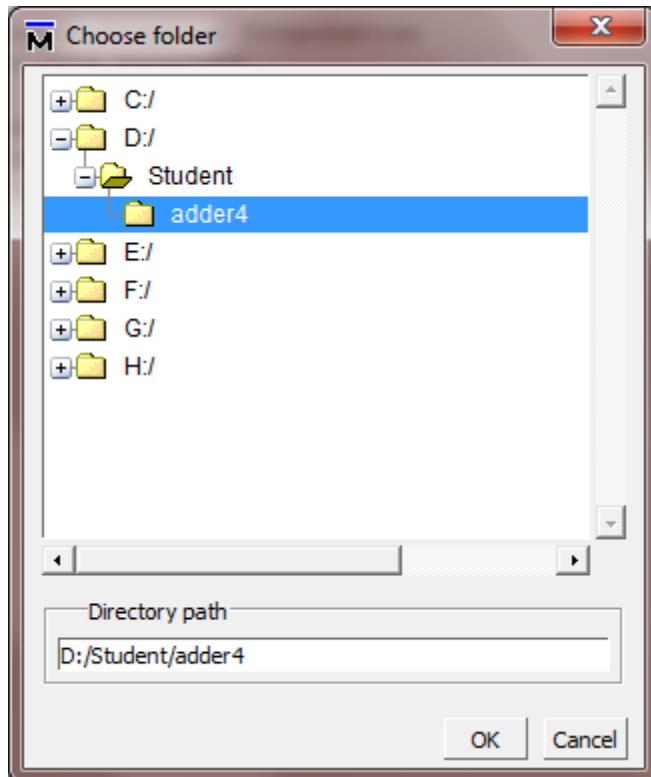
----- 4-bit adder -----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-----
entity adder4 is
port(
    A      : in  std_logic_vector(3 downto 0);
    B      : in  std_logic_vector(3 downto 0);
    CI     : in  std_logic;
    SUM    : out std_logic_vector(3 downto 0);
    CO     : out std_logic);
end adder4;
-----
architecture structure of adder4 is
signal C: std_logic_vector(2 downto 0);
-- declaration of component full_adder
component full_adder
port (
    A      : in  std_logic;
    B      : in  std_logic;
    Cin   : in  std_logic;
    S     : out std_logic;
    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;

```

#### 4. Biên dịch thiết kế.

Để tạo biên dịch thiết kế ta làm lần lượt các bước sau:

**Bước 4:** Khởi động Modelsim, tại menu File chọn Change Directory, tại menu Change directory chọn Chọn đường dẫn tới thư mục làm việc D:\Student\adder4\ chứa các nguồn vừa tạo adder4.vhd, full\_adder.vhd.



**Bước 5:** Tạo thư viện work bằng cách gõ lệnh sau vào cửa sổ Transcript của Modelsim:

```
vlib work
```

```

ModelSim SE PLUS 6.2e
File Edit View Compile Simulate Add Transcript Tools Layout Window Help
Workspace Transcript
Name
+ work
+ sv_std
+ vital2000
+ ieee
+ modelsim_lib
+ std
+ std_develop.
+ synopsys
+ verilog

# Reading C:/Modeltech_6.2e/tcl/vsim/pref.tcl
# // ModelSim SE 6.2e Nov 16 2006
# //
# // Copyright 2006 Mentor Graphics Corporation
# // All Rights Reserved.
# //
# // THIS WORK CONTAINS TRADE SECRET AND
# // PROPRIETARY INFORMATION WHICH IS THE PROPERTY
# // OF MENTOR GRAPHICS CORPORATION OR ITS LICENSED
# // AND IS SUBJECT TO LICENSE TERMS.
# //
cd D:/Student/adder4
ModelSim> vlib work

ModelSim>

```

<No Design Loaded>

**Bước 6:** Biên dịch các mã nguồn bằng cách gõ các lệnh sau vào cửa sổ Transcript

```
vcom full_adder.vhd
vcom adder4.vhd
```

```

ModelSim SE PLUS 6.2e
File Edit View Compile Simulate Add Transcript Tools Layout Window Help
Workspace Transcript
Name
+ work
+ sv_std
+ vital2000
+ ieee
+ modelsim_lib
+ std
+ std_develop.
+ synopsys
+ verilog

# -- Compiling architecture behavioral of full_adder
# -- Compiling architecture dataflow of full_adder
# -- Loading entity full_adder
# -- Compiling architecture structure of full_adder
# -- Loading entity full_adder
ModelSim> vcom adder4.vhd
# Model Technology ModelSim SE vcom 6.2e Compiler 2
# 006.11 Nov 16 2006
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package std_logic_arith
# -- Loading package std_logic_unsigned
# -- Compiling entity adder4
# -- Compiling architecture structure of adder4
# -- Loading entity full_adder

ModelSim>

```

<No Design Loaded>

Khi trình biên dịch phát hiện ra lỗi về mặt cú pháp thì nó sẽ thông báo chính xác dòng tương ứng gây ra lỗi. Nếu như mã nguồn của thiết kế không có lỗi thì biên dịch xong sẽ cho ra kết quả như hình trên.

## 5. Mô phỏng và kiểm tra thiết kế.

Kiểm tra nhanh thiết kế bằng cách đưa vào đầu vào của DUT các giá trị cố định và kiểm tra trực tiếp kết quả đầu ra. Kiểm tra nhanh cho phép phát hiện và sửa những lỗi về mặt chức năng đơn giản trước khi bước vào bước kiểm tra với số lượng lớn tổ hợp giá trị đầu vào.

**Bước 7:** Để kiểm tra nhanh bộ cộng thiết kế ở trên tạo thêm một file adder4\_test.vhd trong thư mục làm việc với nội dung như sau như sau:

```
-----adder4_test-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity adder4_test is
end adder4_test;
-----
architecture test of adder4_test is

component 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 component;
-- khai bao cac tin hieu vao ra cho DUT
signal A  : std_logic_vector(3 downto 0) := "0101";
signal B  : std_logic_vector(3 downto 0) := "1010";
signal CI : std_logic                  := '1';
-- output---
signal SUM : std_logic_vector(3 downto 0);
signal CO : std_logic;

begin
    DUT: component adder4
        port map (
            A => A, B=> B, CI => CI,
            SUM => SUM, CO =>CO
        );

```

```
end test;
```

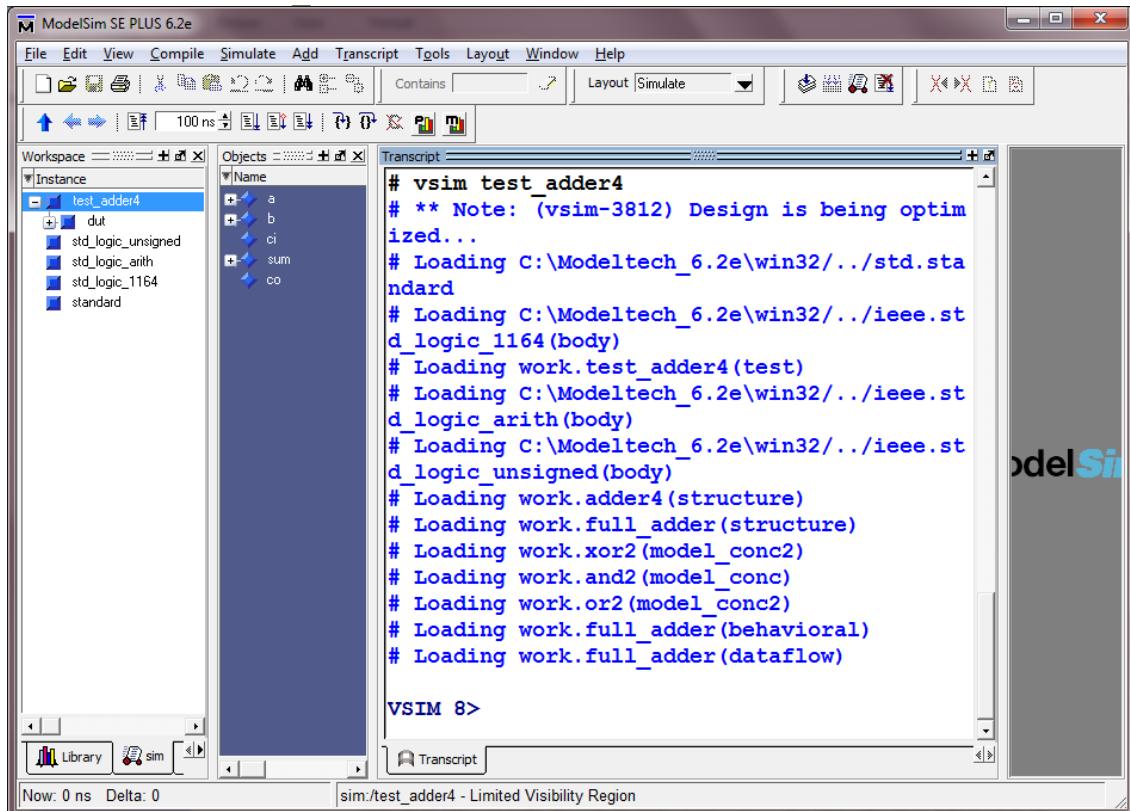
-----  
test\_adder4 là một thiết kế mà không chứa bất cứ cổng vào ra nào ở phần khai báo. Kiến trúc của nó gồm hai phần, phần khai báo tín hiệu sẽ khai báo các tín hiệu vào ra của adder4 trong đó đối với các tín hiệu đầu vào A = “0101”, B = “1010”, CI\_t = ‘1’; đối với các tín hiệu đầu ra thì để trống. Phần hai là khai báo sử dụng adder4 như một khối con có tên là dut (device under test) và gán các cổng vào ra tương ứng như trên.

**Bước 8:** Tiến hành biên dịch file adder4\_test.vhd này bằng lệnh sau trong cửa sổ transcript tương tự làm trong bước 6).

```
vcom adder4_test.vhd
```

**Bước 9: Khởi tạo mô phỏng thiết kế bằng lệnh:**

```
vsim adder4 test
```

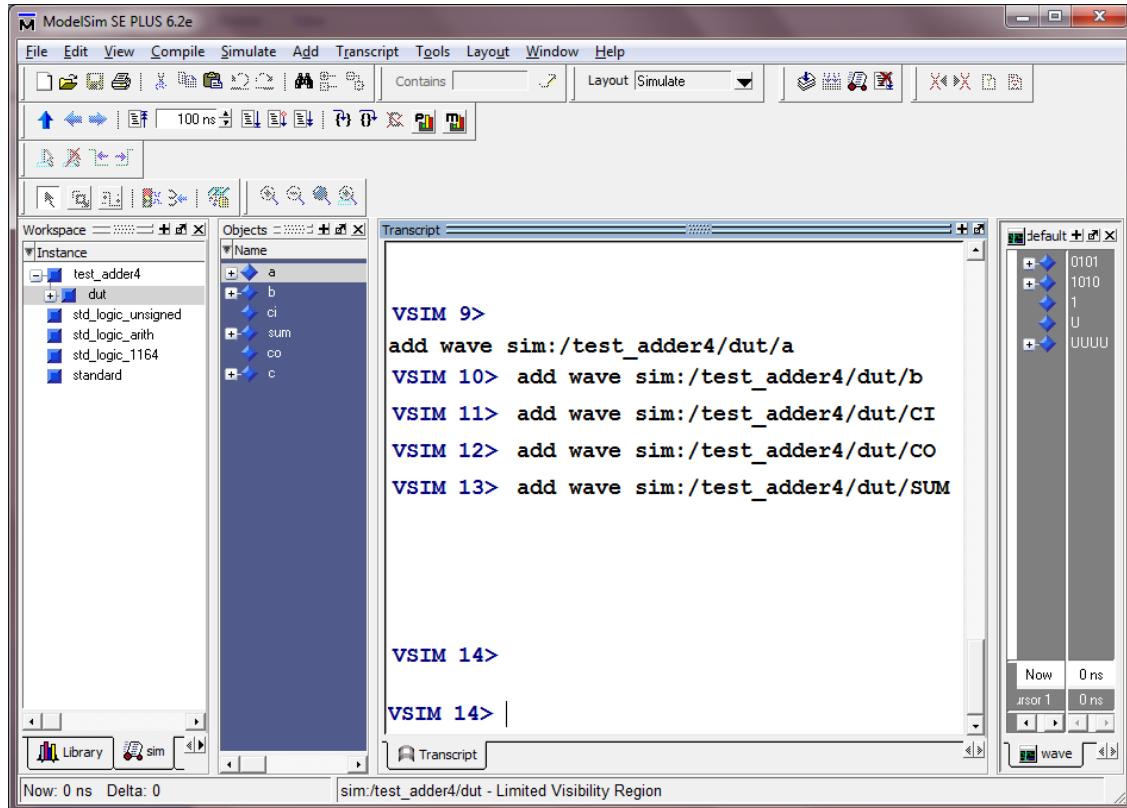


**Bước 10:** Bổ xung các tín hiệu vào cửa sổ wave form để quan sát, để thực hiện gõ các lệnh sau vào cửa sổ Transcript

```
add wave sim:/adder4_test/dut/a
add wave sim:/adder4_test/dut/b
add wave sim:/adder4_test/dut/CI
add wave sim:/adder4_test/dut/CO
```

```
add wave sim:/adder4_test/dut/SUM
```

Mỗi lệnh trên sẽ hiển thị một tín hiệu tương ứng vào giản đồ sóng, bằng cách đó ta có thể lựa chọn các tín hiệu nhất định để theo dõi. Sau khi thực hiện các bước trên thì có thể tiến hành chạy mô phỏng. Mô phỏng có thể chạy bằng nút công cụ Run trên thanh công cụ của cửa sổ giản đồ sóng:



**Bước 11:** Chạy mô phỏng và quan sát kết quả trên waveform bằng cách gõ lệnh run 100 ns vào cửa sổ Transcript sau đó mở rộng cửa sổ Waveform bên phải để quan sát.

run 1000 ns

Khi gõ lệnh này chương trình sẽ chạy mô phỏng trong 1000 ns. Kết quả ra như sau:

Messages	
+ /adder4_test/a	0101
+ /adder4_test/b	1010
+ /adder4_test/ci	1
+ /adder4_test/co	1
+ /adder4_test/sum	0000

Quan sát trên hình có và so sánh với mã nguồn của adder4\_testbench có thể thấy với  $a = "0101" = 5$ ,  $b = "1010" = 15$ ,  $CI = '1'$  thì cho ra kết quả  $sum = "0000" = 0$  và  $CO = '1'$ .

**Bước 12:** Tạo một file run.do lưu vào trong thư mục làm việc với nội dung như sau:

```
quit -sim  
vlib work  
  
vcom full_adder.vhd  
vcom adder4.vhd  
vcom adder4_test.vhd  
  
vsim adder4_test  
  
add wave sim:/adder4_test/a  
add wave sim:/adder4_test/b  
add wave sim:/adder4_test/CI  
add wave sim:/adder4_test/CO  
add wave sim:/adder4_test/SUM  
  
run 1000 ns
```

Dòng thứ nhất để kết thúc bất kỳ mô phỏng nào đang thực thi nếu nó tồn tại, dòng thứ hai tạo thư viện work nếu nó chưa tồn tại, tiếp đến là các lệnh vcom để biên dịch các file mã nguồn từ thấp đến cao. Lệnh vsim để tiến hành mô phỏng, sau đó là các lệnh bổ xung tín hiệu cần theo dõi vào giản đồ sóng. Lệnh cuối cùng là lệnh run dùng để chạy mô phỏng.

**Bước 13:** Trong cửa sổ transcript của modelsim để biên dịch và chạy lại mô phỏng ta chỉ như sau.

```
do run.do
```

## 6. Bài tập sau thực hành

- Dựa trên quy trình đã học, xây dựng bộ cộng 8 bit và thực hiện mô phỏng kiểm tra bộ cộng đó trên modelsim theo các bước ở trên.
- Viết mô tả Full adder bằng kiến trúc kiểu hành vi và thực hiện mô phỏng theo các bước ở trên
- Xây dựng bộ cộng trên cơ sở Full adder theo kiểu hành vi.

**Bài 2: Xây dựng bộ cộng trừ trên cơ sở khôi cộng bằng toán tử**

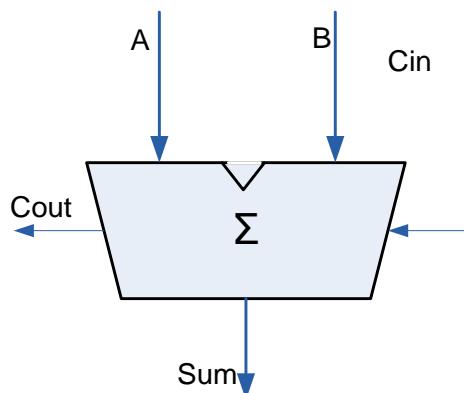
## 1. Mục đích

Qua ví dụ xây dựng khối cộng trừ sử dụng toán tử +, trong bài thực hành này sinh viên tự viết mô tả cho các khối thiết kế, qua đó ôn tập lại các cấu trúc lệnh VHDL, cách sử dụng tham số tĩnh, cách cài đặt khối con, cách thức kiểm tra thiết kế.

Yêu cầu với sinh viên có kiến thức cơ sở về VHDL, sử dụng thành thạo Modelsim.

## 2. Khối cộng đơn giản

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:



**Bước 1:** Viết mô tả (adder.vhd) cho khối cộng sử dụng trực tiếp toán tử cộng, đầu vào A, B và đầu ra Sum có kiểu STD\_LOGIC\_VECTOR 32 bit, Cout và Cin có kiểu STD\_LOGIC.

*Hướng dẫn:* Khi đó buộc phải khai báo thư viện như sau:

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

Khối cộng đơn giản có thể viết bằng cú pháp

```
SUM <= a + b;
```

Lệnh này đặt trực tiếp mô tả kiến trúc (dạng mô tả Dataflow). Tuy vậy để có được bit nhớ Cout thì cần bổ xung thêm 1 bit ‘0’ vào các giá trị A, B như sau:

Khai báo các tín hiệu bổ xung trong phần khai báo kiến trúc:

```
Signal A1 : std_logic_vector(32 downto 0);
Signal B1 : std_logic_vector(32 downto 0);
```

```
Signal Sum1 : std_logic_vector(32 downto 0);
```

Và thực hiện trong phần mô tả kiến trúc như sau:

```
A1 <= '0' & A;
```

```
B1 <= '0' & B;
```

```
Sum1 <= A1 + B1 + Cin;
```

Khi đó giá trị Cout là bit cao nhất của Sum1

```
Cout <= Sum1(32);
```

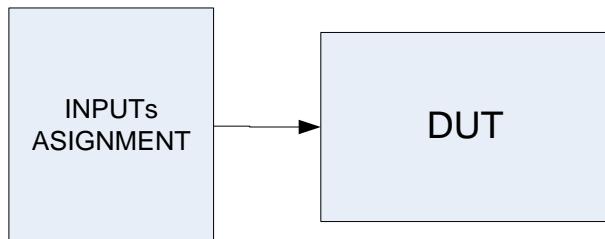
Còn Sum là 32 bit thấp:

```
Sum <= Sum1(31 down to 0);
```

**Bước 2:** Viết khôi kiểm tra cho bộ cộng vừa viết bằng VHDL, thực hiện mô phỏng kiểm tra. Kết quả mô phỏng phải thể hiện được như sau:

◆ /test_add_32/add/cin	0							
+◆ /test_add_32/add/a	17	4369		273		529		17
+◆ /test_add_32/add/b	4	1		2		3		4
+◆ /test_add_32/add/sum	21	4370		275		532		21
◆ /test_add_32/add/cout	0							

Hướng dẫn: Sơ đồ kiểm tra nhanh như sau:



Khôi kiểm tra nhanh là khôi **không có đầu ra đầu vào**, nhiệm vụ chính là đặt các giá trị cho các công đầu vào của khôi kiểm tra.

Khôi kiểm tra được khai báo thực thể là

```
entity test_adder4_gen is  
end test_adder4_gen;
```

Để cài đặt khôi thiết kế con của khôi adder cần khai báo khôi con (khai báo component) như sau trong phần khái báo kiến trúc:

```
component adder is  
port(  
    A      : in std_logic_vector(31 downto 0);  
    B      : in std_logic_vector(31 downto 0);  
    Cin   : in std_logic;  
    SUM   : out std_logic_vector(31 downto 0);  
    Cout  : out std_logic  
);
```

Cài đặt khôi con trực tiếp trong phần mô tả kiến trúc (khối begin/end chính):

```

DUT: component adder
      port map ( A, B, Cin, Sum, Cout);

```

**Bước 3:** Xem lại phần lý thuyết về khai báo tham số tĩnh, bổ xung vào bộ cộng tham số tĩnh N là số bit.

Hướng dẫn: Tham số tĩnh khai báo trong khai báo thực thể của thiết kế. Ví dụ:

```

entity adder is
generic (N : natural :=32);
port (...)
```

Sử dụng giá trị tham số tĩnh này trong thiết kế để tùy biến số Bit của các hạng tử và kết quả như sau:

Với các cổng

```

A    : in  std_logic_vector(N-1 downto 0);
Sum : out std_logic_vector(N-1 downto 0);
```

Với các tín hiệu:

```
signal A1 : std_logic_vector(N downto 0);
```

**Bước 4:** Thực hiện thay đổi khôi kiểm tra để kiểm tra cho bộ cộng dùng tham số tĩnh N, thay đổi giá trị N = 16 và thực hiện kiểm tra lại như bước 2.

Hướng dẫn: Để cài đặt khôi con có tham số tĩnh thì phải cài đặt tham số tĩnh tương tự như cài đặt các tín hiệu cho cổng, xem ví dụ sau đây:

```

adder32: component adder
generic map (32)
port map (A, B, Cin, Sum, Cout);
*Lưu ý là sau generic map () không có dấu ; hay , mà để trống.
```

### 3. Khối trừ

**Bước 5:** Nghiên cứu cấu trúc khối trừ như sau:

Vì các số có dấu trên máy tính được biểu diễn dưới dạng số bù 2 (2'complement), do đó để thực hiện phép trừ A-B thì tương đương với thực hiện A + bù2(B)

Xét ví dụ A = 10 = 1010, B = 5 = 0101 biểu diễn dưới dạng số có dấu 5-bit ta phải thêm bit dấu bằng 0 vào trước.

$$\begin{aligned} A &= 01010, \text{ Bù2}(A) = \text{not } (A) + 1 = 10101 + 1 = 10110 \\ B &= 00101, \text{ Bù2}(B) = \text{not } (B) + 1 = 11010 + 1 = 11011 \end{aligned}$$

Tính A - B:

$$\begin{array}{r} A \quad 01010 \quad 01010 \\ - = - \quad = + \\ B \quad 00101 \quad 11011 \end{array}$$

1 00101

Loại bỏ bit nhớ ở kết quả cuối cùng ta được  $A - B = 00101 = 5$ .

Tính  $B - A$ :

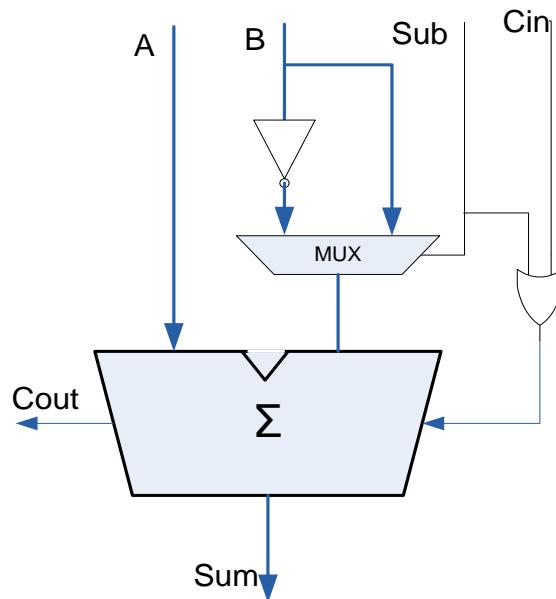
$$\begin{array}{r} B \quad \quad 00101 \quad \quad 00101 \\ - = - \quad \quad = + \\ A \quad \quad 01010 \quad \quad 10110 \\ \hline 0 \quad \quad 11011 \end{array}$$

Loại bỏ bit nhớ ta được  $B - A = 11101$ , đây là số âm, muốn tính giá trị tuyệt đối để kiểm tra lại lấy bù 2 của 11101

$$\text{Bù 2 } (11101) = 00100 + 1 = 00101 = 5$$

vậy  $B - A = -5$

Dựa trên tính chất trên của số bù hai ta chỉ cần thay đổi một chút trong cấu trúc của bộ cộng để nó có khả năng thực hiện cả phép cộng lẫn phép trừ mà không phải thay đổi nhiều về cấu trúc phần cứng. Tại đầu vào có thêm tín hiệu SUB, tín hiệu này quyết định sẽ thực hiện phép cộng hay phép trừ. Khi  $\text{SUB} = 1$  để lấy bù 2 của B sẽ lấy đảo B và cho giá trị đầu vào Cin = 1, để hiện thực trên mạch cấu trúc bộ cộng được bổ xung một khối MUX trước cổng B, khối này có hai đầu vào là B và not B, nếu  $\text{SUB} = 0$  thì B được chọn, nếu  $\text{SUB} = 1$  thì not B được chọn. Đầu vào Cin được OR với SUB trước khi vào bộ cộng.



**Bước 6:** Viết mô tả cho khối MUX 2 đầu vào trên hình vẽ, số bit của các dữ liệu đầu cài đặt bằng tham số tĩnh N như trường hợp của khối cộng, viết khái niệm hoạt động của khái niệm MUX.

Hướng dẫn:

Để viết khối MUX có thể dùng cấu trúc process có danh sách sensitive list là các tín hiệu đầu vào và cấu trúc IF THEN / END IF;

```
process (Sel, data_in1, data_in2)
if Sel = '1' then
    data_out <= data_in1;
else
    data_out <= data_in1;
end if;
```

...  
Để kiểm tra khối MUX có thể thực hiện tương tự như kiểm tra khối cộng.

**Bước 7:** Viết mô tả bộ cộng trừ theo sơ đồ trên, khối trên thực hiện phép cộng nếu như tín hiệu Sub = 0 và thực hiện trừ nếu như Sub = 0.

Hướng dẫn: Khối trừ được ghép bởi khối cộng, khối MUX, và cổng OR hai đầu vào, với cổng OR hai đầu vào dùng trực tiếp toán tử logic OR, còn hai khối cộng và MUX đã mô tả như trên. Khối cộng/trừ viết theo kiểu cấu trúc.

**Bước 8:** Thực hiện mô phỏng kiểm tra khối cộng trừ, tín hiệu SUB cần được thay đổi theo thời gian để quan sát kết quả với các phép toán khác nhau.

Hướng dẫn: Để thay đổi tín hiệu SUB (hoặc bất kỳ tín hiệu nào theo thời gian dùng cấu trúc WAIT FOR của VHDL:

```
...
wait for 20 ns;
SUB <= '0';
wait for 100 ns;
SUB <= '0';
wait for 100 ns;
SUB <= '1';
...

```

#### 4. Bài tập sau thực hành

- Nắm lý thuyết cấu trúc của khối cộng/trừ.
- Thay đổi mô tả của bộ cộng theo sơ đồ trên bằng bộ cộng nối tiếp đã mô tả ở bài thí nghiệm thứ nhất.
  - Xây dựng khối cộng NBCD trên dùng toán tử cộng.
  - Xây dựng khối trừ NBCD dùng toán tử cộng.

### **Bài 3: Khôi dịch và thanh ghi dịch**

## 1. Mục đích

Viết mô tả và kiểm tra cho khối dịch bằng các phương pháp khác nhau: bằng toán tử dịch, bằng sơ đồ thuật toán dịch. Các tạo nhiều giá trị kiểm tra bằng mã VHDL, cách viết và sử dụng thực thể có nhiều kiến trúc.

Yêu cầu với sinh viên Có kiến thức cơ sở về VHDL, sử dụng thành thạo Modelsim.

## 2. Khối dịch dùng toán tử dịch

Các phép toán quan hệ gồm **sll**, **srl**, **sla**, **sra**, **rol**, **ror** được hỗ trợ trong thư viện **ieee.numeric\_bit**, và **ieee.numeric\_std**. Cú pháp của các lệnh dịch có hai tham số là **sho** (shift operand) và **shv** (shift value), ví dụ cú pháp của sll như sau

sha **sll** shv;

Toán tử	Phép toán	Kiểu của sho	Kiểu của shv	Kiểu kết quả
sll	Dịch trái logic	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
srl	Dịch phải logic	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
sla	Dịch trái số học	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
sra	Dịch phải số học	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
rol	Dịch vòng tròn sang trái	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
ror	Dịch vòng tròn phải	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho

Đối với dịch logic thì tại các vị trí bị trống sẽ được điền vào các giá trị ‘0’ còn dịch số học thì các các vị trí trống được thay thế bằng bit có trọng số cao nhất MSB nếu dịch phải, và thay bằng bit có trọng số thấp nhất LSB nếu dịch trái. Đối với dịch vòng thì các vị trí khuyết đi sẽ được điền bằng các bit dịch ra ngoài giới hạn của mảng. Quan sát ví dụ dưới đây

# sho = 11000110

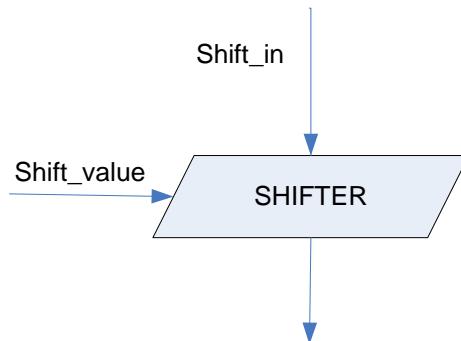
```

# sho sll 2 = 00011000
# sho srl 2 = 00110001
# sho sla 2 = 00011000
# sho sra 2 = 11110001
# sho rol 2 = 00011011
# sho ror 2 = 10110001

```

**Bước 1:** Viết cho khối dịch logic phải.

*Hướng dẫn:* (file mô tả có tên shifter.vhd) Sơ đồ khối của khối dịch như sau:



Toán tử dịch có cấu trúc:

```
Shift_out = Shift_in sll shift_value;
```

Trong đó **bắt buộc** shift\_in và shift\_out có kiểu BIT\_VECTOR, còn shift\_value có kiểu INTEGER.

Trên thực tế các đầu vào này cần được khai báo dạng STD\_LOGIC\_VECTOR để tương thích với các khối khác do vậy có các hàm chuyển đổi sau:

```

TO_BITVECTOR(shift_in) - chuyển sang kiểu BIT_VECTOR;
CONV_INTEGER('0' & shift_value) - chuyển sang kiểu
INTEGER;
TO_STDLOGICVECTOR(sho) - chuyển ngược lại
STD_LOGIC_VECTOR;

```

Chú ý rằng khi chuyển sang kiểu INTEGER cần thêm bit 0 đằng trước để lấy đúng giá trị biểu diễn không dấu của giá trị dịch.

Khi đó buộc phải khai báo thư viện như sau:

```

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;
USE ieee.Numeric_BIT.all;

```

Với các tín hiệu đầu vào là STD\_LOGIC\_VECTOR ta phải khai báo các tín hiệu trung gian kiểu tương thích với lệnh dịch ở phần khai báo kiến trúc:

```
signal shi: bit_vector(31 downto 0);
signal sho: bit_vector(31 downto 0);
signal sa : integer;
```

Dựa trên hướng dẫn trên viết khối dịch dùng toán tử cho phép dịch phải cho chuỗi 32-bit đầu vào.

**Bước 2:** Viết khối kiểm tra cho khối dịch ở bước 1.

*Hướng dẫn:* Ở bài thực hành trước chúng ta quen với cách kiểm tra nhanh với 1 tổ hợp đầu vào. Ở bài này sẽ làm quen với phương pháp kiểm tra nhanh cho một số tổ hợp giá trị đầu vào. Để làm được như cần bổ xung vào phần mô tả kiến trúc của khối kiểm tra đoạn mã tạo các dữ liệu đầu vào bằng lệnh wait for. Ví dụ như sau:

```
create_data: process
begin
    shift_in <= x"19107111" after 50 ns;
    shift_value <= "00011";
    wait for 150 ns;
    shift_in <= x"19107000" after 50 ns;
    shift_value <= "00111";
    wait for 170 ns;
    shift_in <= x"1abc7111" after 50 ns;
    shift_value <= "00011";
    wait;
end process create_data;
```

Đoạn mã trên gán các giá trị khác nhau cho các đầu vào khác nhau, tương ứng trên giàn đồ sóng sẽ quan sát được sự thay đổi này.

**Bước 3:** Viết mô tả khối dịch có kiến trúc mà không sử dụng thuật toán dịch.

*Hướng dẫn:* Một thực thể có thể có nhiều kiến trúc và các kiến trúc có thể được mô tả độc lập với nhau trong 1 mô tả VHDL. Cụ thể ở bước này cần bổ xung vào file nguồn tạo ở **Bước 1** mô tả kiến trúc thứ hai bắt đầu bằng architecture (phần in đậm cuối file)

```
-- SHIFTER -----
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;
USE ieee.Numeric_BIT.all;
-----
```

```

entity 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 shifter_32;

-- khai dich su dung toan tu dich
architecture behavioral of shifter_32 is

begin

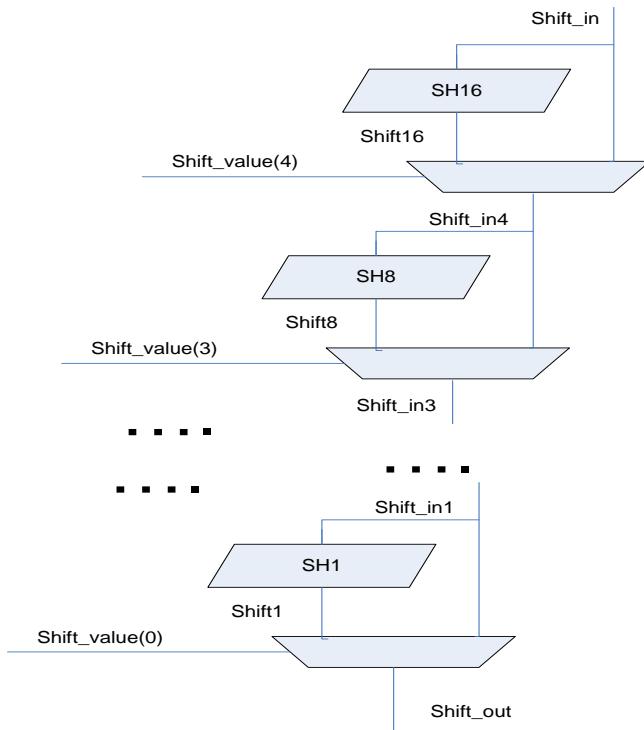
    ...
--Mo ta kien truc 1
end behavioral;

-- khai dich su dung thuat toan dich don gian
architecture rtl of shifter_32 is
begin

    --mota kien truc 2
end rtl;

```

Để hiểu về lý thuyết xây dựng khối dịch không sử dụng toán tử xem mục 1.7 chương III. Sơ đồ khối dịch thể hiện ở hình sau



## Sơ đồ thuật toán khối dịch đơn giản

Để hiện thực sơ đồ trên bằng VHDL ta sử dụng cấu trúc câu lệnh song song WITH/SELECT. Trước hết cần khai báo trong khai báo kiến trúc các tín hiệu trung gian shift\_in4, shift\_in3,...

```
signal shift_in4 : std_logic_vector(31 downto 0);
signal shift_in3 : std_logic_vector(31 downto 0);
signal shift_in2 : std_logic_vector(31 downto 0);
signal shift_in1 : std_logic_vector(31 downto 0);
signal shift_in0 : std_logic_vector(31 downto 0);
```

Sau đó cho mỗi bước dịch trên ta sử dụng câu lệnh WITH/SELECT

```
with shift_value(4) select
shift_in4 <= x"0000" & shift_in(31 downto 16) when '1',
shift_in when others;
```

**Bước 4:** Viết Kiểm tra đồng thời cho hai kiến trúc của khối dịch đã mô tả ở **bước 3.**

*Hướng dẫn:*

Vì ở trên khối dịch có hai kiến trúc nên để kiểm tra khối dịch ta sẽ cài đặt cả hai kiến trúc này trong khối kiểm tra (trong phần mô tả kiến trúc), lưu ý rằng các đầu vào của các khối dịch với kiến trúc khác nhau có thể dùng chung nhưng đầu ra **bắt buộc phải khác nhau**. Ở đây ta khai báo hai đầu ra là shift\_out0, shift\_out1.

```
sh320: component shifter_32 port map (shift_in,
shift_value, shift_out0);
sh321: component shifter_32 port map (shift_in,
shift_value, shift_out1);
```

Tuy vậy để xác định kiến trúc nào sẽ được mô phỏng ta phải có mã quy định trước trong phần khai báo kiến trúc như sau:

```
for sh320: shifter_32 use entity
work.shifter_32(behavioral);
for sh321: shifter_32 use entity work.shifter_32(rtl);
```

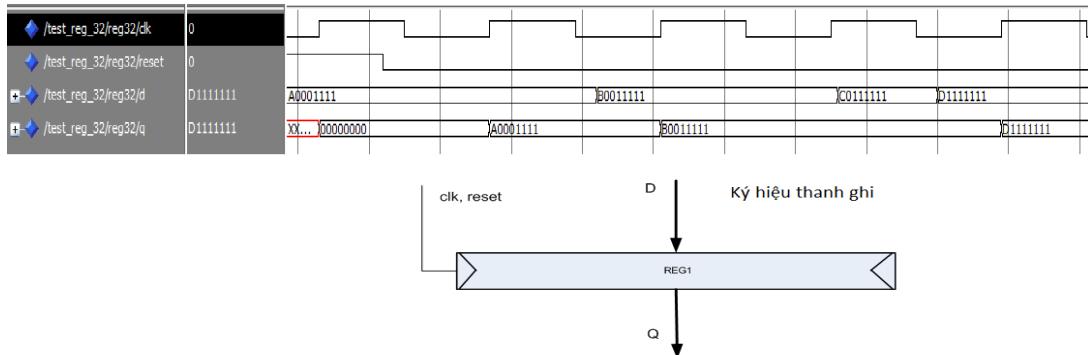
Với quy định trên thì khối con sh320 dùng kiến trúc behavioral còn khối sh321 dùng kiến trúc rtl.

Mô phỏng kiểm tra sẽ thu được kết quả là các kiến trúc khác nhau thực hiện chức năng giống nhau, ví dụ kết quả như hình sau:

Messages						
[+]	/shifter_test/shift_in	1AB12111	xxxxxxxx	19107111	19107100	1AB12111
[+]	/shifter_test/shift_out0	03562422	00000000	19107111	003220E2	03220E20 03562422
[+]	/shifter_test/shift_out1	03562422	xxxxxxxx	19107111	003220E2	03220E20 03562422
[+]	/shifter_test/shift_value	3	X		7	3

### 3. Thanh ghi

**Bước 5:** Viết mô tả cho thanh ghi N-bit có cấu trúc và giản đồ sóng như sau:



Hướng dẫn: Thanh ghi N-bit cần khai báo N là tham biến tĩnh generic.

Tín hiệu RESET ở đây là tín hiệu RESET không đồng bộ, còn tín hiệu CLK kích hoạt tại sườn dương. Thanh ghi viết buộc phải dùng cấu trúc IF/THEN trong khối lệnh tuần tự (khối PROCESS).

```
if RESET = '1' then
    Q <= (others => '0');
elsif CLK = '1' and CLK'event then
    Q <= D;
end if;
```

Ý nghĩa của lệnh `Q <= (others => '0');` là gán tất cả các bit của Q giá trị bằng 0.

**Bước 6:** Mô phỏng kiểm tra thanh ghi, nhận xét về sự thay đổi dữ liệu trên thanh ghi và so sánh với các khái niệm mô tả trước đó.

Hướng dẫn: Khác với các khái niệm mô tả thanh ghi là mạch dãy do đó khi kiểm tra cần tạo đúng dạng cho xung nhịp CLK.

Để tạo xung nhịp CLK dùng một khái niệm process không có danh sách sensitive list, trong đó ta dùng một lệnh wait for để tránh cảnh báo khi biên dịch về việc thiếu danh sách sensitive list. Tín hiệu CLK sẽ được gán bằng đảo của nó sau mỗi nửa chu kỳ. Chú ý rằng con số chỉ thời gian ở đây có thể bất kỳ vì chỉ sử dụng cho mô phỏng.

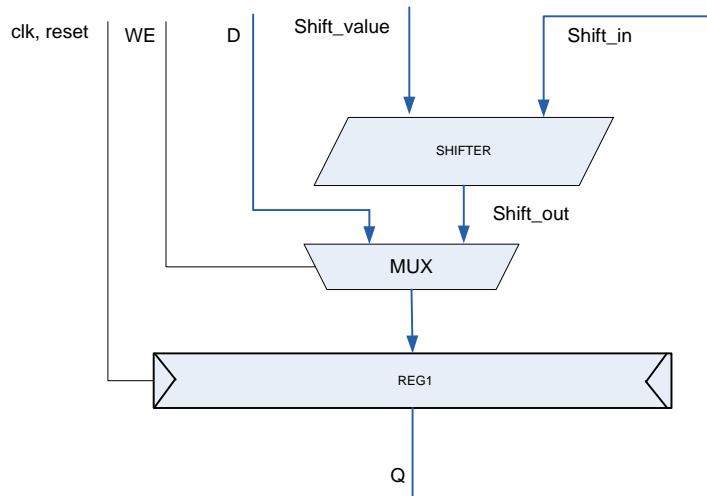
```
--create clock
create_clock: process
begin
    wait for 15 ns;
    CLK <= not CLK after 50 ns;
end process create_clock;
```

Tín hiệu RESET và dữ liệu đầu vào D được tạo bằng lệnh Wait for trong một khối khác, ví dụ như sau:

```
reset <= '0'; enable <= '1';
wait for 10 ns;
reset <= '1';
D <= x"923856c8";
wait for 100 ns;
reset <= '0';
D <= x"10245608";
wait for 250 ns;
D <= x"a23400a8";
wait for 200 ns;
D <= x"c00456c8";
wait;
```

### Bước 7: Viết mô tả cho thanh ghi dịch phải số học.

*Hướng dẫn:* Kết hợp khối dịch phải số học và thanh ghi ta được cấu trúc của thanh ghi dịch phải số học như ở hình sau:

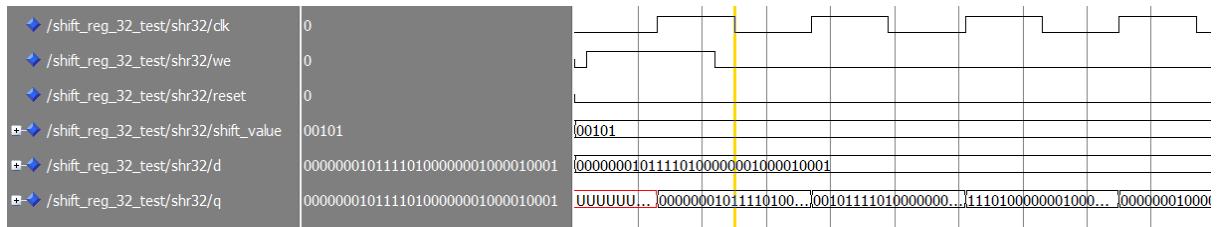


Thanh ghi có thể làm việc ở hai chế độ, chế độ thứ nhất dữ liệu đầu vào được lấy từ đầu vào D nếu tín hiệu WE = 1, chế độ thứ hai là chế độ dịch, khi đó dữ liệu đầu vào của thanh ghi lấy từ khối dịch khi WE = 0, đầu ra của thanh ghi

được gán băng đầu vào của khối dịch. Ở chế độ này dữ liệu sẽ bị dịch liên tục mỗi xung nhịp một lần.

Để viết thanh ghi dịch thì ta viết mô tả của khối dịch, của khối MUX và thanh ghi riêng sau đó ghép lại. Đối với khối dịch sử dụng kiến trúc dịch dùng thuật toán.

Ví dụ giản đồ sóng sau là của thanh ghi dịch trái logic:



#### 4. Bài tập sau thực hành

- Thiết kế các khối dịch không sử dụng toán tử với 6 phép dịch khác nhau
  - Thiết kế thanh ghi dịch cho các khối dịch trên
- Thiết kế khối dịch đa năng có thể thực hiện tất cả các phép dịch trên. Phép dịch được thực hiện quy định bởi tín hiệu điều khiển đầu vào.

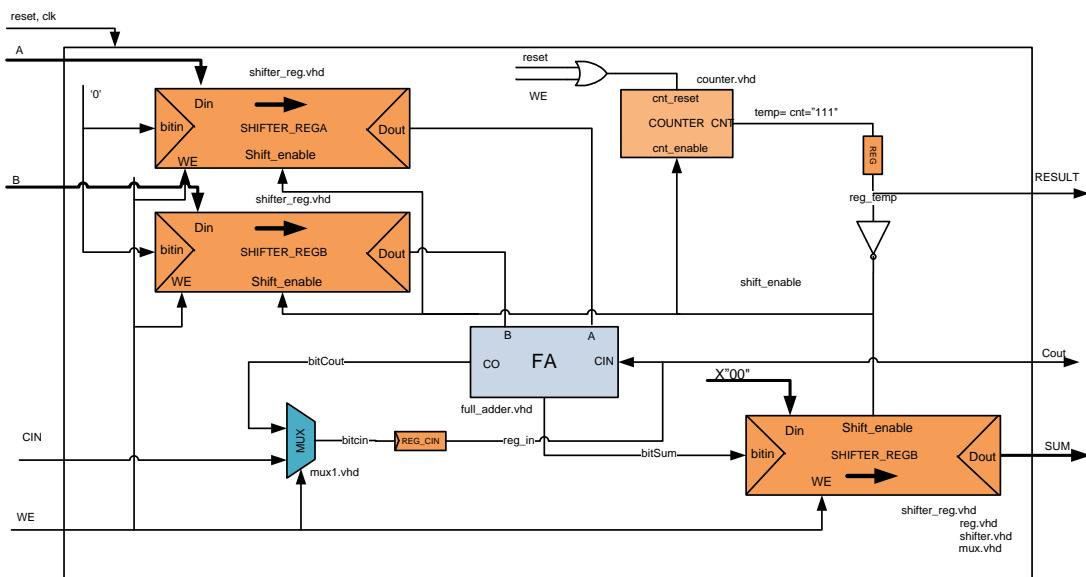
## **Bài 4: Bộ cộng bit nối tiếp dùng 1 FA (serial-bit adder)**

## 1. Mục đích

Viết mô tả và kiểm tra cho khối mạch dãy phức tạp là khối cộng 8 bit dùng một duy nhất 1 full\_adder. Khối mạch cấu trúc từ bộ đếm, thanh ghi dịch một bit nối tiếp và một full\_adder. Kỹ năng chính cần thực hiện ở bài này là cách thức ghép nối các khối đơn giản để tạo thành khối mạch phức tạp và kiểm tra hoạt động của khối mạch dãy.

Yêu cầu sinh viên có kiến thức cơ sở về VHDL và sử dụng thành thạo Modelsim.

## 2. Khối cộng bit nối tiếp (Serial – bit adder)



Sơ đồ tổng quát của khối cộng bit nối tiếp thể hiện trên hình vẽ. Phép cộng được thực hiện bằng cách sử dụng các thanh ghi dịch ở đầu vào để đẩy dần các bit  $A_i$ ,  $B_i$  tương ứng vào một bộ Full\_adder và thực hiện cộng từng bit, kết quả từ full\_adder được đẩy dần ra 1 thanh ghi lưu kết quả. Bộ cộng  $N$  bit sau  $N$  xung nhịp sẽ cho ra kết quả.

**Bước 1:** Chuẩn bị các mã nguồn có sẵn:

Trong sơ đồ trên cho phép sử dụng các mã nguồn có sẵn bao gồm:

1-Khối full\_adder được mô tả trong tệp full\_adder.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.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;
-----
1-Khối chọn kênh MUX được mô tả trong tệp mux.vhd
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity mux is
port(
    Sel      : in  std_logic;
    Din1     : in  std_logic_vector(7 downto 0);
    Din2     : in  std_logic_vector(7 downto 0);
    Dout     : out std_logic_vector(7 downto 0)
);
end mux;
-----
architecture rtl of mux is
begin
    selm: process (Sel, Din1, Din2)
    begin
        if Sel = '1' then
            Dout <= Din1;
        else
            Dout <= Din2;
        end if;
    end process selm;
end rtl;
-----

```

1-Khối shifter được mô tả trong tệp shifter.vhd

```

-----
library ieee;
use IEEE.STD_LOGIC_1164.ALL;
-----
```

```

entity shifter is
    port(
        bitin      : in  std_logic;
        shift_enable : in  std_logic;
        shift_in     : in  std_logic_vector(7 downto 0);
        shift_out    : out std_logic_vector(7 downto 0)
    );
end entity;
-----
architecture rtl of shifter is

begin
    shifting: process (bitin, shift_enable, shift_in)
    begin
        if shift_enable = '1' then
            shift_out <= bitin & shift_in(7 downto 1);
        end if;
    end process shifting;

end architecture;
-----
```

Khối này thực hiện dịch sang bên phải 1 bit và vị trí bít trống bị dịch đi sẽ được lưu giá trị ở đầu vào bitin

### 1-Khối thanh ghi được mô tả trong tệp reg.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----
entity reg is
port(
    D      : in  std_logic_vector(7 downto 0);
    Q      : out std_logic_vector(7 downto 0);
    CLK    : in  std_logic;
    RESET : in  std_logic
);
end reg;
-----
architecture behavioral of reg is
begin
    reg_p: process (CLK, RESET)
    begin
        if RESET = '1' then
            Q <= (others => '0');
        elsif CLK = '1' and CLK'event then
            Q <= D;
```

```

        end if;
    end process reg_p;
end behavioral;
-----

```

Sinh viên có thể sao chép hoặc tạo các mã nguồn có sẵn trên vào trong một thư mục trong D:/student/name/ để bắt đầu làm việc.

**Bước 2:** Biên dịch toàn bộ các file đã tạo ở bước 1.

*Hướng dẫn:* Vì đây là một bài có nhiều mã nguồn nên ở bước này ta sẽ tạo một file script có tên run.do để phục vụ biên dịch mô phỏng cho cả quá trình. Nội dung file ở bước này tạm thời như sau, ở các bước sau ta sẽ bổ sung thêm các lệnh vào script này.

```

quit -sim
vlib work
vcom counter.vhd
vcom full_adder.vhd
vcom mux.vhd
vcom reg.vhd
vcom shifter.vhd

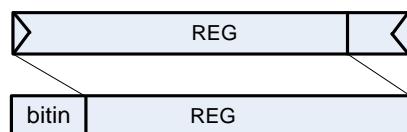
```

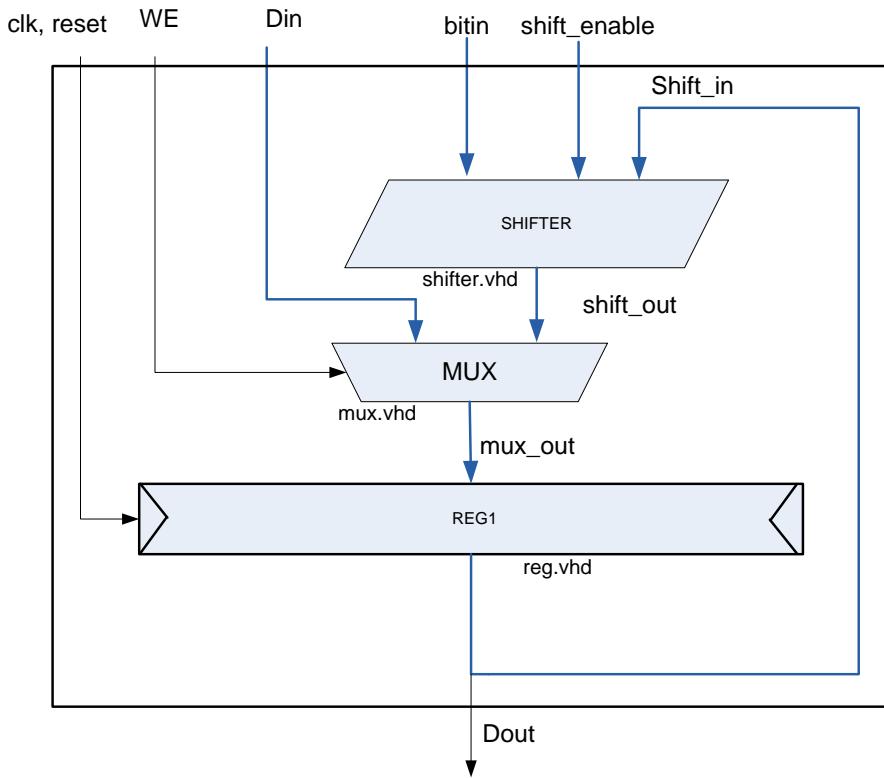
Sau khi tạo xong file này lưu vào trong thư mục làm việc chứa mã nguồn và thực hiện chạy script cho biên dịch bằng lệnh

```
do run.do
```

**Bước 3:** Viết mô tả khối thanh ghi dịch dựa trên các khối có sẵn.

*Hướng dẫn:* Thanh ghi dịch này khác với thanh ghi dịch ở bài thí nghiệm trước. phép dịch luôn dịch sang bên phải 1 bit và bit trống được điền giá trị từ bên ngoài vào từ cổng bitin. Cổng shift\_enable là cổng cho phép dịch hoặc không dịch.





Sơ đồ của khối dịch như ở hình trên, mã nguồn của thanh ghi dịch nối tiếp được liệt kê ở dưới đây. Đây là một ví dụ về cách ghép nối các khối con để tạo thành thiết kế lớn hơn. Các bước cài đặt khối con làm tương tự như khi viết khối kiểm tra nhanh (cài đặt DUT). Trong trường hợp này khối của chúng ta có nhiều khối con hơn.

Đầu tiên trong phần khai báo kiến trúc khai báo các component sẽ sử dụng trong hình ví dụ cho khối reg:

```
component reg is
    port(
        D      : in  std_logic_vector(7 downto 0);
        Q      : out std_logic_vector(7 downto 0);
        CLK   : in  std_logic;
        RESET : in  std_logic
    );
end component;
```

Sau đó cũng trong phần khai báo kiến trúc cần khai báo tất cả các tín hiệu bên trong, tín hiệu bên trong là tín hiệu kết nối giữa các khối của thiết kế, ví dụ:

```
signal shift_out: std_logic_vector(7 downto 0);
signal reg_in: std_logic_vector(7 downto 0);
```

Cuối cùng là cài đặt các khối con tương ứng với các tín hiệu trên hình, phần này viết trực tiếp trong phần mô tả kiến trúc, ví dụ:

```

sh1 : component shifter
    port map (bitin => bitin,
shift_enable => shift_enable, shift_in => shift_in,
                shift_out => shift_out);
mux1: component mux
    port map (Sel => WE, Din1 => Din,
Din2 => shift_out, Dout => reg_in);

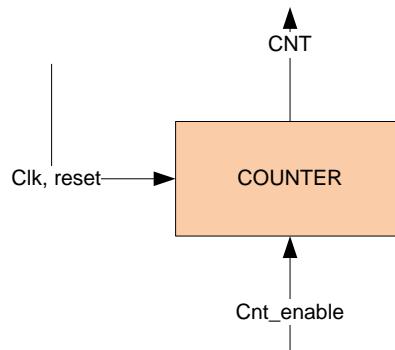
```

Trong ví dụ trên sh1, mux1 là các tên gọi của các khối cài đặt và có thể bất kỳ, tránh trùng với các tên khác đã dùng.

#### Bước 4: Viết mô tả cho bộ đếm.

Hướng dẫn: Mục đích của bộ đếm ở đây là để đếm số lần dịch, vì có 8 bit cộng nên ta sẽ viết một bộ đếm đến 8.

Các cổng của bộ đếm như ở hình vẽ sau:



Trong đó cổng cnt\_enable là cổng cho phép đếm, khi cnt\_enable = 1 thì bộ đếm làm việc bình thường, khi cnt\_enable = 0 thì bộ đếm dừng lại và giữ nguyên giá trị đếm. Cổng clk chính là cổng xung đếm, cổng reset là cổng không đồng bộ, cổng cnt là giá trị hiện tại của bộ đếm.

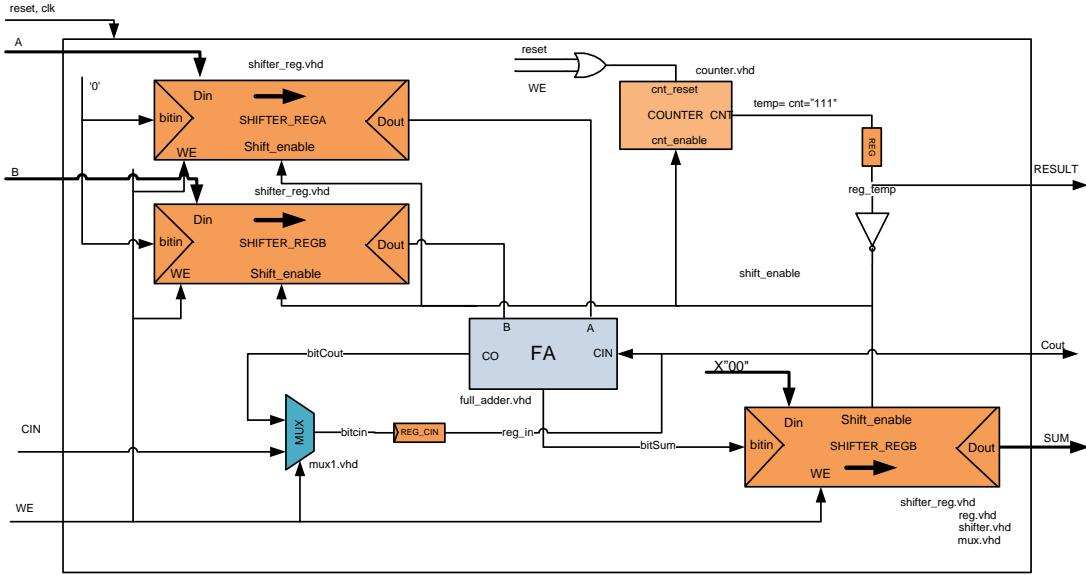
Bản chất của bộ đếm là một bộ cộng tích lũy đặc biệt với giá trị tích lũy bằng 1, thường được mô tả như sau:

```

if reset = '1' then
    temp_cnt <= (others =>'0');
elsif rising_edge(clk) then
    if counter_enable = '1' then
        temp_cnt <= temp_cnt + 1;
    end if;
end if;

```

#### Bước 5: Mô tả khối cộng nối tiếp.



Các thành phần của khối cộng đã đầy đủ bao gồm thanh ghi dịch nối tiếp (shifter\_reg.vhd), khối chọn kênh 1 bit (mux1.vhd), khối đếm (counter.vhd). khối FA (full\_adder.vhd).

Các công vào ra của khối trên bao gồm công A, B 8-bit là các hạng tử, Cin là bit nhớ đầu vào, Sum là kết quả cộng , Cout là bít nhớ ra. Các tín hiệu toàn cục là CLK và RESET, ngoài ra công WE là tín hiệu khi tích cực WE = 1 cho biết là dữ liệu ở các đầu vào A, B đang được nạp vào các thanh ghi SHIFTER\_REGA, SHIFTER\_REGB để thực hiện cộng. Tín hiệu ra RESULT = 1 cho biết giá trị SUM và Cout đã đúng.

- *Thanh ghi dịch SHIFTER\_REGA, SHIFTER\_REGB:*

Các thanh ghi sẽ dịch sang bên phải một bit nếu như không phải thời điểm có kết quả ra, ở thời điểm này bộ đếm đã đếm xong 8 xung nhịp do vậy các tín hiệu này được mô tả như sau:

Tín hiệu temp báo hiệu cnt đã đếm đến “111”

```
with cnt select
    temp <= '1' when "111",
    '0' when others;
```

Tín hiệu này được làm trễ một xung nhịp bằng một D-flip-flop như mô tả sau:

```
reg_p: process (CLK)
begin
if CLK = '1' and CLK'event then
    temp_reg <= temp;
end if;
```

```
end process reg_p;
```

Cuối cùng tín hiệu shift\_enable là đảo của temp đã bị làm trễ.

```
shift_enable <= not temp_reg;
```

Tín hiệu bitin của hai thanh ghi dịch SHIFTER\_REGA, SHIFTER\_REGB không quan trọng và gán bằng hằng số bất kỳ, trên hình ta gán bằng ‘0’.

Tín hiệu vào song song của thanh ghi SHIFTER\_REGA, SHIFTER\_REGB lấy lần lượt từ các cổng A, B 8 bit, cổng WE của các thanh ghi lấy từ cổng WE của khối serial\_bit\_adder.

- *Bộ đếm COUNTER.*

Tín hiệu counter\_enable chính là tín hiệu shift\_enable cho các thanh ghi SHIFTER\_REGA, SHIFTER\_REGB.

```
counter_enable <= shift_enable;
```

Bộ đếm reset nếu bắt đầu nạp dữ liệu để thực hiện phép toán (WE =1 ) hoặc khi có reset không đồng bộ của toàn khối.

```
cnt_reset <= WE or reset;
```

- Khối Full\_Adder

Các tín hiệu bên trong bitA, bitB chính là các bit thấp nhất của hai thanh ghi dịch SHIFTER\_REGA, SHIFTER\_REGB, các bit này được đưa vào khối FA.

```
bitA <= regA(0);
```

```
bitB <= regB(0);
```

Trong đó regA, regB chính là các giá trị của các thanh ghi dịch tương ứng cổng Dout.

Để mô tả chuỗi bit nhớ cần có một khối chọn kênh một bit với các đầu vào là bitCout, và Cin, tín hiệu chọn kênh là WE, nếu như tại thời điểm đầu tiên(WE = 1) thì đầu ra reg\_cin = Cin, còn tại các thời điểm khác đầu vào bit nhớ chính là giá trị đầu ra bit nhớ lần trước nên Reg\_cin = bitCout.

Tín hiệu bitCin vào cổng CIN của khối full\_adder chính là reg\_cin được làm trễ một xung nhịp vad được mô tả như sau:

```
if CLK = '1' and CLK'event then  
    bit_Cin <= Reg_cin;
```

```
end if;
```

*Thanh ghi SHIFTER\_SUM*

Tín hiệu bitSum là đầu ra của Full\_Adder đóng vai trò là đầu vào cho thanh ghi SHIFTER\_SUM.

Tín hiệu RESULT bằng 1 nếu như kết quả đã xuất hiện ở cổng ra SUM và Cout kết quả chính xác có sau 8 xung nhịp cho bộ cộng 8 bit, do đó RESULT trùng với tín hiệu temp\_reg đã trình bày ở trên.

```
--tin hieu result tre nhieu hon temp 1 xung nhieu
RESULT <= temp_reg;
```

Cổng vào dữ liệu song song của thanh ghi REG\_SUM không quan trọng, có thể gán bằng giá trị bất kỳ 8 bit, ở trên hình ta gán bằng giá trị của x”00”. Cổng vào WE của thanh ghi này cũng không quan trọng, ta lấy luôn tín hiệu WE.

#### Bước 6: Kiểm tra khôi thiết kế:

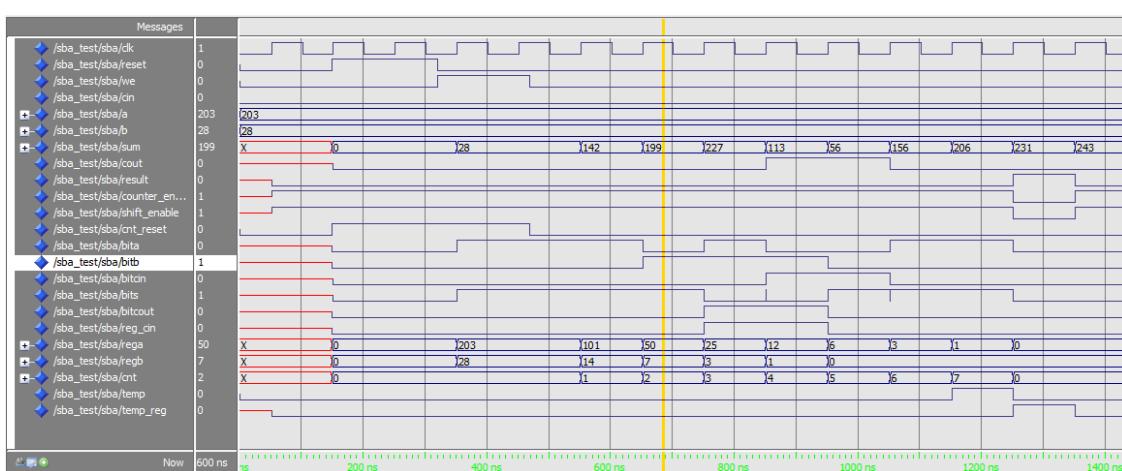
*Hướng dẫn:* Để kiểm tra khôi thiết kế thì cách thức không khác gì với những bài đã làm trước đây. Quan trọng nhất là phải tạo được tín hiệu xung clk:

```
--create clock
create_clock: process
begin
    wait for 2 ns;
    CLK <= not CLK after 50 ns;
end process create_clock;
```

Sau đó là tạo tín hiệu điều khiển hợp lý gồm reset và WE, ví dụ như sau

```
A <= x"1B"; B <= x"0C"; Cin <= '0';
reset <= '0'; WE <= '0';
wait for 150 ns;
reset <= '1';
wait for 170 ns;
reset <= '0';
WE <= '1';
wait for 150 ns;
WE <= '0';
wait;
```

Kết quả mô phỏng có dạng như sau:

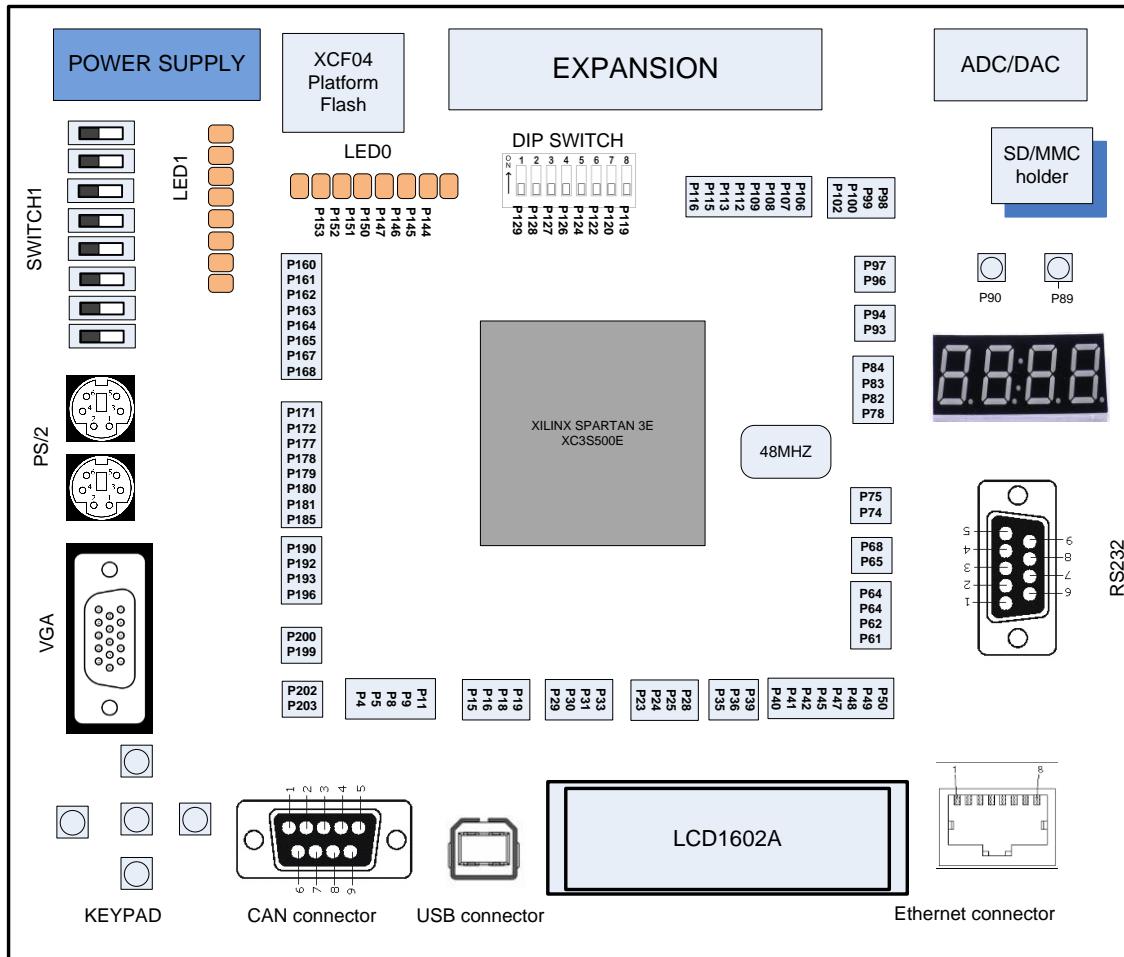


### **3. Bài tập sau thực hành**

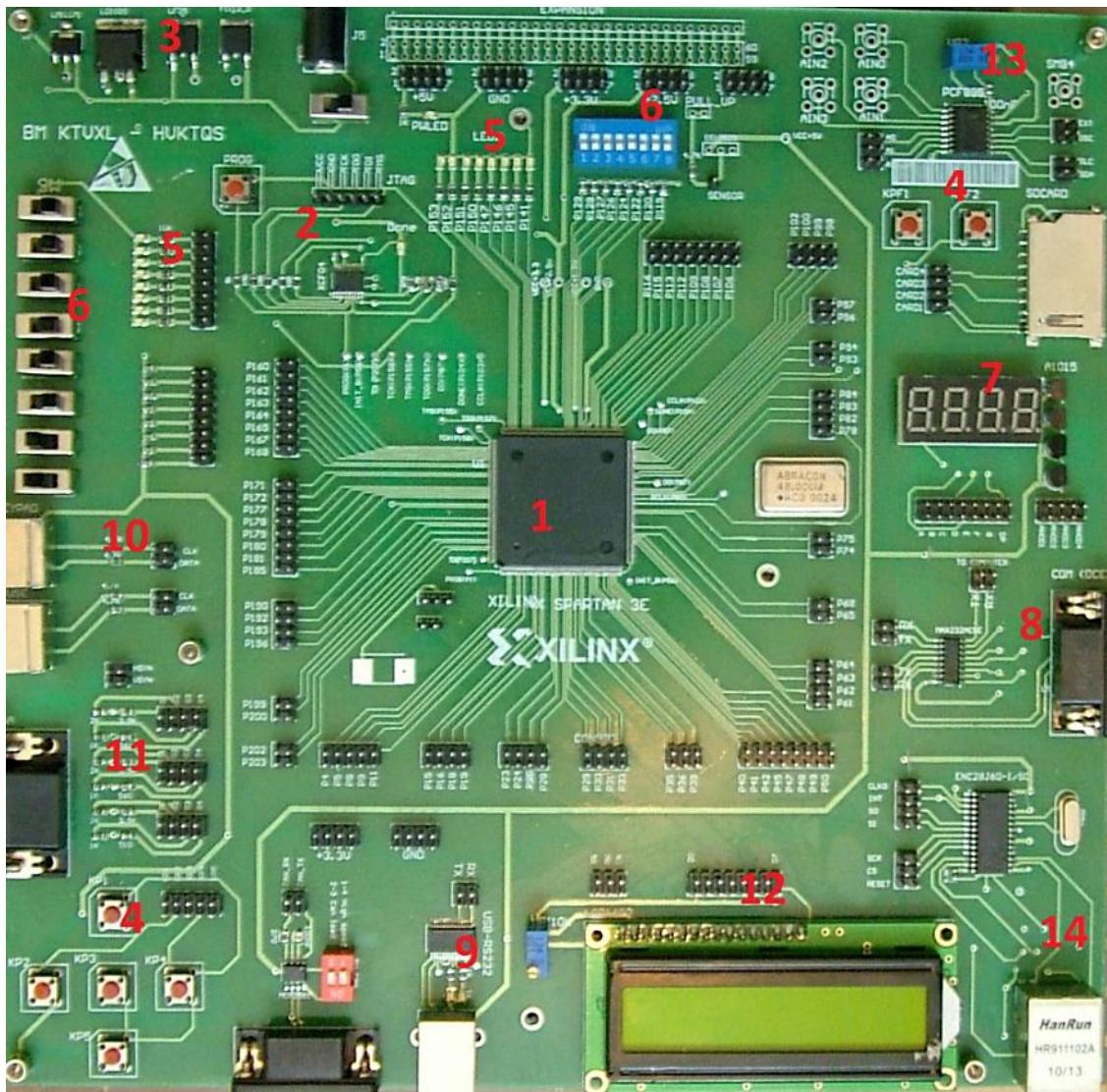
- Xây dựng khối cộng bit nối tiếp cho trường hợp N-bit
- Xây dựng khối cộng bit nối tiếp sử dụng 2 Full-adder
- Tối ưu hóa thiết kế bằng cách sửa lại thiết kế của các thanh ghi dịch đầu vào và đầu ra của khối cộng (bỏ những tín hiệu không sử dụng đến)

## Phụ lục 3: MẠCH PHÁT TRIỂN ỦNG DỤNG FPGA

### 1. Giới thiệu tổng quan



Sơ đồ mạch khi thiết kế



Mạch thí nghiệm hoàn thiện

Mạch thí nghiệm FPGA được xây dựng và chế tạo bởi bộ môn Kỹ thuật xung số, Vi xử lý với mục đích phục vụ các đối tượng đào tạo hệ chính quy dân sự, quân sự, học viên cao học và phục vụ công tác nghiên cứu khoa học. Mạch được thiết kế trên nền tảng Xilinx Spartan 3E FPGA XCS500 PQG208. Mạch in và mạch nguyên lý được vẽ trên công cụ Altium Designer.

Phần trung tâm của mạch là FPGA và FLASH ROM để nạp cấu hình, thiết kế của phần này là nối cứng để đảm bảo tốt sự hoạt động của FPGA. 10 ngoại vi còn lại được thiết kế đa phần mở nhảm tạo sự linh động và dễ kiểm soát trong quá trình làm việc cũng như phù hợp với mục đích nghiên cứu thử nghiệm. Các ngoại vi kia cũng có thể sử dụng độc lập để giao tiếp với các khối bên ngoài

mà không lệ thuộc vào FPGA. Ngoài phần nguồn được nối sẵn, các chân tín hiệu của ngoại vi khi muốn làm việc với FPGA phải được nối bằng cáp chuyên dụng.

## 2. Các khối giao tiếp có trên mạch FPGA

### 2.1. Khối FPGA XCS500

#### Các tham số chính của Xilinx Spartan 3E

Spartan3E dòng FPGA trung bình được thiết kế cho những ứng dụng cỡ vừa và nhỏ với chi phí không lớn. Có tất cả 5 loại Spartan 3E với mật độ tích hợp từ 960 Slices(XC3S200) cho tới 14752 Slices (XC3S1600), nếu tính theo đơn vị cổng tương đương là 2000 cho tới 30000 cổng (cổng tương đương được tính là một cổng AND hoặc OR hai đầu vào).

FPGA sử dụng trong mạch thí nghiệm là XC3S500 PQG208 với số Slices là 4656 tương đương 1164 CLBs (10,476 cổng tương đương) được bố trí trên 46 hàng và 24 cột. Các tài nguyên khác bao gồm 4 khối điều chỉnh/tạo xung nhịp hệ thống Digital Clock Manager (DCM) được bố trí 2 ở trên và 2 ở dưới. Các khói nhớ bao gồm 360K Block RAM và tối đa 73K RAM phân tán. Tích hợp 20 khói nhân 18x18 bít được bố trí sát các Block Ram. Về tài nguyên cổng vào ra XC3S500 với gói PQ208 hỗ trợ 208 chân vào ra trong đó có 8 cổng cho xung nhịp hệ thống, tối đa 232 cổng vào ra sử dụng tự do, trong đó có 158 chân IO, số còn lại là chân Input. XC3S500 được thiết kế trên công nghệ 90nm và cho phép làm việc ở xung nhịp tối đa đến 300Mhz, với tốc độ như vậy XC3S500 có thể đáp ứng hầu hết những bài toán xử lý số cỡ vừa và nhỏ.

### 2.2. Mạch nạp JTAG/PLATFORM FLASH XCF04

Mạch nạp cho FPGA sử dụng Xilinx platform Flash ROM XCF04 dung lượng 4Mb, có khả năng nạp trực tiếp cấu hình vào FPGA thông qua giao tiếp JTAG hoặc nạp gián tiếp cấu hình để lưu trữ cố định trong ROM, khi cần nạp lại cấu hình vào FPGA chỉ việc ấn phím PROG. Chương trình lưu trong ROM có thể được đọc/xóa và ghi mới lại

### 2.3. Khối nguồn Power Supply

Khối nguồn là một khối quan trọng cung cấp cho chip FPGA yêu cầu cấp các nguồn điện áp một chiều 3,3V, 2.5V và 1.2 V. Cho Xilinx Flash Platform XFC04 yêu cầu nguồn 3.3V. Với tất cả các ngoại vi còn lại sử dụng các mức điện áp phổ biến là 5.0V và 3.3V. Tất cả các mức điện áp này được tạo bởi các IC

nguồn chuyên dụng tích hợp trên mạch. Để mạch hoạt động chỉ cần cung cấp điện áp đầu vào DC 5V.

## 2.4. Khối giao tiếp Keypad

Trong sơ đồ sử dụng 5 Keypad độc lập có thể sử dụng cho nhiều mục đích khác nhau, các phím này được treo ở mức cao. Ngoài ra có hai Keypads được nối cứng với FPGA thông qua các chân P90 và P89 để sử dụng trực tiếp trong các thiết kế.

## 2.5. Khối 8x2 Led-Diod

Khối hiển thị LED được thiết kế gồm 2x8 LED đơn treo trở 10K trong đó 8LEDs được gắn cứng với FPGA thông qua các chân P153, P152, P151, P150, P147, P145, P146, P144.

8LEDs còn lại để mở khi sử dụng phải kết nối bằng cáp riêng. Tất cả các LED này được trở 1K và có mức tích cực là 1.

## 2.6. Khối Switch

Khối switch được thiết kế gồm 2x8 SWITCHS, 8 SWITCH sử dụng DIP-SWITCH loại nhỏ và gắn cứng với các chân FPGA qua các chân P129, P128, P127, P126, P124, P122, P120, P119.

8 SWITCHs còn lại được cấu tạo từ các SWITCH đơn loại lớn và đầu ra để mở, khi sử dụng cần nối với các chân FPGA bằng cáp riêng. Tất cả các Switch này đều được treo mức cao thông qua trở 10K.

## 2.7. Khối giao tiếp 4x7-seg Digits

Cung cấp giao phương pháp hiển thị đơn giản thông qua Led 7 thanh, khối Led này được cấu tạo từ 4 số và hoạt động theo nguyên lý quét các cực ANOD chung. Các chân quét ANOD sẽ được khuỷu ch đại bằng pnp transistor A1015 để đảm bảo độ sáng của các LED.

Tốc độ quét cho 4 LED cần duy trì ở tần số cỡ 200Hz để có được hiển thị tốt nhất.

## 2.8. Khối giao tiếp RS232

Khối giao tiếp RS232 thực hiện truyền thông nối tiếp giữa FPGA và các thiết bị số khác. Trong mạch này sử dụng IC MAX232ACSE. Sơ đồ nguyên lý thể hiện ở hình dưới đây. MAX232 có tạo ra hai kết nối truyền thông nối tiếp độc lập giữa FPGA và các thiết bị khác trên máy tính trong đó 1 cổng COM

(DTE) và một cổng khác để dưới dạng kết nối tự do. Tốc độ Baud hỗ trợ từ 110 đến 11520 bit/Sec.

FPGA giao tiếp với khối này thông qua 2 cặp tín hiệu Rx, Tx được để tự do.

## 2.9. Khối giao tiếp USB – RS232

Khối giao tiếp USB được thiết kế dựa trên IC FT232RL của FTDI. IC này chuyển đổi giao tiếp USB sang các giao tiếp RS232 / RS422 / RS485 với tốc độ truyền từ 300-3M Baud. Với thiết kế này giao tiếp USB thực sự được thực hiện như giao tiếp UART, toàn bộ giao thức USB2.0 được thực thi bằng FP232RL. IC này cũng được tích hợp sẵn dao động bên trong và hoạt động chỉ với một điện áp nguồn +5V. Khi được kết nối cổng USB, nguồn 5V từ thiết bị bên ngoài như máy tính có thể dùng để cấp cho mạch FPGA hoạt động bình thường.

Giao tiếp với khối này giống như giao tiếp với khối RS232 trình bày ở trên gồm có 2 tín hiệu Rx, Tx.

Tài liệu chi tiết về FT232RL và Driver cho máy tính có thêm xem trên trang Web của nhà sản xuất

<http://www.ftdichip.com/Products/ICs/FT232R.htm>

## 2.10. Khối giao tiếp PS/2.

FPGA có thể thực hiện điều khiển các ngoại vi cơ bản như màn hình, bàn phím chuột thông qua các giao thức chuẩn PS/2. Mạch đã được thử nghiệm với tốc độ giao tiếp từ 20-30 KHz với bàn phím chuẩn cho kết quả ổn định. Đường truyền PS/2 có giao thức giống đường truyền nối tiếp nhưng các thiết bị sử dụng chung xung nhịp đồng bộ. 2 tín hiệu tối thiểu để giao tiếp là CLK với tần số từ 10-20Khz, và tín hiệu Data cho đường dữ liệu. Trên mạch FPGA các tín hiệu này cũng để dưới dạng tự do và chỉ kết nối khi cần thiết. Trên mạch thiết kế hai cổng PS/2 đủ cho giao tiếp đồng thời cả MOUSE và KEYBOARD.

## 2.11. Khối giao tiếp VGA.

Đối với khối VGA, được thiết kế để có thể giao tiếp với tối đa ở chế độ 12 bit màu. Các chân tín hiệu RGB được treo trờ với giá trị 500, 1K, 2K và 4K. Mạch được kiểm tra với chế độ phân giải 600x800 và tốc độ làm tươi 60Hz cho kết quả ổn định (tương đương xung nhịp hệ thống của mạch điều khiển 50Mhz). Mạch điều khiển VGA là một trong những mạch đòi hỏi tốc độ cao vì vậy khi thiết kế nên sử dụng DCM để đảm bảo độ chính xác cho xung CLK.

Tín hiệu giao tiếp gồm HS, VS tương ứng là các xung quét ngang và dọc. 12 tín hiệu R[3:0], G[3:0] và B[3:0] để đưa 12 bit màu hiển thị lên màn hình.

## 2.12. Khối giao tiếp LCD1602A.

LCD1602A là dạng LCD đơn sắc cho phép hiển thị các ký tự chuẩn ASCII với một giáo thức có thể dễ thực hiện trên các khôi phần cứng bằng FPGA.

Tín hiệu để giao tiếp với LCD gồm 8 chân tín hiệu LCD\_Data[7:0], 3 chân điều khiển LCD\_RS, LCD\_EN, và LCD\_RW. Chi tiết về giao thức với LCD1602A xem trong ví dụ chương 4 và tài liệu hướng dẫn.

LCD được thiết kế để có thể thực hiện giao tiếp ở chế độ 4 bit cũng như 8 bit. Với tốc độ cỡ 100-300Khz các chân giao tiếp này cũng không gắn cứng với các chân FPGA, nghĩa là LCD ngoài làm việc với FPGA có thể sử dụng độc lập với các mục đích khác. LCD được cấp nguồn 5V, có chiết áp điều chỉnh độ tương phản của màn hình.

## 2.13. Khối giao tiếp ADC/DAC

Giao tiếp DAC/ADC được thực hiện trên cùng 1 IC là PCF8591, IC này có khả năng làm việc như một khôi DAC hay một khôi ADC 8-bit phụ thuộc vào cách thức cấu hình cho IC. Thông tin trao đổi giữa PCF8591 và FPGA thông qua giao tiếp chuẩn I2C, tốc độ tối đa hỗ trợ cho là 100Khz.

PCF8591 hoạt động với một nguồn cấp duy nhất với mức điện áp đầu vào cho phép dao động từ 2.5V đến 6V. Trên mạch IC này được cấp nguồn 3.3V, nguồn tham chiếu thay đổi nhờ một biến trở 10K từ 0-3.3V.

PCF8591 có 4 đầu vào tương tự AIN0, AIN1, AIN2, AIN3 với địa chỉ được lập trình, một đầu ra tương tự AOUT. IC hoạt động ở chế độ Slave có thể được địa chỉ hóa bởi 3-bit địa chỉ ở các chân A0, A1, A2, 8 IC PCF8591 có thể kết nối trên cùng một kênh truyền I2C mà không cần thêm phần cứng hỗ trợ. Vì trên mạch chỉ có 1 IC duy nhất nên có thể ngầm định địa chỉ luôn là 000. PCF8591 có thể sử dụng xung nhịp (cho giao tiếp I2C) từ đầu vào SCL hoặc xung nhịp từ một dao động ngoài. Để đơn giản thiết kế nên dùng xung nhịp từ chân SCL khi đó chân EXT cần phải nối đất.

Chi tiết về giao tiếp I2C cũng như cụ thể về cách thức cấu hình điều khiển cho PCF8591 có thể xem trong Datasheet của IC.

## **2.14. Khối giao tiếp Ethernet**

Khối giao tiếp Ethernet được thiết kế trên nền tảng IC ENC28J60 của Microchip thực hiện chức năng giao tiếp ở lớp Media Access Control(MAC) và Physical Layer(PHY) trong đó lớp PHY thực hiện biến đổi xung tín hiệu tương tự từ đường truyền thành tín hiệu số, còn lớp MAC thực hiện đóng gói các tín hiệu này theo chuẩn IEEE 802.3 để truyền tiếp cho các thiết bị xử lý ở lớp trên.

IC điều khiển làm việc với ENC28J60 thông qua giao tiếp chuẩn SPI (Serial Peripheral Interface) với tốc độ tối đa lên tới 20Mhz. Các tham số, chế độ làm việc được lưu trữ trong các thanh ghi điều khiển cho phép đọc và ghi từ bên ngoài. Bộ nhớ đệm của ENC28J60 được thiết kế là một khối RAM hai cổng (Dual-port RAM) với khả năng lưu trữ tạm thời các gói dữ liệu gửi đi, nhận về cũng như hỗ trợ thao tác DMA. Gói dữ liệu ở lớp MAC hoàn toàn tương thích chuẩn.

Các chân tín hiệu giao tiếp SPI trên mạch là CLK0, INT, S0, S1, SCK, CS, RESET.

Chi tiết về cấu trúc và cách thức làm việc của ENC28J60 xem trong Datasheet của IC và trên trang thông tin của Microchip

<http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en022889>.

## **Phụ lục 4: THỰC HÀNH THIẾT KẾ MẠCH SỐ TRÊN FPGA**

# Bài 1: Hướng dẫn thực hành FPGA bằng Xilinx ISE và Kit SPARTAN 3E

## 1. Mục đích

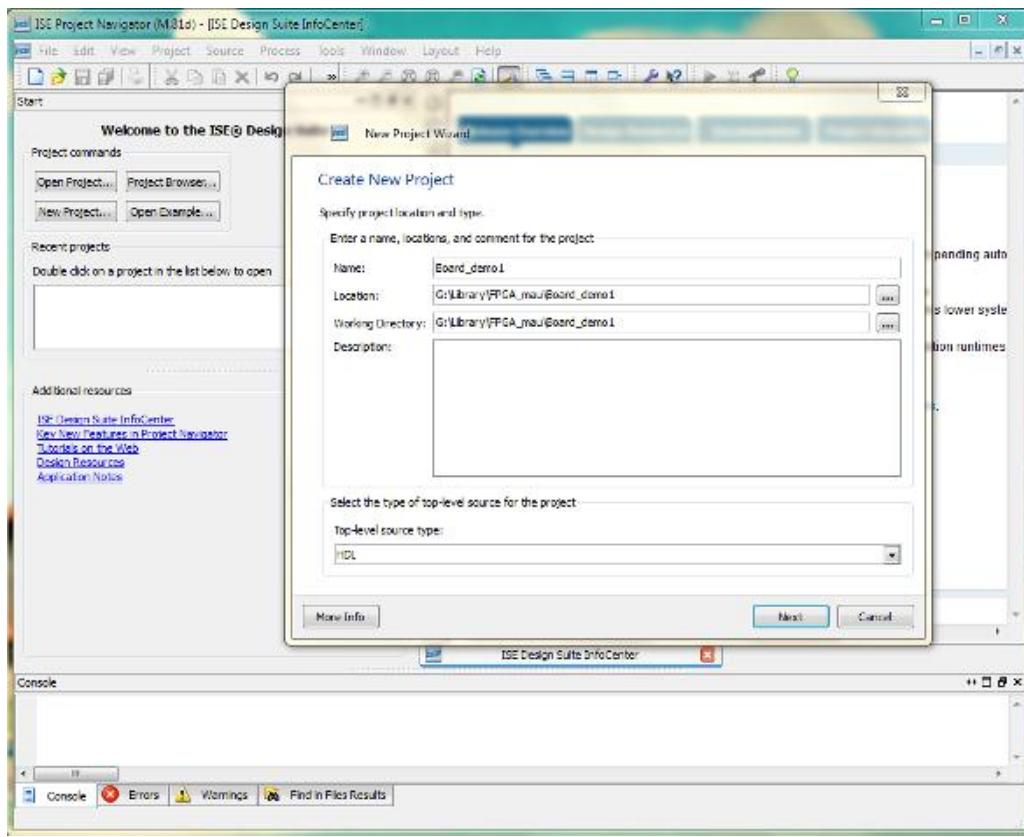
Làm quen với chương trình XilinxISE, học cách nạp cấu hình cho FPGA thông qua một ví dụ đơn giản. Thực hành nạp cấu hình trên mạch Spartan 3E.

Phần dưới đây hướng dẫn sử dụng chương trình bằng một ví dụ hết sức đơn giản là viết khôi điều khiển các 8-led có sẵn ở trên mạch FPGA bằng các 8-Switch tương ứng.

## 2. Hướng dẫn thực hành

### Bước 1: Tạo Project

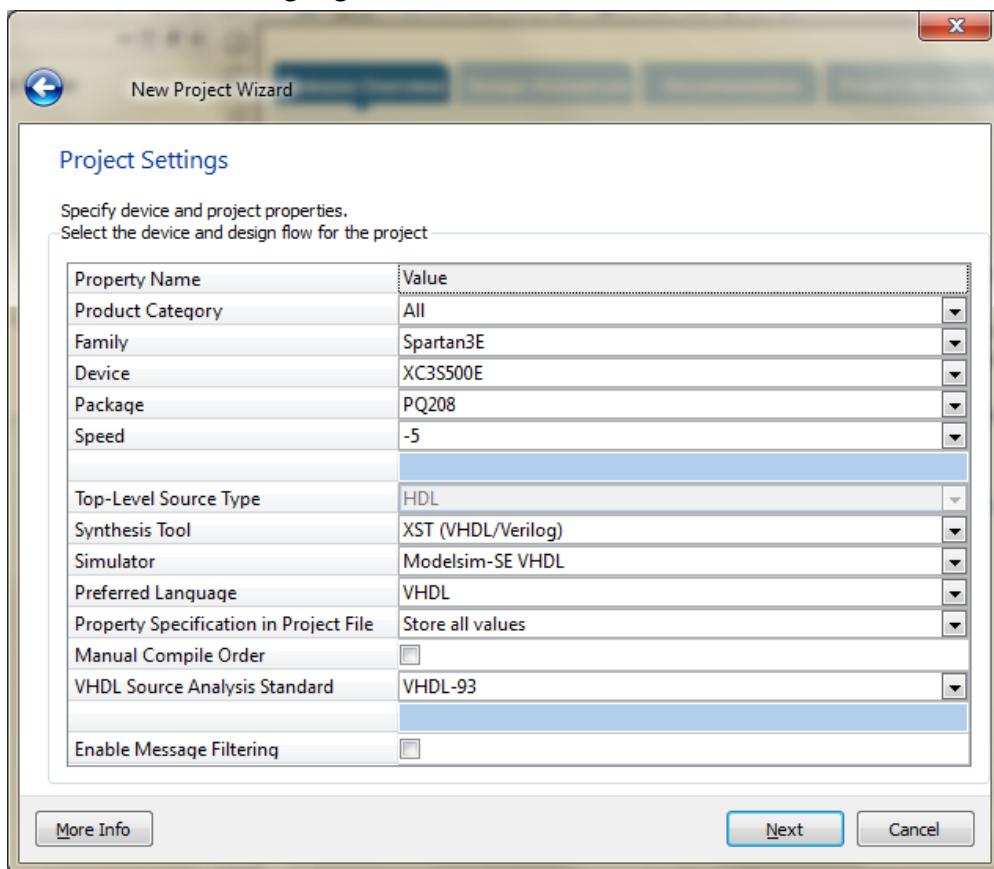
Sau khi khởi động phần mềm Xilinx ISE (ở đây dùng phiên bản 12.4), chọn *File -> New Project*. Đặt tên project là sp3\_led (tên có thể đặt tùy ý). Đồng thời, chọn mục *Top-level source type* là HDL để thiết kế bằng ngôn ngữ mô tả phần cứng.



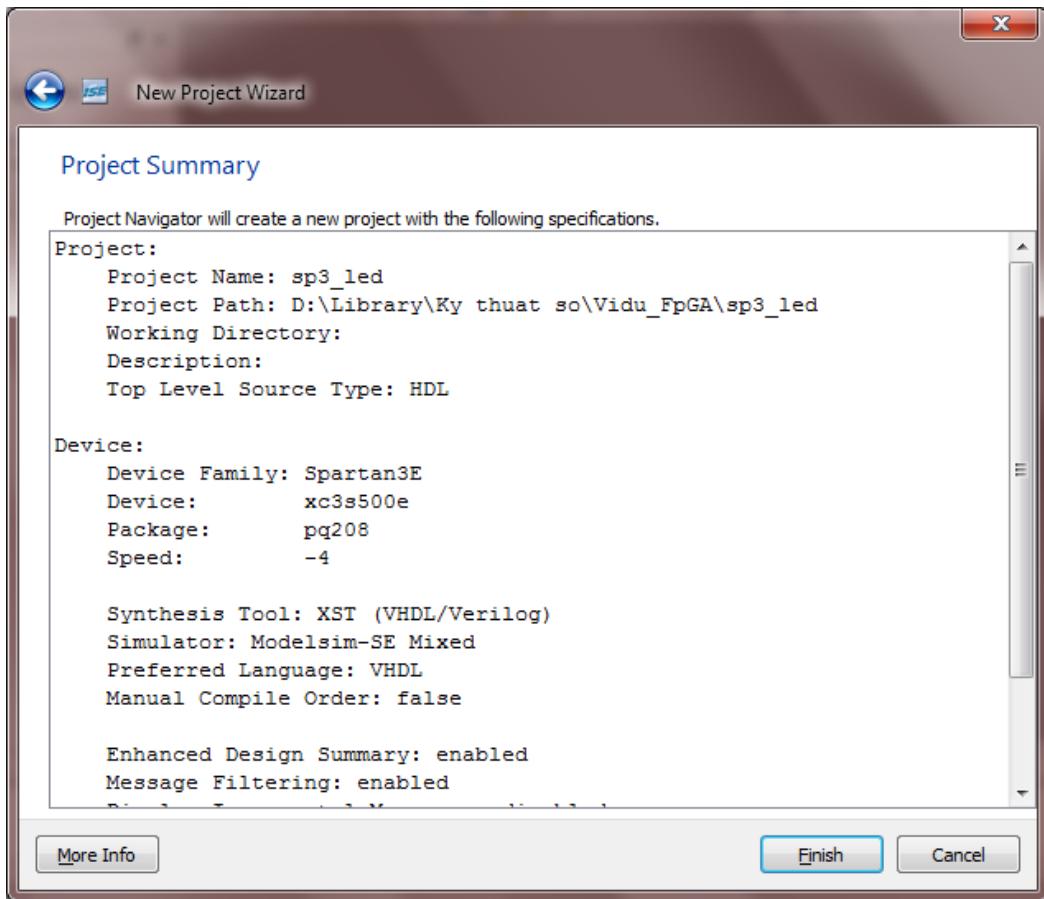
Tiếp theo, chọn NEXT để sang cửa sổ thiết lập các thuộc tính cho project, ở đây ta phải chọn đúng tên, gói (package), và tốc độ cho FPGA mà chúng ta muốn nạp, những thông tin này ghi trên mặt trên của chip FPGA.

Cụ thể đối với Kit của chúng ta sẽ chọn Spartan 3E, gói PQ208 và Speed bằng -5.

Ngầm định thì chương trình dùng để mô phỏng kiểm tra sẽ là Modelsim SE. Đối với ô Prefer Language ta chọn là VHDL



Sau khi ấn Next sẽ xuất hiện hộp thoại Project Summary. Chọn Finish để ISE tạo ra một project mới với các thuộc tính như trên vẽ.

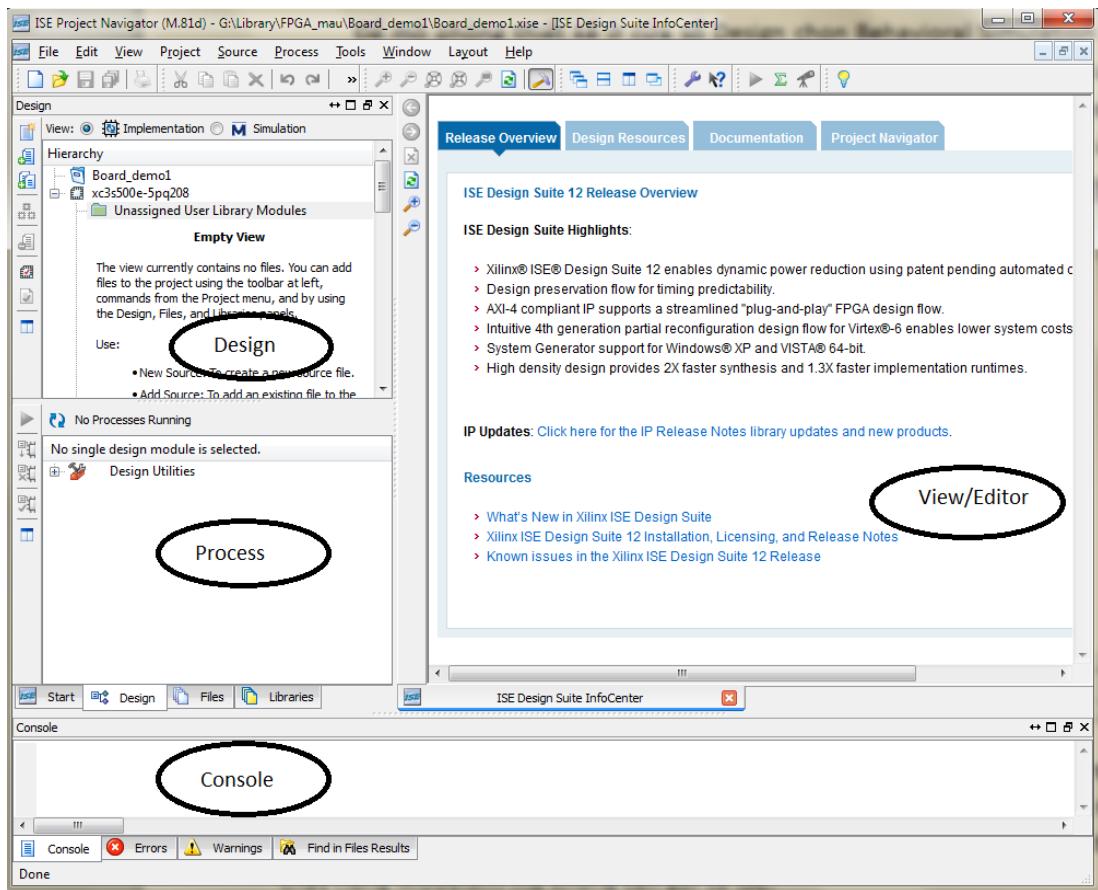


Hình 3.50. Project summary

Sau khi tạo Project thành công, chương trình chuyển sang chế độ làm việc với Project. Có 4 cửa sổ chính bao gồm

- Design: Chứa sơ đồ cấu trúc của các khối thiết kế trong project bao gồm mã nguồn, các file constraint nếu có và các file khác.
- Process: Nơi thực hiện các thao tác của quy trình thiết kế FPGA.
- View/Editor là cửa sổ làm việc chính, nơi có thể chỉnh sửa file nguồn, xem thông tin tổng hợp..vvv
- Command line (Console): Là cửa sổ hỗ trợ dòng lệnh trực tiếp (giống cửa sổ TCL script trong ModelSim)

Từ cửa sổ chương trình ta nhấp chuột đúp vào file sp3\_led.vhd thuộc cửa sổ Design (phía góc trái trên) để bắt đầu tiến hành soạn thảo mã nguồn.



ISE Project Windows

## Bước 2: Chuẩn bị mã nguồn

Tạo một file mã nguồn có tên `Board_demo1.vhd` với nội dung dưới đây. Có thể tạo mã nguồn độc lập bên ngoài (bằng Notepad++) như làm ở những phần trước hoặc có thể tạo trực tiếp trong ISE bằng cách ấn chuột phải vào cửa sổ Design và chọn New Source và làm theo hướng dẫn.

```
-----
-- Company: BMKTVXL
-- Engineer: Trinh Quang Kien
-- Module Name: board_demo1 - Behavioral
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity board_demo2 is
    Port (
        sw0      : in STD_LOGIC_VECTOR (7 downto 0);
        led0     : out STD_LOGIC_VECTOR (7 downto 0);
    );
end board_demo2;
architecture Behavioral of board_demo2 is
```

```

begin

    led0 <= not sw0;

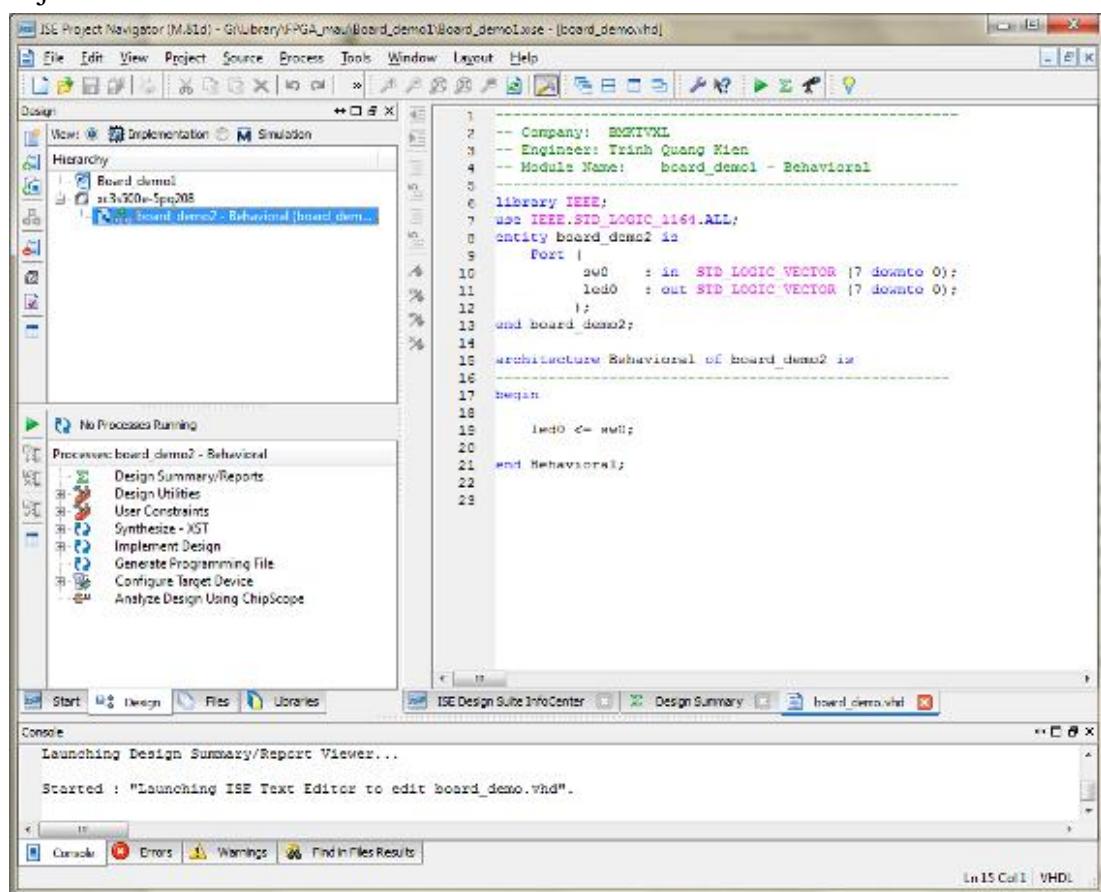
```

```
end Behavioral;
```

### Bước 3: Bổ xung mã nguồn vào Project

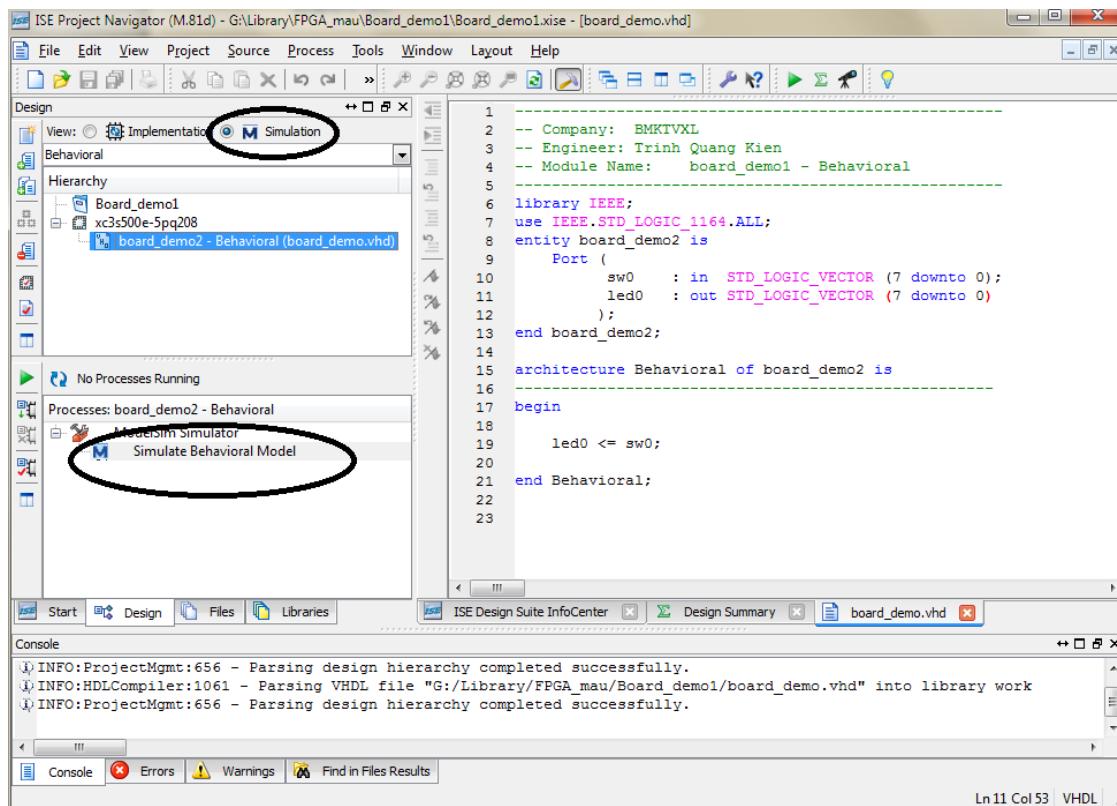
Bổ xung file mã nguồn vào trong Project nếu tạo file độc lập bằng Notepad++. (Án chuột phải vào cửa sổ Design chọn Add Source và làm theo hướng dẫn, chú ý nên để mã nguồn trong thư mục cùng với Project để dễ quản lý.

Nếu file tạo trực tiếp trong ISE thì sẽ tự động được bổ xung vào trong Project.



### Bước 4: Mô phỏng kiểm tra chức năng thiết kế.

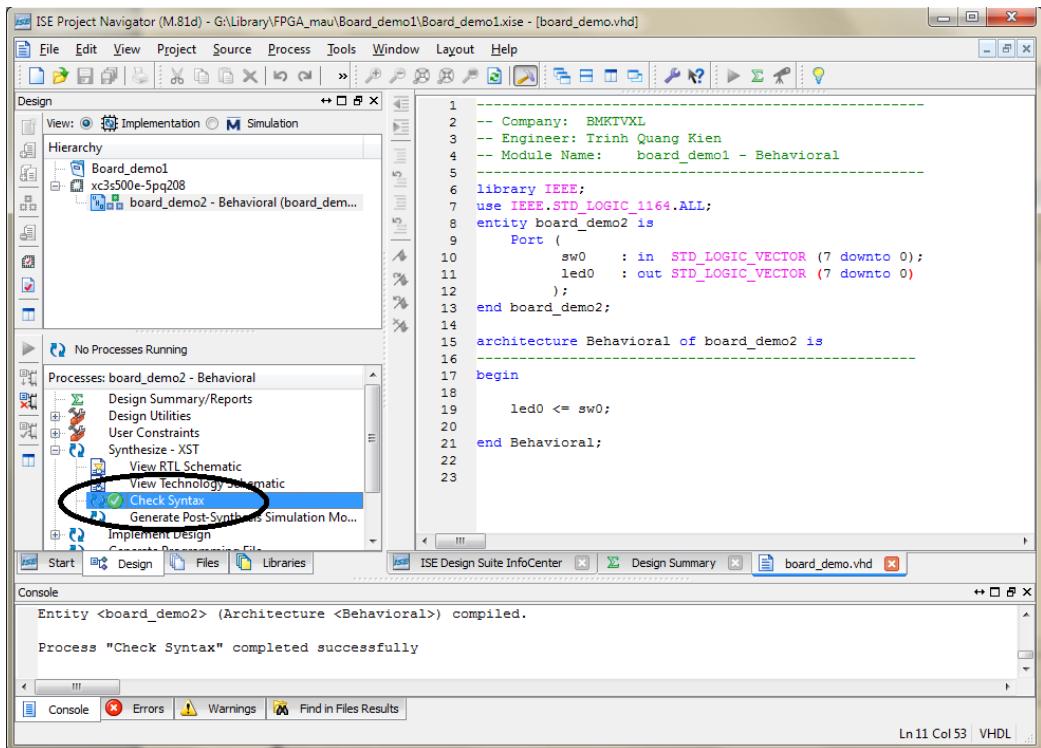
Để mô phỏng thiết kế ở cửa sổ Design chọn Behavioral Simulation sau đó chọn khôi cần kiểm tra ví dụ trong trường hợp này chúng ta chỉ có duy nhất khôi thiết kế board\_demo1. Kích đúp chuột vào biểu tượng Simulate Behavioral Model ở cửa sổ Process chương trình sẽ gọi Modelsim để chạy mô phỏng khôi tương ứng.



Người sử dụng có thể làm theo cách truyền thống là mô phỏng bằng Modelsim mà không nhất thiết phải gọi từ Xilinx ISE. Để đơn giản và thống nhất từ bây giờ về sau các động tác kiểm tra ta đều làm theo phương pháp truyền thống là chạy riêng Modelsim mà không cần gọi từ ISE. Ngoài mô phỏng bằng ModelSim có thể sử dụng trình mô phỏng tích hợp Isim của Xilinx, cách sử dụng trình mô phỏng này người dùng có thể tự tìm hiểu thêm.

### Bước 5: Kiểm tra cú pháp (Biên dịch mã nguồn VHDL)

Tại cửa sổ Process, nhấn chuột mở rộng quá trình *Synthesis – XST* và chọn Check Syntax. Chương trình kích hoạt khối kiểm tra cú pháp mà thực chất là thực hiện biên dịch mã VHDL bằng lệnh Vcom, thông thường nếu chúng ta đã làm mô phỏng kiểm tra chức năng ở bước thi ở bước này không thể phát sinh lỗi. Kết quả thông báo ở cửa sổ Command line



```

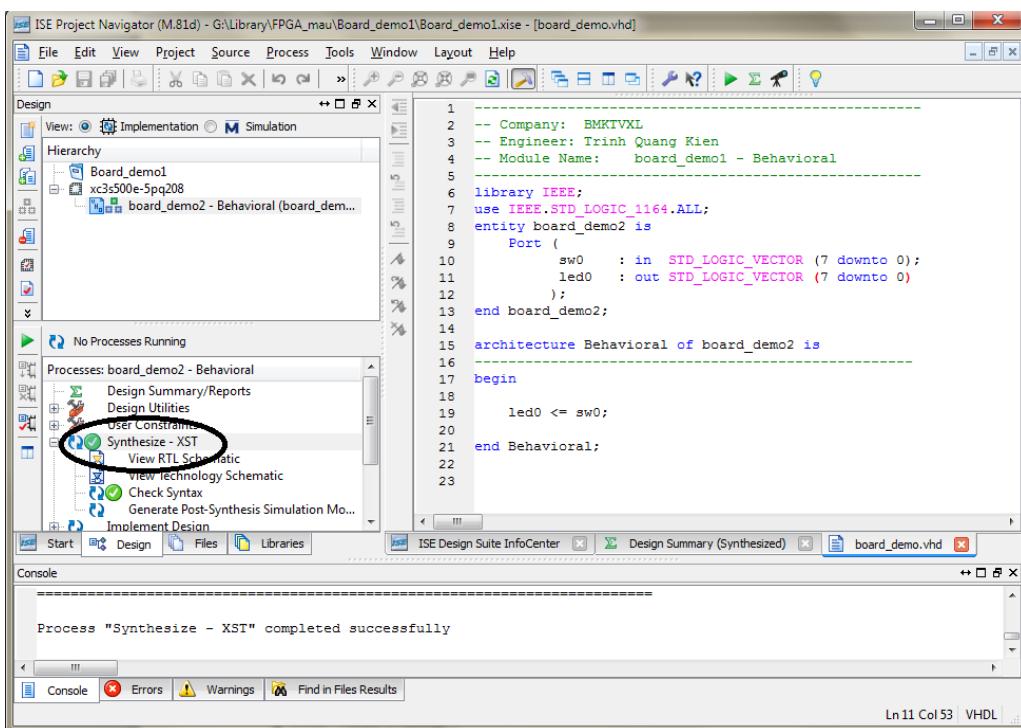
1 --- Company: BMKTVXL
2 --- Engineer: Trinh Quang Kien
3 --- Module Name: board_demo1 - Behavioral
4
5 library IEEE;
6 use IEEE.STD_LOGIC_1164.ALL;
7 entity board_demo2 is
8     Port (
9         sw0 : in STD_LOGIC_VECTOR (7 downto 0);
10        led0 : out STD_LOGIC_VECTOR (7 downto 0)
11    );
12 end board_demo2;
13
14 architecture Behavioral of board_demo2 is
15
16 begin
17
18     led0 <= sw0;
19
20 end Behavioral;
21
22
23

```

The screenshot shows the ISE Project Navigator interface. The left pane displays a project hierarchy for 'Board\_demo1' with a selected 'board\_demo2 - Behavioral' item. The right pane shows a VHDL code editor with the provided code. Below the code editor is a 'Processes' list. The 'Check Syntax' option is highlighted with a blue oval, indicating it has been selected.

## Bước 6: Tổng hợp thiết kế (Synthesis)

Click đúp chuột vào Synthesis –XST để kích hoạt trình tổng hợp thiết kế XST (*Xilinx Synthesis Technology*). Kết quả của quá trình tổng hợp thông báo ở cửa sổ Command Line.



```

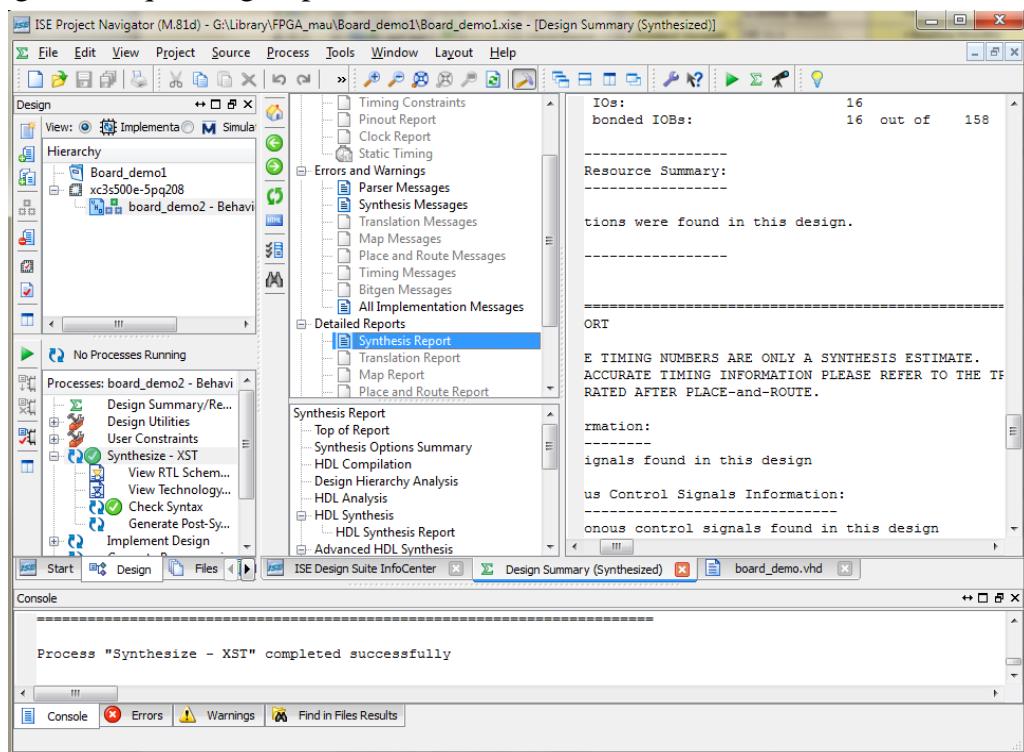
1 --- Company: BMKTVXL
2 --- Engineer: Trinh Quang Kien
3 --- Module Name: board_demo1 - Behavioral
4
5 library IEEE;
6 use IEEE.STD_LOGIC_1164.ALL;
7 entity board_demo2 is
8     Port (
9         sw0 : in STD_LOGIC_VECTOR (7 downto 0);
10        led0 : out STD_LOGIC_VECTOR (7 downto 0)
11    );
12 end board_demo2;
13
14 architecture Behavioral of board_demo2 is
15
16 begin
17
18     led0 <= sw0;
19
20 end Behavioral;
21
22
23

```

The screenshot shows the ISE Project Navigator interface. The left pane displays a project hierarchy for 'Board\_demo1' with a selected 'board\_demo2 - Behavioral' item. The right pane shows a VHDL code editor with the provided code. Below the code editor is a 'Processes' list. The 'Synthesize - XST' option is highlighted with a blue oval, indicating it has been selected.

## Bước 7: Đọc kết quả tổng hợp.

Tại cửa sổ làm việc chính (*View/Editor*) chọn Tab *Design Summary* sau đó chọn *Summary*, Click chuột vào *Synthesis Report* để hiển thị file text chứa thông tin kết quả tổng hợp.



Thiết kế của chúng ta có kết quả tổng hợp khá đơn giản, nội dung thu được có dạng sau:

```
=====
*          Final Report
=====
Final Results
RTL Top Level Output File Name      : board_demo2.ngr
Top Level Output File Name         : board_demo2
Output Format                      : NGC
Optimization Goal                 : Speed
Keep Hierarchy                     : No

Design Statistics
# IOs                            : 16

Cell Usage :
# IO Buffers                     : 16
#        IBUF                       : 8
#        OBUF                       : 8
```

```
=====
Device utilization summary:
-----
Selected Device : 3s500epq208-5
  Number of Slices:          0    out of   4656      0%
  Number of IOs:            16
  Number of bonded IOBs:   16    out of   158      10%
=====
TIMING REPORT
Timing Summary:
-----
Speed Grade: -5
Minimum period: No path found
Minimum input arrival time before clock: No path found
Maximum output required time after clock: No path found
Maximum combinational path delay: 4.632ns
```

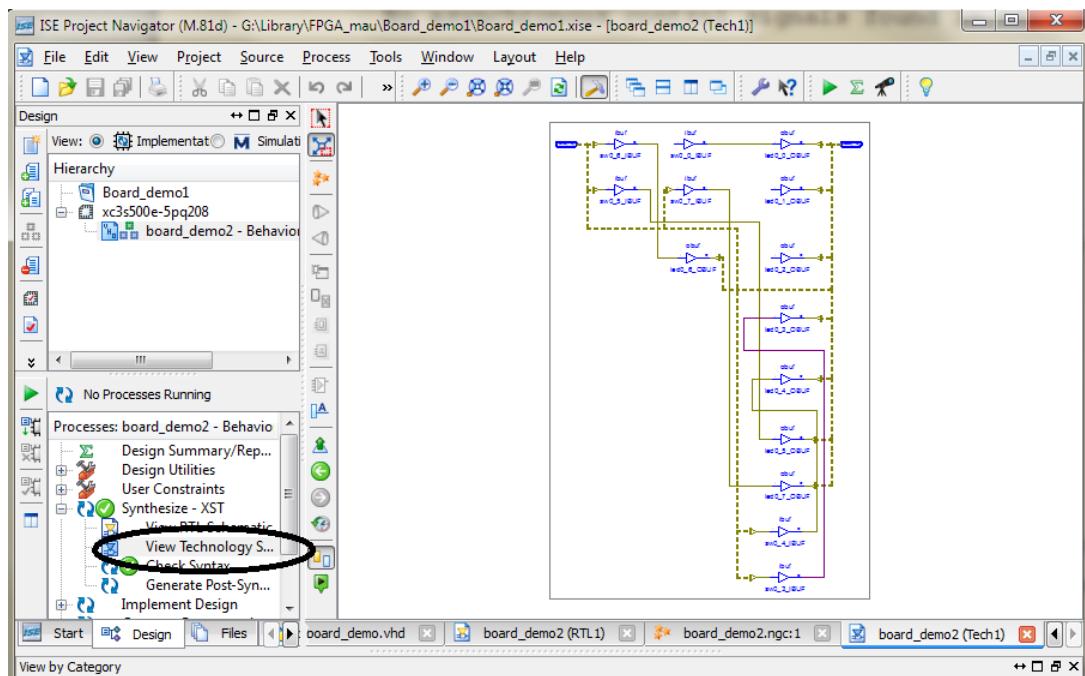
Timing Detail:

All values displayed in nanoseconds (ns)

#### **Bước 8: Kết xuất sơ đồ công nghệ, và sơ đồ RTL (không bắt buộc)**

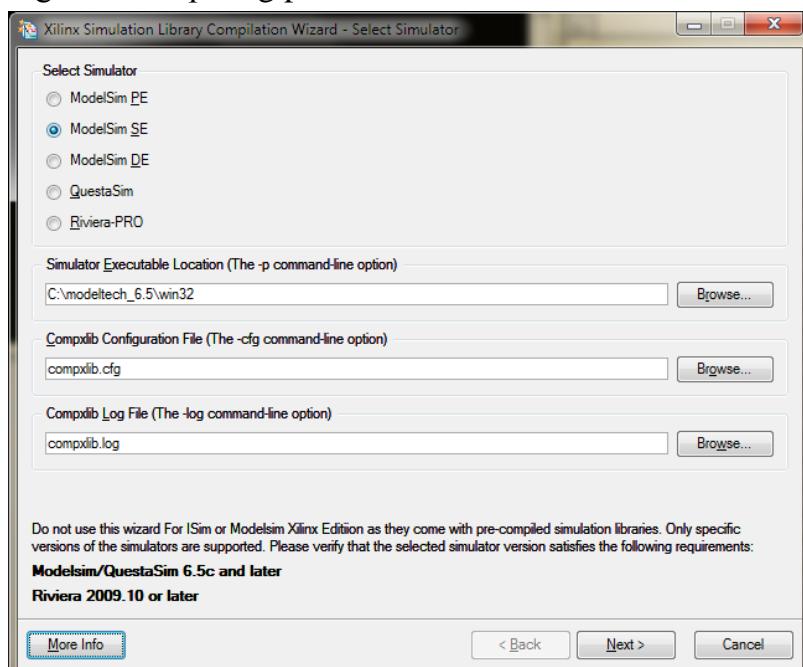
Sơ đồ công nghệ và sơ đồ RTL có thể thu được bằng cách chọn tương ứng *View Technology Schematic* và *View RTL Schematic*. Sau đó chọn tất cả hoặc một phần *Top Level Port* bô xung vào sơ đồ, chọn Create Schematic. Một sơ đồ cơ sở được tạo ra, để quan sát các đối tượng khác có thể click chuột vào các đối tượng hoặc kết nối trên đó để mở rộng hoặc chi tiết hóa sơ đồ.

Sơ đồ công nghệ thu được như ở hình sau, làm tương tự như vậy để thu được sơ đồ RTL.



### Bước 9: Kiểm tra thiết kế sau tổng hợp (không bắt buộc)

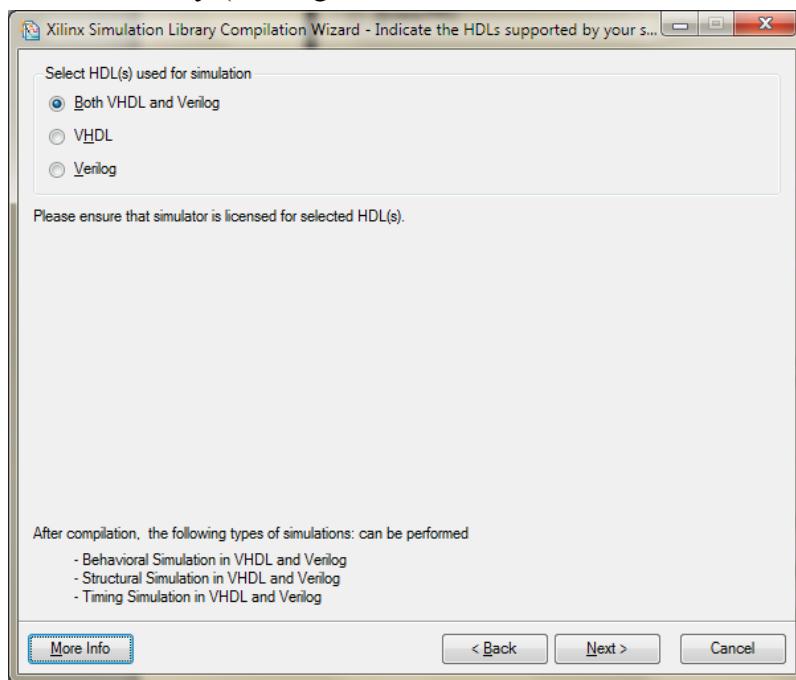
Kiểm tra thiết kế sau tổng hợp là thực hiện mô phỏng file netlist sinh ra sau quá trình tổng hợp. Đây cũng là một mô tả VHDL nhưng ở cấp độ cổng và đòi hỏi chương trình mô phỏng phải hỗ trợ thư viện UNISIM của Xilinx.



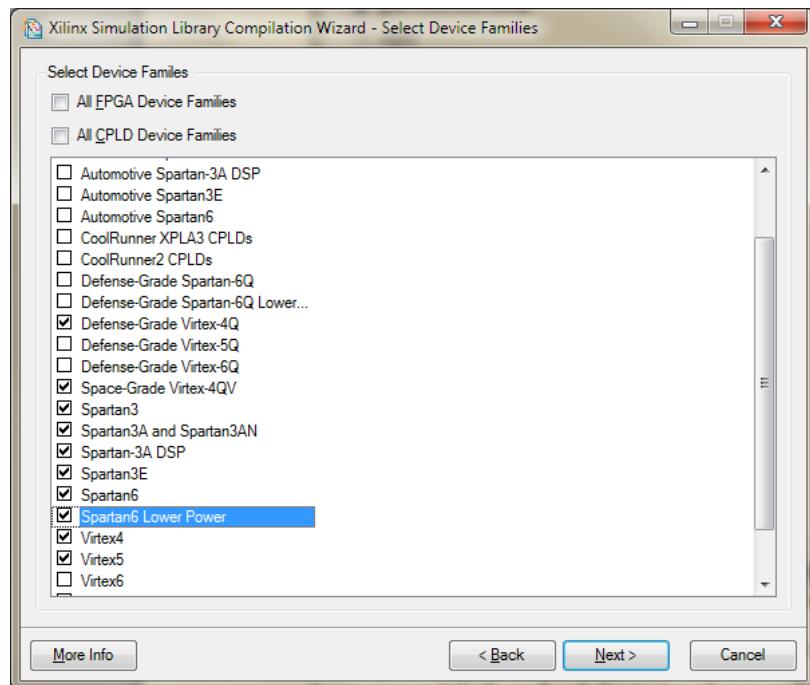
Đối với ModelSim chuẩn không tích hợp thư viện này, để tích hợp các thư viện của Xilinx nói chung chạy trình đơn Compxlib có thể tìm thấy trong thư

mục /bin/nt nơi cài chương trình Xilinx ISE (thông thường là C:\Xilinx\12.4\ISE\_DS\ISE\bin\nt).

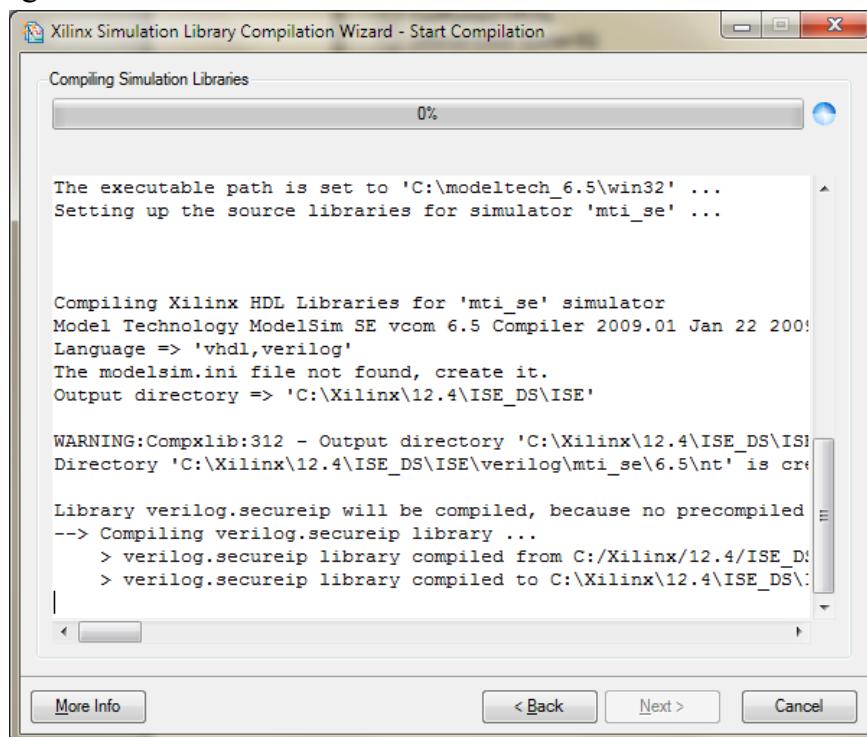
Chọn các tham số cần thiết, ở đây ta chọn ModelSim và kiểm tra lại vị trí cài đặt Modelsim trên máy (thường là C:\Modeltech\_6.x\win32). Chọn Next



Ở cửa sổ này chọn Both VHDL and Verilog nếu ta muốn cài đặt thư viện hỗ trợ cho cả hai loại ngôn ngữ thiết kế. Sau đó chọn Next. Cửa sổ tiếp theo cho phép lựa chọn các thư viện IC FPGA và CPLD sẽ được tích hợp sang trình mô phỏng, ta chọn một số các FPGA điển hình như ở hình sau

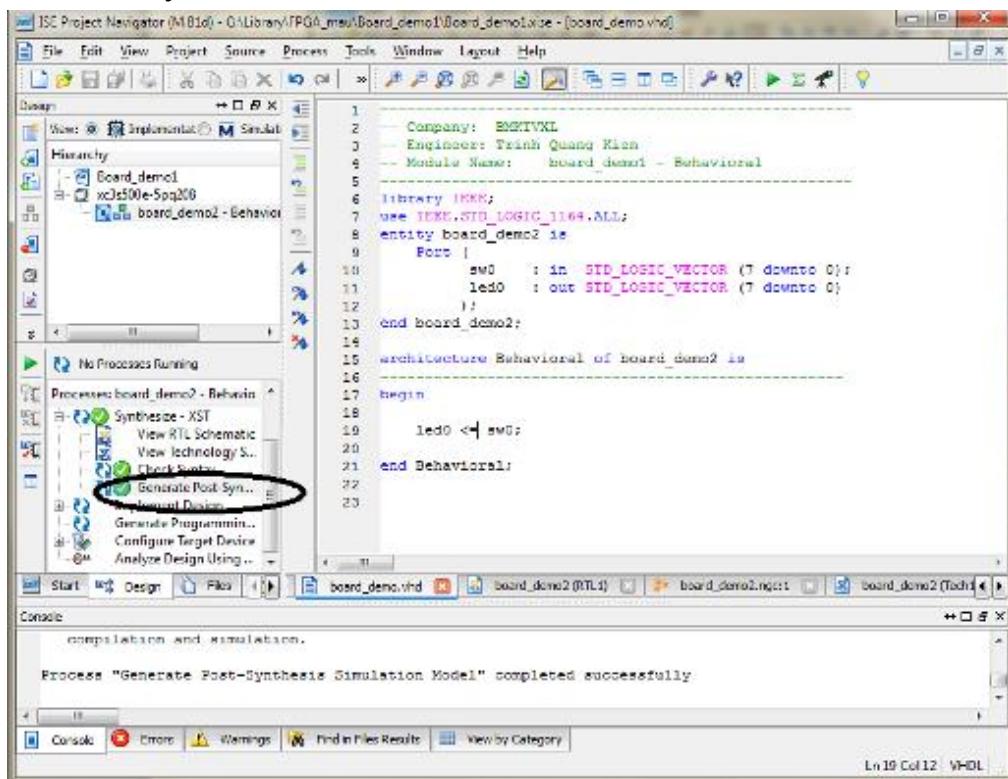


Cửa sổ tiếp cho phép lựa chọn các dạng thư viện sẽ được tích hợp sang trình mô phỏng, ta chọn All Library và Next, quá trình biên dịch sẽ được bắt đầu và thông thường mất từ 10-30 phút để thực hiện xong, phụ thuộc cấu hình máy và số lượng thư viện được chọn.



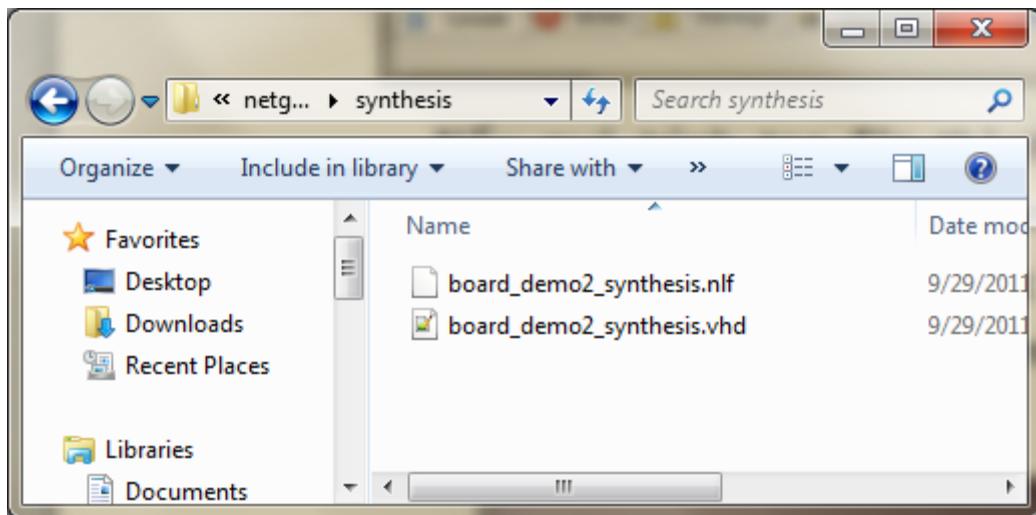
Nếu sử dụng Isim để mô phỏng thì không phải thực hiện các bước như ở trên và lưu ý rằng các bước trên **chỉ phải thực hiện một lần duy nhất**.

Để tạo ra file netlist phục vụ cho việc kiểm tra sau mô phỏng click vào Generate Post-synthesis simulation model như ở hình sau.



The screenshot shows the ISE Project Navigator interface. In the center, there is a code editor window displaying VHDL code for a module named 'board\_demo2'. The code defines an entity 'board\_demo2' with a single port 'sw0' (input) and one output 'led0'. An architecture 'Behavioral' is also defined. Below the code editor, the 'Process' pane is open, showing a list of tasks. One task, 'Generate Post-Synth...', is highlighted with a red oval. Other tasks listed include 'Synthesize - XST', 'View RTL Schematic', 'View Technology S...', 'Check Syntax', 'Generate Design...', 'Generate Programming...', 'Configure Target Device', and 'Analyze Design Using...'. At the bottom of the interface, a 'Console' window displays the message: 'Process "Generate Post-Synthesis Simulation Model" completed successfully.'

Nếu quá trình tạo file thành công sẽ có thông báo tương ứng *Process "Generate Post-Synthesis Simulation Model" completed successfully* ở của sổ *Command line*. Sau đó mở thư mục nơi chứa các file của project ta sẽ thấy xuất hiện thư mục có tên *netgen/synthesis* có chứa file mô tả VHDL có tên *sp3\_led\_synthesis.vhd*.

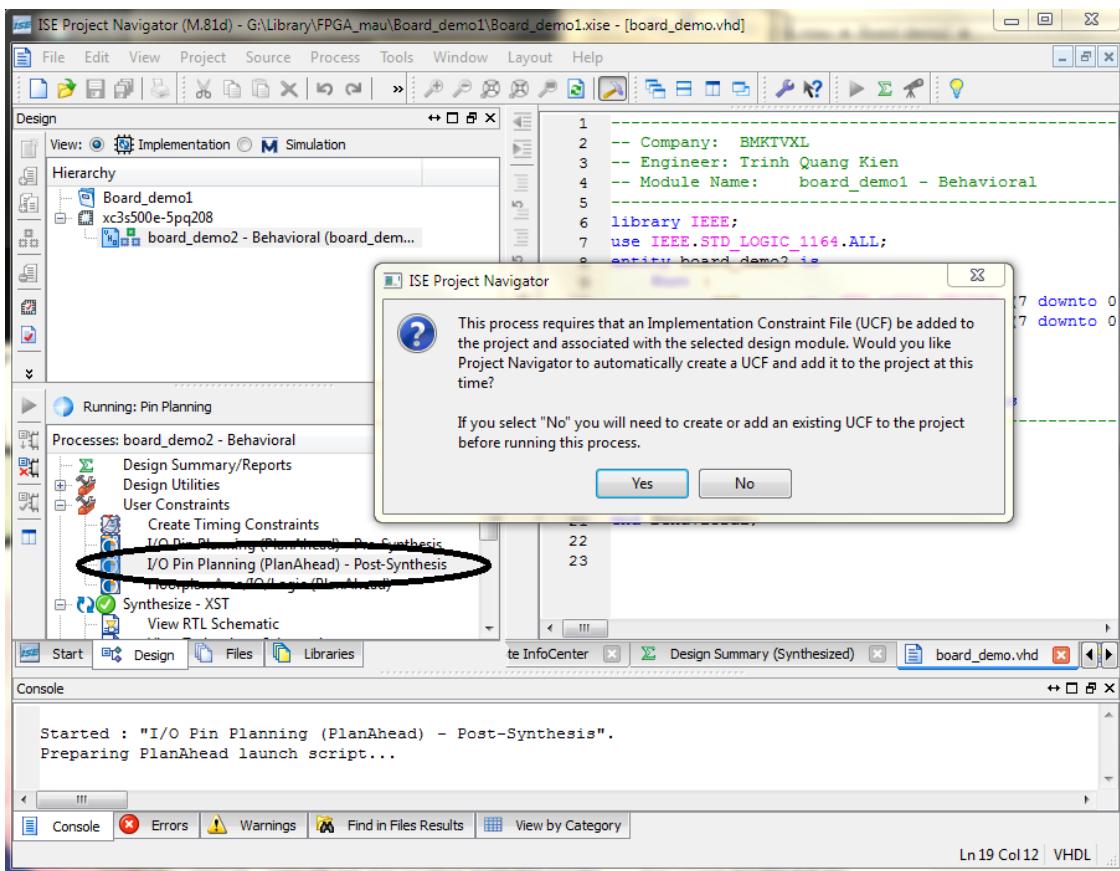


Nếu Modelsim đã hỗ trợ thiết kế trên thư viện UNISIM thì có thể tiến hành mô phỏng kiểm tra bình thường file nguồn này. Bước mô phỏng này nhằm khẳng định chức năng của mạch không bị thay đổi sau khi tổng hợp. Người sử dụng cũng có thể copy file netlist trên và tiến hành mô phỏng độc lập không phụ thuộc vào ISE.

#### **Bước 10. Gán chân vào ra sau tổng hợp**

Bước này làm nếu như chúng có Kit kiểm tra, trong trường hợp không có Kit thì không có thể bỏ qua bước này, chương trình sẽ tự động gán các chân vào ra của thiết kế cho các IO\_pad bất kỳ của FPGA.

Chương trình thực hiện gán chân là PlanAhead, để khởi động ta chọn lệnh *IO planing (PlanAhead) – post synthesis* trong cửa sổ Process mục *User Constraint*. Và chọn Yes để tạo file UCF với tên gọi ngầm định là *Board\_demo1.ucf*

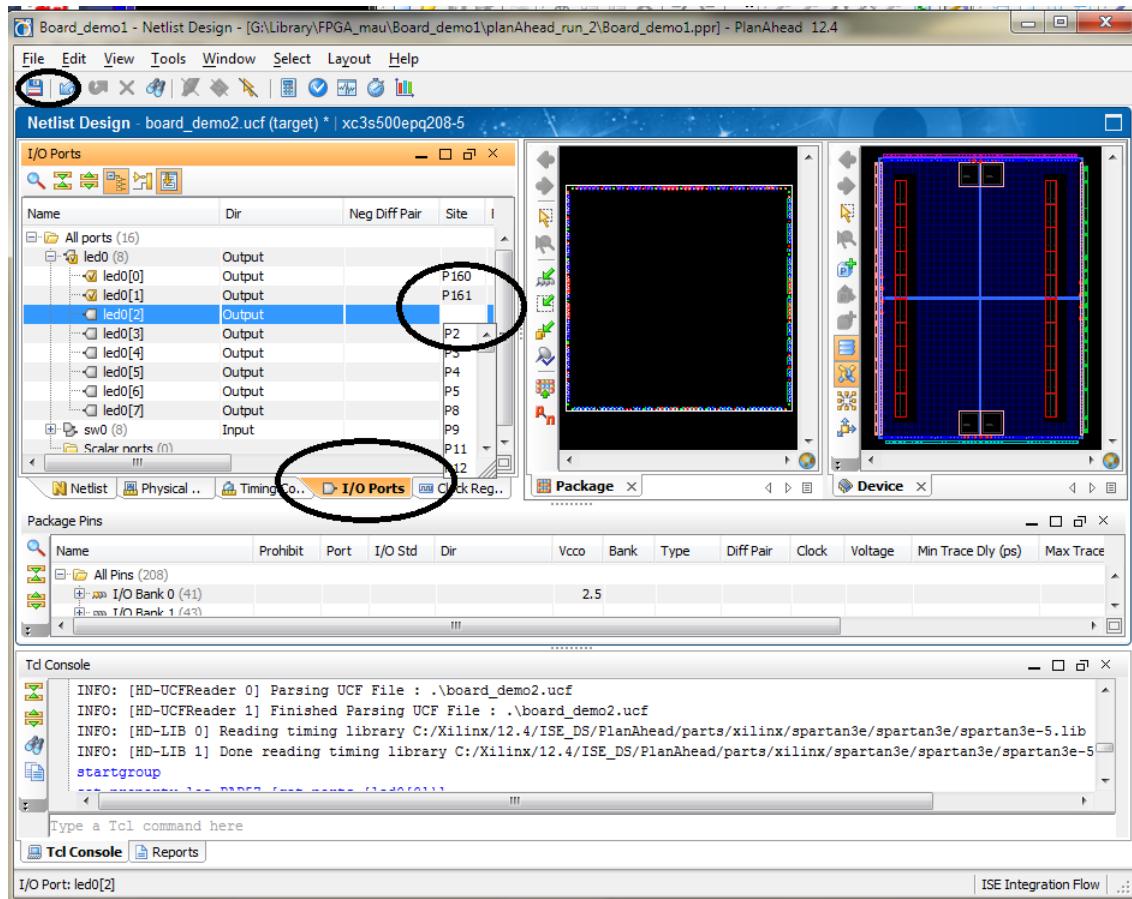


Khi có cửa sổ PlanAhead thực hiện gán địa chỉ cho từng chân tương ứng của thiết kế, ví dụ như ở hình dưới đây ta gán chân P100 cho đầu ra LED2 như trên hình( Ở cửa sổ IO port chọn cổng LED2 sau khi điền chân vào ô site trên cửa sổ IO Properties chọn Apply) trên mạch tương ứng với các vị trí đèn led trên mạch FPGA. Tương tự như vậy ta gán các chân còn lại như sau:

```

NET "led0[0]" LOC = P160;
NET "led0[1]" LOC = P161;
NET "led0[2]" LOC = P162;
NET "led0[3]" LOC = P163;
NET "led0[4]" LOC = P164;
NET "led0[5]" LOC = P165;
NET "led0[6]" LOC = P167;
NET "led0[7]" LOC = P168;
NET "sw0[0]" LOC = P171;
NET "sw0[1]" LOC = P172;
NET "sw0[2]" LOC = P177;
NET "sw0[3]" LOC = P178;
NET "sw0[4]" LOC = P179;
NET "sw0[5]" LOC = P180;
NET "sw0[6]" LOC = P181;
NET "sw0[7]" LOC = P185;

```



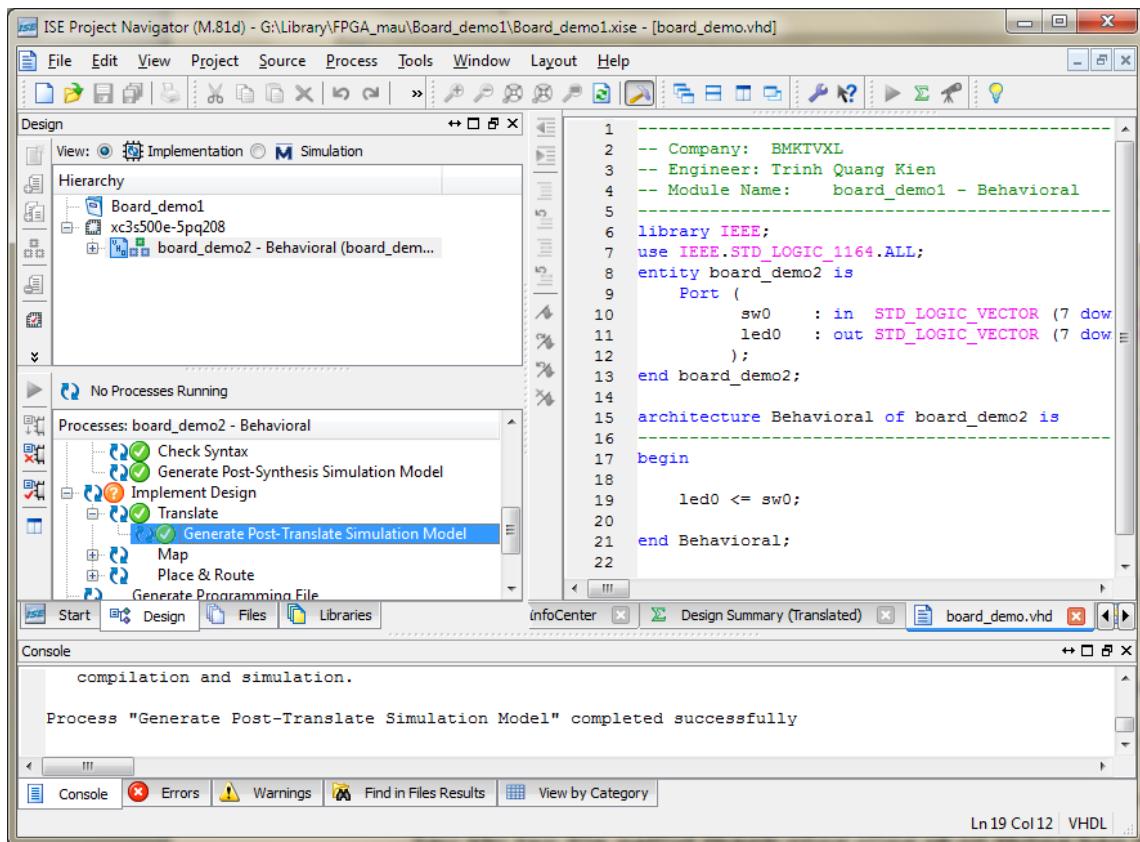
Sau khi hoàn tất việc gán chân *Save* để lưu lại thông tin gán chân và đóng PlanAhead, quay trở lại với chương trình ISE.

Thông tin gán chân được lưu ở trong một file text có tên là *board\_demo1.ucf*. Khi mở file này bằng trình soạn thảo sẽ thấy nội dung giống như ở trên. Một cách khác đơn giản hơn để gán chân là ta tạo một file UCF có tên trùng với tên khối thiết kế chính và soạn thảo với nội dung như vậy.

### Bước 11: Hiện thực hóa thiết kế

#### Bước 11.1 Translate

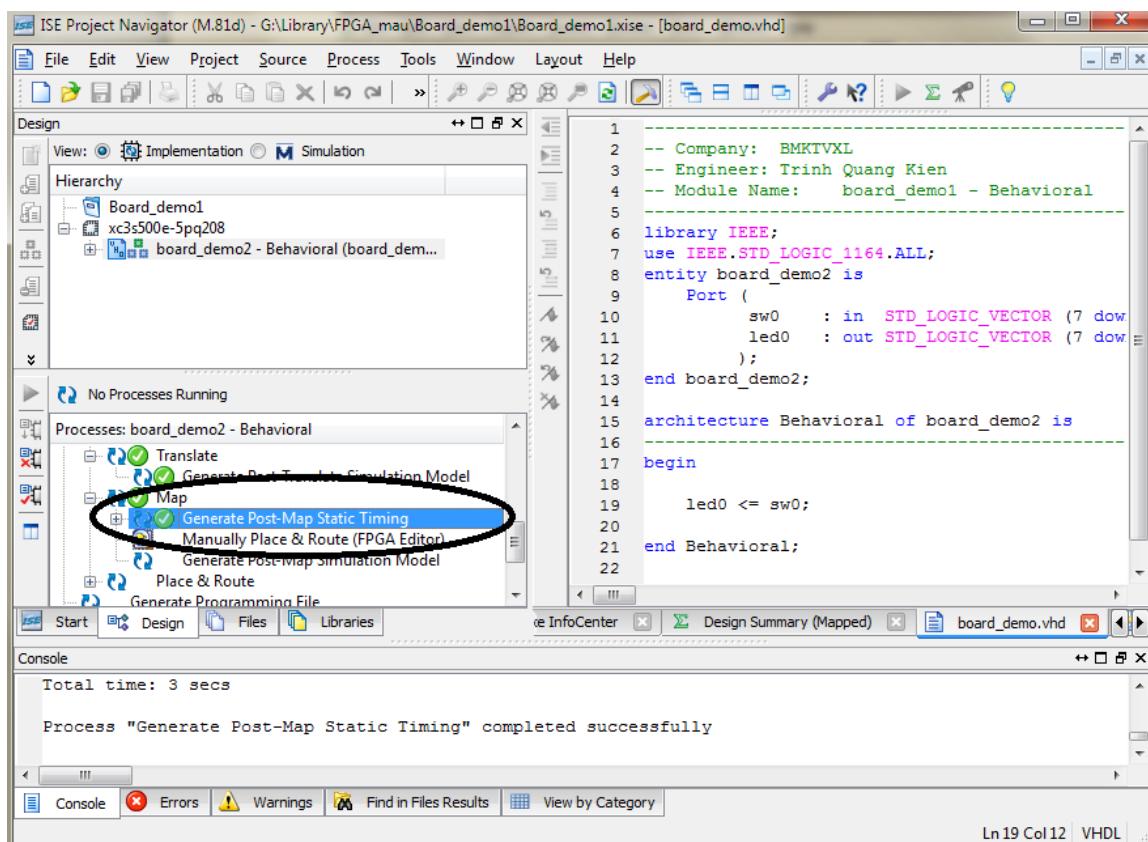
Để thực hiện Translate, chọn lệnh *Translate*, và nếu cần tạo file netlist sau Translate bằng lệnh tương ứng *Generate Post-translate simulation model* như ở hình dưới đây:



Sau khi tạo file netlist thành công cũng sẽ có thông báo tương ứng ở cửa sổ dòng lệnh: Process "Generate Post-Translate Simulation Model" completed successfully. Nếu ta mở lại thư mục *netgen* sẽ thấy xuất hiện thêm thư mục con *translate*, file VHDL tương ứng *board\_demo2\_translate.vhdl* là netlist tạo ra sau bước này, netlist này viết trên thư viện độc lập SIMPRIM. Ta cũng có thể tiến hành mô phỏng kiểm tra thiết kế sau translate giống bước mô phỏng kiểm tra sau Tổng hợp đã trình bày ở trên.

### Bước 11.2 Mapping

Để thực hiện quá trình Mapping chọn lệnh Map trong cửa sổ Process, quá trình sẽ được thực hiện tự động. Ở bước này chúng ta đã bắt đầu có thể kiểm tra các tham số về mặt thời gian đối với thiết kế, để thực hiện bước này (ngầm định khi Mapping bước này không thực hiện) chọn *Analyze Post-map Static Timing* như ở hình dưới đây



Kết quả sẽ được hiện ra ở một file text tương ứng bên cửa sổ VIEW/EDITOR, đối với ví dụ này ta thu được bảng thời gian trễ như sau:

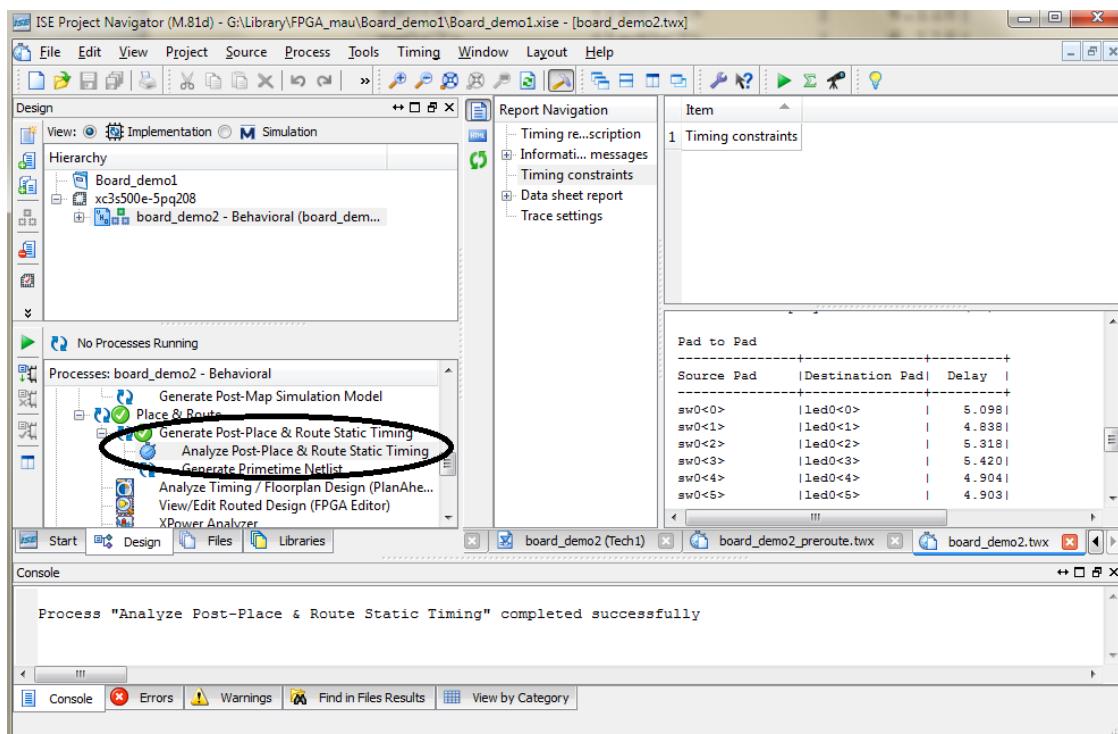
Data Sheet report:

Pad to Pad			
Source Pad	Destination Pad	Delay	
sw0<0>	led0<0>	4.118	
sw0<1>	led0<1>	4.118	
sw0<2>	led0<2>	4.118	
sw0<3>	led0<3>	4.118	
sw0<4>	led0<4>	4.118	
sw0<5>	led0<5>	4.118	
sw0<6>	led0<6>	4.118	
sw0<7>	led0<7>	4.118	

Sau khi maping cũng có thể tạo file netlist để mô phỏng kiểm tra chức năng bằng lệnh *Generate Post-map simulation model* tương tự như với các bước trên đã làm.

### Bước 11.3 Routing

Bước cuối cùng để sẵn sàng tạo ra file bitstream để nạp vào FPGA. Ở bước này ta cũng thực hiện các bước cơ bản như đối với các bước đã liệt kê ở phần Mapping. Điểm khác biệt là ở đây kết quả Analyze về mặt thời gian là chính xác khi mà các khối đã thực sự kết nối với nhau.



Kết quả về mặt thời gian thể hiện ở bảng sau:

Data Sheet report:

All values displayed in nanoseconds (ns)		
Pad to Pad		
Source Pad	Destination Pad	Delay
SW1	LED1	5.095
SW2	LED2	4.835
SW3	LED3	4.835
SW4	LED4	4.835
SW5	LED5	4.835
SW6	LED6	4.835
SW7	LED7	4.818

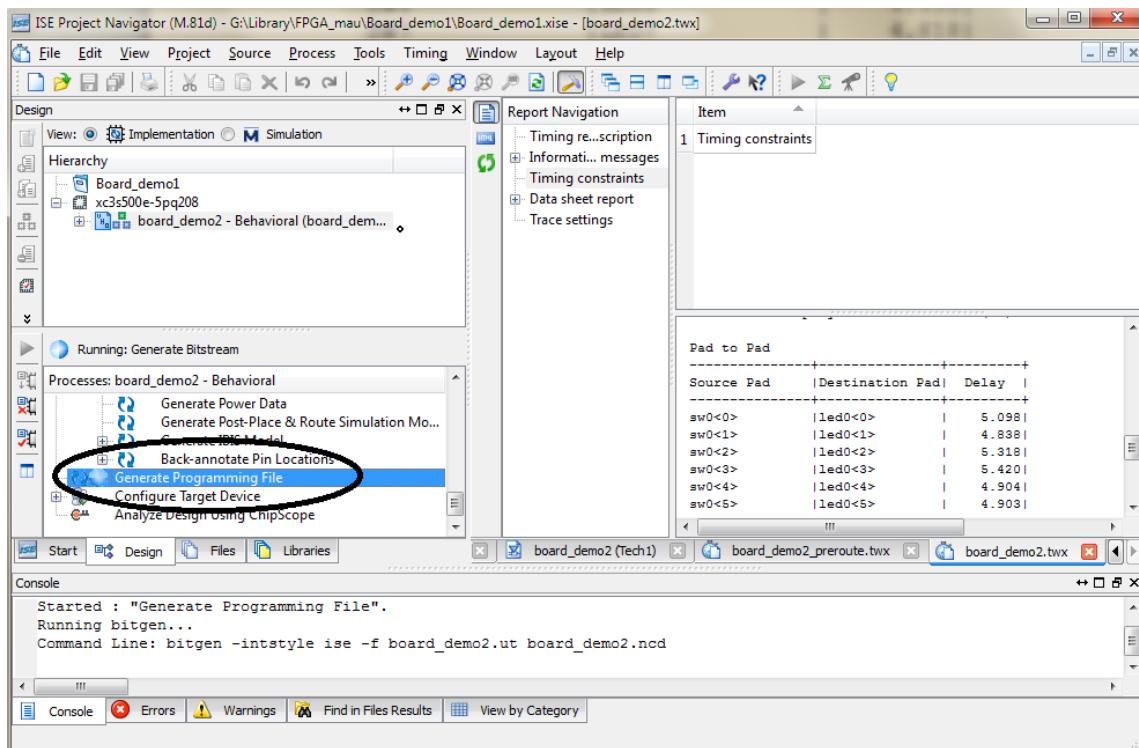
SW8

| LED8

| 4 . 853 |

## Bước 12: Tạo cấu hình FPGA

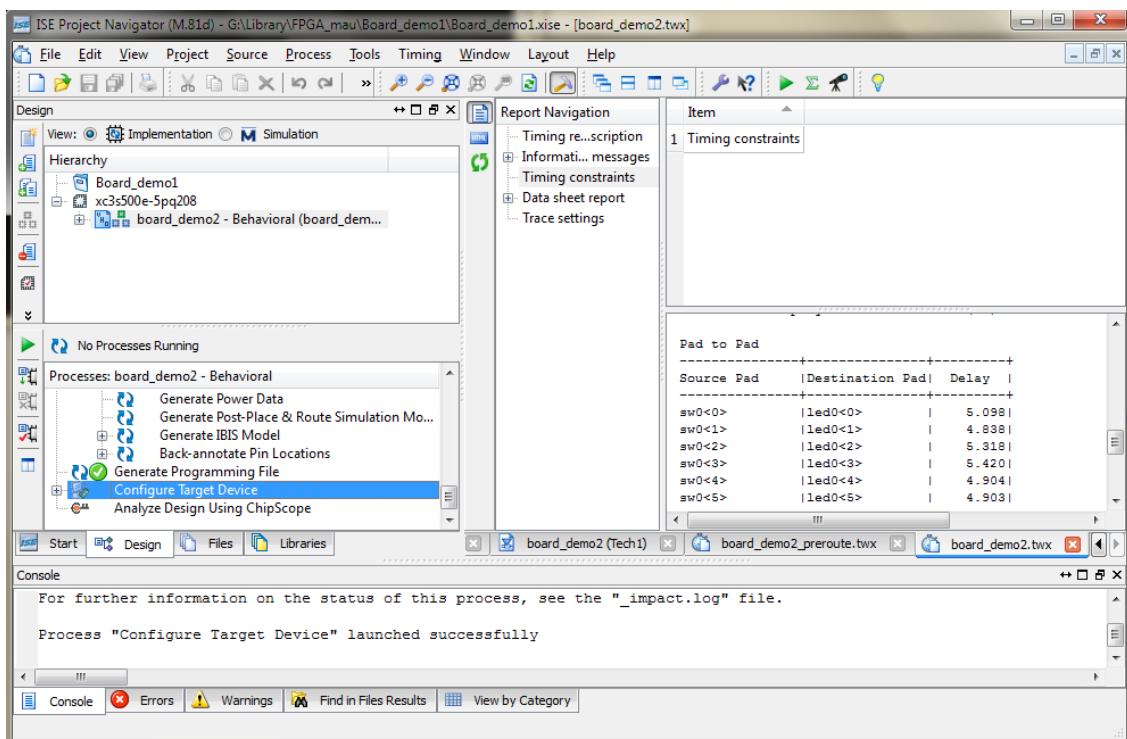
Để tạo tệp *bitstream* để cấu hình FPGA chọn lệnh Generate Programming File như hình dưới đây



Sau quá trình này 1 file có tên *board\_demo1.bit* được tạo ra trong thư mục chứa thiết kế. Thực hiện các bước sau để nạp cấu hình FPGA vào mạch thật.

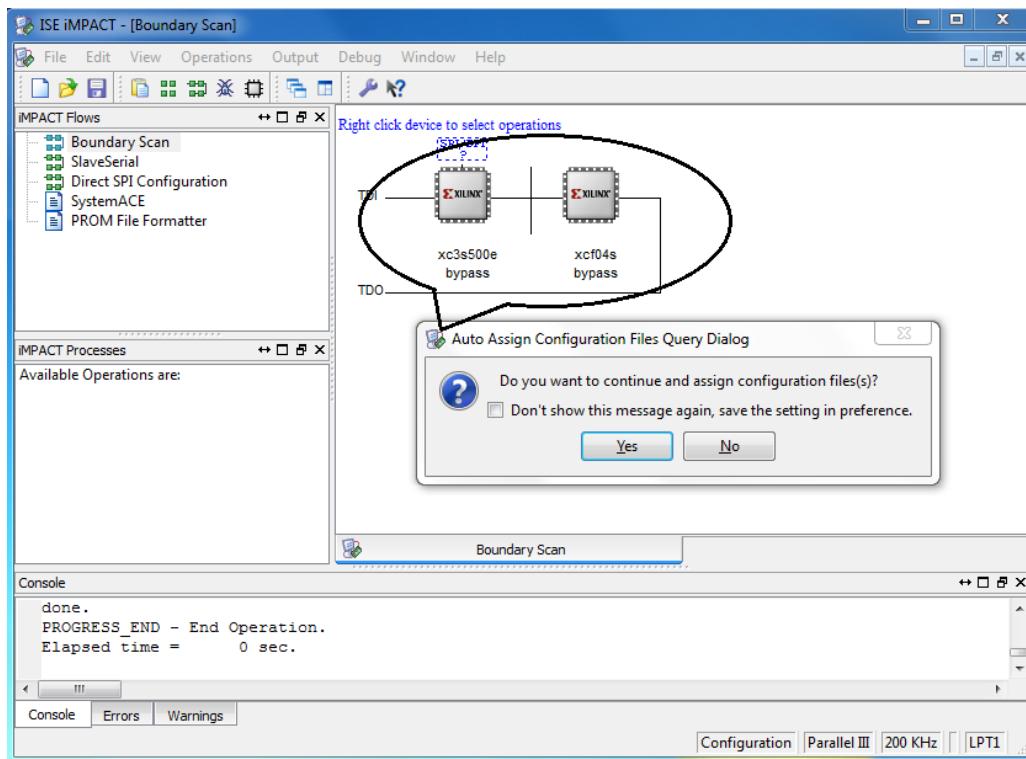
## Bước 13. Nạp cấu hình vào FPGA.

Để nạp cấu hình vào file FPGA cần phải có kit FPGA thật được kết nối với máy tính bằng dây nạp JTAG (thông thường qua cổng LPT hoặc USB). Tại cửa sổ Process chọn *Configure Target Device*

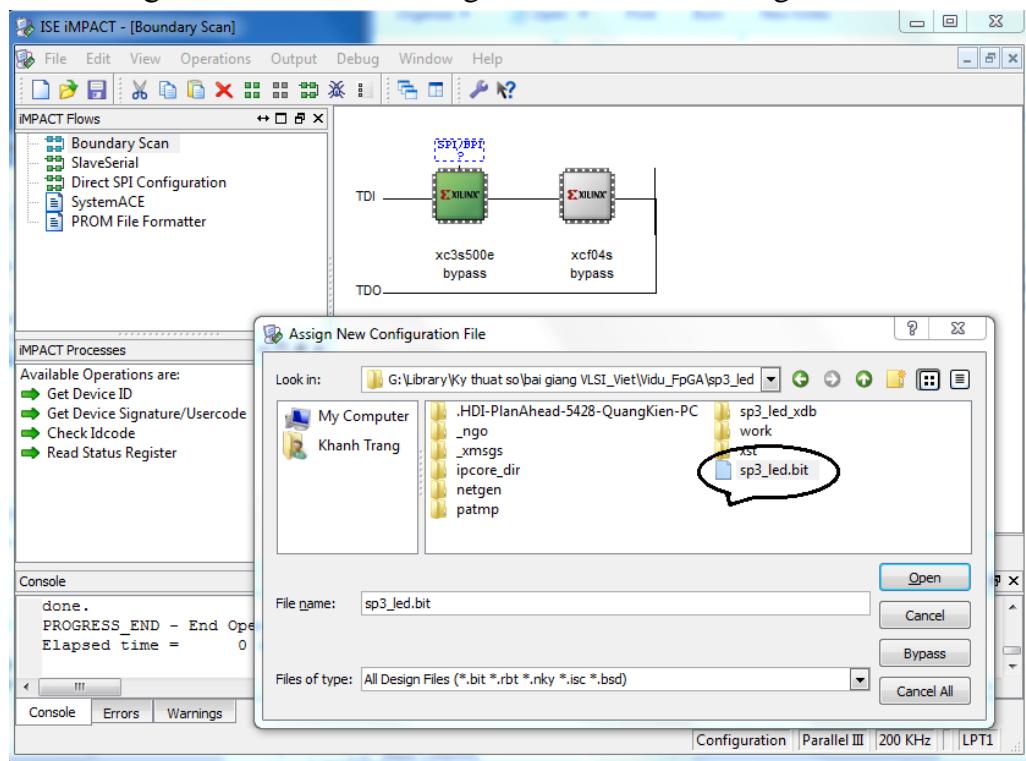


Lệnh này khởi tạo chương trình *ISE imPACT*, chương trình sẽ có thông báo về việc tạo file nạp cấu hình, nhấn *OK* bỏ qua hộp thoại này. Giao diện chương trình *imPACT* như ở dưới đây.

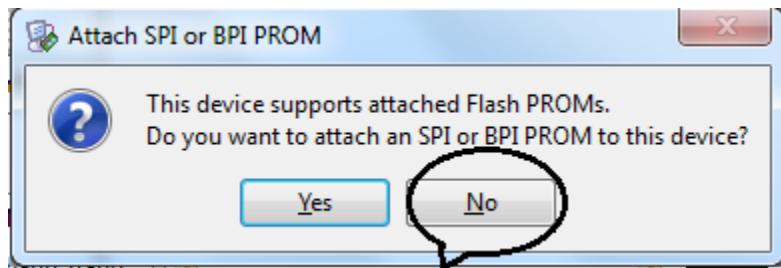
Bên cửa sổ *imPACT Flow* chọn *Boundary Scan*. Click chuột phải bên cửa sổ chính chọn *Initial Chain*. Nếu thành công sẽ thấy xuất hiện biểu tượng FPGA và một khôi ROM như ở hình dưới đây.



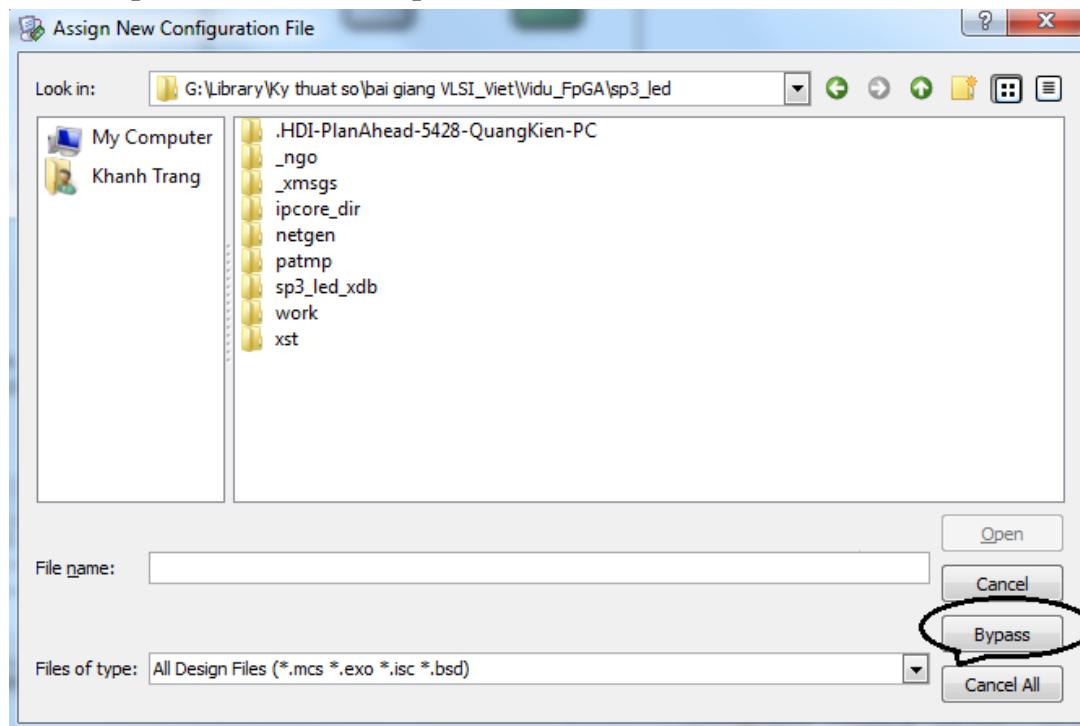
Chương trình sẽ hỏi có muốn gán file cấu hình không chọn Yes.



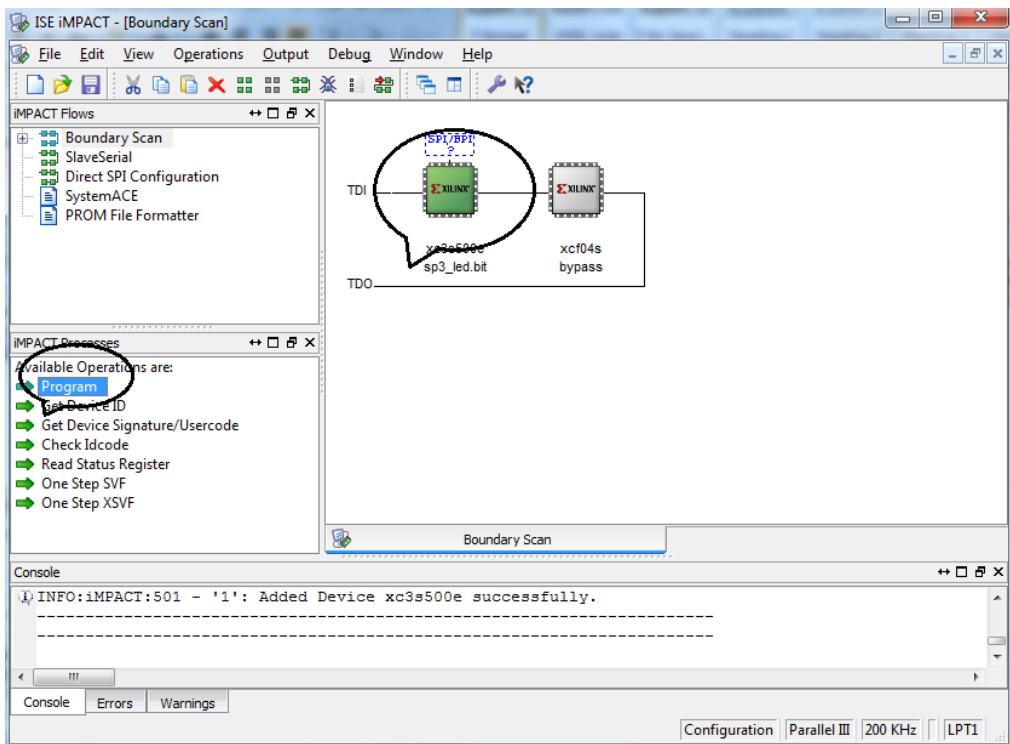
Tại cửa sổ này chọn file bitstream là sp3\_led.bit, chọn Open. Hộp thoại tiếp theo chọn No



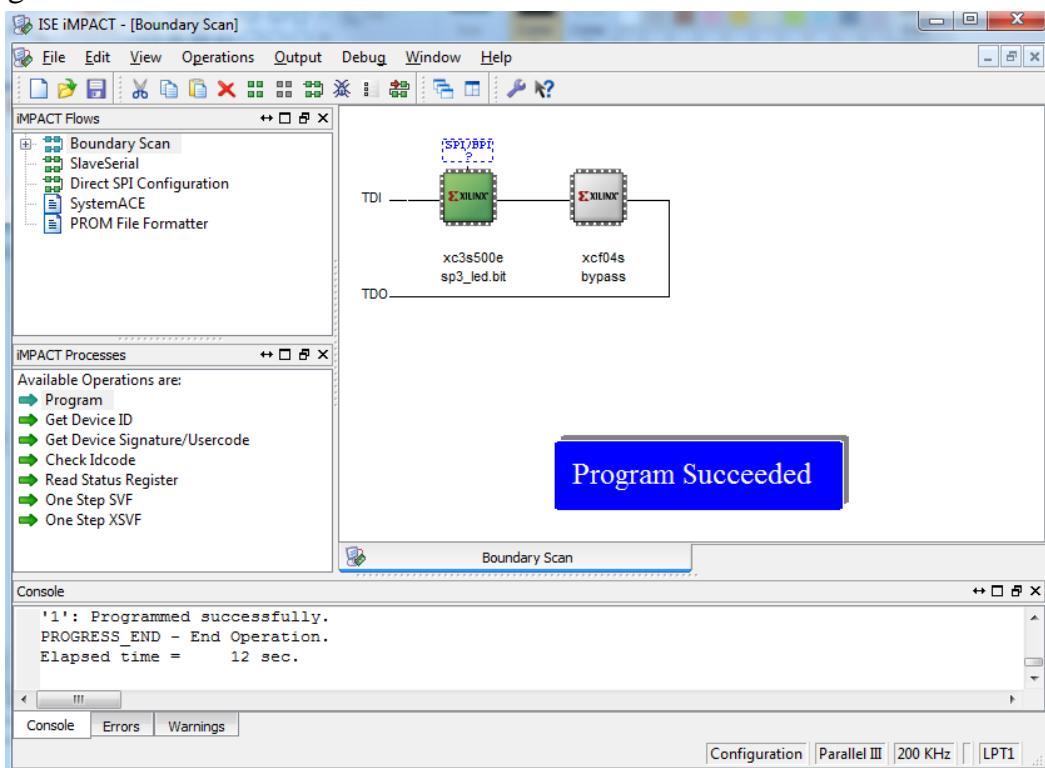
Chương trình tiếp tục hỏi nạp cấu hình vào ROM, chọn Bypass, cấu hình sẽ được nạp vào FPGA trực tiếp.



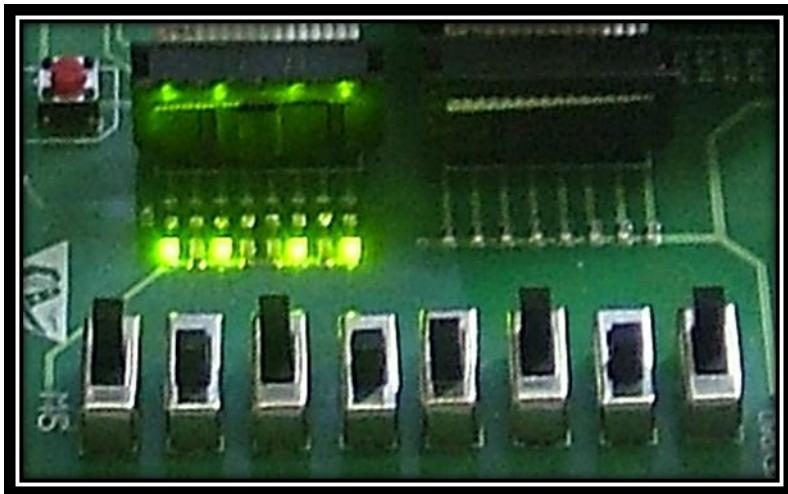
Cửa sổ tiếp theo thông báo về cấu hình đã chọn, ấn *OK* để tiếp tục quay trở về *ISE imPACT*, Click chuột vào biểu tượng Spartan FPGA và chọn lệnh *Program* trong cửa sổ *imPACT Process*.



Chương trình tiến hành nạp cấu hình vào FPGA, nếu quá trình thành công sẽ có thông báo PROGRAM SUCCESSED và đèn DONE trên mạch FPGA sẽ sáng.



Có thể kiểm tra trực tiếp trên mạch thiết kế bằng cách điều chỉnh các SWITCH và quan sát các LED tương ứng.



### 3. Bài tập và câu hỏi sau thực hành

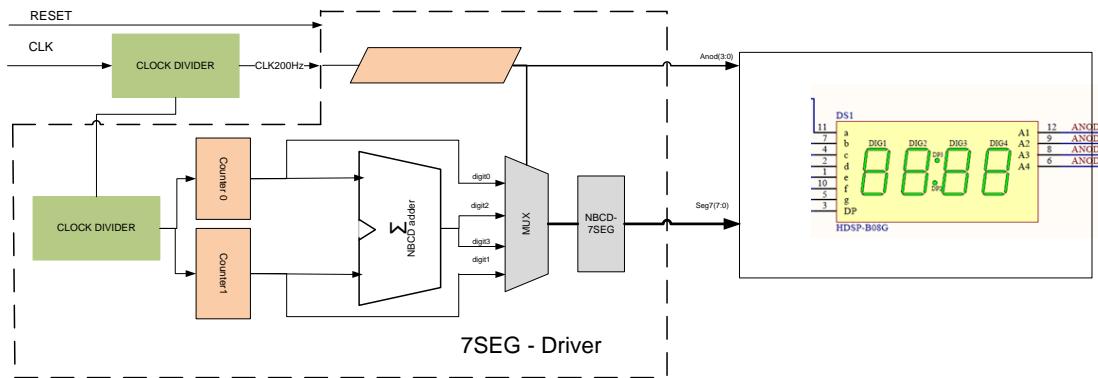
1. Thay đổi file nguồn VHDL theo hướng thay đổi sự phụ thuộc của đầu vào và đầu ra bằng các hàm logic cơ bản AND, OR, XOR ..., thực hiện các bước tổng hợp và hiện thực thiết kế như trên, quan sát kết quả thu được trên mạch thật
2. Thiết kế khối chia tần số, chia tần số có sẵn trên mạch ra tần số cỡ 1Hz có thể quan sát được bằng LED.
3. Thiết kế thanh ghi dịch 8 bit dịch bằng xung Clock 1Hz để quan sát kết quả dịch trên mạch.
4. Sử dụng các khối Led và Switch thiết kế và kiểm tra bộ cộng 4 bit trên mạch.

## Bài 2: Thiết kế khối giao tiếp với 4x7Seg -digits

### 1. Mục đích

Mô hình cách thức xây dựng một thiết kế mạch dãy đơn giản trên FPGA thông qua ví dụ điều khiển khói LED 7 thanh gồm 4 ký tự. Yêu cầu sinh viên phải có kỹ năng cơ bản về VHDL, biết cách sử dụng Xilinx ISE.

### 2. Mô tả hoạt động của khói thiết kế



Sơ đồ khói thiết kế được trình bày như hình vẽ trên. Khối Led 7 đoạn trên mạch có tất cả 4 ký tự số. Nguyên lý cơ bản để sử dụng các led này là sử dụng chân ANOD chung của các đèn để bật hoặc tắt với một tần số đủ lớn làm cho mắt thường không phân biệt được. Giới hạn tối thiểu là 25lần/1s với 1 Led nhưng thực nghiệm cho thấy quét ở tốc độ 50lần/1s với 1Led cho kết quả hiển thị tốt nhất.

Khối quét led được thiết kế là một thanh ghi dịch vòng 4 bit, thanh ghi này được khởi tạo ngầm định giá trị 0111 và dịch với xung clock 200Hz lấy từ khói chia tần.

Dữ liệu cho các Led gồm 7 giá trị a, b, c,d, e ,f g, h và p cho hiển thị dấu chấm. Dữ liệu của các LED từ 0 đến 3 này được đưa vào tương ứng với thời điểm tín hiệu ANOD tương ứng của đèn bằng 0. Vì vậy tín hiệu quét ANOD còn được sử dụng để điều khiển khói chọn kênh chọn 1 trong 4 giá trị đầu vào cho các LED. Để minh họa trong ví dụ này ta thiết kế một khói cộng NBCD và hai bộ đếm, tần số đếm của bộ đếm nhỏ hơn nhiều so với tần số quét và thường đê giá trị cỡ 1-3Hz. Vì vậy ta dùng thêm một khói chia tần để chia tần số 200Hz ra tần số 1Hz. Ở đây ta dùng 2 bộ đếm NBCD nối tầng, một bộ đếm hàng chục và một bộ đếm hàng đơn vị.

Khối cộng NBCD sẽ cộng giá trị các bộ đếm và đưa ra kết quả là số NBCD có 2 chữ số. Giá trị 2 bộ đếm và kết quả này được đưa ra khói chọn kênh để hiển thị trên LED.

Giá trị số sau khói chọn kênh có mã NBCD vì vậy ta sẽ dùng thêm một khói NBCD – 7SEG để giải mã trước khi gửi tới LED.

### 3. Phần thực hành

Các thao tác cơ bản để làm việc với Xilinx ISE đã được cung cấp ở bài số 1. Dưới đây chỉ liệt kê nội dung của các khói thiết kế VHDL.

#### Bước 1: Chuẩn bị mã nguồn thiết kế.

Khối chia tần clk\_div20.vhd:

```
-----
-- Company : BM KTVXL
-- Engineer: Trinh Quang Kien
-----
LIBRARY ieee;
USE ieee.Std_logic_1164.ALL;
USE ieee.Std_logic_unsigned.ALL;
-----
ENTITY clk_div20 IS
  GENERIC
    (
      baudDivide : std_logic_vector(19 downto 0) := 
x"00000"
    );
  PORT(
    clk_in : in    std_logic;
    clk_out : inout Std_logic
  );
END clk_div20;
-----
ARCHITECTURE rtl OF clk_div20 IS
  SIGNAL cnt_div : Std_logic_vector(19 DOWNTO 0);
BEGIN
  process (clk_in, cnt_div)
  begin
    if (clk_in = '1' and clk_in'event) then
      if (cnt_div = baudDivide) then
        cnt_div <= x"00000";
      else
        cnt_div <= cnt_div + 1;
      end if;
    end if;
  end;
```

```

    end process;

    process (cnt_div, clk_out, clk_in)
    begin
        if clk_in = '1' and clk_in'event then
            if cnt_div = baudDivide then
                clk_out <= not clk_out;
            end if;
        end if;
    end process;
END rtl;
-----
```

Bộ đếm BCD counter10.vhd sử dụng tham số tĩnh để thiết lập số bit cho giá trị đếm:

```

-----  

-- Company : BM KTVXL  

-- Engineer: Trinh Quang Kien  

-----  

LIBRARY ieee;  

USE ieee.Std_logic_1164.ALL;  

USE ieee.Std_logic_unsigned.ALL;  

-----  

ENTITY counter10 IS
    PORT (
        clk          : in  std_logic;
        reset        : in  std_logic;
        clock_enable : in  std_logic;
        count_out    : out std_logic_vector(3 DOWNTO 0));
END counter10;  

-----  

ARCHITECTURE rtl OF counter10 IS
    SIGNAL cnt : std_logic_vector(3 DOWNTO 0);
BEGIN
    PROCESS (clk, reset)
    BEGIN
        if reset = '1' then
            cnt <= (others =>'0');
        elsif rising_edge(clk) then
            if clock_enable = '1' then
                if cnt = x"9" then
                    cnt <= x"0";
                else
                    cnt <= cnt + 1;
                end if;
            end if;
        end if;
    end;
```

```

        END PROCESS;
        count_out <= cnt;
END rtl;
-----
Khối cộng NBCD bcd_adder.vhd:
-----
-- Company : BM KTVXL
-- Engineer: Trinh Quang Kien
-- Module Name:      bcd_adder - Behavioral
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
-----
entity bcd_adder is
    Port (
        a : in STD_LOGIC_VECTOR (3 downto 0);
        b : in STD_LOGIC_VECTOR (3 downto 0);
        s1 : out STD_LOGIC_VECTOR (3 downto 0);
        s2 : out STD_LOGIC_VECTOR (3 downto 0));
end bcd_adder;

architecture Behavioral of bcd_adder is
signal opa, opb, sum : STD_LOGIC_VECTOR (4 downto 0);
signal temp           : STD_LOGIC_VECTOR (3 downto 0);

begin
    opa <= '0' & a;
    opb <= '0' & b;
    sum <= opa + opb;
    temp <= sum(3 downto 0) - "1010";
    process (sum, temp)
    begin
        if sum < "1010" then
            s2 <= sum(3 downto 0);
            s1 <= "0000";
        else
            s1 <= "0001";
            s2 <= temp(3 downto 0);
        end if;
    end process;
end Behavioral;
-----
Khối quét LED scan_digit.vhd
-----
```

```

-- Company : BM KTVXL
-- Engineer: Trinh Quang Kien
-- Module Name:      scan_digit - Behavioral
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-----
entity scan_digit is
    Port (
        CLK      : in      STD_LOGIC;
        ANOD    : out     STD_LOGIC_VECTOR(3 downto 0);
        RESET   : in      STD_LOGIC);
end scan_digit;
-----

architecture Behavioral of scan_digit is
signal anod_sig : std_logic_vector(3 downto 0);

begin
    scan: process (CLK, RESET)
begin
    if RESET = '1' then
        ANOD_sig <= "1110";
    elsif CLK = '1' and CLK'event then
        ANOD_sig <= ANOD_sig(0) & ANOD_sig(3 downto 1);
    end if;
end process scan;

    ANOD <= ANOD_sig;
end Behavioral;
-----
Khởi động quát numeric_led.vhd
-----
-- Company:    BM KTVXL
-- Engineer:   Trinh Quang Kien
-- Module Name:      numeric_led - Behavioral
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-----
```

```

entity numeric_led is
    Port
    (
        CLK      : in  STD_LOGIC;
        nRESET  : in  STD_LOGIC;
        ANOD    : out STD_LOGIC_VECTOR (3 downto 0);
        SEG7    : out STD_LOGIC_VECTOR (7 downto 0)
    );
end numeric_led;

architecture Behavioral of numeric_led is
-----
component counter10 IS
    PORT(
        clk          : in  std_logic;
        reset        : in  std_logic;
        clock_enable : in  std_logic;
        count_out    : out std_logic_vector(3 DOWNTO 0)
    );
END component;
-----
component clk_div20 IS
    GENERIC(baudDivide : std_logic_vector(19 downto 0)
:= x"00000");
    PORT(
        clk_in      : in  std_logic;
        clk_out     : inout Std_logic
    );
END component;
-----
component scan_digit is
    Port ( CLK      : in  STD_LOGIC;
            ANOD    : out STD_LOGIC_VECTOR(3 downto 0);
            RESET   : in  STD_LOGIC);
end component;
-----
component bcd_adder is
    Port ( a       : in  STD_LOGIC_VECTOR (3 downto 0);
            b       : in  STD_LOGIC_VECTOR (3 downto 0);
            s1      : out STD_LOGIC_VECTOR (3 downto 0);
            s2      : out STD_LOGIC_VECTOR (3 downto 0));
end component;
-----
component BCD_7seg is
    Port (
        nbcd : in  STD_LOGIC_VECTOR (3 downto 0);

```

```

        seg : out STD_LOGIC_VECTOR (7 downto 0));
end component;
-----
signal reset, cnt1_enable          : std_logic;
signal clk0, clk1                 : std_logic;
signal nbcd0, nbcd1 : std_logic_vector(3 downto 0);

signal nbcd2, nbcd3 : std_logic_vector(3 downto 0);
signal nbcd      : STD_LOGIC_VECTOR(3 downto 0);
signal anod_sig, nanod : std_logic_vector(3 downto 0);
-----
begin
    reset <= not nreset;

cnt0: component counter10
    port map (clk1, reset, '1', nbcd0);

    cnt1_enable <= '1' when nbcd0 = "1001" else
                    '0';
cnt1: component counter10
    port map (clk1, reset, cnt1_enable, nbcd1);

bcda: component bcd_adder
    port map (nbcd0, nbcd1, nbcd2, nbcd3);

    nanod <= not anod_sig;
    nbcd(0) <= (nanod(0) and nbcd0(0)) or (nanod(1) and
nbcd1(0))
                or(nanod(2) and nbcd2(0)) or (nanod(3) and
nbcd3(0));
    nbcd(1) <= (nanod(0) and nbcd0(1)) or (nanod(1) and
nbcd1(1))
                or(nanod(2) and nbcd2(1)) or (nanod(3) and
nbcd3(1));
    nbcd(2) <= (nanod(0) and nbcd0(2)) or (nanod(1) and
nbcd1(2))
                or(nanod(2) and nbcd2(2)) or (nanod(3) and
nbcd3(2));
    nbcd(3) <= (nanod(0) and nbcd0(3)) or (nanod(1) and
nbcd1(3))
                or(nanod(2) and nbcd2(3)) or (nanod(3) and
nbcd3(3));

bcd_seg0: component BCD_7seg

```

```

        port map (nbcd, SEG7);
clk_div0 : component clk_div20
    generic map (x"1D4C0")
    port map (clk, clk0);
clk_div1 : component clk_div20
    generic map (x"00042")
    port map (clk0, clk1);
scan0: scan_digit
    port map (clk0, anod_sig, reset);
ANOD <= anod_sig;
end Behavioral;
-----
```

## Bước 2: Tạo Project

Tạo một Project trên FPGA có tên là Board\_demo2, bô xung các file VHDL trên vào Project, thực hiện tổng hợp thiết kế, nếu quá trình tổng hợp thành công thì chuyển sang bước tiếp theo. Với code trên ta sẽ có kết quả tổng hợp như sau:

### Kết quả tổng hợp trên FPGA

```

=====
*          Final Report          *
=====
Device utilization summary:
-----
Selected Device : 3s500epq208-5

Number of Slices:      45  out of   4656    0%
Number of Slice Flip Flops: 54  out of   9312    0%
Number of 4 input LUTs:  88  out of   9312    0%
Number of IOs:          14
Number of bonded IOBs:  14  out of     158    8%
Number of GCLKs:        2   out of      24    8%
-----
Partition Resource Summary:
-----
No Partitions were found in this design.
-----
TIMING REPORT
Clock Information:
-----+-----+-----+
Clock Signal | Clock buffer(FF name) | Load |
-----+-----+-----+
CLK          | BUFGP                  | 21   |
clk_div0/clk_out1 | BUFG                   | 25   |
clk_div1/clk_out | NONE(cnt0/cnt_0)       | 8    |
```

```
-----+-----+
INFO:Xst:2169 - HDL ADVISOR - Some clock signals were
not automatically buffered by XST with BUFG/BUFR resources.
Please use the buffer_type constraint in order to insert
these buffers to the clock signals to help prevent skew
problems.
```

Timing Summary:

Speed Grade: -5

Minimum period: 4.259ns (Maximum Frequency: 234.811MHz)

Minimum input arrival time before clock: No path found

Maximum output required time after clock: 12.491ns

Maximum combinational path delay: No path found

Timing Detail:

All values displayed in nanoseconds (ns)

### Bước 3: Gán chân sau tổng hợp

Gán các chân cho thiết kế, để gán đúng chân cần xem phụ lục I, tạo một tệp có đuôi mở rộng numeric\_led.ucf với nội dung như sau:

```
INST "CLK_BUFGP" LOC = BUFGMUX_X2Y1;
NET "ANOD[0]" LOC = P78;
NET "ANOD[1]" LOC = P82;
NET "ANOD[2]" LOC = P84;
NET "ANOD[3]" LOC = P83;

NET "CLK" LOC = P80;
NET "SEG7[7]" LOC = P116;
NET "SEG7[6]" LOC = P106;
NET "SEG7[5]" LOC = P107;
NET "SEG7[4]" LOC = P108;
NET "SEG7[3]" LOC = P109;
NET "SEG7[2]" LOC = P112;
NET "SEG7[1]" LOC = P113;
NET "SEG7[0]" LOC = P115;
NET "nRESET" LOC = P90;
```

### Bước 4: Hiện thực hóa thiết kế

Hiện thực hóa thiết kế (Implementation) và đọc kết quả sau quá trình này.  
Kết quả về thời gian tĩnh sau Place & Route

Data Sheet report:

All values displayed in nanoseconds (ns)

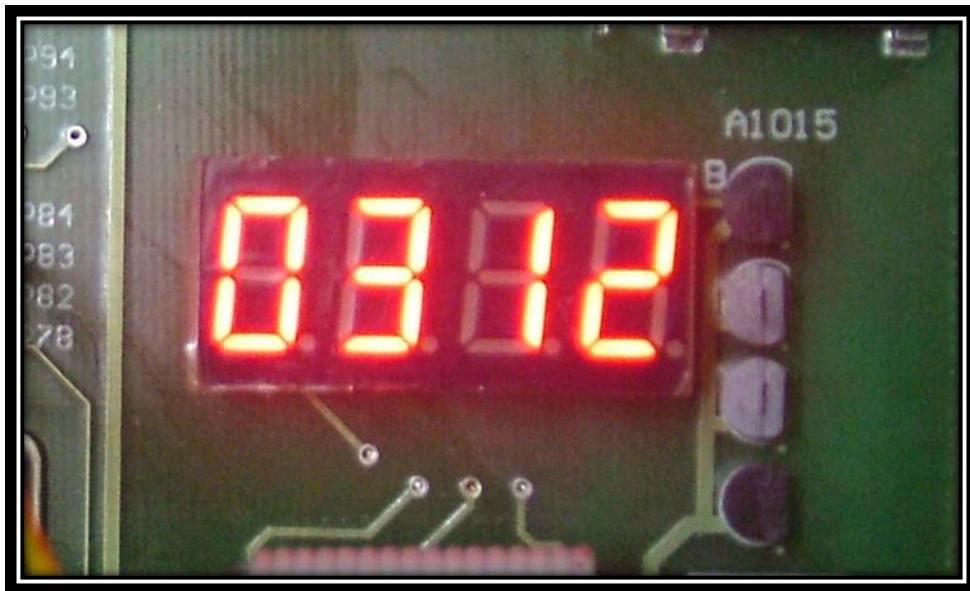
```

Clock to Setup on destination clock CLK
-----+-----+-----+-----+
      | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock|Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
-----+-----+-----+-----+
CLK          |   4.641|           |           |
-----+-----+-----+-----+

```

### Bước 5: Nạp cấu hình

Nạp kết file cấu hình numeric\_led.bit vào FPGA, Quan sát kết quả thu được.



#### 4. Bài tập và câu hỏi sau thực hành

1. Trình bày nguyên lý sử dụng Led 7 đoạn và chế độ quét Led
2. Thay đổi tốc độ quét và quan sát kết quả thu được với tần số quét 100Hz
3. Sửa thiết kế để có thể dùng Switch để điều chỉnh tốc độ quét led
4. Thay thế khối cộng NBCD bằng khối nhân NBCD trong thiết kế trên. Nạp và quan sát kết quả.

## **Phụ lục 5: CÁC BẢNG MÃ THÔNG DỤNG**

## 1. Mã ASCII điều khiển

Hệ 2	Hệ 10	Hệ 16	Viết tắt	Biểu diễn in được	Truy nhập bàn phím	Tên/Y tiếng Anh	nghĩa	Tên/Y nghĩa tiếng Việt
000 0000	0	00	NUL	<sup>N</sup> <sub>0</sub> <sub>L</sub>	<sup>^</sup> @	Null character	Kí tự rỗng	
000 0001	1	01	SOH	<sup>S</sup> <sub>0</sub> <sub>H</sub>	<sup>^</sup> A	Start of Header	Bắt đầu Header	
000 0010	2	02	STX	<sup>S</sup> <sub>T</sub> <sub>X</sub>	<sup>^</sup> B	Start of Text	Bắt đầu văn bản	
000 0011	3	03	ETX	<sup>E</sup> <sub>T</sub> <sub>X</sub>	<sup>^</sup> C	End of Text	Kết thúc văn bản	
000 0100	4	04	EOT	<sup>E</sup> <sub>0</sub> <sub>T</sub>	<sup>^</sup> D	End of Transmission	Kết thúc truyền	
000 0101	5	05	ENQ	<sup>E</sup> <sub>N</sub> <sub>Q</sub>	<sup>^</sup> E	Enquiry	Truy vấn	
000 0110	6	06	ACK	<sup>A</sup> <sub>C</sub> <sub>K</sub>	<sup>^</sup> F	Acknowledgement		
000 0111	7	07	BEL	<sup>B</sup> <sub>E</sub> <sub>L</sub>	<sup>^</sup> G	Bell	Chuông	
000 1000	8	08	BS	<sup>B</sup> <sub>S</sub>	<sup>^</sup> H	Backspace	Xoá ngược	
000 1001	9	09	HT	<sup>H</sup> <sub>T</sub>	<sup>^</sup> I	Horizontal Tab	Tab ngang	
000 1010	10	0A	LF	<sup>L</sup> <sub>F</sub>	<sup>^</sup> J	New Line	Xuống dòng	
000 1011	11	0B	VT	<sup>V</sup> <sub>T</sub>	<sup>^</sup> K	Vertical Tab	Tab dọc	
000 1100	12	0C	FF	<sup>F</sup> <sub>F</sub>	<sup>^</sup> L	Form feed		
000 1101	13	0D	CR	<sup>C</sup> <sub>R</sub>	<sup>^</sup> M	Carriage return		
000 1110	14	0E	SO	<sup>S</sup> <sub>0</sub>	<sup>^</sup> N	Shift Out		

000 1111	15	0F	SI	<sup>s<sub>I</sub></sup>	<sup>^O</sup>	Shift In	
001 0000	16	10	DLE	<sup>d<sub>L_E</sub></sup>	<sup>^P</sup>	Data Link Escape	
001 0001	17	11	DC1	<sup>d<sub>C_1</sub></sup>	<sup>^Q</sup>	Device Control 1 — oft. XON	
001 0010	18	12	DC2	<sup>d<sub>C_2</sub></sup>	<sup>^R</sup>	Device Control 2	
001 0011	19	13	DC3	<sup>d<sub>C_3</sub></sup>	<sup>^S</sup>	Device Control 3 — oft. XOFF	
001 0100	20	14	DC4	<sup>d<sub>C_4</sub></sup>	<sup>^T</sup>	Device Control 4	
001 0101	21	15	NAK	<sup>n<sub>A_K</sub></sup>	<sup>^U</sup>	Negative Acknowledgement	
001 0110	22	16	SYN	<sup>s<sub>Y_N</sub></sup>	<sup>^V</sup>	Synchronous Idle	
001 0111	23	17	ETB	<sup>e<sub>T_B</sub></sup>	<sup>^W</sup>	End of Trans. Block	
001 1000	24	18	CAN	<sup>c<sub>A_N</sub></sup>	<sup>^X</sup>	Cancel	
001 1001	25	19	EM	<sup>e<sub>M</sub></sup>	<sup>^Y</sup>	End of Medium	
001 1010	26	1A	SUB	<sup>s<sub>U_B</sub></sup>	<sup>^Z</sup>	Substitute	
001 1011	27	1B	ESC	<sup>e<sub>S_C</sub></sup>	<sup>^]</sup> [ hay ESC	Escape	
001 1100	28	1C	FS	<sup>f<sub>S</sub></sup>	<sup>^\ \\</sup>	File Separator	
001 1101	29	1D	GS	<sup>g<sub>S</sub></sup>	<sup>^]</sup>	Group Separator	
001 1110	30	1E	RS	<sup>r<sub>S</sub></sup>	<sup>^&amp;</sup>	Record Separator	
001 1111	31	1F	US	<sup>u<sub>S</sub></sup>	<sup>^_</sup>	Unit Separator	
111 1111	127	7F	DEL	<sup>d<sub>E_L</sub></sup>	DEL hay Backspace	Delete	

## 2. Mã ASCII hiển thị

Hệ 2 (Nhị phân)	Hệ 10 (Thập phân)	Hệ 16 (Thập lục phân)	Đồ họa (Hiển thị ra được)
010 0000	32	20	<u>Khoảng trống</u> ( <sup>s<sub>p</sub></sup> )
010 0001	33	21	!
010 0010	34	22	"
010 0011	35	23	#
010 0100	36	24	\$
010 0101	37	25	%
010 0110	38	26	&
010 0111	39	27	'
010 1000	40	28	(
010 1001	41	29	)
010 1010	42	2A	*
010 1011	43	2B	+
010 1100	44	2C	,
010 1101	45	2D	-
010 1110	46	2E	=
010 1111	47	2F	/
011 0000	48	30	0
011 0001	49	31	1
011 0010	50	32	2
011 0011	51	33	3
011 0100	52	34	4
011 0101	53	35	5
011 0110	54	36	6
011 0111	55	37	7

011 1000	56	38	8
011 1001	57	39	9
011 1010	58	3A	<u>:</u>
011 1011	59	3B	<u>:</u>
011 1100	60	3C	<
011 1101	61	3D	=
011 1110	62	3E	>
011 1111	63	3F	<u>?</u>
100 0000	64	40	<u>@</u>
100 0001	65	41	A
100 0010	66	42	B
100 0011	67	43	C
100 0100	68	44	D
100 0101	69	45	E
100 0110	70	46	F
100 0111	71	47	G
100 1000	72	48	H
100 1001	73	49	I
100 1010	74	4A	J
100 1011	75	4B	K
100 1100	76	4C	L
100 1101	77	4D	M
100 1110	78	4E	N
100 1111	79	4F	O
101 0000	80	50	P
101 0001	81	51	Q
101 0010	82	52	R

101 0011	83	53	S
101 0100	84	54	T
101 0101	85	55	U
101 0110	86	56	V
101 0111	87	57	W
101 1000	88	58	X
101 1001	89	59	Y
101 1010	90	5A	Z
101 1011	91	5B	[
101 1100	92	5C	\`
101 1101	93	5D	]
101 1110	94	5E	^
101 1111	95	5F	-
110 0000	96	60	`-
110 0001	97	61	a
110 0010	98	62	b
110 0011	99	63	c
110 0100	100	64	d
110 0101	101	65	e
110 0110	102	66	f
110 0111	103	67	g
110 1000	104	68	h
110 1001	105	69	i
110 1010	106	6A	j
110 1011	107	6B	k
110 1100	108	6C	l
110 1101	109	6D	m

110 1110	110	6E	n
110 1111	111	6F	o
111 0000	112	70	p
111 0001	113	71	q
111 0010	114	72	r
111 0011	115	73	s
111 0100	116	74	t
111 0101	117	75	u
111 0110	118	76	v
111 0111	119	77	w
111 1000	120	78	x
111 1001	121	79	y
111 1010	122	7A	z
111 1011	123	7B	{
111 1100	124	7C	l
111 1101	125	7D	}
111 1110	126	7E	$\simeq$

### 3. Bảng mã ký tự cho LCD 1602A

Higher 4bit Lower 4bit	0000	0010 0011 0100 0101 0110 0111	1010 1011 1100 1101 1110 1111
xxxx0000		Q@P^F	-9Eop
xxxx0001		!1AQaaq	a7848g
xxxx0010		"2BRbr	54Wx88
xxxx0011		*3CScs	u9TEsc
xxxx0100		\$4DTdt	.ITfu0
xxxx0101		%5EUeu	.9*1c0
xxxx0110		&6FUfV	2023o2
xxxx0111		?7GU9w	7*87gR
xxxx1000		(8HKhx	49*Ujx
xxxx1001		)9IYiy	uTJu~u
xxxx1010		*:JZjz	zDnV1?
xxxx1011		+;KCkC	*7C0*x
xxxx1100		:<L#11	7577*4
xxxx1101		==M1m>	u782k*
xxxx1110		#>N^m*	a7b^R
xxxx1111		/20_o*	w97P8

## **TÀI LIỆU THAM KHẢO**

- 1- Lê Xuân Bằng - Kỹ thuật số (Tập 1) - NXB Khoa học kỹ thuật – Năm 2008
- 2- Đỗ Xuân Tiến - Kỹ thuật Vi xử lý và lập trình Assembly –NXB Khoa học kỹ thuật – Năm 2002
- 3- Nguyễn Thúy Vân - Thiết kế logic mạch số - NXB Khoa học kỹ thuật - Năm 2005
- 4- Nguyễn Linh Giang - Thiết kế mạch bằng máy tính –NXB Khoa học kỹ thuật - Năm 2005
- 5- IEEE VHDL Standard reference 2002 – Năm 2002.
- 6- Douglas L Perry - VHDL Programming by Example 4th Edition - 2002
- 7- S. S. Limaye Digital Design with VHDL – 2002
- 8- Enoch O. Hwang - Microprocessor Design Principles and Practices with VHDL - 2004
- 9- Mark Balch - Complete digital design - 2003
- 10-Maxfield, Brawn - The Definitive Guide to How Computers Do Math - Wiley Publisher - 2005
- 11-Behrooz Parhami - Computer ArithmeticAlgorithms and Hardware Designs – Oxford University Press - 2000
- 12-Mi Lu - Arithmetic and Logic in Computer Systems – John Wiley & Sons Inc. Publication - 2004
- 13-Milos D. Ercegovac and Tomas Lang - Digital Arithmetic - San Francisco Morgan Kaufmann Publishers - 2004
- 14-IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Standard No. 754. American National Standards Institute –Washington, DC - 1985.
- 15-Wakerly J.F - Digital design principles and practices - 1999
- 16-Wai-Kai Chen - The VLSI Handbook, 2<sup>nd</sup> Edition - University of Illinois - 2007
- 17-Naveed A. Sherwani - Algorithms for VLSI Physical Design Automation, 3<sup>rd</sup> Edition - Kluwer Academic Publishers - 2002
- 18-David A. Paterson, John L. Hennessy - Computer Organization and Design,Third Edition - 2005.
- 19-Uwe Meyer-Baese -Digital Signal Processing with FPGA – Springer - 2007
- 20-Xilinx - Spartan-3 Generation FPGA User Guide (ug331.pdf)
- 21-Xilinx - Spartan-3E FPGA Family Datasheet (ds312.pdf)

- 22-MIPS Technology - MIPS Instruction Set Reference Vol I - 2003  
23-Xilinx - Spartan 3A/3AN Starter Kit User Guide (ug334.pdf)  
24-16COM / 40SEG DRIVER & CONTROLLER FOR DOT MATRIX LCD  
(ks0066u.pdf)
- 25-<http://www.csee.umbc.edu/portal/help/VHDL/>
- 26-<http://www.model.com>
- 27-[http://www.doulos.com/knowhow/vhdl\\_designers\\_guide/](http://www.doulos.com/knowhow/vhdl_designers_guide/)
- 28-<http://www.synopsys.com>
- 29-<http://www.xilinx.com>
- 30-<http://vlsiip.com/tutorial/>
- 31-<http://www.opencores.org/>
- 32-<http://www.asic-world.com/>
- 33-<http://www.cs.umbc.edu/portal/help/VHDL/stdpkg.html>
- 34-<http://opencores.org>
- 35-<http://en.wikipedia.org/wiki/CORDIC>
- 36-[http://en.wikipedia.org/wiki/Advanced\\_Encryption\\_Standard](http://en.wikipedia.org/wiki/Advanced_Encryption_Standard)
- 37-[http://en.wikipedia.org/wiki/Data\\_Encryption\\_Standard](http://en.wikipedia.org/wiki/Data_Encryption_Standard)
- 38-<http://en.wikipedia.org/wiki/RSA>