

ĐẠI HỌC QUỐC GIA TP HỒ CHÍ MINH  
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

2008



THIẾT KẾ HỆ THỐNG SOC TRÊN FPGA

Tác giả: Trần Thị Điểm

Phạm Hoài Luân

Nguyễn Văn Tình

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA  
THÀNH PHỐ HỒ CHÍ MINH – NĂM 2024

ĐẠI HỌC QUỐC GIA TP HỒ CHÍ MINH  
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

ĐHQT



THIẾT KẾ HỆ THỐNG SOC TRÊN FPGA

Tác giả: Trần Thị Điểm

Phạm Hoài Luân

Nguyễn Văn Tình

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA  
THÀNH PHỐ HỒ CHÍ MINH – NĂM 2024

## LỜI NÓI ĐẦU

Giáo trình Thiết kế hệ thống SoC trên FPGA được biên soạn nhằm cung cấp cho sinh viên những kiến thức từ cơ bản đến nâng cao về thiết kế hệ thống nhúng trên chip FPGA. Sinh viên hiểu được SoC là gì, SoC gồm những thành phần cơ bản nào, và nắm được quy trình thiết kế SoC trên FPGA. Hơn thế nữa, sinh viên sẽ được cung cấp các kỹ năng tối ưu thiết kế về tốc độ, tài nguyên thiết kế.

Đối tượng chủ yếu của tài liệu này là sinh viên các chuyên ngành kỹ thuật máy tính, điện tử - viễn thông, vĩ điện tử. Kiến thức người đọc cần trang bị trước để nắm bắt tốt nội dung của giáo trình này gồm các môn học: Nhập môn mạch số, lập trình C/C++, kiến trúc máy tính, ngôn ngữ thiết kế HDL.

Tài liệu được chia thành 6 chương:

Chương 1: Tổng quan hệ thống SoC.

Chương 2: Vi xử lý trên SoC FPGA.

Chương 3: Thiết kế bộ nhớ.

Chương 4: Hệ thống kết nối.

Chương 5: Tiêu chuẩn đánh giá hệ thống soc trên FPGA

Chương 6: Tổng hợp cấp cao trong SoC FPGA.

Trong quá trình sử dụng tài liệu này, nếu bạn đọc có bất kỳ góp ý nào về nội dung, hình thức trình bày hay các góp ý khác về tài liệu, xin vui lòng gửi góp ý về thư điện tử: [ttdiem@uit.edu.vn](mailto:ttdiem@uit.edu.vn). Đóng góp của quý bạn đọc là nguồn tham khảo hữu ích để nhóm tác giả có thể cải tiến tài liệu này trong những lần tái bản sau nhằm tạo ra một tài liệu tin cậy, dễ hiểu và hữu dụng dành cho sinh viên nhóm ngành máy tính và điện tử viễn thông.

CÁC TÁC GIẢ.

## MỤC LỤC

LỜI NÓI ĐẦU .....	i
MỤC LỤC.....	ii
DANH MỤC HÌNH.....	vi
DANH MỤC BẢNG.....	xiv
CHỮ VIẾT TẮT, THUẬT NGỮ, QUY ƯỚC.....	xv
<b>CHƯƠNG 1: TỔNG QUAN HỆ THỐNG SOC .....</b>	<b>1</b>
1.1. TỔNG QUAN .....	1
1.2. FPGA VÀ ASIC .....	5
1.2.1. <i>FPGA</i> .....	6
1.2.2. <i>ASIC</i> .....	9
1.3. SOC TRÊN FPGA .....	10
1.3.1. Các thành phần của SoC trên <i>FPGA</i> .....	10
1.3.2. <i>Altera SoC FPGA</i> .....	13
1.3.3. <i>Xilinx SoC FPGA</i> .....	16
1.3.4. So sánh các SoC <i>FPGA</i> của <i>Altera</i> và <i>Xilinx</i> .....	19
1.4. QUY TRÌNH THIẾT KẾ SoC.....	19
1.4.1. Thiết kế SoC trên <i>FPGA</i> .....	22
1.5. CÁC YẾU TỐ ẢNH HƯỞNG ĐẾN THIẾT KẾ SoC.....	25
1.6. ỨNG DỤNG CỦA SOC TRÊN <i>FPGA</i> .....	31
1.7. TÓM TẮT .....	32
1.8. CÂU HỎI VÀ BÀI TẬP .....	32
<b>CHƯƠNG 2: VI XỬ LÝ TRÊN SOC FPGA .....</b>	<b>34</b>
2.1. GIỚI THIỆU.....	34
2.2. KIẾN TRÚC VI XỬ LÝ ĐA NĂNG LÔI MỀM TRÊN <i>FPGA</i> .....	36
2.2.1. Kiến trúc vi xử lý <i>Nios II</i> .....	38
2.2.2. Kiến trúc vi xử lý <i>MicroBlaze</i> .....	43
2.2.3. Cách kết nối vi xử lý mềm vào trong hệ thống SoC. ....	46
2.3. KIẾN TRÚC VI XỬ LÝ ĐA NĂNG LÔI CÙNG TRÊN <i>FPGA</i> .....	49
2.3.1. Tổng quan kiến trúc <i>ARM CPU</i> .....	51
2.3.2. <i>CPU lõi cùng</i> trên <i>Altera</i> và <i>Xilinx FPGA</i> .....	55
2.3.3. Cách kết nối vi xử lý <i>cùng</i> vào trong hệ thống SoC .....	60

2.4. VI XỬ LÝ CHUYÊN DỤNG CHO HỌC SÂU TRÊN SoC FPGA .....	62
2.4.1. Kiến trúc của DPU .....	65
2.5. SO SÁNH CÁC LOẠI VI XỬ LÝ .....	71
2.6. TÍCH HỢP VI XỬ LÝ TỰ THIẾT KẾ VÀO HỆ THỐNG SOC .....	72
2.7. TÓM TẮT .....	80
2.8. CÂU HỎI VÀ BÀI TẬP .....	81
<b>CHƯƠNG 3: THIẾT KẾ BỘ NHỚ .....</b>	<b>82</b>
3.1. GIỚI THIỆU .....	82
3.2. BỘ NHỚ TRÊN CHIP (ON-CHIP MEMORY) .....	85
3.2.1. Giới thiệu .....	85
3.2.2. LUTRAM hay Distributed RAM .....	87
3.2.3. Block RAM .....	89
3.2.4. Ultra RAM .....	100
3.2.5. FIFO .....	104
3.3. BỘ NHỚ NGOÀI CHIP (OFF-CHIP MEMORY) .....	109
3.3.1. Giới thiệu .....	109
3.3.2. Bộ nhớ đệm (cache) L1, L2, L3 .....	112
3.3.3. Bộ nhớ DRAM .....	115
3.3.4. Bộ nhớ NAND Flash .....	123
3.4. BỘ NHỚ CÁU HÌNH .....	124
3.5. TÓM TẮT .....	126
3.6. CÂU HỎI VÀ BÀI TẬP .....	126
<b>CHƯƠNG 4: HỆ THỐNG KẾT NỐI .....</b>	<b>128</b>
4.1. GIỚI THIỆU TỔNG QUAN KIẾN TRÚC KẾT NỐI .....	128
4.2. GIAO THỨC AMBA .....	130
4.3. HỆ THỐNG BUS APB .....	132
4.3.1. Giới thiệu về hệ thống bus APB .....	132
4.3.2. Tín hiệu và kết nối APB .....	133
4.3.3. Chi tiết tín hiệu của APB trong việc ghi đọc dữ liệu .....	136
4.4. HỆ THỐNG BUS AHB .....	140
4.4.1. Giới thiệu về hệ thống bus AHB .....	140
4.4.2. Tín hiệu và kết nối AHB .....	141
4.4.3. Chi tiết tín hiệu của AHB trong việc ghi đọc dữ liệu cơ bản .....	146

4.4.4. Chi tiết tín hiệu của AHB trong việc ghi đọc dữ liệu với Burst .....	149
4.5. HỆ THỐNG BUS AXI .....	154
4.5.1. Giới thiệu về hệ thống bus AXI .....	154
4.5.2. Tín hiệu và kết nối AXI4 Full .....	156
4.5.3. Chi tiết tín hiệu của AXI4 Full trong việc đọc và ghi dữ liệu .....	163
4.5.4. Tín hiệu và kết nối AXI4 Lite.....	165
4.5.5. Chi tiết tín hiệu của AXI4-Lite trong việc đọc và ghi dữ liệu .....	167
4.5.6. Tín hiệu và kết nối AXI4 Stream .....	169
4.5.7. Chi tiết tín hiệu của AXI4-Stream trong việc ghi dữ liệu.....	171
4.6. HỆ THỐNG BUS AVALON .....	172
4.6.1. Giới thiệu về hệ thống bus Avalon .....	172
4.6.2. Tín hiệu và kết nối Avalon Bus.....	175
4.6.3. Chi tiết tín hiệu của Avalon bus trong việc ghi đọc dữ liệu .....	179
4.7. KẾT NỐI IP TỰ THIẾT KẾ VÀO HỆ THỐNG BUS.....	182
4.7.1. Kết nối IP tự thiết kế vào hệ thống AXI bus .....	183
4.7.2. Kết nối IP tự thiết kế vào hệ thống Avalon bus.....	190
4.8. TÓM TẮT .....	192
4.9. CÂU HỎI VÀ BÀI TẬP.....	193

## CHƯƠNG 5: TIÊU CHUẨN ĐÁNH GIÁ HỆ THỐNG SOC TRÊN FPGA..... 195

5.1. GIỚI THIỆU .....	195
5.2. ĐÁNH GIÁ TÀI NGUYÊN SỬ DỤNG .....	197
5.2.1. Đánh giá diện tích .....	197
5.2.2. Đánh giá năng lượng .....	207
5.3. ĐÁNH GIÁ HIỆU NĂNG CỦA HỆ THỐNG .....	211
5.3.1. Ước tính chu kỳ dài nhất và tần số hoạt động tối đa .....	213
5.3.2. Đánh giá độ trễ .....	214
5.3.3. Đánh giá thông lượng .....	218
5.4. ĐÁNH GIÁ HIỆU QUẢ CỦA HỆ THỐNG .....	221
5.4.1. Đánh giá hiệu quả diện tích .....	221
5.4.2. Đánh giá hiệu quả năng lượng .....	224
5.5. ĐỘ TIN CẬY .....	227
5.5.1. Lỗi và sửa lỗi.....	228
5.5.2. Kỹ thuật cải tiến độ tin cậy.....	229

5.5.3. Khắc phục lỗi từ nhà máy sản xuất chip .....	231
5.6. SỰ ĐÁNH ĐỒI TRONG HỆ THỐNG SOC .....	232
5.7. TÍCH HỢP HỆ THỐNG .....	235
5.8. TÓM TẮT .....	236
5.9. CÂU HỎI VÀ BÀI TẬP .....	237
<b>CHƯƠNG 6: TỔNG HỢP CẤP CAO TRONG SOC FPGA .....</b>	<b>240</b>
6.1. GIỚI THIỆU .....	240
6.2. GIỚI THIỆU VIVADO HLS VÀ INTEL HLS .....	240
6.2.1. Quy trình tạo IP từ công cụ HLS .....	243
6.2.2. Các lợi ích của phương pháp tổng hợp cấp cao .....	244
6.3. CÁC KỸ THUẬT TỐI UU HÓA THIẾT KẾ VỚI HLS .....	245
6.3.1. Phân giải vòng lặp .....	245
6.3.2. Phân chia mảng (Array Partitioning) .....	247
6.3.3. Pragma trong tổng hợp cấp cao .....	249
6.3.4 Kỹ thuật lượng tử hóa .....	251
6.4. QUY TRÌNH THIẾT KẾ SOC THEO PHƯƠNG PHÁP TỔNG HỢP CẤP CAO .....	253
6.5. VÍ DỤ THIẾT KẾ SOC BẰNG PHƯƠNG PHÁP TỔNG HỢP CẤP CAO .....	255
6.5.1. Ví dụ tạo IP từ phương pháp tổng hợp cấp cao .....	256
6.5.2. Ví dụ thiết kế một SoC sử dụng phương pháp tổng hợp cấp cao cho ứng dụng mạng trí tuệ nhân tạo .....	258
Mạng tích chập .....	259
6.6. TÓM TẮT .....	270
6.7. CÂU HỎI VÀ BÀI TẬP .....	270
<b>TÀI LIỆU THAM KHẢO .....</b>	<b>272</b>

## DANH MỤC HÌNH

Hình 1.1: Số lượng transistor thay đổi theo năm trên một chip bán dẫn.....	2
Hình 1.2: Ví dụ minh họa hệ thống SoC từ chip VN32 của ICDREC. ....	3
Hình 1.3: Minh họa bảng mạch gồm các linh kiện điện tử .....	5
Hình 1.4: Cấu trúc mang tính khái niệm của một FPGA .....	6
Hình 1.5: Sơ đồ kết nối chi tiết của ma trận chuyển mạch .....	7
Hình 1.6: Minh họa cấu trúc một khối logic trong FPGA Xilinx [10].....	7
Hình 1.7: Thành phần cơ bản trong hệ thống SoC. ....	10
Hình 1.8: Ví dụ minh họa hệ thống SoC trên board DE2-115 của Altera.[5].....	14
Hình 1.9: Ví dụ minh họa hệ thống SoC trên board Zynq-7000 của Xilinx.[3].....	17
Hình 1.10: Minh họa quy trình thiết kế SoC tổng quát .....	20
Hình 1.11: Quy trình thiết kế vật lý cho hệ thống SoC .....	21
Hình 1.12: Quy trình thiết kế SoC trên FPGA. ....	22
Hình 1.13: Ví dụ kiến trúc SoC tích hợp IP CNN trên board Altera với phần mềm .....	23
Hình 1.14: Ví dụ kiến trúc SoC tích hợp IP CNN trên board Xilinx với phần mềm Vivado. ....	24
Hình 1.15: Chương trình điều khiển khối IP CNN theo kiến trúc SoC Altera.....	25
Hình 1.16: Chương trình điều khiển khối IP CNN theo kiến trúc SoC Xilinx .....	25
Hình 2.1: Cấu trúc hệ thống SoC FPGA với vi xử lý đa năng và vi xử lý chuyên dụng. ....	34
Hình 2.2: Cấu trúc hệ thống SoC FPGA với vi xử lý đa năng lõi cứng (a) và mềm (b). ....	35
Hình 2.3: Kiến trúc vi xử lý Nios II [9].....	38
Hình 2.4: So sánh hiệu suất và lượng logic cần dùng cho từng loại vi xử lý Nios II [9] .....	42
Hình 2.5: Nơi lấy IP Nios II và cấu hình vi xử lý theo bộ công cụ Qsys.....	43
Hình 2.6: Kiến trúc vi xử lý Microblaze .....	44
Hình 2.7: Yêu cầu xây dựng hệ thống SoC đơn giản .....	46
Hình 2.8: Minh họa lựa chọn CPU Nios II cho ví dụ .....	47
Hình 2.9: Hệ thống SoC tích hợp CPU Nios II. ....	48
Hình 2.10: Đoạn mã đơn giản dùng chạy cho CPU Nios II .....	48
Hình 2.11: Giản đồ hệ thống SoC tích hợp CPU Microblaze. ....	49
Hình 2.12: Dòng sản phẩm bộ vi xử lý ARM. ....	51
Hình 2.13: Hệ thống xử lý với đa CPU lõi cứng Arm Cortex-A9 trên Altera và Xilinx FPGA. ....	56
Hình 2.14: Hệ thống SoC đơn giản với vi xử lý lõi cứng trên FPGA. ....	61

Hình 2.15: Hệ thống SoC tích hợp CPU lõi cứng ARM Cortex-A9.....	61
Hình 2.16: Đoạn mã đơn giản dùng chạy cho CPU lõi cứng ARM Cortex-A9.....	62
Hình 2.17: Giản đồ hệ thống SoC tích hợp CPU lõi cứng ARM Cortex-A9.....	62
Hình 2.18: Vị trí IP DPU trong một board mạch FPGA .....	63
Hình 2.19: Kiến trúc chi tiết của DPU tích hợp trên board FPGA [21] .....	65
Hình 2.20: Bảng đóng gói của IP DPU được cung cấp bởi Xilinx .....	67
Hình 2.21: Quy trình hiện thực một ứng dụng trên DPU bằng công cụ Vitis AI [21] .....	69
Hình 2.22: Cấu trúc hệ thống SoC FPGA với vi xử lý tự thiết kế.....	73
Hình 2.23: Kiến trúc vi xử lý RISC-V cơ bản với khói xử lý lệnh tùy chỉnh để hỗ trợ các tính toán chuyên biệt phù hợp với yêu cầu của từng ứng dụng cụ thể. ....	75
Hình 2.24: Kiến trúc bộ tập lệnh I. ....	78
Hình 2.25: Giản đồ hệ thống SoC tích hợp CPU RISC-V đa năng tự thiết kế.....	79
Hình 2.26: Giản đồ hệ thống SoC tích hợp CPU RISC-V tùy chỉnh tự thiết kế. ....	80
Hình 3.1: Một số bộ nhớ và vị trí của chúng trên bảng mạch ZCU102. ....	82
Hình 3.2: Kiến trúc tổng quan một SoC trên FPGA với bộ nhớ trên chip (on-chip memory), bộ nhớ ngoài chip (off-chip memory), và bộ nhớ cấu hình (configuration memory). ....	83
Hình 3.3: Một SoC đơn giản với thiết kế IP trên mảng logic lập trình được sử dụng nhiều loại bộ nhớ trên chip để lưu trữ dữ liệu tính toán. ....	85
Hình 3.4: Vị trí LUT trong khói logic FPGA có thể tạo thành LUTRAM [13].....	87
Hình 3.5 Cấu trúc bên trong LUT khi được cấu hình thành bộ nhớ LUTRAM [13]. ....	88
Hình 3.6: Mô tả LUTRAM bằng Verilog.....	88
Hình 3.7: Sơ đồ thời gian chi tiết của tín hiệu trong LUT cho việc đọc và ghi. ....	89
Hình 3.8: Vị trí BRAM trong FPGA [13]. ....	90
Hình 3.9: Giao diện bộ nhớ BRAM [14]. ....	91
Hình 3.10: Sơ đồ thời gian chi tiết của tín hiệu trong BRAM cho việc đọc và ghi dữ liệu ở chế độ NO_CHANGE [14]. ....	93
Hình 3.11: Sơ đồ thời gian chi tiết của tín hiệu trong BRAM cho việc đọc và ghi dữ liệu ở chế độ READ_FIRST [14]. ....	93
Hình 3.12: Sơ đồ thời gian chi tiết của tín hiệu trong BRAM cho việc đọc và ghi dữ liệu ở chế độ WRITE_FIRST [14]. ....	94
Hình 3.13: Code Verilog mô tả Dual-Port BRAM với hai cổng đọc và ghi, sử dụng bộ nhớ BRAM trong thiết kế IP.....	95

Hình 3.14: Code Verilog để điều chỉnh mô đun BRAM với các chế độ NO_CHANGE, READ_FIRST, và WRITE_FIRST ở Công A, cổng B sẽ tương tự hệt như cổng A. ....	96
Hình 3.15: Giao diện IP Catalog trong Vivado, nơi tìm kiếm và chọn mô-đun Block Memory Generator để tạo và cấu hình BRAM trong thiết kế IP ở FPGA. ....	97
Hình 3.16: Giao diện cấu hình mô-đun Block Memory Generator trong Vivado, nơi người thiết kế có thể đặt tên cho mô-đun BRAM và lựa chọn loại bộ nhớ thích hợp. ....	98
Hình 3.17: Giao diện cấu hình mô-đun Block Memory Generator trong Vivado, nơi người thiết kế điều chỉnh độ rộng, độ sâu bộ nhớ BRAM và chọn chế độ hoạt động. ....	99
Hình 3.18: Giao diện hiển thị các cổng input và output của IP BRAM trong tab Design Sources, nơi mô-đun BRAM được gọi để các mô-đun khác có thể kết nối và sử dụng. ....	100
Hình 3.19: So sánh dung lượng giữa BRAM và URAM trong FPGA. ....	101
Hình 3.20: Code Verilog mô tả URAM với một cổng đọc và ghi trong thiết kế IP. ....	102
Hình 3.21: Giao diện cấu hình mô-đun Block Memory Generator trong Vivado, nơi người thiết kế điều chỉnh bộ nhớ sử dụng BRAM hay URAM. ....	103
Hình 3.22: So sánh cách lưu trữ và truy xuất dữ liệu giữa (a) FIFO và (b) BRAM/URAM. ....	104
Hình 3.23: Mã Verilog mô tả FIFO với độ rộng địa chỉ 10 bit và dữ liệu 32 bit. ....	106
Hình 3.24: Giao diện IP Catalog trong Vivado, nơi tìm kiếm và chọn mô-đun FIFO Generator để tạo và cấu hình FIFO trong thiết kế IP ở FPGA. ....	107
Hình 3.25: Giao diện cấu hình mô-đun FIFO Generator trong Vivado, nơi người thiết kế điều chỉnh độ rộng, độ sâu bộ nhớ FIFO và chọn các tham số liên quan đến chế độ đọc/ghi. ....	108
Hình 3.27: Các bộ nhớ ngoài chip trên FPGA ZCU102, bao gồm bộ nhớ DDR4 4GB, bộ nhớ NAND Flash, và các bộ nhớ cache L1, L2 của CPU ARM Cortex-A53 trong hệ thống SoC....	112
Hình 3.28: Tỷ lệ lỗi cache khi truy cập mảng theo thứ tự khác nhau với bộ nhớ cache 4 byte và mỗi block chứa 7 từ. (a) Lỗi cache 12.5%, (b) Lỗi cache 100%. ....	114
Hình 3.29: Các phương thức truyền dữ liệu giữa bộ nhớ DRAM và IP người dùng thiết kế (a) PIO, (b) DMA. ....	116
Hình 3.30: (a) Mô tả hệ thống SoC đơn giản với PS và IP thiết kế với tên MY_IP. (b) Địa chỉ được chỉ định của MY_IP trên SoC. ....	117
Hình 3.31: Quá trình truyền PIO từ DRAM vào MY_IP thông qua ánh xạ bộ nhớ mmap(). ....	118
Hình 3.33: (a) Truyền đơn lẻ (single transfer) trong phương thức truyền PIO và (b) truyền theo dạng gói (burst transfer) trong phương thức truyền DMA. ....	120
Hình 3.34: Khối FPD DMA trong hệ thống xử lý (ZYNQ PS) hỗ trợ truyền DMA từ DRAM sang MY_IP. ....	121

Hình 3.35: Code C mô tả quá trình truyền DMA từ DRAM vào MY_IP thông qua FPD DMA.	122
Hình 3.36: NAND Flash lưu trữ dữ liệu cho hệ điều hành nhúng trên SoC FPGA như một ổ cứng trên các máy tính thông thường.	123
Hình 3.37: Ứng dụng của thẻ nhớ SD trong việc lưu trữ dữ liệu cho IP thiết kế trong SoC và quy trình truyền dữ liệu từ thẻ nhớ SD lên DRAM và MY_IP.	124
Hình 3.38: Quy trình cấu hình SoC dùng bộ nhớ cấu hình trên FPGA.	125
<b>Hình 4.1:</b> Sơ đồ mô tả kiến trúc kết nối trên chip và ngoài chip trong hệ thống SoC [1].	128
<b>Hình 4.2:</b> Các phiên bản và đặc tả chính của kiến trúc bus AMBA qua các thế hệ [15].	130
<b>Hình 4.3:</b> Một ví dụ về hệ thống APB [16].	135
<b>Hình 4.4:</b> Sơ đồ thời gian chi tiết của tín hiệu đọc trong APB theo chuẩn AMBA 2.	136
<b>Hình 4.5:</b> Sơ đồ thời gian chi tiết của tín hiệu ghi trong APB theo chuẩn AMBA 2.	137
<b>Hình 4.6:</b> Sơ đồ thời gian chi tiết của tín hiệu đọc trong APB với 3 trạng thái chờ và tín hiệu phản hồi theo chuẩn AMBA 3.	138
<b>Hình 4.7:</b> Sơ đồ thời gian chi tiết của tín hiệu ghi trong APB với 3 trạng thái chờ và tín hiệu phản hồi theo chuẩn AMBA 3.	139
<b>Hình 4.8:</b> Một hệ thống điển hình dựa trên bus AMBA dùng AHB và APB [16].	140
<b>Hình 4.9:</b> Hệ thống AHB đơn giản với một AHB chủ và hai AHB tớ [16].	144
<b>Hình 4.10:</b> Phân chia quá trình truyền dữ liệu thành giai đoạn địa chỉ và dữ liệu.	146
<b>Hình 4.11:</b> Sơ đồ thời gian chi tiết của tín hiệu đọc cơ bản trong AHB.	147
<b>Hình 4.12:</b> Sơ đồ thời gian chi tiết của tín hiệu ghi cơ bản trong AHB.	147
<b>Hình 4.13:</b> Sơ đồ thời gian chi tiết của tín hiệu đọc cơ bản với hai trạng thái chờ.	148
<b>Hình 4.14:</b> Sơ đồ thời gian chi tiết của tín hiệu ghi cơ bản với hai trạng thái chờ.	149
<b>Hình 4.15:</b> Sơ đồ thời gian chi tiết của tín hiệu kết hợp đọc và ghi cơ bản trong AHB.	149
<b>Hình 4.16:</b> Sơ đồ thời gian chi tiết của tín hiệu kết hợp đọc với Burst vòng 4 nhịp.	151
<b>Hình 4.17:</b> Sơ đồ thời gian chi tiết của tín hiệu kết hợp đọc với Burst gia tăng 4 nhịp.	153
<b>Hình 4.18:</b> Hệ thống kết hợp các bus AXI, AHB, và APB trên nhiều Xilinx FPGA [17].	154
<b>Hình 4.19:</b> Quá trình ghi dữ liệu trên AMBA AXI4 Full sử dụng ba kênh chính: kênh địa chỉ ghi, kênh dữ liệu ghi, và kênh phản hồi ghi [18].	156
<b>Hình 4.20:</b> Quá trình đọc dữ liệu trên AMBA AXI4 Full sử dụng hai kênh chính: kênh địa chỉ đọc và kênh dữ liệu đọc[18].	157
<b>Hình 4.21:</b> Sơ đồ thời gian chi tiết của tín hiệu chính cho ghi dữ liệu với Burst trên kênh địa chỉ ghi, kênh dữ liệu ghi, và kênh phản hồi ghi.	163

Hình 4.22: Sơ đồ thời gian chi tiết của tín hiệu chính cho đọc dữ liệu với Burst trên kênh địa chỉ đọc và kênh dữ liệu đọc.....	164
Hình 4.23: Sơ đồ thời gian chi tiết của tín hiệu chính cho đọc dữ liệu chồng chéo với Burst trên kênh địa chỉ đọc và kênh dữ liệu đọc.....	165
Hình 4.24: Quá trình ghi dữ liệu trên AMBA AXI4-Lite sử dụng ba kênh chính: kênh địa chỉ ghi, kênh dữ liệu ghi, và kênh phản hồi ghi [18].....	166
Hình 4.25: Quá trình đọc dữ liệu trên AMBA AXI4-Lite sử dụng hai kênh chính: kênh địa chỉ đọc và kênh dữ liệu đọc [18].....	166
Hình 4.26: Sơ đồ thời gian chi tiết của tín hiệu chính cho ghi dữ liệu của AXI4-Lite trên kênh địa chỉ ghi, kênh dữ liệu ghi, và kênh phản hồi ghi.....	168
Hình 4.27: Sơ đồ thời gian chi tiết của tín hiệu chính cho đọc dữ liệu của AXI4-Lite trên kênh địa chỉ đọc và kênh dữ liệu đọc.....	169
Hình 4.28: Sơ đồ thời gian chi tiết của tín hiệu chính cho ghi dữ liệu của AXI4-Stream. ....	172
Hình 4.29: Sơ đồ khái mô-đun bus Avalon trong hệ thống ví dụ đơn giản [19].....	173
Hình 4.30: Sơ đồ sử dụng Avalon Bus cơ bản cho quá trình đọc để giao tiếp giữa bộ xử lý Nios II và phần cứng IP tự thiết kế.....	178
Hình 4.31: Sơ đồ sử dụng Avalon Bus cơ bản cho quá trình ghi để giao tiếp giữa bộ xử lý Nios II và phần cứng IP tự thiết kế.....	179
Hình 4.32: Sơ đồ thời gian chi tiết của tín hiệu chính cho đọc dữ liệu cơ bản của Avalon Bus..	180
Hình 4.33: Sơ đồ thời gian chi tiết của tín hiệu chính cho đọc dữ liệu với một trạng thái chờ cố định của Avalon Bus.....	180
Hình 4.34: Sơ đồ thời gian chi tiết của tín hiệu chính cho ghi dữ liệu cơ bản của Avalon Bus. .	181
Hình 4.35: Sơ đồ thời gian chi tiết của tín hiệu chính cho ghi dữ liệu với một trạng thái chờ cố định của Avalon Bus.....	182
Hình 4.36: Minh họa hệ thống đơn giản kết nối giữa CPU và IP tự thiết kế thông qua AXI/Avalon Bus, với IP tự thiết kế đóng vai trò Tờ kết nối vào hệ thống AXI/Avalon bus.....	182
Hình 4.37: Minh họa hệ thống kết nối giữa CPU và IP tự thiết kế qua AXI Full/Lite/Stream Bus.	183
Hình 4.38: Bước tạo một vỏ bọc giao diện AXI bus từ mẫu có sẵn cho IP tự thiết kế trong phần mềm Xilinx Vivado. ..	183
Hình 4.39: Chọn loại giao diện (Interface Type) khi tạo vỏ bọc AXI, với các tùy chọn AXI Full hoặc AXI Lite để xác định ba chuẩn khác nhau của giao diện AXI. ....	184

Hình 4.40: Hai file code Verilog tương ứng với vỏ bọc giao diện AXI Full/Lite/Stream và mô đun xử lý các tín hiệu trong AXI Full/Lite/Stream bus được tạo ra sau bước chọn giao diện trong phần mềm Xilinx Vivado. ....	184
Hình 4.41: Code Verilog mô tả IP tự thiết kế sử dụng 6 tín hiệu từ mô đun xử lý các tín hiệu AXI Full bus để ghi, đọc dữ liệu vào các biến và thực hiện tính toán $y = a*x + b$ . ....	185
Hình 4.42: Code Verilog mô tả IP tự thiết kế sử dụng 6 tín hiệu từ mô đun xử lý các tín hiệu AXI Lite bus để ghi, đọc dữ liệu vào các biến và thực hiện tính toán $y = a*x + b$ . ....	186
Hình 4.43: Khác biệt cấu hình vỏ bọc giao diện AXI Full/Lite và AXI Stream: (a) Vỏ bọc giao diện AXI Full/Lite với chế độ Tớ hỗ trợ cả tín hiệu đọc và ghi; (b) Vỏ bọc giao diện AXI Stream với chế độ Tớ chỉ hỗ trợ tín hiệu ghi và chế độ Chủ chỉ hỗ trợ tín hiệu đọc.....	187
Hình 4.44: Thêm giao diện AXI Stream cho IP tự thiết kế, với M00_AXIS là chế độ giao diện Chủ và S00_AXIS là chế độ giao diện Tớ. ....	188
Hình 4.45: Code Verilog mô tả IP tự thiết kế sử dụng 5 tín hiệu từ mô đun xử lý các tín hiệu AXI Stream bus của giao diện Chủ và Tớ để ghi, đọc dữ liệu vào các biến và thực hiện tính toán $y = a*x + b$ . ....	189
Hình 4.46: Minh họa hệ thống kết nối giữa CPU và IP tự thiết kế qua Avalon Bus. ....	190
Hình 4.47: Code Verilog mô tả IP tự thiết kế kết hợp với mô-đun xử lý các tín hiệu Avalon bus để ghi, đọc dữ liệu vào các biến và thực hiện phép tính $y = a * x + b$ . ....	191
Hình 4.48: Cấu hình các tín hiệu của IP tự thiết kế (mô đun Compute) trên phần mềm Quartus Prime. ....	192
Hình 5.1: Đánh giá diện tích của IP trên chip ASIC và FPGA: (a) Diện tích của IP trên chip ASIC được đo bằng các đơn vị vật lý; (b) Diện tích của IP trên FPGA là tổng diện tích của các thành phần cơ bản trong cấu trúc FPGA. ....	197
Hình 5.2: Diện tích trên Altera FPGA trích xuất tài nguyên từ phần mềm Quartus Prime. ....	199
Hình 5.3: Diện tích SoC trên ZCU102 FPGA trích xuất tài nguyên từ phần mềm Vivado. ....	200
Hình 5.4: Cấu hình phép nhân với độ rộng 16 bit cho hai toán hạng, chọn tối ưu tốc độ, và chọn tầng ống dẫn là 4, sử dụng LUT thay vì DSP trên phần mềm Vivado. ....	202
Hình 5.5: Diện tích phép nhân với độ rộng 16 bit cho hai toán hạng, tối ưu hóa tốc độ, và 4 tầng đường ống trên ZCU102 FPGA.....	202
Hình 5.6: Diện tích phép nhân với độ rộng 32 bit cho hai toán hạng, tối ưu hóa diện tích, và 8 tầng đường ống trên ZCU102 FPGA.....	203
Hình 5.7: Code Verilog mô tả phép nhân với độ rộng 16 bit cho hai toán hạng mà không sử dụng IP phép nhân có sẵn trong IP Catalog của Vivado. ....	204

Hình 5.8: Diện tích phép nhân với độ rộng 16 bit cho hai toán hạng và 1 tầng ống dẫn của IP tương đương với mã Verilog mô tả trong Hình 5.6.....	204
Hình 5.9: Quy trình phân tích công suất cho hệ thống SoC trên FPGA. [7] .....	207
Hình 5.10: Năng lượng trên Altera FPGA trích xuất tài nguyên từ phần mềm Quartus Prime. .	208
Hình 5.11: Ví dụ trích xuất thông tin công suất tiêu thụ từ công cụ phân tích công suất được tích hợp trong Vivado Xilinx [6] .....	209
Hình 5.12: Hình ảnh trực quan giải thích khái niệm về độ trễ và thông lượng.....	212
Hình 5.13: Độ tin cậy TMR so sánh với độ tin cậy Simplex .....	230
Hình 5.14: Minh họa sự đánh đổi trong thiết kế vi xử lý .....	233
Hình 6.9: Quy trình tạo IP tự thiết kế với công cụ Vivado HLS/Intel HLS .....	244
Hình 6.11: Minh họa đoạn mã xử lý với kỹ thuật phân giải vòng lặp từng phần.....	246
Hình 6.12: Minh họa một số kỹ thuật phân giải vòng lặp .....	246
Hình 6.13: Minh họa các kỹ thuật phân chia mảng trong tổng hợp cấp cao .....	249
Hình 6.14: Minh họa đoạn mã xử lý có tích hợp các chỉ định pragma .....	250
Hình 6.15: Mã C sử dụng dấu phẩy động để tính toán.....	252
Hình 6.16: Mã C sử dụng dấu chấm cố định để tính toán .....	252
Hình 6.17: Quy trình thiết kế hệ thống từ HLS xuống SoC .....	254
Hình 6.18: Minh họa đoạn mã nguồn để tạo IP nhân hai ma trận từ mã C++. .....	256
Hình 6.19: Minh họa đoạn mã nguồn kiểm tra C++ dùng để kiểm tra chức năng bài toán nhân hai trận. ....	257
Hình 6.20: Tạo tần số hoạt động ban đầu cho thiết kế. ....	258
Hình 6.21: Tổ chức lưu trữ trong bộ công cụ Vitis HLS cho một dự án.....	258
Hình 6.22: Cấu trúc một mô hình mạng tích chập cơ bản.....	259
Hình 6.23: Ví dụ về cách tính nhân chập của lớp tích chập .....	261
Hình 6.24: Công thức và đồ thị một số hàm kích hoạt phổ biến .....	262
Hình 6.25: Ví dụ về max pooling và average pooling .....	264
Hình 6.26: Cấu trúc lớp làm phẳng và kết nối đầy đủ .....	265
Hình 6.27: Minh họa hình ảnh cụ thể trong tập dữ liệu MNIST .....	265
Hình 6.28: Minh họa đoạn mã Python của mô hình mạng Lenet5 được tạo bằng nền tảng Keras .....	267
Hình 6.29: Minh họa đoạn mã C dùng để tính toán lớp tích chập, lớp kích hoạt (relu) và lớp maxpooling. ....	268
Hình 6.30: Hệ thống SoC thiết kế cho ứng dụng nhận dạng chữ số trên FPGA Altera .....	269

Hình 6.31: Minh họa lập trình code trên phần mềm để điều khiển và lấy kết quả từ phần cứng. 269

# Thiết kế hệ thống SOC, JIT

## DANH MỤC BẢNG

Bảng 1.1: Các dòng sản phẩm SoC chuyên dụng của hãng Altera .....	15
Bảng 1.2: Một số SoC do hãng Xilinx cung cấp .....	18
Bảng 1.3: Minh họa tích hợp các thành phần khác nhau trong các loại SoC .....	26
Bảng 1.4: So sánh công suất tiêu thụ trên các dòng SoC FPGA khác nhau.....	27
Bảng 1.5: Các loại nhân vi xử lý tích hợp dưới dạng IP cứng hoặc mềm.....	30
Bảng 2.1: So sánh các loại CPU Nios II.....	40
Bảng 2.2: So sánh các loại CPU ARM Cortex-A được sử dụng làm CPU lõi cứng.....	53
Bảng 2.3: Chi tiết ý nghĩa của các chân tín hiệu trong IP DPU .....	67
Bảng 2.4: So sánh 3 loại vi xử lý trên SoC FPGA .....	71
Bảng 3.1: Các tín hiệu của bộ nhớ BRAM .....	91
Bảng 3.2: So sánh giữa các loại bộ nhớ khác nhau .....	110
Bảng 4.1: Các tín hiệu điển hình của APB .....	133
Bảng 4.2: Các tín hiệu điển hình của AHB .....	142
Bảng 4.3: Mã hóa tín hiệu Burst.....	150
Bảng 4.4: Mã hóa loại truyền dữ liệu .....	152
Bảng 4.5: Các tín hiệu ở 5 kênh dùng trong AXI Full .....	158
Bảng 4.6: Các tín hiệu ở 5 kênh dùng trong AXI Lite .....	167
Bảng 4.7: Các tín hiệu trong AXI Stream .....	169
Bảng 4.8: Các tín hiệu trong Avalon Bus .....	175
Bảng 6.1: So sánh các đặc trưng chính giữa hai bộ công cụ Vivado HLS và Intel HLS .....	241
Bảng 6.2: Báo cáo tài nguyên phân cứng với dữ liệu đầu vào ở hai kiểu dữ liệu: dấu chấm cố định và dấu phẩy động.....	253

## CHỮ VIẾT TẮT, THUẬT NGỮ, QUY ƯỚC

<b>SoC</b>	System on Chip
<b>IoT</b>	Internet of Things
<b>CPU</b>	Central Processor Unit
<b>MMU</b>	Memory Management Unit
<b>GPU</b>	Graphics Processing Unit
<b>DSP</b>	Digital Signal Processor
<b>SIMD</b>	Single Instruction, Multiple Data
<b>VLIW</b>	Very Long Instruction Word
<b>NOC</b>	Network on Chip
<b>FPGA</b>	Field-Programmable Gate Array
<b>ASIC</b>	Application-Specific Integrated Circuit
<b>RISC</b>	Reduced Instruction Set Computer
<b>AMBA</b>	Advanced Microcontroller Bus Architecture
<b>AXI</b>	Advanced eXtensible Interface
<b>AHB</b>	Advanced High-performance Bus
<b>APB</b>	Advanced Peripheral Bus
<b>EDS</b>	Embedded Design Suite
<b>FIFO</b>	First-In, First-Out
<b>ECC</b>	Error Correction Code
<b>TMR</b>	Triple modular redundancy
<b>DFT</b>	Design for Testability
<b>JTAG</b>	Joint Test Action Group
<b>HLS</b>	High Level Synthesis
<b>RTL</b>	Register Transfer Level
<b>VLSI</b>	Very Large Scale Integration
<b>IP</b>	Intellectual Property
<b>HDL</b>	Hardware Description Language
<b>CNN</b>	Convolutional Neural Network
<b>MNIST</b>	Modified National Institute of Standards and Technology Database
<b>ALU</b>	Arithmetic and Logic Unit

AU	Arithmetic Unit
PE	Processing Element
PL	Programmable Logic
PS	Processing System
HPC	High-Performance Computing

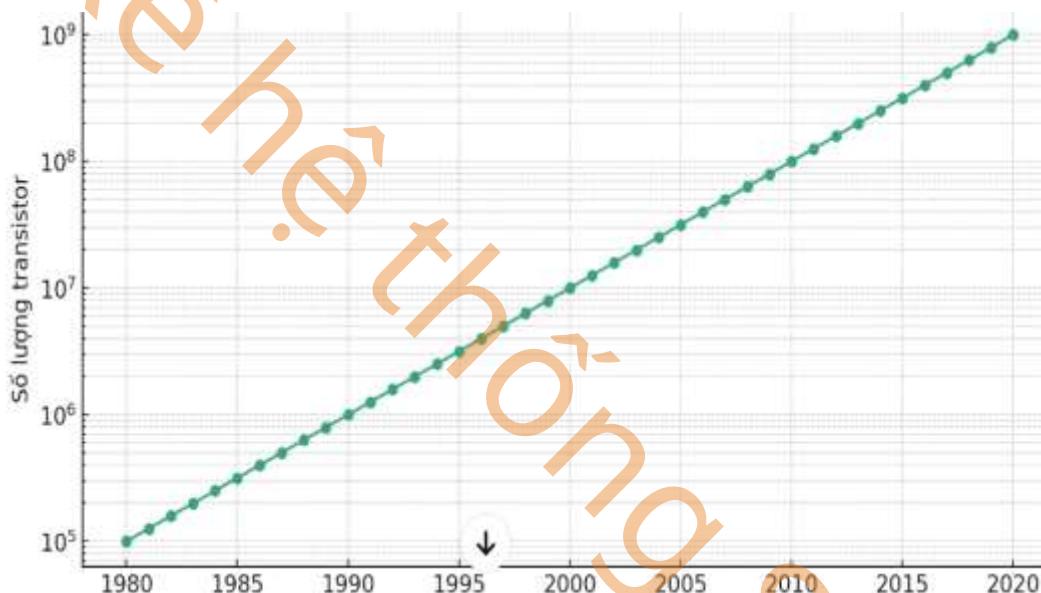
# CHƯƠNG 1: TỔNG QUAN HỆ THỐNG SOC

## 1.1. Tổng quan

Trong 40 năm qua, công nghệ bán dẫn đã chứng kiến những bước tiến đáng kinh ngạc, dẫn đến sự tăng vọt về mật độ transistor cũng như hiệu suất của linh kiện điện tử. Vào năm 1966, nhà máy bán dẫn Fairchild lần đầu tiên giới thiệu một công logic NAND hai đầu vào với khoảng 10 transistor trên một linh kiện (die). Đến nay, hàng tỷ transistor đã được tích hợp trên một bộ vi xử lý đa nhân của Intel, với kích thước mỗi transistor chỉ nhỏ bằng một vi khuẩn. Sự phát triển này không chỉ minh chứng cho khả năng tích hợp số lượng transistor trên chip tăng theo đúng như dự đoán của định luật Moore, mà còn cho thấy mức độ sáng tạo không ngừng của con người trong việc đẩy mạnh giới hạn của khoa học và kỹ thuật. Ngoài ra, sự thay đổi này không chỉ làm tăng tốc độ xử lý và khả năng lưu trữ của máy tính, mà còn giúp giảm đáng kể kích thước cũng như mức tiêu thụ năng lượng của chúng. Các bước tiến trong công nghệ bán dẫn đã mở ra cánh cửa cho sự phát triển của công nghệ di động, trí tuệ nhân tạo, và Internet vạn vật, mang lại lợi ích to lớn cho xã hội và kinh tế. Biểu đồ trong hình 1.1 minh họa sự tăng trưởng theo thời gian của số lượng transistor trong các vi mạch, phù hợp với định luật Moore. Từ năm 1980 đến năm 2020, biểu đồ cho thấy số lượng transistor tăng theo cấp số nhân, từ hàng trăm nghìn lên đến hàng tỷ. Xu hướng này phản ánh sự phát triển vượt bậc của công nghệ bán dẫn và quá trình thu nhỏ kích thước transistor, giúp thúc đẩy sự đổi mới trong lĩnh vực điện tử và công nghệ thông tin. Đây là minh chứng rõ ràng về tính chính xác của định luật Moore trong suốt nhiều thập kỷ qua.

Hệ thống trên chip (SoC - System on Chip) đánh dấu một bước tiến quan trọng trong ngành công nghiệp điện tử bán dẫn, mang lại giải pháp tối ưu cho việc thiết kế và sản xuất các thiết bị điện tử hiện đại. Với khả năng tích hợp tất cả các thành phần cần thiết của một hệ thống máy tính vào một con chip duy nhất, SoC không chỉ tối ưu hóa diện tích sử dụng mà còn cải thiện đáng kể hiệu suất và hiệu quả năng lượng. Nhờ những lợi ích vượt trội này, các thiết bị di động như điện thoại thông minh và máy tính bảng đã trở nên mỏng nhẹ hơn, mạnh mẽ hơn và tiết kiệm năng lượng hơn. Đặc biệt, trong thời đại Internet vạn vật (IoT), SoC đóng vai trò không thể thay thế nhờ khả năng tích hợp cao, đáp ứng yêu cầu kết

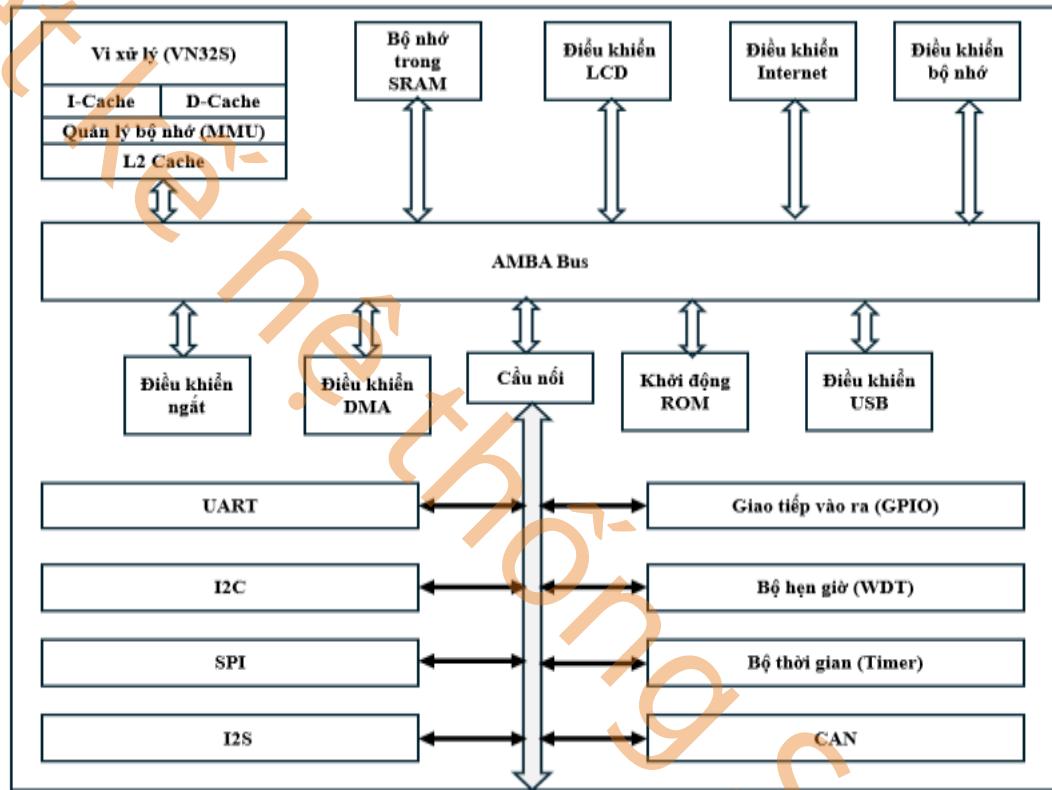
nối liên tục, xử lý dữ liệu nhanh chóng và tiết kiệm năng lượng. SoC cũng thúc đẩy những đổi mới trong nhiều lĩnh vực như y tế, tự động hóa công nghiệp và giải trí, nơi mà sự nhỏ gọn, hiệu quả và tính năng đa dạng là những yếu tố quyết định. Điểm nổi bật của SoC là khả năng tích hợp các bộ vi xử lý mạnh mẽ, bộ xử lý đồ họa tiên tiến, bộ nhớ đa dạng và các giao tiếp đa năng, tất cả trong một không gian vô cùng hạn chế. Sự linh hoạt này cho phép các nhà sản xuất điều chỉnh SoC để phù hợp với yêu cầu cụ thể của từng sản phẩm, từ đó tối ưu hóa hiệu năng cho các ứng dụng khác nhau. Tóm lại, SoC không chỉ là nền tảng công nghệ mang đến cơ hội mới cho các thiết kế điện tử tiên tiến mà còn là chìa khóa mở rộng tiềm năng của các thiết bị thông minh trong tương lai.



Hình 1.1: Số lượng transistor thay đổi theo năm trên một chip bán dẫn.

Hình 1.2 minh họa kiến trúc của một hệ thống SoC, trong đó các thành phần được kết nối thông qua bus vi điều khiển nâng cao (AMBA - Advanced Microcontroller Bus Architecture). Tâm điểm của hệ thống là vi xử lý VN32S (CPU - Central Processor Unit) được trang bị bộ nhớ đệm L2 (cache) dung lượng 128 KB và bộ quản lý bộ nhớ (MMU - Memory Management Unit). Hệ thống tích hợp nhiều mô-đun phục vụ các chức năng khác nhau, bao gồm bộ điều khiển LCD, Ethernet, bộ nhớ, và bộ điều khiển USB. Đồng thời, nó hỗ trợ các giao thức giao tiếp phổ biến như UART, I2C, SPI, I2S và CAN, cho phép kết nối linh hoạt với các thiết bị ngoại vi. Ngoài ra, hệ thống được thiết kế để cung cấp các chức năng bổ trợ như điều khiển ngắn, truy cập bộ nhớ trực tiếp (DMA - Direct Memory

Access) và giao diện vào/ra đa năng GPIO (General Purpose Input/Output). SoC này còn được tích hợp bộ nhớ ROM khởi động dung lượng 1 KB, cùng với các tính năng bảo mật tiên tiến nhờ IP bảo mật (AES-128 bit). Bên cạnh đó, hệ thống hỗ trợ các bộ đếm (counter) và bộ định thời (timer) với chức năng điều chỉnh độ rộng xung (PWM - Pulse Width Modulation), cũng như bộ hẹn giờ giám sát (WDT - Watchdog Timer), cung cấp các giải pháp hiệu quả cho việc giám sát và quản lý toàn diện hệ thống.



Hình 1.2: Ví dụ minh họa hệ thống SoC từ chip VN32 của ICDREC.

SoC được định nghĩa là sự tích hợp toàn bộ hệ thống vào trong một chip. Tuy nhiên, như minh họa trong hình 1.2, một hệ thống hoàn chỉnh bao gồm nhiều bộ vi xử lý, bộ nhớ và các giao tiếp ngoại vi, được lắp đặt trên một bảng mạch lớn, vẫn có thể được coi là một SoC thay vì một máy tính truyền thống. Điểm làm nên sự khác biệt của hệ thống SoC so với máy tính chung chung là mục tiêu thiết kế cụ thể của nó. Các hệ thống SoC được tạo ra sẽ nhắm vào một hoặc một vài ứng dụng được giả định là đã biết và xác định trước, cho phép trong quá trình thiết kế có thể lựa chọn kích thước và tối ưu hóa các thành phần của hệ thống một cách hiệu quả nhất cho ứng dụng đó mà không quan tâm tới khả năng thực

thi của SoC trong nhiều ứng dụng tổng quát như cách giải quyết của máy tính cá nhân hoặc máy tính để bàn.

SoC thường tích hợp vi xử lý, các ngoại vi, và chip xử lý chuyên dụng trên một board mạch duy nhất, được thiết kế tối ưu hóa cho các tác vụ đặc thù của ứng dụng mục tiêu. Do đó, SoC thường không thể nâng cấp hoặc thay đổi các thành phần phần cứng vì ý định ngay từ ban đầu là tối ưu hóa cho một hoặc một vài ứng dụng cụ thể. Các thành phần như vi xử lý, bộ nhớ, ngoại vi, và các bộ tăng tốc phần cứng được tích hợp cố định trên cùng một chip hoặc bảng mạch, giúp tăng hiệu suất, giảm kích thước vật lý và tiết kiệm năng lượng. Ngược lại, máy tính cá nhân thường được lắp ghép từ các thiết bị có thể tháo rời và nâng cấp được, chẳng hạn như CPU, RAM, ổ cứng, card đồ họa, và các ngoại vi khác. Mục tiêu thiết kế của PC là đáp ứng nhu cầu đa dụng và linh hoạt, cho phép người dùng thay đổi hoặc nâng cấp các thành phần phần cứng khi cần thiết để phù hợp với nhiều ứng dụng và yêu cầu sử dụng khác nhau. Điều này khác biệt hoàn toàn với SoC, nơi mọi thành phần đều được tích hợp cố định và tối ưu hóa cho ứng dụng cụ thể ngay từ khâu thiết kế ban đầu. Sự khác biệt này giúp SoC trở nên lý tưởng cho các thiết bị nhúng, điện thoại thông minh, thiết bị IoT và các hệ thống tự động hóa, nơi yêu cầu cao về hiệu năng, tiết kiệm năng lượng, và kích thước nhỏ gọn, trong khi máy tính cá nhân và máy tính để bàn lại phù hợp hơn cho các ứng dụng tổng quát và khả năng mở rộng linh hoạt trong tương lai.

Quá trình thiết kế một hệ thống SoC bắt đầu từ việc hình thành ý tưởng cho một ứng dụng cụ thể và kết thúc bằng việc tạo ra một IP (Intellectual Property) hoặc một bảng mạch hoàn chỉnh. Quy trình này bao gồm hai giai đoạn chính: thiết kế SoC ở mức logic và thiết kế SoC ở mức vật lý. Do tính chất phức tạp và số lượng bước lớn trong quy trình thiết kế SoC, giáo trình này sẽ tập trung vào giai đoạn thiết kế SoC ở mức logic. Cụ thể, nội dung sẽ giới thiệu các kiến thức cơ bản về hệ thống SoC, các thành phần chính cấu thành hệ thống, quy trình thiết kế một hệ thống SoC hoàn chỉnh trên FPGA, phương pháp điều khiển SoC bằng phần mềm, cách đánh giá hiệu năng hệ thống SoC trên FPGA, và phương pháp tiếp cận thiết kế SoC từ ngôn ngữ cấp cao (HLS - High Level Synthesis).

Giai đoạn thiết kế SoC ở mức vật lý sẽ được trình bày chi tiết trong một giáo trình khác.

Ngoài ra, để nắm rõ và hiểu tường tận nội dung của giáo trình, người đọc cần có kiến thức cơ bản về mạch số, kiến trúc máy tính, lập trình C/C++, và lập trình bằng ngôn ngữ mô tả phần cứng (HDL - Hardware Description Language ).

## 1.2. FPGA và ASIC



Hình 1.3: Minh họa bảng mạch gồm các linh kiện điện tử

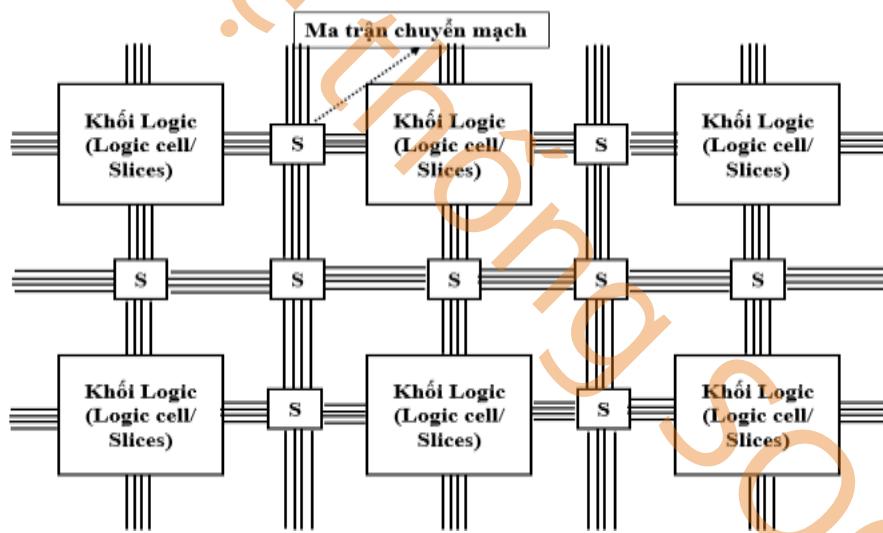
Mạch tích hợp (IC - Integrated Circuit) đóng vai trò nền tảng cơ bản cho hầu hết các thiết bị điện tử hiện đại. IC là sự kết hợp tinh vi của nhiều thành phần điện tử như transistor, điện trở, tụ điện và các linh kiện khác, được tích hợp khéo léo trên một lớp chất bán dẫn đơn lẻ, thường là silicon. Quá trình này đã cách mạng hóa ngành công nghiệp điện tử bằng cách thu nhỏ kích thước, tăng cường hiệu quả năng lượng và mở rộng khả năng thực hiện các chức năng phức tạp mà không cần đến các linh kiện rời rạc truyền thống. Mạch tích hợp được phân loại dựa trên số lượng transistor, từ các mạch đơn giản (SSI - Small Scale Integration), chứa hàng trăm hoặc hàng nghìn transistor, đến các mạch phức tạp (VLSI - Very Large Scale Integration), với hàng triệu transistor trên một chip duy nhất. Hình 1.3 minh họa một ví dụ về bảng mạch tích hợp chứa nhiều linh kiện điện tử.

Trong thế giới mạch tích hợp, mạch tích hợp chuyên dụng (ASIC - Application-Specific Integrated Circuit) và mảng cổng có thể lập trình (FPGA - Field-Programmable Gate Array) là hai loại phổ biến nhất. ASIC là loại mạch được thiết kế đặc biệt cho một ứng dụng cụ thể và không thể chỉnh sửa sau khi đã được sản xuất thành chip. Điều này giúp ASIC đạt hiệu suất tối ưu và tiết kiệm năng lượng tối đa cho các tác vụ chuyên biệt.

Ngược lại, FPGA có thể được lập trình lại nhiều lần sau khi sản xuất, cho phép linh hoạt tùy chỉnh để phù hợp với nhiều ứng dụng khác nhau. Cả hai loại IC này đều đóng vai trò quan trọng trong ngành công nghiệp điện tử, từ việc phát triển các thiết bị thông minh cho đến tối ưu hóa các hệ thống tự động hóa, đáp ứng nhu cầu ngày càng cao về hiệu suất và tính linh hoạt.

### 1.2.1. *FPGA*

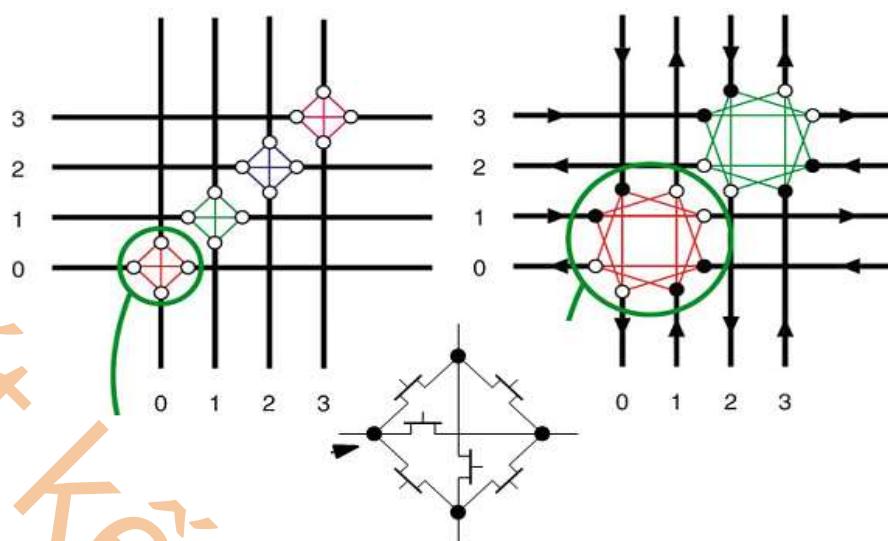
FPGA là một loại vi mạch tích hợp có khả năng lập trình lại, cho phép người dùng tùy chỉnh cấu trúc và chức năng của mạch điện ngay cả sau khi sản xuất. Về cấu tạo, FPGA bao gồm một mảng hai chiều các khối logic đa năng và các ma trận chuyển mạch (switch matrix) như minh họa trong hình 1.4. Các khối logic có khả năng thực hiện các chức năng logic cơ bản, trong khi các chuyển mạch đóng vai trò kết nối giữa các khối logic, từ đó tạo nên các mạch điện tử hoàn chỉnh theo thiết kế cụ thể của người dùng.



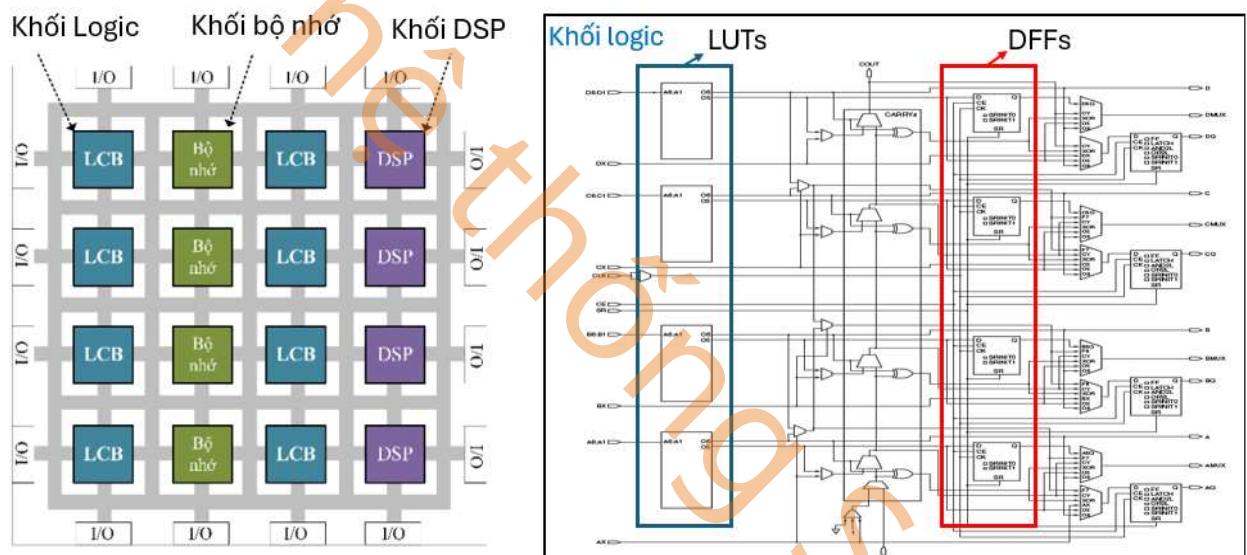
Hình 1.4: Cấu trúc mang tính khái niệm của một *FPGA*

Người dùng có thể cấu hình các khối logic để thực hiện các chức năng logic cụ thể và thiết lập các chuyển mạch nhằm liên kết các khối này theo cấu trúc mong muốn. Sau khi thiết kế và tổng hợp mạch trên phần mềm máy tính, cấu hình được nạp vào chip FPGA để triển khai mạch logic theo yêu cầu. Điều đặc biệt là quá trình này có thể thực hiện ngay tại chỗ (in the field), mà không cần phải sản xuất lại chip tại nhà máy (fab). Chính khả năng lập trình linh hoạt này là lý do cho tên gọi "field-programmable" của FPGA.

# Thiết kế bộ nhớ



Hình 1.5: Sơ đồ kết nối chi tiết của ma trận chuyển mạch



Hình 1.6: Minh họa cấu trúc một khối logic trong FPGA Xilinx [10]

Một khối logic thường bao gồm một mạch tổ hợp nhỏ có thể cấu hình, với các thành phần chính là bảng tra cứu (LUT - Look-Up Table) và flip-flop D (DFF), như minh họa trong hình 1.6. LUT hoạt động bằng cách sử dụng các ngõ vào làm tín hiệu định địa chỉ cho bộ nhớ. Nếu LUT có  $n$  ngõ vào, nó sẽ hình thành một bộ nhớ gồm  $2^n$  ô, mỗi ô chứa một giá trị 1 bit. LUT có thể được sử dụng để thực hiện các hàm logic tổ hợp bất kỳ, với kết quả được lưu trữ trong các ô nhớ tương ứng. Ngõ ra của bảng tra cứu (LUT) có thể được sử dụng trực tiếp trong thiết kế mạch tổ hợp hoặc được lưu trữ vào (DFF) trong thiết kế mạch tuần tự. Ngoài các khối logic thông thường, hầu hết FPGA còn tích hợp các khối

chức năng chuyên dụng, thường được gọi là macro cells hoặc macro blocks bao gồm: các khối bộ nhớ, khối xử lý tín hiệu số (DSP – Digital Signal Processing) chuyên dụng, và mạch quản lý đồng hồ (clock management circuits).

Dưới đây là một số đánh giá khách quan khi sử dụng FPGA trong việc tái cấu hình lại thiết kế:

**Về mặt thời gian:** FPGA là một thiết bị được cấu trúc từ các khối logic, hỗ trợ thiết kế xử lý đường ống (pipeline) ở mức độ cao. Điều này cho phép FPGA đạt hiệu suất tương đương hoặc thậm chí cao hơn các bộ xử lý truyền thống bằng cách thực thi các mạch tùy chỉnh theo cách song song, ngay cả khi tốc độ xung nhịp thấp hơn. Không giống như bộ xử lý vi mô, nơi cấu trúc pipeline cố định có thể không phù hợp với các yêu cầu cụ thể của ứng dụng, FPGA mang lại khả năng tái cấu hình linh hoạt, đáp ứng nhanh chóng và hiệu quả các thiết kế đặc thù. Nhờ khả năng này, FPGA không chỉ tối ưu hóa hiệu suất xử lý mà còn cải thiện hiệu suất tổng thể của hệ thống một cách đáng kể.

**Về mặt diện tích:** FPGA với khả năng tái cấu hình và linh hoạt cao thường được ưu tiên sử dụng trong giai đoạn phát triển và thử nghiệm sản phẩm nhờ khả năng thích ứng nhanh với những thay đổi trong thiết kế. Tuy nhiên, do cấu trúc lập trình được, FPGA thường chiếm nhiều diện tích hơn so với ASIC, vốn được tối ưu hóa đặc biệt cho một ứng dụng cụ thể nhằm giảm thiểu không gian trên chip. ASIC, mặc dù có chi phí và thời gian phát triển ban đầu cao, nhưng lại có lợi ích lớn về diện tích và hiệu suất trong sản xuất hàng loạt. Sự đánh đổi giữa FPGA và ASIC trong thiết kế SoC thể hiện sự cân nhắc giữa tính linh hoạt và hiệu quả diện tích. Trong khi FPGA cung cấp giải pháp linh hoạt và thích nghi tốt trong các giai đoạn phát triển và đánh giá thiết kế, ASIC lại vượt trội trong việc tối ưu hóa diện tích và hiệu suất cho các sản phẩm quy mô lớn. Lựa chọn giữa hai công nghệ này phụ thuộc vào nhiều yếu tố, bao gồm giai đoạn phát triển sản phẩm, yêu cầu về hiệu suất, chi phí, và thời gian đưa sản phẩm ra thị trường. FPGA thích hợp cho giai đoạn phát triển linh hoạt, trong khi ASIC là giải pháp tối ưu khi mục tiêu là hiệu quả tối đa trong sản xuất hàng loạt.

**Về mặt độ tin cậy:** Khả năng tái cấu hình của FPGA mang lại lợi thế đáng kể trong việc duy trì hiệu suất mạch và ổn định hoạt động, đặc biệt khi đối mặt với sự biến đổi trong quá

trình sản xuất bán dẫn. Tính năng tái cấu hình không chỉ giúp giảm thiểu tác động của các lỗi tiềm ẩn mà còn cho phép FPGA tự động "tự sửa chữa" bằng cách tái cấu hình để bỏ qua các phần mạch bị hỏng. So với các công nghệ mạch tích hợp truyền thống, FPGA vượt trội trong việc duy trì độ tin cậy ngay cả dưới các điều kiện biến đổi và không chắc chắn trong sản xuất. Điều này làm tăng đáng kể tính ứng dụng của FPGA trong các hệ thống yêu cầu độ ổn định cao, đặc biệt trong các lĩnh vực mà độ tin cậy là yếu tố quan trọng hàng đầu.

### 1.2.2. ASIC

ASIC, hay mạch tích hợp chuyên dụng cho ứng dụng, là một phần tử quan trọng trong lĩnh vực thiết kế điện tử, được thiết kế riêng để phục vụ một chức năng đặc biệt. Sự đặc biệt này tạo nên bản sắc rõ ràng của ASIC so với các mạch tích hợp đa năng khác như FPGA hay vi xử lý, vốn là những tiếp cận được thiết kế để xử lý nhiều tác vụ khác nhau hoặc có khả năng lập trình sau khi sản xuất để phù hợp với nhiều mục đích khác nhau. ASIC tỏa sáng nhờ khả năng tối ưu hóa tuyệt vời của nó từ hiệu suất công việc đến mức tiêu thụ năng lượng. Những điểm mạnh này đều được cải tiến đáng kể thông qua thiết kế cụ thể cho từng ứng dụng. Điều này không chỉ mang lại hiệu quả cao trong hoạt động mà còn giúp giảm đáng kể chi phí năng lượng, đặc biệt là khi sản xuất hàng loạt. Do đó, trong một số lĩnh vực cần đến sự chuyên biệt cao và hiệu quả năng lượng, ASIC thường là lựa chọn hàng đầu. Điều này chứng tỏ rằng việc đầu tư vào thiết kế ASIC có thể mang lại lợi ích lâu dài cả về kinh tế lẫn kỹ thuật cho các sản phẩm và dịch vụ đòi hỏi sự chính xác cao và đặc thù.

#### Đặc Điểm của ASIC

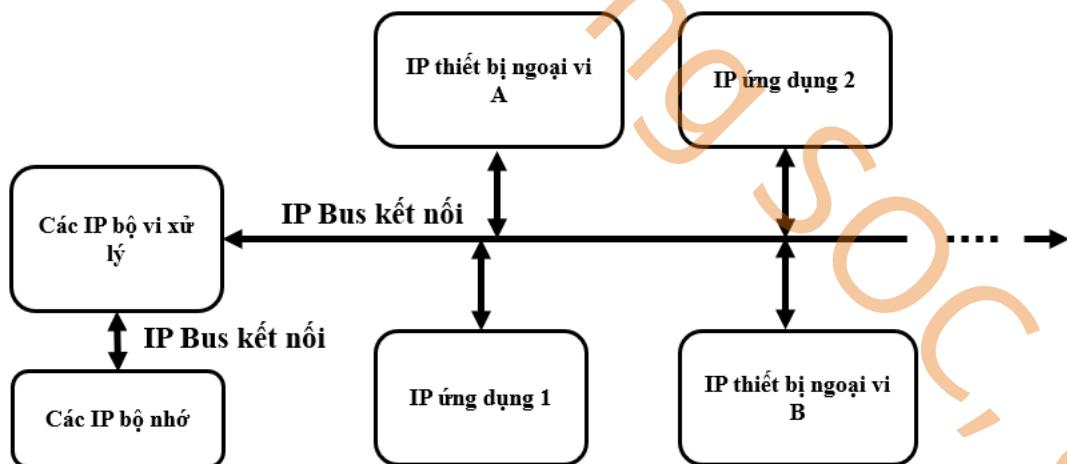
- ✓ **Tối ưu hóa hiệu suất:** Do được thiết kế riêng cho một ứng dụng cụ thể, ASIC có thể được tối ưu hóa để đạt hiệu suất cao nhất cho ứng dụng.
- ✓ **Tiêu thụ năng lượng thấp:** ASIC thường tiêu thụ ít năng lượng hơn so với các giải pháp dựa trên FPGA hoặc vi xử lý, nhờ loại bỏ các chức năng không cần thiết.
- ✓ **Chi phí sản xuất thấp:** Mặc dù chi phí phát triển ban đầu cao do yêu cầu thiết kế và sản xuất khuôn mẫu, nhưng khi sản xuất hàng loạt, chi phí trên mỗi đơn vị sẽ giảm đáng kể.

- ✓ **Bảo mật cao:** ASIC cung cấp mức độ bảo mật vượt trội vì chúng khó bị sao chép hoặc thay đổi sau khi sản xuất, nhờ thiết kế cố định và không thể tái lập trình.

### 1.3. SoC trên FPGA

SoC trên FPGA là minh chứng rõ ràng cho khái niệm hệ thống tích hợp trên chip. Công nghệ này cho phép tích hợp nhiều thành phần như bộ xử lý, bộ nhớ, giao tiếp ngoại vi và các khối logic tùy chỉnh, tất cả được cấu hình trên một chip FPGA duy nhất. Công ty Altera (nay thuộc Intel) và Xilinx (hiện thuộc sở hữu của AMD) là hai tên tuổi hàng đầu trong lĩnh vực phát triển SoC dựa trên FPGA. Các sản phẩm của họ cung cấp đa dạng sự lựa chọn cho nhiều ngành công nghiệp, từ tự động hóa trong sản xuất cho đến viễn thông. Sự đa dạng này cho phép tối ưu hóa và tùy chỉnh chuyên sâu theo nhu cầu cụ thể của từng ứng dụng. Nhờ vậy, các nhà phát triển có thể tận dụng tối đa hiệu suất của FPGA trong khi vẫn duy trì sự linh hoạt để cập nhật và mở rộng hệ thống. Điều này không chỉ đẩy nhanh tiến độ phát triển mà còn giúp giảm chi phí và thời gian đưa sản phẩm ra thị trường, một yếu tố then chốt trong môi trường cạnh tranh hiện nay.

#### 1.3.1. Các thành phần của SoC trên FPGA



Hình 1.7: Thành phần cơ bản trong hệ thống SoC.

Kiến trúc cơ bản của một SoC bao gồm bộ vi xử lý, bộ nhớ, hệ thống bus và các thành phần ngoại vi khác. Như minh họa trong hình 1.7, hệ thống SoC này được tạo thành từ nhiều tài sản sở hữu trí tuệ (IP - Intellectual Property). Hệ thống bao gồm một IP CPU, một

IP bộ nhớ, hai IP bus để kết nối, hai IP thiết bị ngoại vi và hai IP ứng dụng. Các IP này có thể được tạo ra theo hai cách: tự thiết kế để đáp ứng mục đích cụ thể hoặc sử dụng các IP có sẵn, được cung cấp bởi các nền tảng phần mềm chuyên dụng từ các nhà sản xuất FPGA.

**Bộ vi xử lý (CPU):** Trên các bảng mạch FPGA tích hợp hệ thống SoC, CPU đóng vai trò quan trọng trong việc xử lý các tác vụ nhúng, thực thi chương trình điều khiển, quản lý giao tiếp với các thiết bị ngoại vi, và điều phối hoạt động của các thành phần trong SoC. Tùy thuộc vào ứng dụng, CPU trong FPGA có thể đảm nhiệm các nhiệm vụ từ điều khiển thiết bị đơn giản đến xử lý dữ liệu phức tạp, như phân tích hình ảnh và tín hiệu số. CPU trong SoC thường được chia thành hai loại chính: CPU mềm (Soft CPU) và CPU cứng (Hard CPU).

- ✓ **CPU mềm:** Loại CPU này được triển khai bằng cách sử dụng các tài nguyên logic có thể lập trình của FPGA, mang lại tính linh hoạt cao và khả năng tùy chỉnh theo nhu cầu thiết kế. Một số ví dụ tiêu biểu của CPU mềm bao gồm MicroBlaze (Xilinx) và Nios II (Intel/Altera). CPU mềm thường được sử dụng trong các ứng dụng nhúng yêu cầu chi phí thấp hoặc không đòi hỏi hiệu suất xử lý quá cao. Với khả năng tùy chỉnh linh hoạt, chúng đặc biệt phù hợp trong các tình huống cần tối ưu hóa thiết kế cho các yêu cầu cụ thể.
- ✓ **CPU cứng:** Loại CPU này được tích hợp trực tiếp vào chip FPGA dưới dạng phần cứng cố định, mang lại hiệu suất cao và tiết kiệm tài nguyên logic của FPGA. Một số ví dụ phổ biến của CPU cứng bao gồm các dòng ARM Cortex, như Cortex-A9, Cortex-A53 hoặc Cortex-R5. Những CPU này thường được tích hợp trong các SoC FPGA nổi bật như Xilinx Zynq hoặc Intel Stratix 10, giúp tối ưu hóa hiệu năng và khả năng xử lý của hệ thống.

**Bộ nhớ (memory):** là một thành phần không thể thiếu trong hệ thống SoC, đảm nhiệm vai trò lưu trữ dữ liệu và chương trình cần thiết cho hoạt động của hệ thống. Trong SoC trên FPGA, bộ nhớ có chức năng chính là cung cấp không gian lưu trữ cho dữ liệu và mã lệnh, đồng thời hỗ trợ CPU và các thành phần khác thực hiện các tác vụ như xử lý tín hiệu hoặc

lưu trữ tạm thời. Bộ nhớ trong SoC trên FPGA thường được phân loại thành ba loại chính, mỗi loại phục vụ những mục đích khác nhau:

- ✓ **Bộ nhớ trên chip (On-chip Memory):** là loại bộ nhớ được tích hợp trực tiếp bên trong FPGA, thường bao gồm các loại như bộ nhớ RAM (Random Access Memory) với các khối bộ nhớ (BRAM - Block RAM) hoặc bộ nhớ chỉ đọc (ROM - Read Only Memory). Mặc dù dung lượng của bộ nhớ trên chip thường bị giới hạn, nhưng nó lại có tốc độ truy cập rất cao, đáp ứng tốt các ứng dụng yêu cầu xử lý dữ liệu nhanh. Loại bộ nhớ này thường được sử dụng cho các mục đích như làm bộ đệm (cache), lưu trữ tạm thời, hoặc lưu trữ các hằng số cấu hình đảm bảo hiệu suất vượt trội trong các tác vụ cần tốc độ cao.
- ✓ **Bộ nhớ ngoài chip (Off-chip Memory):** Được kết nối với FPGA thông qua các bus giao tiếp, loại bộ nhớ này có dung lượng lớn hơn, điển hình như SDRAM, DDR/DDR2/DDR3. Bộ nhớ ngoài thường được sử dụng để lưu trữ hệ điều hành, chương trình ứng dụng hoặc các dữ liệu lớn cần xử lý. Ví dụ, DDR3 là một loại bộ nhớ phổ biến được sử dụng trong các bảng mạch FPGA như Xilinx Zynq hoặc Intel Cyclone.
- ✓ **Bộ nhớ cấu hình (Configuration Memory):** Loại bộ nhớ này được sử dụng để lưu trữ cấu hình của FPGA (bitstream), giúp nạp cấu hình vào FPGA trong quá trình khởi động. Các loại bộ nhớ điển hình như Flash hoặc EEPROM thường được sử dụng để lưu trữ bitstream, đảm bảo tính ổn định và khả năng tái cấu hình nhanh chóng cho FPGA.

**Bus kết nối:** Bus kết nối trên các bảng mạch FPGA tích hợp SoC đóng vai trò trung tâm trong việc đảm bảo sự phối hợp nhịp nhàng và hiệu quả giữa các thành phần của hệ thống. Chúng chịu trách nhiệm truyền tải dữ liệu, tín hiệu điều khiển và duy trì sự đồng bộ giữa CPU, bộ nhớ và các thiết bị ngoại vi. Với việc áp dụng các chuẩn bus hiện đại như bus vi điều khiển nâng cao (AMBA - Advanced Microcontroller Bus Architecture), bus mở rộng nâng cao (AXI - Advanced eXtensible Interface), hoặc Avalon, hệ thống SoC trên FPGA

có thể linh hoạt đáp ứng các yêu cầu đa dạng. Những yêu cầu này bao gồm từ các ứng dụng điều khiển nhúng đơn giản đến các tác vụ xử lý dữ liệu phức tạp và hiệu suất cao.

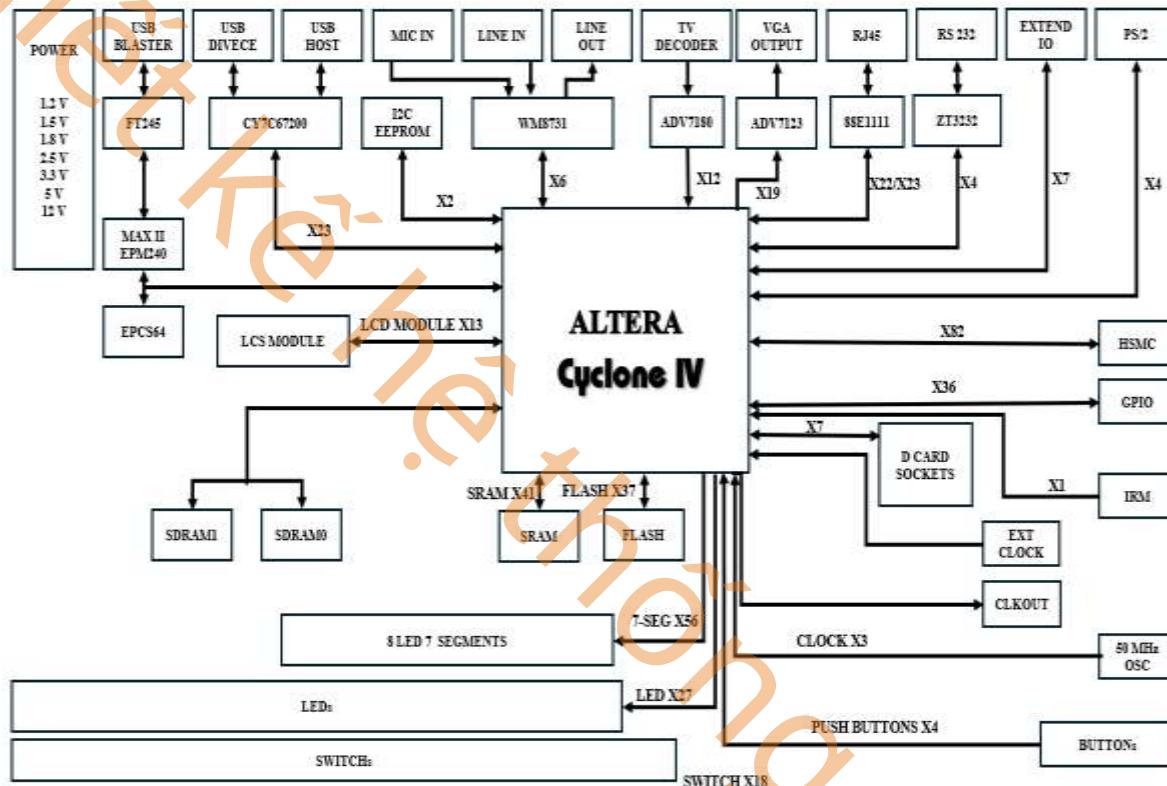
**IP thiết bị ngoại vi và IP tùy chỉnh:** Ngoài các thành phần chính bắt buộc trong một hệ thống SoC như CPU, bộ nhớ và bus, các hệ thống SoC trên FPGA còn có thể tích hợp thêm các IP tùy chỉnh (Custom IP), và IP thiết bị ngoại vi (Peripheral IP)..

- ✓ IP tùy chỉnh (Custom IP): Đây là các IP do người thiết kế tạo ra và tích hợp vào hệ thống SoC để thực hiện các chức năng đặc thù. Các chức năng này bao gồm xử lý tín hiệu số (DSP), nén hình ảnh, mã hóa dữ liệu, hoặc triển khai các thuật toán chuyên biệt cho mạng nhân tạo (AI – Artificial Intelligent) và học máy. Các công cụ phát triển như Xilinx Vivado hoặc Intel Quartus cung cấp môi trường thuận tiện để thiết kế và tích hợp các IP này.
- ✓ IP thiết bị ngoại vi (Peripheral IP): Đây là các IP được cung cấp sẵn trong thư viện FPGA, giúp kết nối và điều khiển các thiết bị ngoại vi thông qua các giao tiếp tiêu chuẩn. Một số IP phổ biến bao gồm:
  - UART, SPI, I2C: Hỗ trợ giao tiếp nối tiếp với cảm biến, thiết bị ngoại vi hoặc các mô-đun khác.
  - Ethernet: Cung cấp kết nối mạng, phù hợp cho các ứng dụng IoT hoặc truyền dữ liệu tốc độ cao.
  - DMA (Direct Memory Access): Tăng hiệu suất bằng cách truyền dữ liệu trực tiếp giữa bộ nhớ và thiết bị ngoại vi mà không phụ thuộc vào CPU.
  - GPIO (General Purpose Input/Output): Cung cấp giao tiếp linh hoạt để kết nối với các thiết bị phần cứng khác.

### 1.3.2. Altera SoC FPGA

Kết hợp cấu trúc FPGA hiệu suất cao với hệ thống bộ xử lý mạnh mẽ dựa trên nền tảng ARM, các SoC của Altera đã mang lại nhiều cơ hội mới trong thiết kế hệ thống nhúng. Dòng sản phẩm Stratix của họ, nổi bật với khả năng xử lý mạnh mẽ và tính năng an toàn cao, được các nhà thiết kế đánh giá cao trong các ứng dụng yêu cầu độ tin cậy và hiệu suất, như hàng không vũ trụ và y tế. Dòng Arria của Altera, nổi tiếng nhờ sự cân bằng tốt giữa

chi phí, hiệu suất và hiệu quả năng lượng, trở thành lựa chọn lý tưởng cho các ứng dụng như truyền thông dữ liệu, xử lý tín hiệu số và các hệ thống nhúng trong công nghiệp. Bên cạnh đó, dòng Cyclone với mức giá hợp lý và khả năng tích hợp cao đã trở thành lựa chọn phổ biến trong giảng dạy và phát triển các dự án nghiên cứu hoặc ứng dụng có quy mô nhỏ hơn, chẳng hạn như hệ thống học tập và nguyên mẫu thử nghiệm.



Hình 1.8: Ví dụ minh họa hệ thống SoC trên board DE2-115 của Altera.[5]

Hình 1.8 minh họa bảng mạch tích hợp SoC Cyclone của hãng Altera. Trung tâm của hệ thống là CPU Altera Cyclone IV, đóng vai trò bộ xử lý chính, kết nối và điều khiển các thành phần khác. Bộ nhớ của hệ thống bao gồm SRAM để lưu dữ liệu tạm thời, Flash để lưu dữ liệu cố định, và một bộ nhớ cấu hình EPROM dành cho thông tin khởi tạo. Hệ thống hỗ trợ nhiều giao tiếp ngoại vi, như USB, Ethernet, VGA, và giao tiếp âm thanh thông qua các chip chuyên dụng. Ngoài ra, hệ thống còn cung cấp giao diện người dùng như màn hình LCD, đèn LED, nút nhấn và công tắc, giúp hiển thị trạng thái và điều khiển trực tiếp. Bộ dao động xung nhịp và các thành phần cấp nguồn đảm bảo cho hệ thống hoạt động ổn

định. Tất cả các thành phần này được kết nối thông qua Avalon bus, tạo thành một hệ thống nhúng mạnh mẽ và linh hoạt.

Các SoC FPGA của Altera cung cấp nhiều giao tiếp ngoại vi và hỗ trợ số lượng lớn lõi IP, cho phép các nhà phát triển tùy chỉnh phần cứng mà không cần chuyển đổi sang nền tảng khác. Kết hợp với các công cụ phát triển phần mềm mạnh mẽ như Quartus II và bộ phát triển nhúng (EDS - Embedded Design Suite), Altera đã giữ vững vị thế là một trong những nhà cung cấp SoC FPGA hàng đầu thế giới. Sự hợp nhất giữa phần cứng mạnh mẽ và phần mềm linh hoạt tạo ra một giải pháp độc đáo, đáp ứng hiệu quả nhu cầu đa dạng của thị trường thiết kế hệ thống nhúng hiện đại.

**Bảng 1.1: Các dòng sản phẩm SoC chuyên dụng của hãng Altera**

Tên SoC	Lõi CPU	Bộ nhớ	Bus giao tiếp
Cyclone IV SoC	Lõi mềm 32-bit RISC	DDR3	Avalon
Cyclone V SoC	Hai lõi ARM Cortex-A9	DDR3	Avalon/AXI
Arria V SoC	Hai lõi ARM Cortex-A9	DDR3	Avalon/AXI
Stratix V SoC	Hai lõi ARM Cortex-A9	DDR3	Avalon/AXI
Arria 10 SoC	Hai lõi ARM Cortex-A9	DDR4	Avalon/AXI
Stratix 10 SoC	Bốn lõi ARM Cortex-A53	DDR4	Avalon/AXI

Bảng 1.1 liệt kê các dòng sản phẩm SoC hiện được tích hợp trong các bảng mạch FPGA của hãng Altera. Mỗi dòng SoC được thiết kế với các đặc điểm riêng để đáp ứng nhiều yêu cầu ứng dụng khác nhau. Về cấu trúc CPU, Cyclone IV SoC sử dụng lõi mềm 32-bit RISC, trong khi các dòng Cyclone V, Arria V, Stratix V, và Arria 10 SoC được trang bị hai lõi ARM Cortex-A9, mang lại hiệu năng xử lý cao. Đặc biệt, Stratix 10 SoC tích hợp bốn lõi ARM Cortex-A53, cung cấp khả năng xử lý đa luồng vượt trội. Về bộ nhớ, các dòng SoC này hỗ trợ DDR3 hoặc DDR4, đáp ứng nhu cầu băng thông cao và tốc độ truy cập nhanh trong các ứng dụng hiện đại. Ngoài ra, các SoC đều sử dụng các giao thức bus giao tiếp như Avalon và AXI, đảm bảo khả năng kết nối linh hoạt và trao đổi dữ liệu tốc độ cao giữa các thành phần hệ thống.

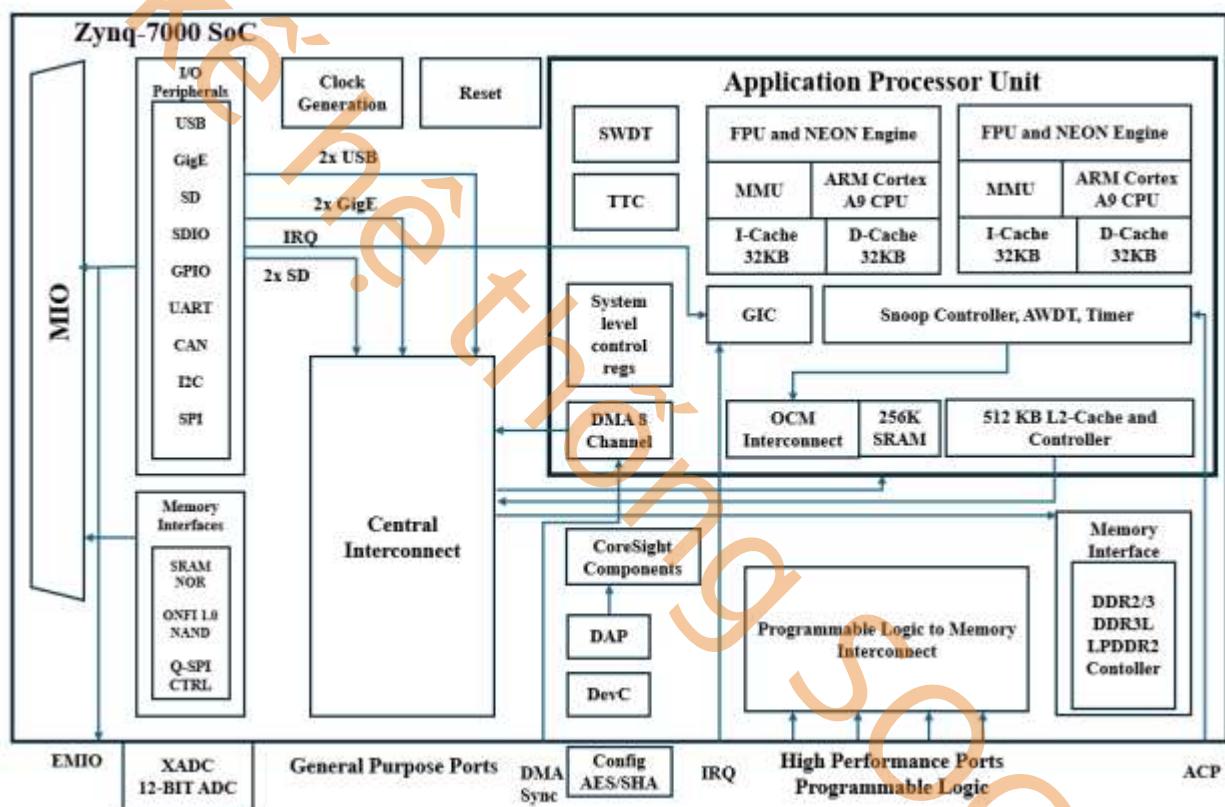
### **Một số đặc tính SoC FPGA của Altera:**

- ✓ Bộ xử lý ARM Cortex-A9 tích hợp: Cung cấp nền tảng xử lý mạnh mẽ, hỗ trợ chạy các hệ điều hành và phần mềm ứng dụng cấp cao.
- ✓ Cấu trúc FPGA hiệu suất cao: Cho phép thực hiện các chức năng tùy chỉnh và tăng tốc phần cứng, đặc biệt lý tưởng cho các tác vụ xử lý chuyên sâu, như xử lý tín hiệu số hoặc hình ảnh.
- ✓ Hỗ trợ thiết bị ngoại vi toàn diện: Tích hợp nhiều thiết bị ngoại vi, như Ethernet, USB, UART, và các giao thức khác, tạo điều kiện thuận lợi cho thiết kế hệ thống linh hoạt.
- ✓ Hỗ trợ phát triển phần cứng và phần mềm: Cung cấp các công cụ mạnh mẽ, như Quartus Prime để thiết kế phần cứng và EDS để phát triển phần mềm. Các công cụ này hỗ trợ nhiều hệ điều hành phổ biến, bao gồm Windows và Linux.

### 1.3.3. Xilinx SoC FPGA

Xilinx, một trong những nhà cung cấp hàng đầu trong lĩnh vực công nghệ FPGA, đã vượt qua nhiều thách thức kỹ thuật để phát triển các dòng sản phẩm SoC FPGA tiên tiến. Dòng Artix-7 và Spartan-7 là những sản phẩm được thiết kế đặc biệt để đáp ứng nhu cầu của các ứng dụng nhúng với chi phí thấp nhưng vẫn đảm bảo hiệu suất xử lý cao. Cả hai dòng chip này đều hỗ trợ MicroBlaze, một CPU lõi mềm mạnh mẽ, linh hoạt và có khả năng tùy chỉnh theo yêu cầu cụ thể của từng ứng dụng. Dòng Zynq-7000, được minh họa trong hình 1.9, đã đặt nền móng cho các hệ thống nhúng với khả năng tối ưu hóa chi phí và hiệu suất đáng kể. Trong hệ thống này, hai lõi ARM Cortex-A9 đóng vai trò xử lý chính, được hỗ trợ bởi bộ nhớ đệm cấp một (L1) và cấp hai (L2) để tăng tốc độ truy cập dữ liệu. Giao tiếp nội bộ thông qua AXI Bus được quản lý bởi một kênh kết nối trung tâm, đảm bảo các luồng dữ liệu giữa các thành phần hoạt động mượt mà. Hệ thống hỗ trợ nhiều giao diện ngoại vi như USB, Ethernet, UART, và SPI, giúp kết nối dễ dàng với các thiết bị bên ngoài. Ngoài ra, các giao diện bộ nhớ tích hợp hỗ trợ nhiều loại bộ nhớ như SRAM và DDR3, cung cấp không gian lưu trữ dữ liệu và chương trình. Phần logic lập trình được kết nối chặt chẽ với phần xử lý, cho phép mở rộng các chức năng tùy chỉnh theo yêu cầu cụ thể. Tiếp nối thành công của dòng Zynq-7000, dòng Zynq UltraScale+ MPSoC nâng tầm

hiệu suất với cấu trúc đa nhân và khả năng mở rộng mạnh mẽ. Hơn thế nữa, kiến trúc Versal ACAP, một nền tảng mới và cực kỳ mạnh mẽ, đã mở ra kỷ nguyên mới cho các giải pháp tính toán hiệu năng cao. Versal tích hợp các công cụ mạng nhân tạo (AI Engines) và công cụ xử lý tín hiệu (DSP Engines), mang đến sức mạnh vượt trội cho các ứng dụng tiên tiến như mạng máy tính, điện toán đám mây, và xe tự lái. Xilinx không ngừng cải tiến và mở rộng danh mục sản phẩm để phục vụ đa dạng các ngành công nghiệp, từ viễn thông, quốc phòng đến tiêu dùng. Điều này chứng tỏ Xilinx không chỉ dẫn đầu về công nghệ mà còn cung cấp các giải pháp toàn diện cho thị trường toàn cầu.



Hình 1.9: Ví dụ minh họa hệ thống SoC trên board Zynq-7000 của Xilinx.[3]

Bảng 1.2 trình bày thông tin về các dòng sản phẩm SoC và FPGA tiêu biểu từ Xilinx bao gồm , Artix-7, Spartan-7, Zynq-7000, Zynq UltraScale+ MPSoC, Versal ACAP. Các dòng này được thiết kế với những đặc điểm phần cứng và giao tiếp khác nhau để phục vụ nhiều mục đích ứng dụng từ nhúng cơ bản đến hệ thống AI. Đầu tiên với dòng FPGA Artix-7 và Spartan-7 sử dụng bộ xử lý mềm MicroBlaze, cho phép tùy chỉnh cấu trúc CPU theo

thiết kế. Cả hai dòng này phụ thuộc vào thiết kế trên FPGA để xác định các thông số RAM và giao tiếp, như AXI và liên kết đơn giản nhanh (FSL-Fast Simplex Link). Các dòng này phù hợp với ứng dụng nhúng nhỏ gọn và chi phí thấp. Tiếp theo với dòng Zynq-7000 tích hợp hai nhân ARM Cortex-A9 và hỗ trợ RAM DDR3, DDR3L, cùng các giao tiếp như AXI, PLB, và các cổng tốc độ cao (HP-High Performance ports). Dòng này thích hợp cho các ứng dụng nhúng và điều khiển công nghiệp. Dòng Zynq UltraScale+ MPSoC mang lại hiệu suất vượt trội với bốn nhân ARM Cortex-A53 (cho ứng dụng tổng quát), hai nhân ARM Cortex-R5 (cho ứng dụng thời gian thực), và GPU ARM Mali-400 MP2, hỗ trợ RAM DDR4, LPDDR4 và giao tiếp AXI, AHB, APB. Dòng này phù hợp với các ứng dụng xử lý tín hiệu số, AI, và các hệ thống phức tạp. Cuối cùng là dòng sản phẩm Versal ACAP (Adaptive Compute Acceleration Platform) được tối ưu hóa cho các ứng dụng AI, DSP, và tính toán hiệu năng cao, với hai nhân ARM Cortex-A72, hai nhân ARM Cortex-R5F, và hỗ trợ RAM DDR4, LPDDR4 cùng bus giao tiếp NoC (Network on Chip), mang lại khả năng xử lý và truyền dữ liệu nhanh chóng.

**Bảng 1.2: Một số SoC do hãng Xilinx cung cấp**

Tên SoC	Lõi CPU	Bộ nhớ	Bus giao tiếp
Artix-7	MicroBlaze (bộ xử lý lõi mềm)	Phụ thuộc vào thiết kế trên FPGA	AXI, FSL
Spartan-7	MicroBlaze (bộ xử lý lõi mềm)	Phụ thuộc vào thiết kế trên FPGA	AXI, FSL
Zynq-7000	Hai lõi ARM Cortex-A9	DDR3, DDR3L	AXI, PLB, HP
Zynq UltraScale+ MPSoC	Bốn lõi ARM Cortex-A53	DDR4, LPDDR4	AXI, AHB, APB
	Hai lõi ARM Cortex-R5		
	ARM Mali-400 MP2		
Versal ACAP	Hai lõi ARM Cortex-A72	DDR4, LPDDR4	AXI, NoC
	Hai lõi ARM Cortex-R5F		
	Tích hợp có chế AI, DSP		

### **Một số đặc tính SoC FPGA của Xilinx:**

- ✓ Bộ xử lý ARM tích hợp: Tùy thuộc vào dòng, Xilinx SoC FPGA tích hợp ARM Cortex-A9 (Zynq-7000) hoặc bộ xử lý ARM Cortex-A53 và Cortex-R5 tiên tiến hơn (Zynq UltraScale+ MPSoC), đáp ứng nhiều nhu cầu xử lý.
- ✓ Cấu trúc FPGA đa năng: cung cấp logic lập trình cho các bộ tăng tốc và giao diện tùy chỉnh, với UltraScale+ mang đến những cải tiến về hiệu suất, hiệu quả sử dụng điện năng và chức năng.
- ✓ Tính năng nâng cao: UltraScale+ MPSoC bao gồm các tính năng như GPU (dành cho các ứng dụng yêu cầu xử lý đồ họa), tính năng bảo mật nâng cao và tùy chọn kết nối tốc độ cao.
- ✓ Hệ sinh thái phần mềm: Xilinx cung cấp Vivado Design Suite cho thiết kế phần cứng và nền tảng phần mềm hợp nhất Vitis để phát triển các ứng dụng tận dụng cả hệ thống xử lý ARM và logic lập trình.

#### **1.3.4. So sánh các SoC FPGA của Altera và Xilinx**

Mặc dù cả hai công ty cung cấp các dòng SoC FPGA khác nhau, nhưng giữa chúng vẫn có những điểm khác biệt rõ rệt về cách tiếp cận, tính năng, và mục tiêu ứng dụng:

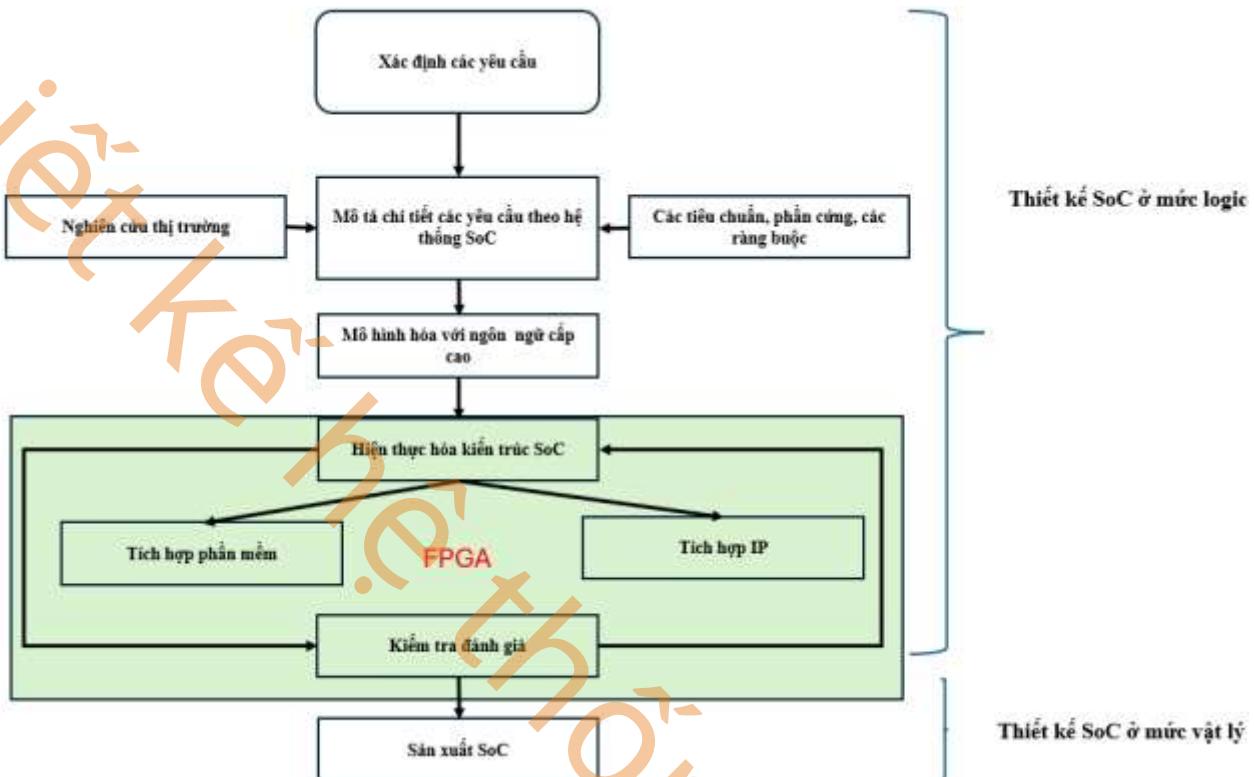
**Hiệu suất và năng lượng:** Cả Altera và Xilinx đều cung cấp các sản phẩm có phạm vi hiệu suất rộng, nhưng số liệu hiệu suất cụ thể có thể khác nhau tùy theo dòng sản phẩm. Ví dụ, UltraScale+ MPSoC của Xilinx được thiết kế cho các ứng dụng hiệu suất cao với lõi xử lý tiên tiến và kết cấu FPGA.

**Bảo mật và kết nối:** Xilinx đặc biệt chú trọng đến các tính năng bảo mật và kết nối tốc độ cao trong dòng UltraScale+, giúp nó phù hợp với các ứng dụng có yêu cầu nghiêm ngặt về bảo mật hoặc thông lượng dữ liệu.

**Công cụ phát triển và phần mềm:** Cả hai công ty đều cung cấp bộ công cụ phát triển phần mềm toàn diện, nhưng việc lựa chọn giữa Quartus Prime (Altera) và Vivado/Vitis (Xilinx) có thể tùy thuộc vào sở thích của người dùng, các tính năng cụ thể hoặc tối ưu hóa hiệu suất.

### **1.4. Quy trình thiết kế SoC**

Thiết kế hệ thống SoC, là một quá trình phức tạp và đa bước bao gồm việc lựa chọn, tối ưu hóa, và tích hợp các khối chức năng khác nhau trên một chip bán dẫn đơn lẻ. Hình 1.10 mô tả chi tiết các bước chính trong quá trình thiết kế SoC.

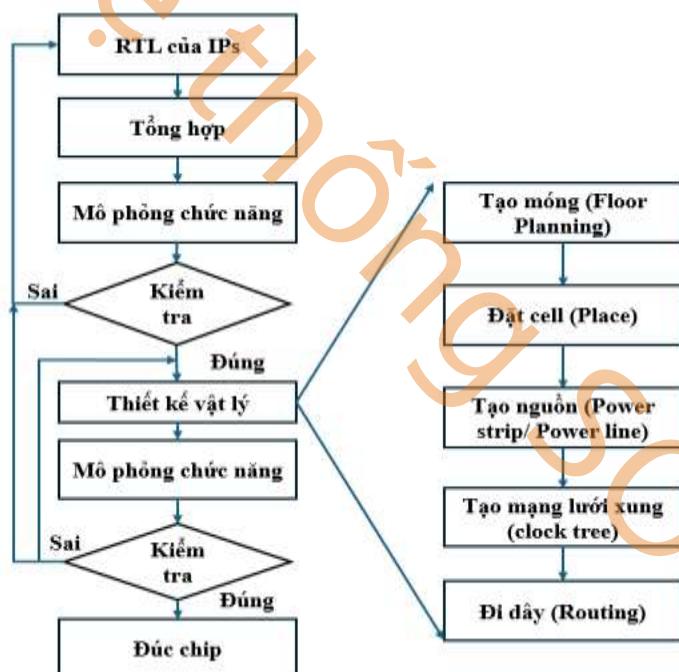


Hình 1.10: Minh họa quy trình thiết kế SoC tổng quát

Quy trình thiết kế SoC được chia thành hai giai đoạn chính: thiết kế logic và thiết kế vật lý. Giai đoạn thiết kế logic bắt đầu bằng việc xác định các yêu cầu cụ thể, dựa trên nghiên cứu thị trường và các tiêu chuẩn kỹ thuật, nhằm đảm bảo sản phẩm đáp ứng đúng mục tiêu. Tại bước này, các yêu cầu được mô tả chi tiết và chuyển đổi thành các mô hình khái niệm bằng các ngôn ngữ thiết kế cấp cao như Matlab, Python hoặc C/C++. Những mô hình này đóng vai trò định hình ý tưởng và xây dựng kiến trúc hệ thống. Khi kiến trúc được hoàn thiện, thiết kế ở mức RTL (Register Transfer Level) được tiến hành để hiện thực hóa cấu trúc SoC. Các khối IP được thiết kế riêng lẻ, sau đó kiểm tra và đánh giá nhằm đảm bảo chức năng của chúng đáp ứng yêu cầu ban đầu. Sau đó, các thành phần IP được tích hợp thành một hệ thống SoC hoàn chỉnh. Song song với quá trình này, phần mềm điều khiển cũng được phát triển và tối ưu hóa để đảm bảo sự phối hợp nhịp nhàng giữa phần

cứng và phần mềm, duy trì tính nhất quán và hiệu suất cao của hệ thống. Cuối cùng, hệ thống SoC hoàn chỉnh được kiểm tra toàn diện trước khi chuyển sang giai đoạn sản xuất.

*Lưu ý rằng SoC được thiết kế trên FPGA có thể dùng trực tiếp để chạy ứng dụng thực tế nhờ tính linh hoạt và khả năng tái cấu hình mạnh mẽ của FPGA. Ví dụ, trong các hệ thống xe tự hành ở châu Âu, FPGA có thể được sử dụng để nhận diện làn đường khi đi qua nhiều quốc gia khác nhau, nhờ khả năng xử lý song song mạnh mẽ và độ trễ thấp, giúp hệ thống phản ứng nhanh với các thay đổi trong môi trường thực tế. Ngoài ra, SoC trên FPGA cũng có thể được sử dụng như một thiết bị để mô phỏng (emulate), cho phép kiểm thử và đánh giá hiệu năng trước khi chuyển sang giai đoạn sản xuất. Điều này giúp phát hiện sớm lỗi thiết kế, tối ưu hóa hiệu suất, và giảm chi phí phát triển, đảm bảo rằng SoC khi chuyển sang thiết kế vật lý và sản xuất chip ASIC sẽ đạt được hiệu năng và chức năng như mong đợi.*

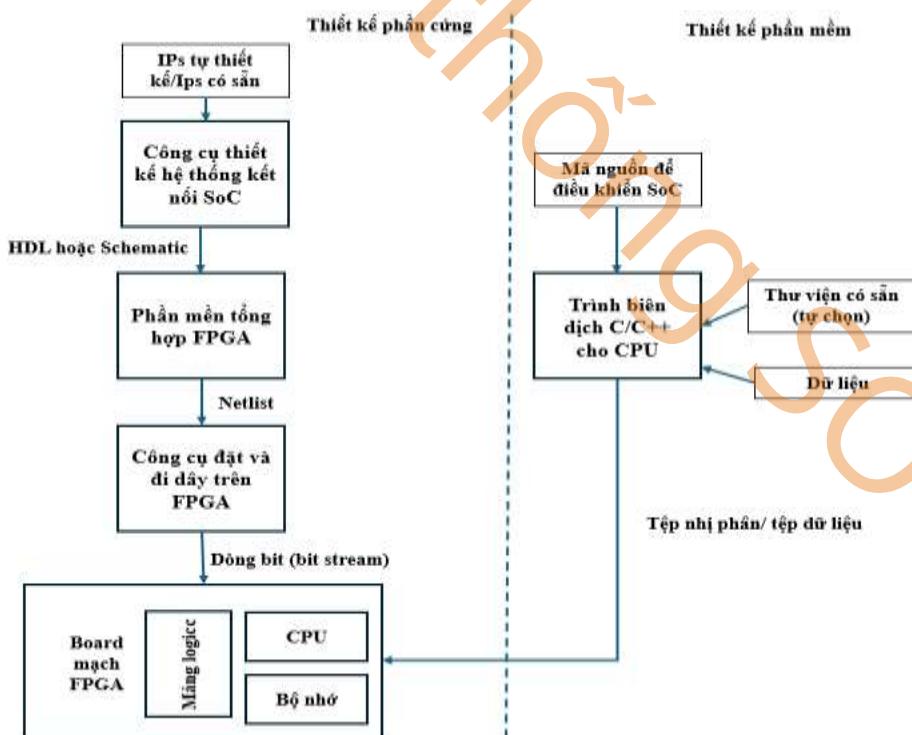


Hình 1.11: Quy trình thiết kế vật lý cho hệ thống SoC

Quy trình thiết kế vật lý (sản xuất chip) được minh họa trong hình 1.11. Quy trình thiết kế vật lý hệ thống SoC được thực hiện sau khi hoàn thành thiết kế mức logic và tổng hợp RTL của các IP. Giai đoạn đầu tiên là tạo kế hoạch mặt bằng (Floor Planning), nhằm xác định bố cục tổng thể của các thành phần trên chip, đảm bảo tối ưu không gian và hiệu

suất. Tiếp theo, các cổng logic (cell) được đặt vào các vị trí cụ thể (Place), dựa trên cấu trúc thiết kế và yêu cầu kết nối. Sau khi hoàn tất việc đặt cell, quy trình chuyển sang giai đoạn tạo nguồn (Power Planning), bao gồm việc thiết kế các đường cung cấp nguồn và các mạch phân phối năng lượng (Power Strip/Power Line), đảm bảo hệ thống nhận đủ năng lượng trong quá trình hoạt động. Tiếp đến là tạo mạng lưới xung nhịp (Clock Tree Synthesis), giúp đảm bảo tín hiệu xung nhịp được phân phối đồng đều và ổn định đến tất cả các thành phần của hệ thống. Giai đoạn cuối cùng trong thiết kế vật lý là đi dây (Routing), nơi các kết nối vật lý giữa các cell được thiết lập để hoàn thành mạch tích hợp. Sau khi đi dây, hệ thống sẽ được kiểm tra và mô phỏng chức năng để đảm bảo thiết kế đáp ứng các yêu cầu kỹ thuật. Nếu phát hiện lỗi, quy trình sẽ quay lại các bước trước đó để sửa đổi và tối ưu. Khi mọi kiểm tra đạt yêu cầu, hệ thống sẽ được đưa vào quy trình sản xuất để đúc chip.

#### 1.4.1. Thiết kế SoC trên FPGA



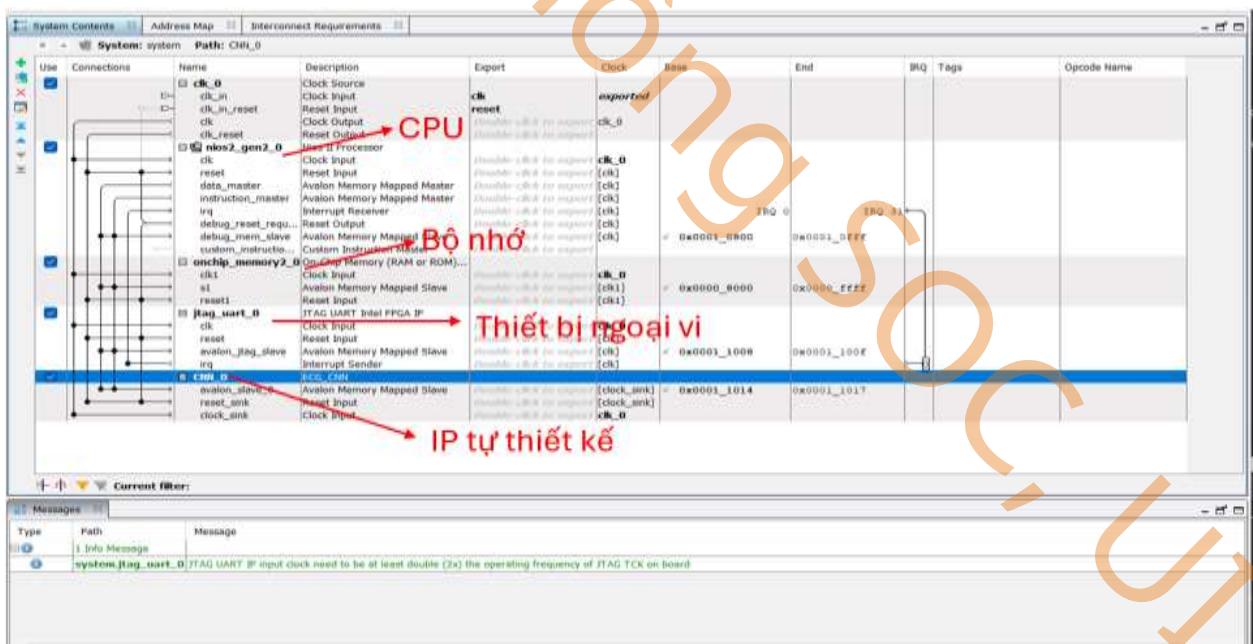
Hình 1.12: Quy trình thiết kế SoC trên FPGA.

Như đã đề cập, giáo trình này tập trung vào việc thiết kế hệ thống SoC ở mức logic. Theo mô tả ở hình 1.10, trước khi bắt đầu thiết kế một hệ thống SoC trên FPGA, người

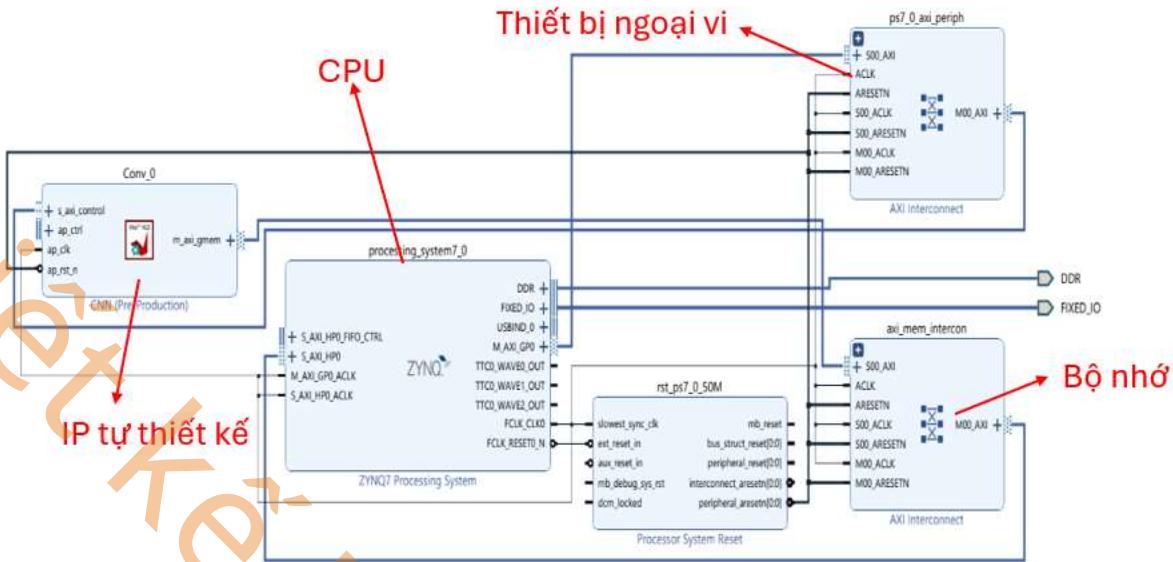
thiết kế cần hiểu rõ các khối chức năng của ứng dụng và hoàn thiện thiết kế RTL. Đồng thời, các khối chức năng này phải được kiểm tra kỹ lưỡng để đảm bảo đáp ứng các tiêu chí về tài nguyên phần cứng, chi phí và yêu cầu thị trường của ứng dụng. Khi thiết kế RTL đã được xác nhận đáp ứng đầy đủ các yêu cầu của ứng dụng, chúng sẽ được đóng gói thành các IP do người dùng tự thiết kế (Custom IP). Sau đó, quá trình thiết kế hệ thống SoC trên FPGA sẽ được triển khai. Một thiết kế hệ thống SoC hoàn chỉnh trên FPGA thường bao gồm hai phần chính: thiết kế phần cứng và thiết kế phần mềm. Hình 1.12 minh họa lưu đồ chi tiết của quy trình thiết kế một hệ thống SoC trên FPGA.

### Thiết kế phần cứng

Trong quá trình thiết kế phần cứng, người thiết kế có thể tích hợp các IP (bao gồm IP do nhà sản xuất cung cấp hoặc IP được phát triển riêng) bằng các công cụ thiết kế hệ thống SoC, chẳng hạn như Qsys (SoC Builder) hoặc Vivado. Sản phẩm của bước thiết kế phần cứng là một hệ thống SoC hoàn chỉnh, bao gồm CPU, bộ nhớ, các bộ tính toán phần cứng chuyên dụng, và giao tiếp ngoại vi. Kết quả ở giai đoạn này được mô tả dưới dạng ngôn ngữ mô tả phần cứng (HDL) hoặc một giản đồ (schematic).



Hình 1.13: Ví dụ kiến trúc SoC tích hợp IP CNN trên board Altera với phần mềm Quartus.



Hình 1.14: Ví dụ kiến trúc SoC tích hợp IP CNN trên board Xilinx với phần mềm Vivado.

Hình 1.13 minh họa một hệ thống phần cứng SoC được xây dựng trên phần mềm Qsys của Altera. Trong đó có một IP CPU Nios, một IP bộ nhớ trong, một IP thiết bị ngoại vi UART. Đây là những IP do nhà sản xuất cung cấp. Ngoài ra hệ thống còn tích hợp thêm một IP tự thiết kế. Toàn bộ hệ thống được kết nối thông qua bus Avalon. Tương tự trong hình 1.14 là phiên bản hiện thực hệ thống SoC bằng phần mềm Vivado của Xilinx với cùng ứng dụng như hình 1.13. Hệ thống SoC trong hình 1.14 cũng gồm các thành phần như vi xử lý CPU ARM Cortex-A9, bộ nhớ trong, thiết bị ngoại vi UART và IP tự thiết kế. Hệ thống kết nối thông qua bus AXI là điểm khác biệt so với kiến trúc SoC của Altera.

### Thiết kế phần mềm

Thiết kế phần mềm là dùng các phần mềm như NIOS II Embedded Design Suite khi làm việc với SoC Altera hoặc SDK khi làm việc với SoC Xilinx để viết mã điều khiển phần cứng. Phần mềm thường được viết bằng ngôn ngữ C/C++. Phần mềm này sẽ chạy trên lõi CPU mềm (NIOS II hoặc Microblaze) hoặc CPU cứng (ARM CPU) đã được tạo ra ở giai đoạn thiết kế phần cứng. Sau khi biên dịch các mã nguồn tương ứng với các lõi CPU từ phần cứng, trình biên dịch sẽ tạo ra các tệp dữ liệu/ chương trình nhị phân có thể chạy trên SoC. Hình 1.15 minh họa đoạn code viết ra từ phần mềm để điều khiển một hệ thống SoC phần cứng đã thiết kế ở hình 1.13 theo SoC FPGA Altera và hình 1.16 minh họa đoạn code C điều khiển hệ thống SoC FPGA Xilinx ở hình 1.14. Nhìn vào cú pháp của hai đoạn code

gần như giống với cách viết ngôn ngữ C/C++ thông thường do vậy người thiết kế rất dễ tiếp cận và điều khiển hệ thống phần cứng SoC.

```
#include "stdio.h"
#include "io.h"
#include "system.h"
void imagedtest(){
    IOWR(CNN_0_BASE, 0, 0x000000f8);
}
int main()
{
    print("Hello World\n\r");
    init_platform();
    imagedtest();
    parameters1();
    parameters2();
    parameters3();
    IOWR(CNN_0_BASE, 1);
    return 0;
}
```

Hình 1.15: Chương trình điều khiển khói IP CNN theo kiến trúc SoC Altera

```
#include "xil_io.h"
void imagedtest(){
    xil_Out32(0xA0000000+844, 0x000000f8);
}
int main()
{
    print("Hello World\n\r");

    init_platform();
    imagedtest();
    parameters1();
    parameters2();
    parameters3();
    xil_Out32(0xA4000000+0, 1);

    cleanup_platform();
    return 0;
}
```

Hình 1.16: Chương trình điều khiển khói IP CNN theo kiến trúc SoC Xilinx

Sau khi hoàn thiện thiết kế phần cứng và phần mềm, người thiết kế sẽ nạp phần cứng xuống FPGA tương ứng và sử dụng phần mềm để điều khiển hệ thống phần cứng đó. Tiếp theo, chức năng của hệ thống sẽ được kiểm tra để đảm bảo hoạt động chính xác. Quá trình này sẽ được lặp lại nhiều lần cho đến khi thiết kế đáp ứng đầy đủ các tiêu chuẩn đã đề ra.

## 1.5. Các yếu tố ảnh hưởng đến thiết kế SoC

Thiết kế SoC chịu ảnh hưởng bởi nhiều yếu tố, trong đó bao gồm mức độ tích hợp của các thành phần như CPU, GPU, và bộ nhớ. Độ phức tạp của ứng dụng đóng vai trò quyết định trong việc xác định quy mô và các chức năng cần thiết của SoC. Bên cạnh đó, các yếu

cần về tiêu thụ năng lượng và hiệu suất xử lý là những yếu tố quan trọng cần được tối ưu hóa. Đồng thời, việc đảm bảo khả năng kết nối và giao tiếp hiệu quả giữa các thành phần, cũng như tích hợp các tiêu chuẩn bảo mật, là những yếu tố không thể thiếu trong quá trình thiết kế SoC.

**Khả năng tích hợp:** SoC nổi bật với khả năng tích hợp cao, kết hợp các thành phần chính như CPU, GPU, và bộ nhớ trên cùng một chip. CPU trong SoC thường được tối ưu hóa để xử lý các tác vụ đa dụng, trong khi GPU đảm nhận các nhiệm vụ xử lý đồ họa và tính toán song song. Bộ nhớ, bao gồm RAM và bộ nhớ lưu trữ, được thiết kế tích hợp trực tiếp trên chip, giúp giảm độ trễ, tăng tốc độ truy cập dữ liệu, đồng thời tiết kiệm không gian và năng lượng. Khả năng tích hợp này không chỉ giúp tối ưu hóa hiệu suất trong một diện tích nhỏ mà còn tăng cường khả năng tương tác giữa các thành phần. Điều này mang lại một giải pháp hiệu quả cho các thiết bị di động và hệ thống nhúng. Bảng 1.3 minh họa các loại SoC hiện đại tích hợp nhiều loại vi xử lý và bộ nhớ khác nhau trên cùng một hệ thống. Ví dụ, SoC Apple A14 Bionic tích hợp cả CPU và GPU, mang lại hiệu suất vượt trội trong một thiết kế nhỏ gọn.

**Bảng 1.3: Minh họa tích hợp các thành phần khác nhau trong các loại SoC**

Tên SoC	CPU	GPU	Bộ nhớ
Qualcomm Snapdragon 888	1x Cortex-X1 @ 2.84GHz + 3x Cortex-A78 @ 2.42GHz + 4x Cortex-A55 @ 1.8GHz	Adreno 660	LPDDR5, tốc độ lên đến 3200MHz
Apple A14 Bionic	2x Firestorm + 4x Icestorm	Apple GPU 4-core	LPDDR4X
Samsung Exynos 2100	1x Cortex-X1 @ 2.9GHz + 3x Cortex-A78 @ 2.8GHz + 4x Cortex-A55 @ 2.2GHz	Mali-G78 MP14	LPDDR5, tốc độ lên đến 2750MHz

MediaTek Dimensity 1000	4x Cortex-A77 @ 2.6GHz + 4x Cortex-A55 @ 2.0GHz	Mali-G77 MC9	LPDDR4X
NVIDIA Tegra X1	4x Cortex-A57 + 4x Cortex-A53	NVIDIA Maxwell architecture 256-core	LPDDR4

Bảng 1.3 minh họa sự tích hợp các thành phần chính trong các loại SoC từ các nhà sản xuất hàng đầu như Qualcomm, Apple, Samsung, MediaTek và NVIDIA. Mỗi SoC kết hợp giữa các thành phần CPU, GPU và bộ nhớ để tối ưu hóa hiệu năng và đáp ứng các yêu cầu đặc thù của từng ứng dụng. Các CPU sử dụng kiến trúc đa lõi, bao gồm lõi hiệu năng cao và lõi tiết kiệm năng lượng, giúp cân bằng giữa tốc độ xử lý và hiệu quả năng lượng. GPU được thiết kế để xử lý đồ họa và tính toán, mang lại khả năng tăng tốc trong các ứng dụng AI, chơi game, và đồ họa cao cấp. Bộ nhớ RAM, với công nghệ LPDDR4X hoặc LPDDR5, hỗ trợ tốc độ cao và băng thông lớn, đảm bảo khả năng truyền dữ liệu nhanh chóng và giảm độ trễ. Sự kết hợp chặt chẽ giữa các thành phần này giúp các SoC đáp ứng hiệu quả các yêu cầu phức tạp trong thiết bị di động, máy tính bảng, và hệ thống chơi game.

**Công suất tiêu thụ:** là lượng năng lượng được tiêu thụ trong quá trình hoạt động. Điều này cực kỳ quan trọng trong các hệ thống nhúng, nhất là các thiết bị cầm tay sử dụng Pin. Một đơn vị thường dùng để so sánh là mW/MIPS (Miliwatts/MIPS). Giá trị này càng lớn thì công suất tiêu thụ càng lớn. Việc hướng đến công suất tiêu thụ thấp còn dẫn đến các đặc trưng khác của các thiết bị cầm tay như ít tỏa nhiệt, kích thước nhỏ, nhẹ, và thiết kế cơ khí đơn giản.

**Bảng 1.4: So sánh công suất tiêu thụ trên các dòng SoC FPGA khác nhau**

SoC	CPU	Bộ nhớ	Công suất (W)	MIPS	mW/MIPS

Xilinx Zynq UltraScale+ MPSoC ZU3EG	2x ARM Cortex-A53 @ 1.5GHz + 2x ARM Cortex-R5 @ 600MHz	DDR4, tốc độ lên đến 2.400GHz	15W - 60W	4200	8.928571
Intel Stratix 10 SX SoC	Quad-core ARM Cortex-A53 @ 1.5GHz	DDR4, tốc độ lên đến 2.6GHz	Khoảng 30W	6000	5
Xilinx Zynq-7000	Dual-core ARM Cortex-A9 @ 667MHz	LPDDR2, tốc độ lên đến 533MHz	3W - 15W	1334	6.746627
Intel Arria 10 SoC	Dual-core ARM Cortex-A9 @ 800MHz	DDR4, tốc độ lên đến 2.1GHz	Khoảng 15W	1600	9.375

Bảng 1.4 cung cấp thông tin so sánh giữa các dòng SoC FPGA khác nhau về mức tiêu thụ điện năng và hiệu suất. Cụ thể, dòng Xilinx Zynq UltraScale+ MPSoC ZU3EG được trang bị CPU với hai lõi ARM Cortex-A53 tốc độ 1.5GHz kết hợp với hai lõi ARM Cortex-R5 tốc độ 600MHz. Hệ thống sử dụng bộ nhớ DDR4 với tốc độ lên đến 2400MHz, và mức tiêu thụ điện năng dao động từ 15W đến 60W. Hiệu suất tính toán của SoC này đạt 4200 MIPS, tương ứng với mức tiêu thụ năng lượng 8.93 mW/MIPS. Trong khi đó, Intel Stratix 10 SX SoC sử dụng CPU Quad-core ARM Cortex-A53 tốc độ 1.5GHz, cùng bộ nhớ DDR4 có tốc độ tối đa 2666MHz. Mức tiêu thụ điện năng của SoC này vào khoảng 30W, với hiệu suất đạt 6000 MIPS, mang lại hiệu quả năng lượng cao nhất trong các SoC được so sánh, với chỉ 5 mW/MIPS. Với dòng Xilinx Zynq-7000, SoC này tích hợp CPU Dual-core ARM

Cortex-A9 tốc độ 667MHz, sử dụng bộ nhớ LPDDR2 có tốc độ tối đa 533MHz. Mức tiêu thụ điện năng của thiết bị dao động từ 3W đến 15W, với hiệu suất tính toán đạt 1334 MIPS, tương ứng với mức tiêu thụ năng lượng 6.75 mW/MIPS. Cuối cùng, Intel Arria 10 SoC được trang bị CPU Dual-core ARM Cortex-A9 tốc độ 800MHz và bộ nhớ DDR4 với tốc độ lên đến 2133MHz. Mức tiêu thụ điện năng của SoC này vào khoảng 15W, với hiệu suất tính toán đạt 1600 MIPS, và mức tiêu thụ năng lượng ở mức 9.38 mW/MIPS.

**Khả năng xử lý:** Trong các hệ thống SoC tốc độ xử lý được xem là yếu tố cốt lõi đánh giá sức mạnh của bộ xử lý. Thông số MIPS, viết tắt cho "Millions of instructions per second", là một chỉ số quan trọng giúp đo lường số lệnh mà bộ xử lý có thể xử lý trong một giây. Bộ xử lý có chỉ số MIPS cao thường mang lại tốc độ xử lý nhanh hơn, điều này đặc biệt quan trọng khi xét trong cùng một bối cảnh kỹ thuật. Máy tính đa năng hiện đại thường sử dụng các bộ xử lý 32-bit hoặc 64-bit để xử lý các tác vụ ngày càng phức tạp. Tuy nhiên, các hệ thống SoC nhỏ gọn như trong thiết bị điều khiển công nghiệp hay thiết bị đeo cá nhân không nhất thiết cần những bộ xử lý lớn này. Chúng thường được trang bị bộ xử lý có số bit ít hơn, ví dụ như 8-bit hoặc 16-bit, nhưng vẫn đảm bảo hiệu suất cao thông qua việc tối ưu hóa cho các ứng dụng cụ thể, chứng tỏ rằng không phải càng nhiều bit thì càng tốt, mà quan trọng là sự phù hợp của bộ xử lý với nhu cầu cụ thể của hệ thống.

**Chi phí phát triển sản phẩm:** Khi chi phí và độ phức tạp trong thiết kế tăng cao, một sự đánh đổi cơ bản xuất hiện giữa việc tối ưu hóa thiết kế và chi phí thực hiện. Để giải quyết vấn đề này, một số phương pháp tiếp cận đã được áp dụng. Phương pháp phổ biến nhất là sử dụng các thành phần đã được thiết kế sẵn hoặc áp dụng các thiết bị có khả năng tái cấu hình. Việc sử dụng các linh kiện thiết kế sẵn giúp giảm đáng kể thời gian và công sức cần thiết để phát triển từ đầu. Trong khi đó, các thiết bị tái cấu hình mang lại mức độ linh hoạt cao, cho phép sản phẩm dễ dàng điều chỉnh để phù hợp với nhiều yêu cầu khác nhau mà không cần thực hiện lại toàn bộ quá trình thiết kế. Khi được kết hợp một cách thông minh, các phương pháp này giúp cân bằng giữa chi phí và sự tối ưu hóa, đồng thời vẫn đảm bảo chất lượng và hiệu suất của sản phẩm cuối cùng.

**Mua IP:** Khi mục tiêu là tạo ra một thiết kế tối ưu hóa việc sử dụng công nghệ, chi phí cố định thường rất cao. Do đó, sản phẩm cuối cùng cần phải có tính ứng dụng rộng rãi để bù đắp khoản đầu tư ban đầu. Một phương án thay thế cho việc tối ưu từng phần của thiết kế là tái sử dụng các thiết kế hiện có. Mặc dù các thiết kế này có thể không đáp ứng hoàn hảo mọi yêu cầu của một công nghệ quy trình cụ thể, nhưng việc tiết kiệm thời gian và công sức thiết kế là lợi ích đáng kể. Việc mua lại các thiết kế từ các bên thứ ba được gọi là "bán quyền sở hữu trí tuệ bán dẫn" (Semiconductor Intellectual Property Core - SIPC). Việc sử dụng IP không chỉ giảm thiểu rủi ro trong phát triển thiết kế mà còn giúp giảm chi phí và rút ngắn thời gian đưa sản phẩm ra thị trường. Chi phí của một IP thường được xác định bởi quy mô sản xuất: khối lượng sản xuất càng lớn, chi phí trên mỗi đơn vị càng giảm. Điều này tạo điều kiện thuận lợi cho việc phân phối sản phẩm rộng rãi hơn và nhanh chóng tiếp cận thị trường.

**Bảng 1.5: Các loại nhân vi xử lý tích hợp dưới dạng IP cứng hoặc mềm**

Loại IP	Mức thiết kế	Mô tả
IP cứng có thể tùy chỉnh (Hard IP)	Mức vật lý	IP được sử dụng trong vi xử lý cứng, nhưng có thể tối ưu một số chức năng
IP được tổng hợp từ nhà máy (Firm IP)	Mức công	IP được xử dụng trong bộ đa vi xử lý nhưng cũng có thể tối ưu hóa
IP mềm có thể tổng hợp (Soft IP)	Mức RTL	IP được dùng trong bất kỳ thiết kế, không có thể chỉnh sửa

Bảng 1.5 mô tả đặc tính của các loại IP CPU khác nhau. Hard IP (sở hữu trí tuệ cứng) là các thiết kế ở cấp độ vật lý, tận dụng tất cả các tính năng có sẵn theo quy trình công nghệ, bao gồm thiết kế mạch và bố trí vật lý. Nhiều IP tương tự và IP tín hiệu hỗn hợp, chẳng hạn như SRAM và vòng khóa pha (PLL), được thiết kế theo định dạng này để đảm bảo hiệu suất tối ưu và các đặc tính thiết kế khác. Firm IP (sở hữu trí tuệ cố định) là các thiết

kế ở cấp độ cổng (gate level), có khả năng áp dụng trên nhiều cơ sở sản xuất với các công nghệ xử lý khác nhau. Soft IP (sở hữu trí tuệ mềm) là các thiết kế ở cấp độ logic, được cung cấp dưới dạng có thể tổng hợp (synthesizable) và áp dụng trực tiếp vào các công nghệ cell tiêu chuẩn. Phương pháp này cho phép người dùng tùy chỉnh mã nguồn để phù hợp với yêu cầu thiết kế trong nhiều trường hợp khác nhau. Soft IP mang lại mức độ linh hoạt cao trong quá trình tích hợp và tối ưu hóa IP vào hệ thống.

## 1.6. Ứng dụng của SoC trên FPGA

SoC (System on Chip) là sự tích hợp toàn bộ các thành phần vào trong một con chip hoặc trên một board mạch, mang lại những lợi ích vượt trội về hiệu suất, hiệu quả năng lượng, và kích thước nhỏ gọn. Sự kết hợp giữa bộ xử lý và các khối logic lập trình trong SoC trên FPGA đã tạo ra một nền tảng mạnh mẽ, lý tưởng để triển khai các hệ thống phức tạp và đáp ứng linh hoạt các yêu cầu khác nhau trong nhiều lĩnh vực công nghệ cao. Dưới đây là tổng quan các lĩnh vực ứng dụng chính của SoC trên FPGA:

Trong lĩnh vực trí tuệ nhân tạo (AI) và học sâu (Deep Learning): SoC trên FPGA được sử dụng để tăng tốc các mô hình xử lý dữ liệu, như mạng nơ-ron sâu (DNN – Deep Neural Network) và mạng tích chập (CNN – Convolutional Neural Network). FPGA cho phép người dùng tối ưu hóa các khối tính toán ma trận, giúp tăng tốc độ xử lý và giảm độ trễ, điều này rất quan trọng trong các ứng dụng yêu cầu thời gian thực như nhận diện hình ảnh, phân tích video và xe tự hành.

Trong viễn thông: SoC trên FPGA đóng vai trò cốt lõi trong các hệ thống 5G, với khả năng triển khai các giao thức truyền thông phức tạp như MIMO (Multiple Input Multiple Output) và xử lý tín hiệu số. FPGA cung cấp băng thông cao và khả năng xử lý song song, giúp đáp ứng các yêu cầu nghiêm ngặt về tốc độ truyền dữ liệu và độ trễ thấp trong các hệ thống mạng hiện đại.

Trong ứng dụng nhúng: SoC trên FPGA được sử dụng trong các hệ thống điều khiển công nghiệp, thiết bị y tế và IoT thông minh. Nhờ khả năng lập trình và tích hợp linh hoạt, SoC trên FPGA có thể đảm nhận vai trò của một bộ điều khiển tùy chỉnh. Đồng thời, nó cung cấp các giao diện kết nối với cảm biến và các thiết bị ngoại vi khác.

Ngoài ra, SoC trên FPGA cũng được ứng dụng rộng rãi trong hệ thống an ninh và quốc phòng, nơi cần hiệu năng cao và độ bảo mật cao. FPGA có khả năng thực thi các thuật toán mã hóa và giải mã phức tạp, xử lý dữ liệu thời gian thực và đáp ứng các yêu cầu khắt khe về tính bảo mật trong các hệ thống radar, truyền thông mã hóa và giám sát.

Nhờ sự kết hợp giữa phần cứng linh hoạt và phần mềm mạnh mẽ, SoC trên FPGA đã trở thành một lựa chọn lý tưởng trong việc triển khai các giải pháp công nghệ phức tạp và đáp ứng các yêu cầu ngày càng cao của ngành công nghiệp hiện đại.

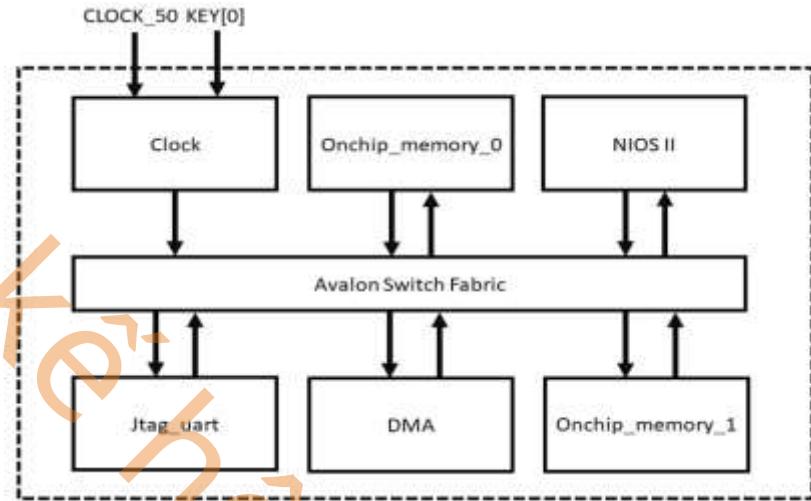
### 1.7. Tóm tắt

Trong chương này, chúng tôi đã giới thiệu khái niệm về FPGA và ASIC, đồng thời phân tích sự khác biệt giữa hai loại vi mạch tích hợp này. Tiếp theo, chương trình bày khái niệm về hệ thống SoC trên FPGA, bao gồm các thành phần cơ bản của một hệ thống SoC. Chúng tôi cũng điểm qua một số loại SoC khác nhau, phân loại dựa trên kiến trúc của Altera và Xilinx, giúp người đọc nhận thấy sự khác biệt giữa hai kiến trúc này. Phần tiếp theo của chương đi sâu vào mô tả chi tiết các bước thiết kế một hệ thống SoC, từ quy trình thiết kế chung đến các bước cụ thể trên FPGA. Đồng thời, chương cung cấp kiến thức về các yếu tố ảnh hưởng đến hiệu năng và hiệu quả của hệ thống SoC, giúp người đọc cân nhắc những đánh giá cần thiết khi thiết kế cho một ứng dụng cụ thể. Cuối cùng, chương kết thúc với phần trình bày về các ứng dụng phổ biến hiện nay có tích hợp hệ thống SoC, mang đến một cái nhìn thực tế và toàn diện hơn về lĩnh vực này.

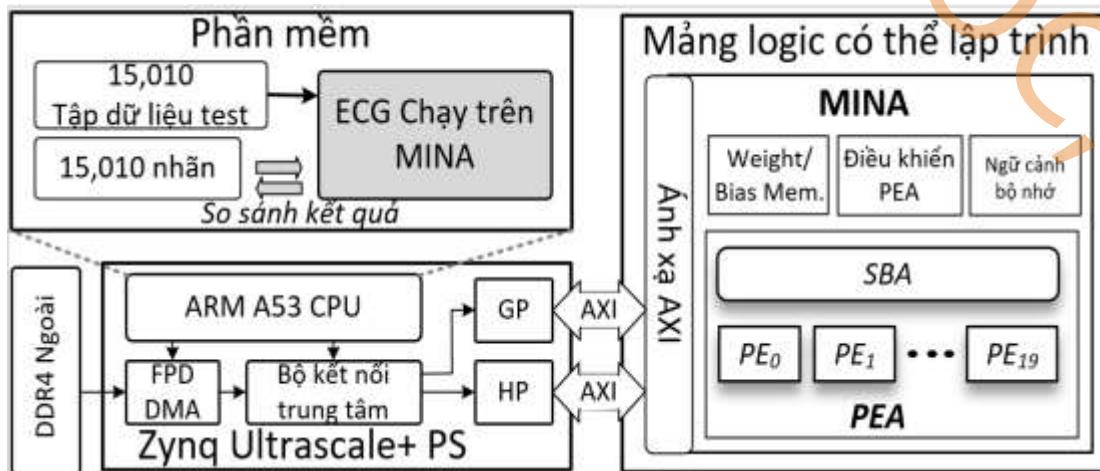
### 1.8. Câu hỏi và bài tập

1. FPGA là gì, cấu trúc cơ bản của một FPGA gồm những thành phần nào?
2. Phân biệt sự giống và khác nhau giữa FPGA và ASIC
3. Nêu khái niệm như thế nào là một hệ thống SoC. Cho ví dụ minh họa.
4. Một hệ thống SoC tối thiểu cần thiết phải có những thành phần nào. Kể tên một số ví dụ cho từng loại thành phần trong hệ thống SoC đã nêu.

5. Nhìn vào hình bên dưới, cho biết các thành phần có trong hệ thống SoC trên. Theo bạn thành phần nào tự thiết kế, thành phần nào là những IP có sẵn. Lưu ý trong đó Clock là khối chức năng đã có không xem là một IP.



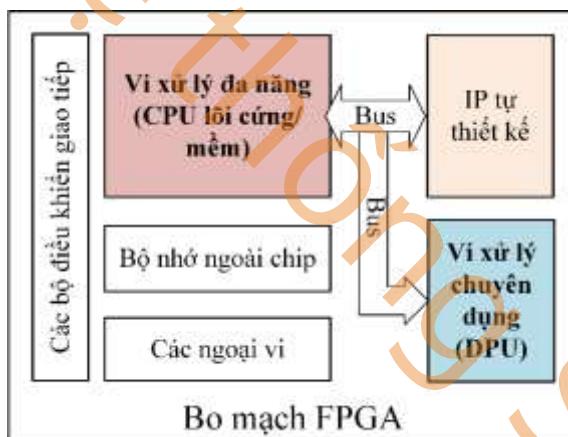
6. Nêu quy trình thiết kế SoC tổng quát.  
 7. Nêu quy trình thiết kế SoC trên FPGA. Phân tích ý nghĩa từng bước nêu trong quy trình trên.  
 8. Nêu điểm giống và khác nhau giữa SoC Altera và SoC Xilinx.  
 9. Kể tên một số ứng dụng được thiết kế dựa trên nền tảng SoC FPGA.  
 10. Theo bạn trong hệ thống SoC FPGA, IP CPU, bộ nhớ nên được lấy từ nhà cung cấp hay tự thiết kế. Vì sao?  
 11. Nhìn vào hình sau và cho biết trong hệ thống SoC trên gồm những thành phần nào. Liệt kê và nêu chức năng của các thành phần đó.



## CHƯƠNG 2: VI XỬ LÝ TRÊN SOC FPGA

### 2.1. Giới Thiệu

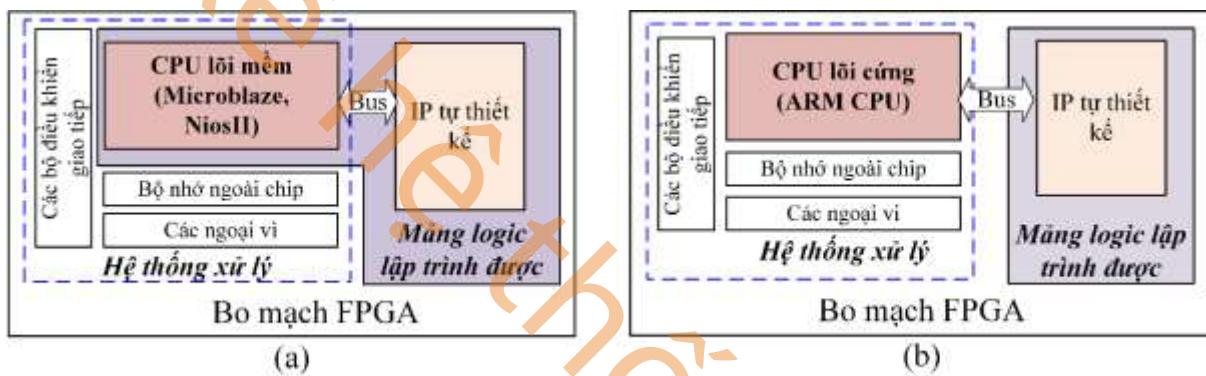
Vi xử lý trên SoC FPGA là một thành phần quan trọng trong các hệ thống điện tử hiện đại, kết hợp giữa tính linh hoạt của FPGA và khả năng xử lý của các vi xử lý. SoC FPGA cho phép kết hợp nhiều chức năng trong một hệ thống duy nhất, bao gồm các vi xử lý đa năng (CPU lõi cứng hoặc lõi mềm) và các vi xử lý chuyên dụng như DPU (Deep Learning Processing Unit). Vi xử lý đóng vai trò như "bộ não" của hệ thống, tiếp nhận các tín hiệu, xử lý thông tin và đưa ra quyết định để điều phối hoạt động của toàn bộ hệ thống, từ việc thực hiện các phép toán phức tạp đến việc quản lý giao tiếp giữa các thiết bị ngoại vi và bộ nhớ. Việc sử dụng vi xử lý trong SoC FPGA giúp tăng cường khả năng linh hoạt và tối ưu hóa hệ thống, đồng thời giảm thiểu kích thước và chi phí của hệ thống, vì các chức năng điều khiển, tính toán và giao tiếp đều có thể tích hợp vào một hệ thống duy nhất.



Hình 2.1: Cấu trúc hệ thống SoC FPGA với vi xử lý đa năng và vi xử lý chuyên dụng.

Hình 2.1 mô tả cấu trúc của một hệ thống SoC FPGA, bao gồm các thành phần chính như vi xử lý đa năng (CPU lõi cứng/lõi mềm) và vi xử lý chuyên dụng, kết nối với nhau qua hệ thống kết nối bus. Vi xử lý đa năng không chỉ có nhiệm vụ điều khiển toàn bộ hoạt động của hệ thống mà còn điều khiển các IP tự thiết kế hoặc vi xử lý chuyên dụng, quản lý giao tiếp với bộ nhớ ngoài chip (ví dụ bộ nhớ DRAM), quản lý giao tiếp với các bộ điều khiển giao tiếp như Ethernet MAC, đầu cảm USB, SD/MMC, bộ điều khiển bộ nhớ NAND Flash, v.v., và các thiết bị ngoại vi như UART, JTAG, GPIO, v.v.. IP tự thiết kế có thể là

phần cứng chuyên dụng giúp tăng tốc một ứng dụng cụ thể nào đó, chẳng hạn như xử lý tín hiệu số, mã hóa/giải mã dữ liệu, hoặc thực hiện các phép toán phức tạp trong thời gian ngắn hơn so với khi xử lý trên CPU. Bên cạnh đó, vi xử lý chuyên dụng cũng có thể được tích hợp vào hệ thống tương tự như các IP tự thiết kế, giúp tăng cường hiệu suất cho các tác vụ tính toán đặc thù, chẳng hạn như các ứng dụng AI và học sâu (deep learning). DPU, với khả năng tối ưu hóa các phép toán tensor và ma trận, xử lý các tác vụ phức tạp mà vi xử lý tổng quát không thể xử lý hiệu quả, giảm tải cho CPU và mang lại hiệu suất vượt trội trong các ứng dụng yêu cầu tính toán cao như nhận dạng hình ảnh, xử lý ngôn ngữ tự nhiên, và các mô hình học máy.



**Lưu ý:** Mặc dù cụm từ hệ thống xử lý thường được sử dụng trong các dòng Xilinx Zynq FPGA, chẳng hạn như ZCU102 và ZCU104, để chỉ một IP tích hợp bao gồm CPU

*lõi cứng ARM, các ngoại vi và bộ nhớ DRAM, nhưng trong sách này, hệ thống xử lý sẽ được hiểu rộng hơn. Cụ thể, hệ thống xử lý sẽ bao gồm cả CPU lõi cứng hoặc lõi mềm cùng với các ngoại vi, thay vì chỉ giới hạn ở CPU lõi cứng. Do đó, khi nhắc đến hệ thống xử lý, ý nghĩa của nó sẽ không bị giới hạn như cách gọi IP hệ thống xử lý trên các dòng Xilinx Zynq FPGA, mà sẽ được hiểu là một nhóm các thành phần có nhiệm vụ xử lý và điều khiển chung cho hệ thống SoC FPGA. Điều này giúp dễ hình dung hơn và phù hợp với cách trình bày nhất quán trong toàn bộ sách.*

Trong phần tiếp theo, chúng ta sẽ mô tả chi tiết về các loại vi xử lý mềm, vi xử lý cứng, vi xử lý chuyên dụng, cũng như giới thiệu sơ qua về cách tích hợp vi xử lý tự thiết kế vào SoC FPGA để đáp ứng các yêu cầu tính toán đặc thù của hệ thống.

## 2.2. Kiến trúc vi xử lý đa năng lõi mềm trên FPGA

Kiến trúc vi xử lý (CPU) lõi mềm trên board FPGA là một giải pháp linh hoạt và mạnh mẽ trong thiết kế hệ thống nhúng. Khác với các vi xử lý cứng, CPU lõi mềm cho phép người thiết kế cấu hình và tùy chỉnh nhiều yếu tố của vi xử lý như: số lượng thanh ghi, bộ nhớ cache, các đơn vị chức năng, cũng như các giao thức kết nối. Các CPU lõi mềm như Nios II của Altera (Intel) và MicroBlaze của Xilinx cho phép tích hợp trực tiếp vào các bảng mạch FPGA, tận dụng khả năng tái cấu hình của FPGA để điều chỉnh vi xử lý sao cho phù hợp với yêu cầu của hệ thống. CPU lõi mềm trên SoC FPGA có một số ưu điểm và nhược điểm như sau:

### **Ưu điểm:**

- ✓ Linh hoạt và có thể tùy chỉnh: CPU lõi mềm cho phép thay đổi cấu trúc và các tính năng của vi xử lý để đáp ứng yêu cầu cụ thể của từng ứng dụng. Người thiết kế có thể dễ dàng điều chỉnh các tham số như số lượng thanh ghi, kích thước bộ nhớ, hoặc các mô-đun giao tiếp mà không cần thay đổi phần cứng.
- ✓ Tiết kiệm chi phí và diện tích: Mặc dù CPU lõi mềm tiêu tốn tài nguyên phần cứng của FPGA, nhưng việc sử dụng CPU lõi mềm có thể giúp giảm chi phí và diện tích trong một số trường hợp, vì CPU này có thể tích hợp trực tiếp vào FPGA mà không

cần một CPU riêng biệt. Việc tối ưu hóa phần cứng cho các tác vụ cụ thể giúp giảm bớt sự cần thiết phải sử dụng các vi xử lý hoặc phần cứng tách biệt.

- ✓ **Khả năng mở rộng và thích ứng:** CPU lõi mềm có thể dễ dàng mở rộng và điều chỉnh khi hệ thống phát triển hoặc có sự thay đổi trong yêu cầu ứng dụng. Điều này rất quan trọng trong các môi trường phát triển nhanh và thay đổi liên tục, giúp ứng dụng linh hoạt đáp ứng nhu cầu thay đổi.
- ✓ **Tích hợp dễ dàng với các phần cứng khác:** CPU lõi mềm có khả năng tích hợp linh hoạt với các IP tự thiết kế và các mô-đun phần cứng khác trong SoC FPGA, giúp tối ưu hóa hệ thống cho các ứng dụng đặc thù mà không cần thay đổi cấu trúc phần cứng tổng thể.

#### **Nhược điểm:**

- ❖ **Tần số hoạt động thấp:** CPU lõi mềm thường có tần số hoạt động trong khoảng 100-200 MHz, trong khi CPU lõi cứng có thể đạt tới 1.3 GHz hoặc cao hơn. Sự khác biệt này làm giảm hiệu suất xử lý, đặc biệt trong các ứng dụng yêu cầu tính toán nhanh và xử lý dữ liệu lớn.
- ❖ **Hiệu suất thấp hơn CPU lõi cứng:** Do CPU lõi mềm phải sử dụng các phần tử phần cứng lập trình trên FPGA nên tần số hoạt động thấp, hiệu suất của nó không cao bằng CPU lõi cứng đã được tối ưu hóa sẵn cho các tác vụ tính toán phức tạp.
- ❖ **Tiêu thụ tài nguyên FPGA:** CPU lõi mềm sử dụng tài nguyên FPGA như logic cells, bộ nhớ, và các I/O blocks. Việc này có thể làm giảm số lượng tài nguyên còn lại cho các ứng dụng khác hoặc IP tự thiết kế. Trong một số trường hợp, việc sử dụng CPU lõi mềm có thể chiếm một phần lớn tài nguyên của FPGA, hạn chế khả năng sử dụng cho các tác vụ khác.
- ❖ **Tiêu thụ năng lượng cao hơn:** CPU lõi mềm, mặc dù linh hoạt, có thể tiêu thụ năng lượng cao hơn so với CPU lõi cứng do phải sử dụng thêm các tài nguyên phần cứng để thực thi các tác vụ tính toán ở các hệ thống yêu cầu vi xử lý đa năng phức tạp.
- ❖ **Khả năng xử lý song song hạn chế:** CPU lõi mềm yếu hơn nhiều so với CPU lõi cứng về mặt hiệu suất tính toán. Do CPU lõi mềm phải chia sẻ tài nguyên FPGA và

không được tối ưu hóa như CPU lõi cứng, nó không hiệu quả trong việc xử lý các tác vụ tính toán song song quy mô lớn hoặc các ứng dụng đòi hỏi tối ưu hóa phần cứng chuyên biệt, như học sâu hoặc xử lý tín hiệu số quy mô lớn.

### 2.2.1. Kiến trúc vi xử lý Nios II



Hình 2.3: Kiến trúc vi xử lý Nios II [9]

CPU Nios II hiện được tích hợp trên các bảng mạch FPGA thuộc họ Cyclone (II, III, IV) hoặc họ Stratix (II, III, IV). Kiến trúc của bộ vi xử lý Nios II tuân theo mô hình RISC pipelined cổ điển, bao gồm 32 thanh ghi đa dụng, ba định dạng lệnh (R, I, J) và hỗ trợ lệnh 32-bit cùng với bus dữ liệu 32-bit. Hệ thống sử dụng cấu trúc thanh ghi phẳng để tối ưu hóa hiệu suất, đồng thời có bộ nhớ cache riêng biệt cho lệnh và dữ liệu với kích thước có thể cấu hình. Ngoài ra, Nios II cung cấp tùy chọn bộ nhớ kết chặt chẽ nhằm tăng tốc độ truy xuất dữ liệu, tích hợp cơ chế dự đoán nhánh giúp cải thiện hiệu quả thực thi lệnh. Bộ vi xử lý này hỗ trợ tối đa 32 ngắt ưu tiên và tích hợp các phần cứng chuyên dụng để thực hiện các phép toán như nhân, dịch và xoay bit ngay trên chip, giúp tăng hiệu suất xử lý. Hệ thống còn bao gồm các đơn vị quản lý bộ nhớ (MMU) và bảo vệ bộ nhớ (MPU), giúp kiểm soát truy cập bộ nhớ tốt hơn, đặc biệt hữu ích trong các hệ thống đa nhiệm. Nios II được gọi là CPU mềm vì người dùng có thể thêm hoặc loại bỏ các thành phần như bộ

nhớ cache, đơn vị xử lý số thực (FPU), quản lý bộ nhớ (MMU), điều khiển ngắt, giúp tối ưu tài nguyên phần cứng FPGA.

Hình 2.3 mô tả sơ đồ khái niệm của bộ vi xử lý Nios II, thể hiện các thành phần chính và cách chúng kết nối với nhau trong vi xử lý. Các mô-đun gồm:

➤ Thành phần lõi của vi xử lý Nios II:

- Bộ điều khiển chương trình & tạo địa chỉ (Program Controller & Address Generation): Chịu trách nhiệm điều phối luồng thực thi chương trình và quản lý địa chỉ bộ nhớ.
- Thanh ghi đa dụng (General Purpose Registers) và thanh ghi trạng thái & điều khiển (Status & Control Registers): Lưu trữ dữ liệu tạm thời và thông tin điều khiển hoạt động của bộ xử lý.
- Bộ nhớ lệnh cache (Instruction Cache) và bộ nhớ dữ liệu cache (Data Cache): Giúp tăng tốc độ truy xuất dữ liệu bằng cách lưu trữ tạm thời các lệnh và dữ liệu thường xuyên sử dụng.
- Bộ nhớ liên kết chặt (Tightly Coupled I-Memory & D-Memory): Được sử dụng để giảm độ trễ trong quá trình truy cập dữ liệu và lệnh quan trọng.
- Bộ xử lý số học và logic (Arithmetic Logic Unit - ALU): Thực hiện các phép toán số học và logic cơ bản.
- Bộ xử lý lệnh logic tùy chỉnh (Custom Instruction Logic): Cho phép mở rộng tập lệnh để đáp ứng các yêu cầu đặc biệt của ứng dụng.

➤ Hệ thống quản lý bộ nhớ và ngắt

- Bộ quản lý bộ nhớ (Memory Management Unit - MMU) và bộ bảo vệ bộ nhớ (Memory Protection Unit - MPU): Đảm bảo quản lý và bảo vệ truy cập bộ nhớ, đặc biệt trong các hệ thống đa nhiệm.
- Bộ điều khiển ngắt (Interrupt Controller): Quản lý các tín hiệu ngắt (irq[31..0]), giúp xử lý các sự kiện từ ngoại vi hoặc các tác vụ ưu tiên.
- Bộ điều khiển ngoại lệ (Exception Controller): Xử lý các lỗi và sự kiện ngoại lệ trong quá trình thực thi chương trình.

- Hỗ trợ gỡ lỗi và theo dõi hệ thống
  - Bộ hỗ trợ gỡ lỗi phần cứng (Hardware-Assisted Debug Module): Cung cấp giao diện JTAG để kết nối với trình gỡ lỗi phần mềm.
  - Cổng truy vết (Trace Port) và Bộ nhớ truy vết (Trace Memory): Cho phép giám sát và phân tích quá trình thực thi chương trình.
  - Bộ truy vết lệnh & dữ liệu (Instruction and Data Trace) và Điểm dừng phần cứng (HW Breakpoints): Giúp kiểm tra và tối ưu hóa hiệu suất hệ thống.
- Các cổng giao tiếp bên ngoài
  - Giao tiếp với bộ nhớ và các ngoại vi để lấy lệnh và dữ liệu.
  - Hỗ trợ kết nối với các thiết bị ngoại vi tùy chỉnh.
  - Điều khiển xung nhịp và thiết lập lại bộ xử lý.
- Màu sắc đại diện trong sơ đồ
  - Xanh lá (Fixed): Thành phần bắt buộc của Nios II.
  - Xanh dương (Optional): Thành phần tùy chọn có thể được thêm vào.
  - Xanh lam đậm (Configurable): Thành phần có thể cấu hình theo yêu cầu.
  - Đỏ (Debug): Thành phần hỗ trợ gỡ lỗi và theo dõi hệ thống.

**Bảng 2.1: So sánh các loại CPU Nios II**

	Nios II /f (Nhanh)	Nios II /s (Tiêu chuẩn)	Nios II /e (Tiết kiệm)
<b>Thông lượng lệnh (Tối đa)</b>	1 lệnh / chu kỳ	1 lệnh / chu kỳ	1 lệnh / 6 chu kỳ
<b>Số lệnh hoạt động tối đa trong pipeline</b>	6	5	1
<b>Bộ nhân &amp; dịch bit bằng phần cứng</b>	1 chu kỳ	3 chu kỳ	Mô phỏng bằng phần mềm
<b>Dự đoán nhánh</b>	Động	Tĩnh	Không có
<b>Bộ nhớ đệm lệnh</b>	Cấu hình được	Cấu hình được	Không có
<b>Bộ nhớ đệm dữ liệu</b>	Cấu hình được	Không có	Không có

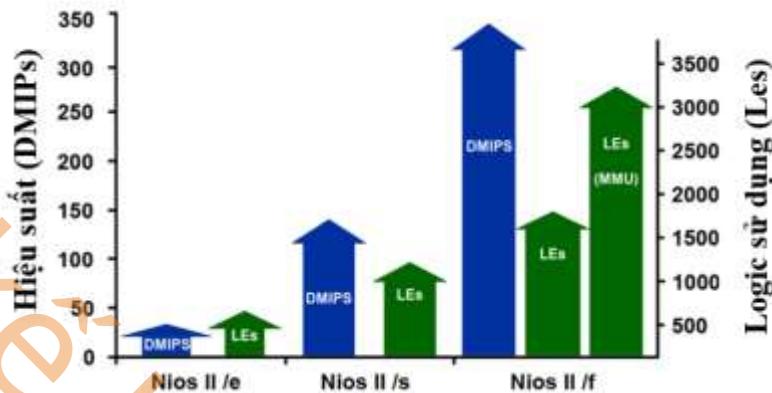
<b>Yêu cầu tài nguyên logic (Số LEs điển hình)</b>	1800 không hoặc không có MMU hoặc 3200 có MMU	1200	600
<b>Lệnh tùy chỉnh</b>	Tối đa 256	Tối đa 256	Không có

Dòng vi xử lý Nios II có ba phiên bản chính, bao gồm Nios II/f (nhanh), Nios II/s (chuẩn) và Nios II/e (tiết kiệm), mỗi loại được tối ưu hóa cho những mục đích khác nhau như thông tin so sánh cụ thể trong bảng 2.1.

- Nios II/f (nhanh (Fast)): Phiên bản này được tối ưu hóa cho tốc độ, hỗ trợ dự đoán rẽ nhánh động, bộ nhân phần cứng hoạt động trong 1 chu kỳ, bộ nhớ cache có thể cấu hình và có thể thực thi một lệnh trên mỗi chu kỳ xung nhịp. Đây là lựa chọn phù hợp cho các ứng dụng yêu cầu hiệu suất cao.
- Nios II/s (chuẩn (Standard)): Đây là phiên bản cân bằng giữa tốc độ và kích thước, sử dụng dự đoán rẽ nhánh tĩnh, bộ nhân phần cứng hoạt động trong 3 chu kỳ, bộ nhớ cache có thể cấu hình và cũng có thể thực thi một lệnh trên mỗi chu kỳ xung nhịp. Nó phù hợp cho các ứng dụng yêu cầu sự cân đối giữa hiệu suất và tài nguyên phần cứng.
- Nios II/e (tiết kiệm (Economy)): Phiên bản này được tối ưu hóa để tiết kiệm tài nguyên phần cứng, không có bộ nhớ cache, không hỗ trợ dự đoán rẽ nhánh và thực hiện phép nhân bằng phần mềm thay vì phần cứng, dẫn đến hiệu suất thấp hơn. Nó chỉ có thể thực hiện một lệnh trên mỗi 6 chu kỳ xung nhịp, nhưng sử dụng ít tài nguyên logic hơn, phù hợp với các thiết kế có yêu cầu thấp về hiệu suất nhưng cần tiết kiệm tài nguyên.

Ngoài ra, khả năng nhân phần cứng của vi xử lý Nios II cũng có sự khác biệt giữa các phiên bản. Phiên bản tiết kiệm không có bộ nhân phần cứng và phải sử dụng thư viện toán học phần mềm để thực hiện phép nhân, dẫn đến thời gian xử lý lâu hơn (khoảng 250 chu kỳ cho phép nhân 32x32). Trong khi đó, phiên bản chuẩn và nhanh đều hỗ trợ bộ nhân phần cứng, giúp thực hiện phép nhân nhanh hơn đáng kể, lần lượt trong 3 chu kỳ và 1 chu kỳ khi sử dụng khói DSP của Stratix FPGA. Các bộ xử lý Nios II có thể hỗ trợ phần cứng

nhân thông qua các khối DSP của dòng thiết bị Stratix, các khối nhân của dòng Cyclone hoặc thông qua các phần tử logic (LE) với phương pháp nhân bằng dịch và quay (~11 chu kỳ cho mỗi phép nhân).

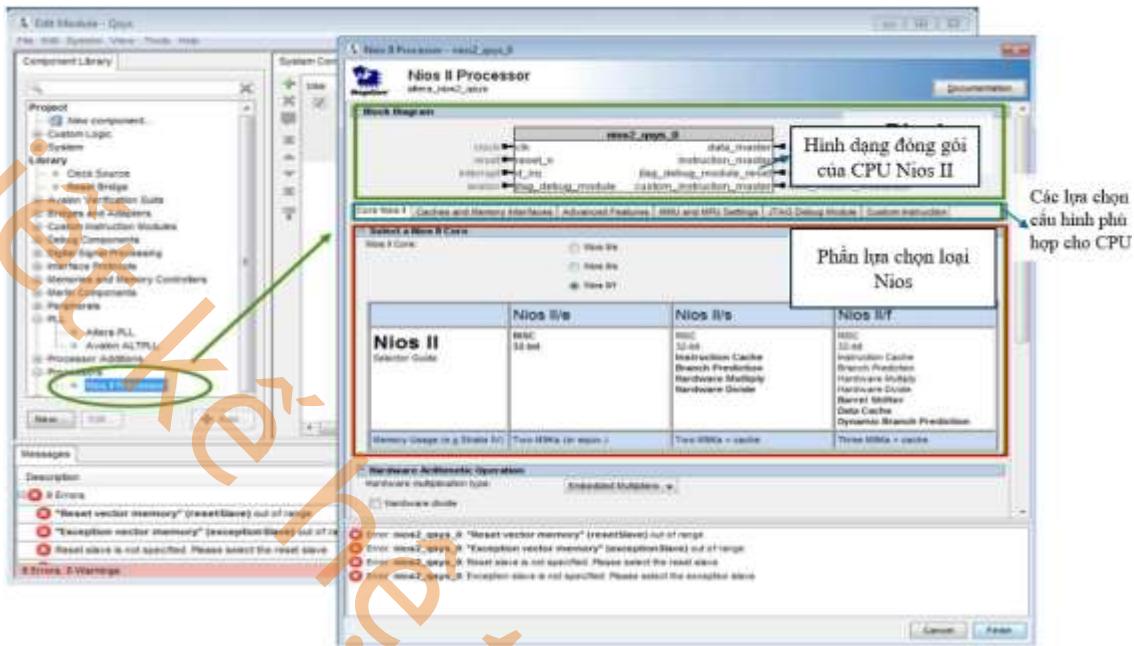


Hình 2.4: So sánh hiệu suất và lượng logic cần dùng cho từng loại vi xử lý Nios II [9]

Hình 2.4 thể hiện phạm vi hiệu suất của các phiên bản vi xử lý Nios II, được đo bằng chỉ số Dhystone MIPS (DMIPS) và số lượng phần tử logic (LEs) sử dụng. Với đơn vị (Dhystone Dhystone per second (Dhystones/s)): số vòng lặp của bài test thực hiện được trong một giây. Nios II/e có hiệu suất thấp nhất với chỉ số DMIPS khá nhỏ, đồng thời sử dụng ít tài nguyên logic nhất. Điều này phù hợp với các ứng dụng yêu cầu tiết kiệm tài nguyên phần cứng nhưng không đòi hỏi hiệu suất cao. Nios II/s có hiệu suất trung bình, cao hơn đáng kể so với phiên bản tiết kiệm. Số lượng phần tử logic sử dụng cũng nhiều hơn một chút, thể hiện sự cân bằng giữa hiệu suất và tài nguyên phần cứng. Nios II/f đạt hiệu suất cao nhất, với DMIPS cao hơn hẳn so với hai phiên bản còn lại. Tuy nhiên, số lượng phần tử logic sử dụng cũng cao nhất, đặc biệt khi có thêm đơn vị quản lý bộ nhớ, yêu cầu tài nguyên phần cứng lớn hơn nhiều so với các phiên bản khác.

Hình 2.5 mô tả giao diện phần mềm Qsys, một công cụ của Intel Quartus Prime dùng để thiết kế hệ thống phần cứng trên FPGA. Bên trái là cửa sổ thư viện IP (Component Library), nơi người dùng có thể thêm các thành phần phần cứng vào thiết kế, trong hình đang chọn Nios II. Bên phải là cửa sổ cấu hình bộ vi xử lý Nios II, cho phép chọn các thông số như loại lõi CPU (Nios II/e, Nios II/s, Nios II/f), các tính năng phần cứng bổ sung (bộ nhớ cache, đơn vị nhân phần cứng, bộ bảo vệ bộ nhớ, v.v.). Những lựa chọn này minh họa

cho tính năng mềm của CPU Nios II, một đặc tính khác biệt so với các loại CPU cứng được nhắc đến ở phần kế tiếp.



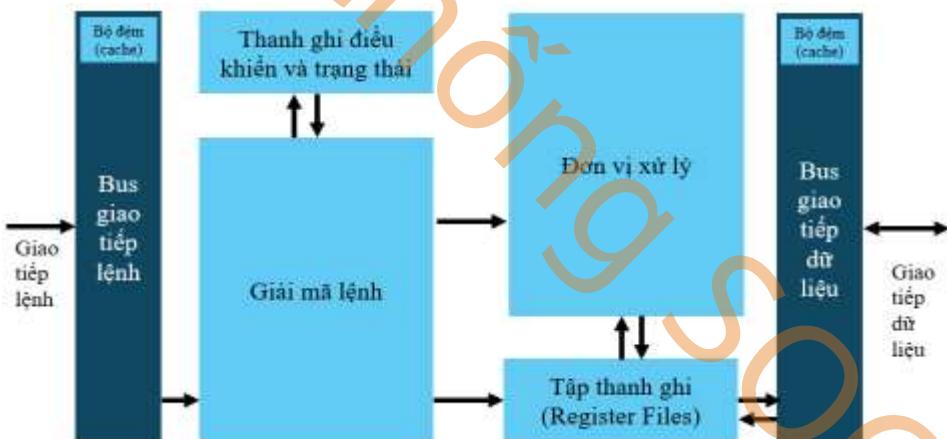
Hình 2.5: Nơi lấy IP Nios II và cấu hình vi xử lý theo bộ công cụ Qsys.

### 2.2.2. Kiến trúc vi xử lý MicroBlaze

MicroBlaze là một bộ vi xử lý mềm (soft CPU) do Xilinx phát triển, được thiết kế để hoạt động trên các thiết bị FPGA thuộc họ như Spartan-6, Artix-7, Virtex-7. Đây là một vi xử lý dựa trên kiến trúc Harvard RISC, hỗ trợ cả cấu hình 32-bit và 64-bit, mang lại hiệu suất cao và khả năng tùy chỉnh linh hoạt. MicroBlaze sở hữu tập lệnh độc lập (orthogonal), cho phép các lệnh sử dụng bất kỳ thanh ghi nào một cách linh hoạt, tối ưu hóa việc lập trình và hiệu suất. Kiến trúc Harvard của MicroBlaze tách biệt hoàn toàn bus lệnh và bus dữ liệu, giúp tăng cường băng thông và giảm xung đột khi truy cập bộ nhớ. Bộ xử lý này tích hợp 32 thanh ghi đa dụng 32-bit, cung cấp không gian lưu trữ tạm thời lớn cho các phép toán phức tạp. Ngoài ra, MicroBlaze còn có tùy chọn bộ dịch vòng (barrel shifter) 32-bit, hỗ trợ dịch và xoay bit một cách hiệu quả trong một chu kỳ xung nhịp. Đặc biệt, MicroBlaze tích hợp các giao diện bộ nhớ nhanh như bộ nhớ trên chip (OCM - On-Chip Memory) và bus ngoại vi trên chip (OPB - On-chip Peripheral Bus) tiêu chuẩn của IBM, giúp tăng cường khả năng giao tiếp với bộ nhớ trong và các ngoại vi, từ đó nâng cao hiệu suất toàn hệ thống. Điểm mạnh của MicroBlaze là khả năng tích hợp dễ dàng với các ngoại

vi, bộ nhớ và giao diện khác nhau, giúp nhà phát triển tùy chỉnh phù hợp với từng ứng dụng cụ thể. Ngoài ra, nó hỗ trợ các tính năng nâng cao như các bộ tăng tốc phần cứng, bộ nhớ cache, bộ quản lý bộ nhớ (MMU) và đơn vị xử lý số thực (FPU - Floating Point Unit), giúp tăng tốc xử lý các tác vụ phức tạp.

MicroBlaze được gọi là một bộ xử lý mềm vì nó cho phép cấu hình linh hoạt trên FPGA mà không bị ràng buộc vào phần cứng cố định. Người dùng có thể tùy chỉnh kiến trúc, bật hoặc tắt các thành phần như bộ nhớ cache, đơn vị xử lý số thực, bộ quản lý bộ nhớ và bộ điều khiển ngắt để tối ưu hiệu suất hoặc tiết kiệm tài nguyên. Nhờ được lập trình hoàn toàn trên tài nguyên logic của FPGA, MicroBlaze có thể được mở rộng hoặc thu gọn tùy vào yêu cầu ứng dụng mà không cần thay đổi phần cứng vật lý. MicroBlaze cũng hỗ trợ nhiều mức hiệu suất khác nhau, từ phiên bản tối giản tiết kiệm tài nguyên đến phiên bản mạnh mẽ với 5 tầng pipeline, bộ nhớ cache và khả năng chạy hệ điều hành Linux. Ngoài ra, khả năng nâng cấp và thay đổi linh hoạt giúp MicroBlaze có thể mở rộng quy mô hệ thống bằng cách tăng số lõi xử lý hoặc tích hợp thêm các bộ gia tốc phần cứng.



Hình 2.6: Kiến trúc vi xử lý Microblaze [22]

Hình 2.6 mô tả kiến trúc bên trong của bộ xử lý MicroBlaze. Các thành phần chính trong sơ đồ kiến trúc MicroBlaze gồm:

- **Bus giao tiếp lệnh (Instruction Bus Interface):** Kết nối với bộ nhớ lệnh và bộ nhớ đệm để lấy lệnh từ bộ nhớ ngoài.

- Giải mã lệnh (Instruction Decode): Nhận các lệnh từ bộ nhớ và giải mã chúng để gửi đến các đơn vị xử lý (Execution Units). Kết nối với thanh ghi điều khiển và trạng thái để theo dõi và điều chỉnh hoạt động của CPU.
- Các đơn vị thực thi (Execution Units): Xử lý các lệnh đã được giải mã, thực hiện các phép toán số học, logic và điều khiển.
- Tập thanh ghi (Register File): Chứa các thanh ghi dùng để lưu trữ dữ liệu tạm thời trong quá trình thực thi lệnh. Giúp giảm số lần truy cập bộ nhớ ngoài, tăng tốc độ xử lý.
- Bus giao tiếp dữ liệu (Data Bus Interface): Truy xuất dữ liệu từ bộ nhớ và các thiết bị ngoại vi qua bộ nhớ đệm dữ liệu. Tách biệt với bus lệnh để đảm bảo hiệu suất cao hơn.
- Bộ nhớ đệm: Gồm hai bộ nhớ đệm riêng biệt: một cho lệnh và một cho dữ liệu, giúp tăng tốc độ truy xuất bộ nhớ.

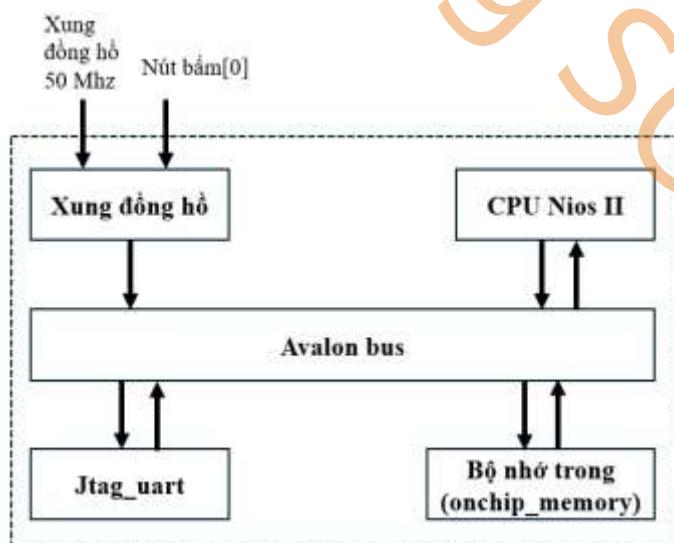
Tập lệnh của MicroBlaze bao gồm các nhóm lệnh chính như số học, logic, rẽ nhánh, tải/lưu trữ, và các lệnh khác. Tất cả các lệnh đều có kích thước cố định, giúp tối ưu hóa quá trình giải mã và thực thi. Mỗi lệnh có thể sử dụng tối đa ba thanh ghi làm toán hạng, mang lại sự linh hoạt trong xử lý dữ liệu. MicroBlaze sử dụng hai định dạng lệnh chính: Loại A (Type A) và Loại B (Type B):

- Loại A: Được sử dụng chủ yếu cho các lệnh thao tác giữa các thanh ghi (register-register). Định dạng này bao gồm mã lệnh (opcode và func), một thanh ghi đích (Rd), và hai thanh ghi nguồn (Ra, Rb).
- Loại B: Được sử dụng cho các lệnh thao tác giữa thanh ghi và giá trị tức thời (immediate value). Định dạng này bao gồm mã lệnh (opcode), một thanh ghi đích (Rd), một thanh ghi nguồn (Ra), và một giá trị tức thời 16-bit làm nguồn dữ liệu.

<b>Loại A</b>	Mã lệnh (opcode)	Rd	Ra	Rb	Chức năng (func)
<b>Loại B</b>	Mã lệnh (opcode)	Rd	Ra	Giá trị tức thời (Immediate value)	

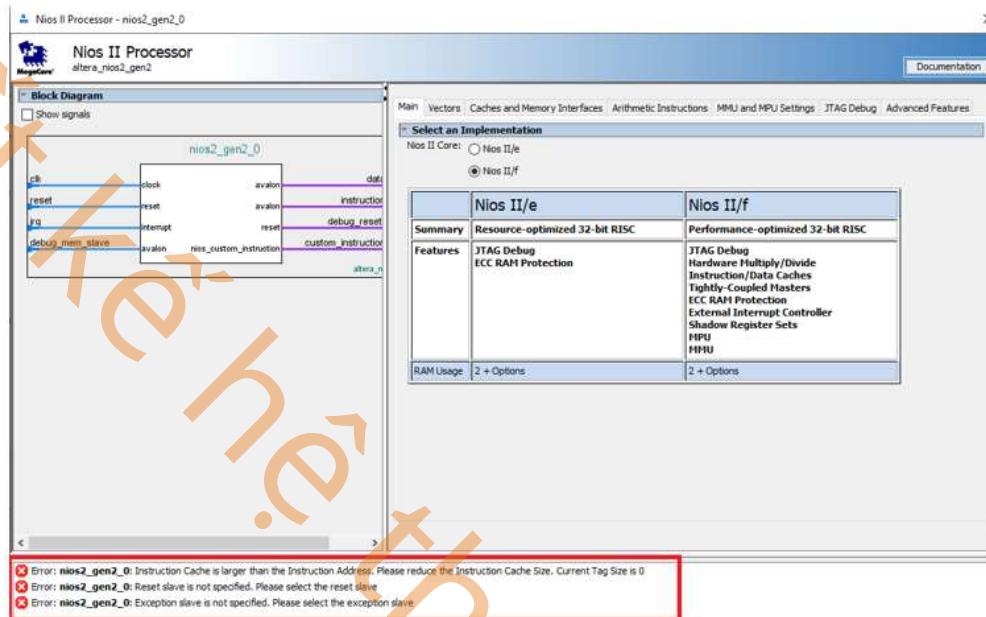
Bộ vi xử lý MicroBlaze sử dụng kiến trúc pipeline gồm 3 giai đoạn: nạp lệnh (fetch), giải mã (decode), và hoàn thành (complete). Việc chuyển tiếp dữ liệu (data forwarding), xử lý rẽ nhánh (branches), và dừng pipeline (pipeline stall) đều được phần cứng tự động xác định, giúp tối ưu hóa hiệu suất hoạt động. MicroBlaze hỗ trợ ba kích thước dữ liệu trong bộ nhớ: 8-bit (Byte), 16-bit (Halfword), và 32-bit (Word). Các thao tác truy xuất bộ nhớ luôn được căn chỉnh theo kích thước dữ liệu, đảm bảo tính nhất quán. Đây là bộ xử lý Big-Endian, nghĩa là nó sử dụng địa chỉ và quy tắc gán nhãn theo chuẩn Big-Endian khi truy cập bộ nhớ. Khi có một ngắt (interrupt) xảy ra, MicroBlaze sẽ dừng thực thi lệnh hiện tại để xử lý yêu cầu ngắt bằng cách chuyển đến địa chỉ vector ngắt. Đồng thời, nó cũng lưu lại địa chỉ lệnh tiếp theo cần thực thi để có thể quay lại sau khi xử lý xong ngắt. Trong quá trình này, MicroBlaze sẽ chặn các ngắt tiếp theo bằng cách xóa cờ IE (Interrupt Enable) trong MSR (Machine Status Register) để đảm bảo tính toàn vẹn của dữ liệu. MicroBlaze hỗ trợ buss rộng bus 32-bit và là một bộ xử lý RISC có tập lệnh riêng biệt cho truy cập bộ nhớ và dữ liệu. Nó sử dụng RAM LUT 32-bit để lưu trữ thanh ghi. Bộ xử lý này hỗ trợ cả bộ nhớ trong và bộ nhớ ngoài. Với MicroBlaze, người dùng có thể linh hoạt lựa chọn bộ nhớ, thiết bị ngoại vi và giao diện để tạo ra một hệ thống tùy chỉnh chính xác trên một FPGA duy nhất với chi phí thấp, mang lại sự tối ưu hóa cả về hiệu năng lẫn tài nguyên.

### 2.2.3. Cách kết nối vi xử lý mềm vào trong hệ thống SoC.



Hình 2.7: Yêu cầu xây dựng hệ thống SoC đơn giản

Phần này đưa ra một ví dụ minh họa cụ thể cách gọi và kết nối một vi xử lý mềm vào trong hệ thống SoC trên FPGA. Để đơn giản hóa, tác giả chỉ giới thiệu một kiến trúc SoC đơn giản gồm NIOS II đóng vai trò là CPU và Onchip\_memory là bộ nhớ để lưu trữ dữ liệu của hệ thống như kết nối trong Hình 2.7.

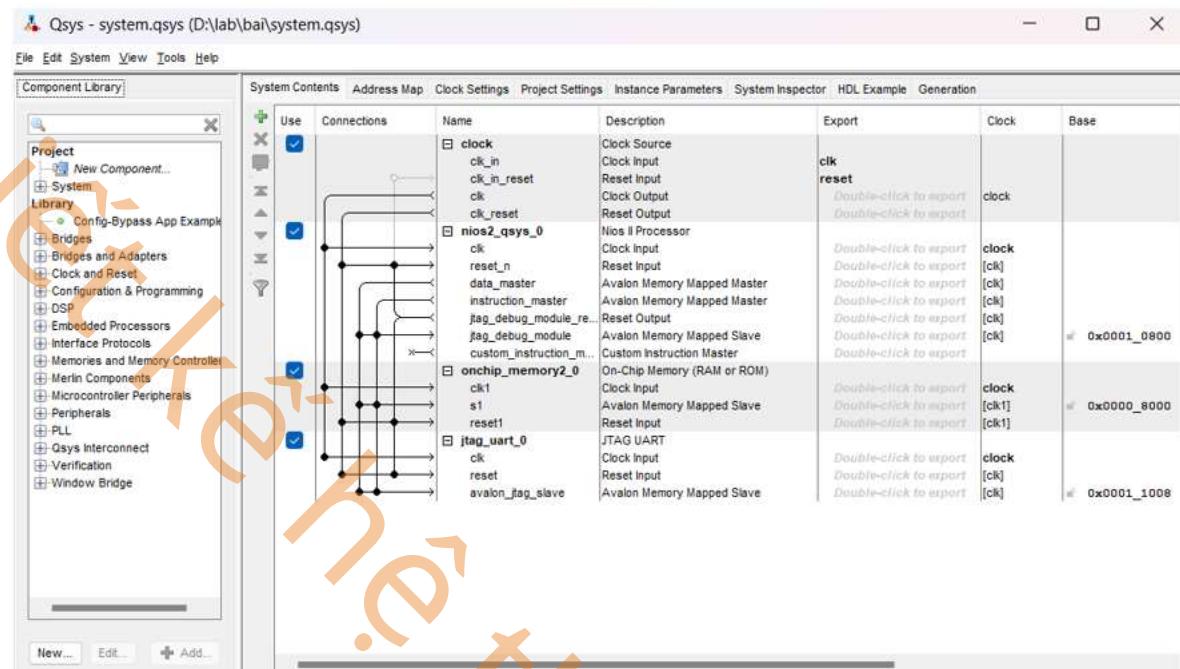


Hình 2.8: Minh họa lựa chọn CPU Nios II cho ví dụ

Hình 2.8 minh họa cách lấy CPU Nios II trên board DE2115. Trước tiên cần mở Quartus Prime và tạo một project mới, sau đó truy cập Platform Designer (Qsys) để thiết kế hệ thống phần cứng. Trong cửa sổ Component Library, chọn Nios II Processor từ mục Embedded Processors và thêm vào hệ thống. Khi cửa sổ cấu hình CPU hiện ra, người dùng có thể chọn giữa Nios II/e (tiết kiệm tài nguyên) hoặc Nios II/f (hiệu suất cao) tùy theo yêu cầu thiết kế. Trong ví dụ này ta chọn CPU Nios II/f.

Hình 2.9 minh họa hệ thống SoC hoàn chỉnh sau khi kết nối CPU Nios II với các thành phần như yêu cầu ban đầu. Trong hình bao gồm bộ xử lý Nios II (nios2\_qsys\_0) đóng vai trò là CPU chính, được kết nối với bộ nhớ trong (onchip\_memory2\_0), giúp lưu trữ chương trình và dữ liệu trong hệ thống. Khối clock (clk) cung cấp tín hiệu xung nhịp điều khiển toàn bộ hệ thống, đảm bảo sự đồng bộ giữa các thành phần. Cổng giao tiếp JTAG UART (jtag\_uart\_0) cho phép truyền thông giữa hệ thống và máy tính thông qua giao diện JTAG, giúp lập trình viên có thể tải chương trình, kiểm tra và gỡ lỗi một cách dễ dàng. Các kết

nối trong sơ đồ được thiết lập theo chuẩn giao tiếp Avalon bus, giúp CPU có thể truy xuất bộ nhớ và giao tiếp với các ngoại vi một cách hiệu quả.



Hình 2.9: Hệ thống SoC tích hợp CPU Nios II.

Để thực hiện kiểm tra CPU Nios II đã hoạt động chưa, ta chỉ cần thiết lập 1 phần mềm đơn giản như hình. Đoạn mã phần mềm này chỉ thực hiện tương tác với CPU và hệ thống SoC thông qua lệnh in ra của số phần mềm 1 câu phát biểu đơn giản “Bài 1 thực hành SoC”, như Hình 2.10.

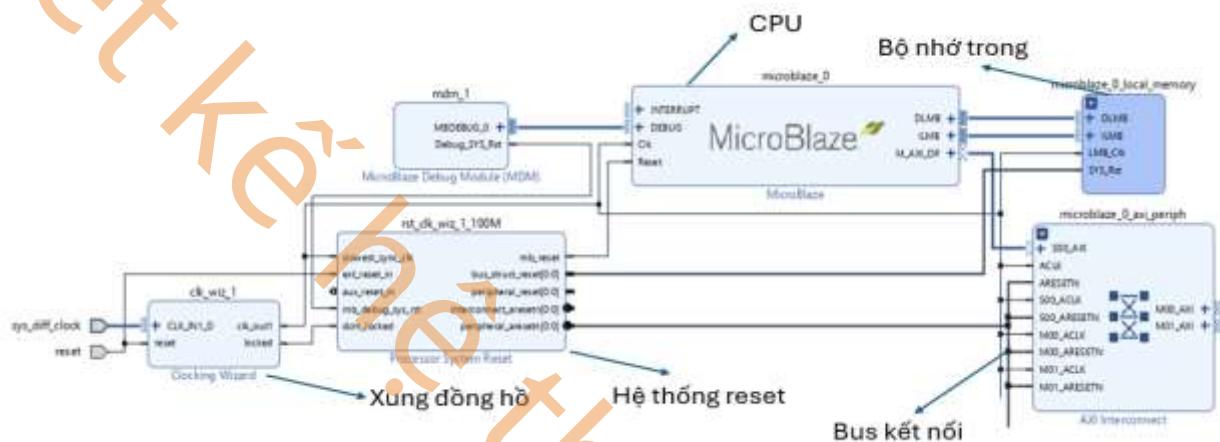
```
#include <stdio.h>

void main() {
    printf("Bai 1 thuc hanh SoC\n");
}
```

Hình 2.10: Đoạn mã đơn giản dùng chạy cho CPU Nios II

Tương tự như cách tạo một hệ thống SoC với CPU mềm Nios II, để thiết lập một sơ đồ hệ thống SoC như Hình 2.11 bằng phần mềm Vivado với CPU mềm Microblaze trước tiên ta cần tạo một thiết kế mới và thêm các khối IP có sẵn trong thư viện gồm MicroBlaze, Clocking Wizard, Processor System Reset, AXI Interconnect, và bộ nhớ trong. Clocking Wizard được sử dụng để tạo và quản lý xung đồng hồ, cung cấp tín hiệu đồng hồ chính cho toàn bộ hệ thống. Processor System Reset chịu trách nhiệm xử lý các tín hiệu reset từ bên ngoài và tạo ra các tín hiệu reset đồng bộ cho các thành phần khác. MicroBlaze là vi xử lý

chính, được kết nối với bộ nhớ trong qua bus LMB và với các ngoại vi khác qua AXI Interconnect. AXI Interconnect đóng vai trò là bus kết nối, cho phép truyền dữ liệu giữa MicroBlaze và các ngoại vi sử dụng giao tiếp AXI. Ngoài ra, khối MicroBlaze Debug Module (MDM) cũng được tích hợp để hỗ trợ việc gỡ lỗi. Khi tất cả các thành phần đã được cấu hình đúng cách, cần thực hiện kết nối các tín hiệu đồng hồ, reset và bus truyền dữ liệu theo sơ đồ như thể hiện trong hình, sau đó tạo bitstream để nạp vào FPGA.



Hình 2.11: Giải đố hệ thống SoC tích hợp CPU Microblaze.

### 2.3. Kiến trúc vi xử lý đa năng lõi cứng trên FPGA

CPU lõi cứng trên FPGA là một giải pháp mạnh mẽ và hiệu quả trong thiết kế các hệ thống nhúng, đặc biệt là những hệ thống yêu cầu hiệu suất tính toán cao và khả năng thực thi các tác vụ phức tạp. CPU lõi cứng được tích hợp sẵn vào chip FPGA, cho phép thực hiện các phép toán nhanh chóng và hiệu quả mà không cần phải thay đổi phần cứng. Các CPU lõi cứng, như ARM Cortex-A9 và ARM Cortex-A53, được thiết kế sẵn trên các bo mạch FPGA của các nhà sản xuất như Xilinx và Altera (Intel). Lý do là ARM CPU được ưa chuộng trong thiết kế hệ thống nhúng nhờ vào hiệu suất cao, tiết kiệm năng lượng và khả năng tùy chỉnh linh hoạt, phù hợp với nhiều ứng dụng đa dạng. Những CPU lõi cứng này thường có khả năng thực thi các hệ điều hành phức tạp như Linux, đồng thời có thể thực hiện các phép toán tổng quát trong các ứng dụng từ viễn thông, xử lý tín hiệu số, đến các hệ thống nhúng. CPU lõi cứng trên SoC FPGA cũng có một số ưu điểm và nhược điểm như sau:

### **Ưu điểm:**

- ✓ Hiệu suất cao và khả năng tối ưu hóa phần cứng: CPU lõi cứng, ví dụ như CPU ARM Cortex-A9 và A53, cung cấp khả năng xử lý mạnh mẽ, giúp thực hiện các tác vụ xử lý phức tạp.
- ✓ Tiết kiệm chi phí và diện tích: Việc tích hợp CPU lõi cứng trực tiếp vào SoC FPGA giúp giảm thiểu diện tích và chi phí, vì không cần phải sử dụng một CPU riêng biệt và các phần cứng tách biệt.
- ✓ **Khả năng chạy hệ điều hành phức tạp:** CPU lõi cứng có khả năng hỗ trợ các hệ điều hành phức tạp như Linux, mở ra khả năng phát triển ứng dụng đa dạng.
- ✓ Tính tương thích cao với các công nghệ khác: CPU lõi cứng dễ dàng tích hợp với nhiều công nghệ và phần cứng khác nhau, từ mạng không dây đến cảm biến và thiết bị ngoại vi, mang lại khả năng mở rộng hệ thống linh hoạt.
- ✓ Tiết kiệm năng lượng: Các CPU lõi cứng trên SoC FPGA được tối ưu hóa để tiết kiệm năng lượng, điều này rất quan trọng trong các ứng dụng nhúng, nơi yêu cầu hoạt động lâu dài với mức tiêu thụ năng lượng thấp mà không cần nguồn cung cấp lớn.

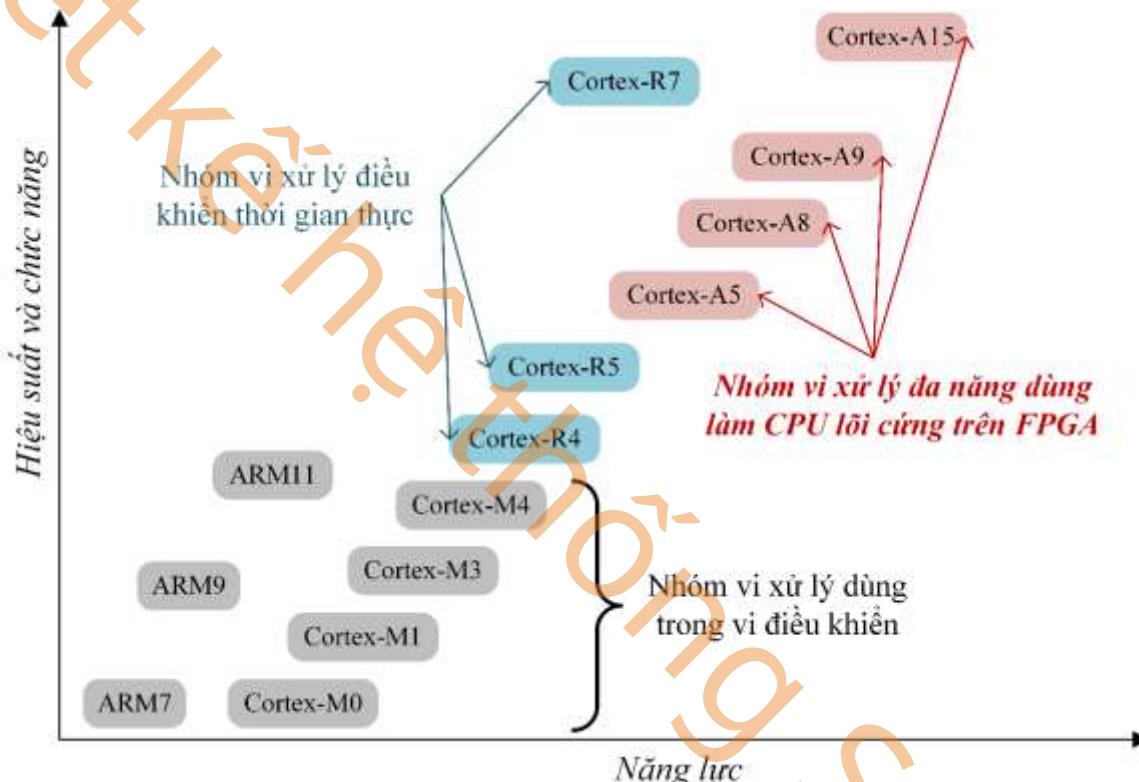
### **Nhược điểm:**

- ❖ **Khả năng tùy chỉnh phần cứng hạn chế:** CPU lõi cứng không thể thay đổi cấu trúc phần cứng như các giải pháp tùy chỉnh hoàn toàn. Điều này có thể hạn chế khả năng tối ưu hóa cho các ứng dụng đặc thù, nơi cần điều chỉnh phần cứng để đáp ứng yêu cầu tính toán đặc biệt.
- ❖ **Khả năng xử lý song song hạn chế:** Mặc dù CPU lõi cứng trên FPGA vẫn đủ mạnh để tính toán và xử lý nhanh trong hệ thống nhúng, nhưng chúng không tối ưu cho các tác vụ tính toán song song quy mô lớn như các vi xử lý chuyên dụng hoặc các cấu trúc phần cứng tùy chỉnh, điều này có thể ảnh hưởng đến hiệu suất trong một số ứng dụng tính toán phức tạp.

- ❖ Chi phí phát triển và phần mềm: Việc tích hợp CPU lõi cứng vào hệ thống SoC FPGA có thể yêu cầu các công cụ phát triển phần mềm và các giấy phép phần mềm, điều này có thể làm tăng chi phí tổng thể của dự án.

Phản kê tiếp sẽ mô tả tổng quan kiến trúc ARM CPU để làm CPU lõi cứng trên Altera và Xilinx FPGA.

### 2.3.1. Tổng quan kiến trúc ARM CPU



Hình 2.12: Dòng sản phẩm bộ vi xử lý ARM.

ARM (Advanced RISC Machine) là kiến trúc vi xử lý dựa trên RISC (Reduced Instruction Set Computer), nổi bật với hiệu năng cao, tiết kiệm năng lượng và độ linh hoạt cao trong thiết kế. ARM CPU được sử dụng rộng rãi trong các thiết bị di động, hệ thống nhúng, và các ứng dụng IoT nhờ thiết kế đơn giản nhưng mạnh mẽ, cho phép tối ưu hóa hiệu năng và giảm thiểu tiêu thụ năng lượng.

Kiến trúc ARM được chia thành nhiều dòng sản phẩm, phù hợp với các mục đích sử dụng khác nhau, từ vi điều khiển hiệu năng thấp đến vi xử lý đa năng hiệu năng cao. Hình 2.12 minh họa dòng sản phẩm vi xử lý ARM, được phân loại theo hiệu suất và chức năng

(trục dọc) và năng lực (trục ngang). Các dòng ARM cũ bao gồm ARM7, ARM9 và ARM11. Đây là các dòng vi xử lý thế hệ trước, được sử dụng trong nhiều ứng dụng nhúng và thiết bị di động. Mặc dù hiệu năng không cao như các dòng mới, nhưng chúng đã đặt nền tảng cho sự phát triển của kiến trúc ARM hiện đại. Các dòng này chủ yếu tập trung vào tính đơn giản, chi phí thấp và tiết kiệm năng lượng, phù hợp cho các hệ thống điều khiển cơ bản và các thiết bị nhúng không yêu cầu tính toán phức tạp. Nhóm vi xử lý dùng trong vi điều khiển (Cortex-M Series) được thiết kế cho các ứng dụng nhúng yêu cầu hiệu năng thấp và tiêu thụ năng lượng tối thiểu, như các thiết bị IoT và cảm biến. Dòng Cortex-M0 và M1 có hiệu năng thấp nhất, phù hợp với các ứng dụng cảm biến và hệ thống nhúng cơ bản. Trong khi đó, Cortex-M3 và M4 cung cấp hiệu năng cao hơn cùng khả năng xử lý tín hiệu số, phù hợp cho các ứng dụng điều khiển, âm thanh và xử lý tín hiệu. Với tính năng dễ lập trình và chi phí thấp, nhóm Cortex-M trở thành lựa chọn phổ biến trong các hệ thống nhúng thời gian thực. Nhóm vi xử lý điều khiển thời gian thực (Cortex-R Series) hướng đến các ứng dụng yêu cầu độ tin cậy cao và khả năng xử lý thời gian thực, chẳng hạn như hệ thống ô tô, thiết bị y tế và các hệ thống nhúng công nghiệp. Cortex-R4, R5 và R7 tăng dần về hiệu năng và khả năng kiểm soát thời gian thực. Các vi xử lý này được trang bị bộ nhớ tự sửa lỗi ECC để đảm bảo độ tin cậy dữ liệu cao, cùng khả năng xử lý ngắn với độ trễ thấp, phù hợp cho các ứng dụng yêu cầu độ chính xác cao và phản hồi nhanh. Nhóm vi xử lý đa năng ứng dụng (Cortex-A Series) được thiết kế cho các ứng dụng yêu cầu hiệu năng cao và khả năng xử lý phức tạp, như điện thoại thông minh, máy tính bảng và các thiết bị đa phương tiện. Các vi xử lý tiêu biểu trong dòng Cortex-A bao gồm Cortex-A5, A7, A8, A9, A12, A15 và A17. Nhóm Cortex-A nổi bật với khả năng hỗ trợ đa lõi (SMP), cho phép thực hiện đa nhiệm hiệu quả và tăng cường hiệu suất xử lý.

Trong các hệ thống SoC FPGA, vi xử lý Cortex-A thường được tích hợp làm CPU lõi cứng, đảm nhiệm vai trò điều khiển chính và xử lý các tác vụ phức tạp. Việc tích hợp này mang lại nhiều lợi ích như hiệu năng cao, khả năng xử lý đa nhiệm, và hỗ trợ các hệ điều hành phổ biến như Linux. Đồng thời, việc sử dụng CPU lõi cứng giúp giải phóng tài nguyên

logic trên FPGA, cho phép sử dụng cho các tác vụ khác, đặc biệt là các IP tự thiết kế trên mảng logic lập trình được.

**Bảng 2.2: So sánh các loại CPU ARM Cortex-A được sử dụng làm CPU lõi cứng**

Ví xử lý ARM Cortex-A	Tần số hoạt động	Bộ nhớ đệm (L1/L2/L3)	Hiệu suất	FPGA tích hợp
Cortex-A5	Lên đến 1.5 GHz	32 KB I-cache + 32 KB D-cache / 128 KB – 512 KB / Không có	Trung bình	Không phổ biến
Cortex-A7	Lên đến 1.5 GHz	32 KB I-cache + 32 KB D-cache / 256 KB – 1 MB / Không có	Trung bình	Không phổ biến
Cortex-A9	Lên đến 2.0 GHz	32 KB I-cache + 32 KB D-cache / 128 KB – 1 MB / Không có	Trung bình đến cao	Xilinx Zynq-700 FPGA và Cyclone V FPGA
Cortex-A53	Lên đến 2.3 GHz	32 KB I-cache + 32 KB D-cache / 512 KB – 2 MB / Tùy chọn, lên đến 2 MB	Trung bình đến cao	Zynq UltraScale+ MPSoC FPGA và Intel Stratix 10 SoC
Cortex-A57	Lên đến 2.5 GHz	48 KB I-cache + 32 KB D-cache / 1 MB – 2 MB / Tùy chọn, lên đến 4 MB	Cao	Không phổ biến
Cortex-A72	Lên đến 2.8 GHz	48 KB I-cache + 32 KB D-cache / 1 MB – 2 MB / Tùy chọn, lên đến 4 MB	Cao	Xilinx Versal FPGA

Cortex-A78	Lên đến 3.3 GHz	32 KB hoặc 64 KB I-cache + 32 KB hoặc 64 KB D-cache / Lên đến 512 KB / Tùy chọn, lên đến 4 MB	Rất Cao	Không phổ biến
------------	-----------------	---	---------	----------------

Bảng 2.2 cung cấp cái nhìn tổng quan về các loại vi xử lý ARM Cortex-A phổ biến được sử dụng làm CPU lõi cứng trong các hệ thống SoC, đặc biệt là trên các dòng FPGA của Altera (nay là Intel) và Xilinx. Bảng này so sánh chi tiết về tần số hoạt động, kích thước bộ nhớ đệm (L1, L2, L3), hiệu suất, cũng như các dòng FPGA tích hợp từng loại CPU. Cortex-A5 là vi xử lý cơ bản nhất trong dòng Cortex-A, với tần số hoạt động lên đến 1.5 GHz và bộ nhớ đệm L1 bao gồm 32 KB I-cache và 32 KB D-cache. Bộ nhớ đệm L2 có dung lượng từ 128 KB đến 512 KB nhưng không có L3. Hiệu suất của Cortex-A5 chỉ ở mức trung bình và ít được sử dụng trong các hệ thống FPGA hiện đại. Cortex-A7 cũng có tần số hoạt động lên đến 1.5 GHz nhưng được trang bị bộ nhớ đệm L2 lớn hơn (từ 256 KB đến 1 MB). Hiệu suất của Cortex-A7 cũng ở mức trung bình và chủ yếu được sử dụng trong các ứng dụng nhúng tiết kiệm năng lượng. Tuy nhiên, nó không phổ biến trên các dòng FPGA của Altera và Xilinx. Cortex-A9 là một trong những vi xử lý phổ biến nhất trong dòng Cortex-A, với tần số hoạt động lên đến 2.0 GHz. Cortex-A9 cung cấp hiệu suất từ trung bình đến cao và được tích hợp trong dòng FPGA Xilinx Zynq-7000 cũng như Cyclone V của Intel. Sự phổ biến của Cortex-A9 xuất phát từ kiến trúc đa lõi và hiệu suất cao, phù hợp cho các ứng dụng nhúng và điều khiển thời gian thực. Cortex-A53 là vi xử lý 64-bit đầu tiên trong dòng Cortex-A, hoạt động ở tần số lên đến 2.3 GHz. Với bộ nhớ đệm L2 từ 512 KB đến 2 MB và L3 tùy chọn lên đến 2 MB, Cortex-A53 cung cấp hiệu suất từ trung bình đến cao và hiệu quả năng lượng tốt. Nó được tích hợp trong các dòng FPGA cao cấp như Zynq UltraScale+ MPSoC của Xilinx và Intel Stratix 10 SoC, phù hợp cho các ứng dụng tính toán hiệu năng cao và tiết kiệm năng lượng. Cortex-A57 có tần số hoạt động lên đến 2.5 GHz và được trang bị bộ nhớ đệm L2 lớn hơn từ 1 MB đến 2 MB và L3 tùy

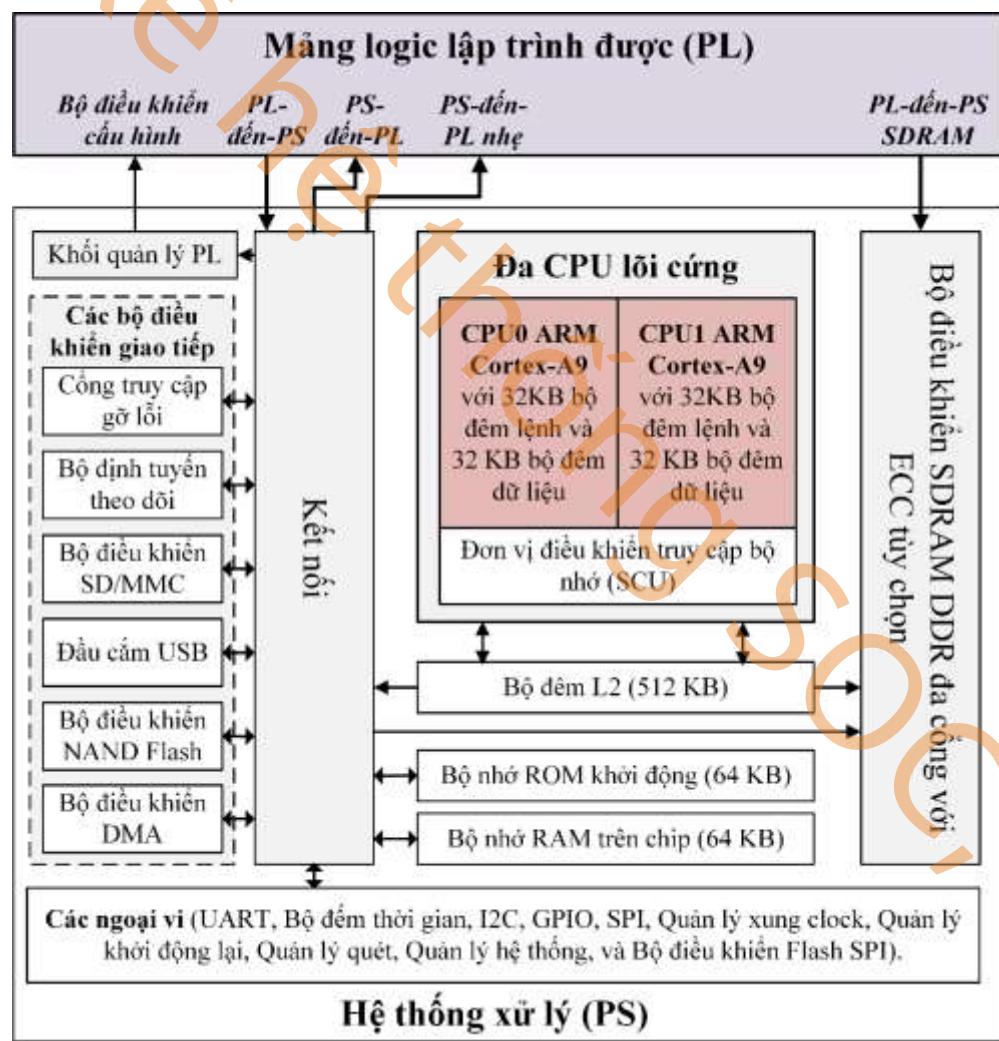
chọn lên đến 4 MB. Với hiệu suất cao, Cortex-A57 phù hợp cho các ứng dụng đòi hỏi tính toán mạnh mẽ. Tuy nhiên, nó không phổ biến trên các dòng FPGA của Altera và Xilinx. Cortex-A72 là một trong những vi xử lý mạnh mẽ nhất trong dòng Cortex-A với tần số hoạt động lên đến 2.8 GHz và bộ nhớ đệm L3 lên đến 4 MB. Với hiệu suất cao và khả năng xử lý đa nhiệm hiệu quả, Cortex-A72 được tích hợp trong dòng FPGA cao cấp Xilinx Versal, phù hợp cho các ứng dụng AI và học máy. Cortex-A78 là phiên bản cải tiến của Cortex-A77 với tần số hoạt động lên đến 3.3 GHz và hiệu suất rất cao. Nó có bộ nhớ đệm L1 linh hoạt (32 KB hoặc 64 KB) và L3 tùy chọn lên đến 4 MB. Mặc dù hiệu suất rất cao, Cortex-A78 không phổ biến trên các dòng FPGA hiện tại của Altera và Xilinx.

Nhìn chung, kiến trúc ARM Cortex-A9, Cortex-A53 và Cortex-A72 là những kiến trúc phổ biến nhất được sử dụng làm CPU lõi cứng trong các dòng FPGA của Xilinx và Altera (Intel). Trong khi đó, mặc dù Cortex-A5, A7, A57, và A78 không phổ biến trong các dòng FPGA thương mại của Xilinx và Intel, nhưng chúng vẫn có thể được sử dụng làm lõi cứng đối với các dòng FPGA tự chế (custom FPGA) hoặc các thiết kế SoC đặc biệt. Điều này mang lại sự linh hoạt cho các nhà phát triển khi lựa chọn vi xử lý phù hợp với yêu cầu cụ thể của hệ thống, tận dụng ưu điểm của cả CPU lõi cứng hiệu năng cao và mảng logic lập trình được để tối ưu hóa hiệu suất tổng thể của hệ thống.

### **2.3.2. CPU lõi cứng trên Altera và Xilinx FPGA**

Như đã giới thiệu ở nội dung trước, các dòng Cyclone V, Arria 10 SoC, và Stratix 10 SoC của Altera (Intel) và các dòng Zynq-7000, Zynq UltraScale+ MPSoC ZCU102, và Zynq UltraScale+ MPSoC ZCU104 của Xilinx là những ví dụ điển hình về FPGA tích hợp CPU lõi cứng ARM, mang lại khả năng xử lý mạnh mẽ và hiệu quả năng lượng. Trong đó, Cyclone V và Zynq-7000 sử dụng ARM Cortex-A9 làm CPU lõi cứng, nổi bật với hiệu suất ổn định và khả năng xử lý đa nhiệm, đặc biệt phổ biến trong các ứng dụng nhúng và thiết kế hệ thống nhờ cấu trúc linh hoạt và dễ áp dụng. Tương tự, Zynq UltraScale+ MPSoC trên ZCU102 và ZCU104 sử dụng ARM Cortex-A53 với hiệu suất cao hơn, phù hợp cho các ứng dụng yêu cầu tính toán phức tạp và xử lý đa nhiệm hiệu quả hơn. Trong khi đó, Versal FPGA đời mới của Xilinx tích hợp ARM Cortex-A72, cung cấp hiệu năng mạnh

mẽ và khả năng mở rộng vượt trội cho các ứng dụng tính toán phức tạp và AI. Mặc dù có sự khác biệt về số lượng lõi, tần số xung nhịp và các thành phần hỗ trợ giao tiếp, cấu trúc vi xử lý lõi cứng trong các FPGA của Altera và Xilinx về cơ bản là tương đồng, mang lại sự linh hoạt trong thiết kế hệ thống. Chương này sẽ tập trung giới thiệu chi tiết về ARM Cortex-A9 trên Cyclone V và Zynq-7000 do tính phổ biến và ứng dụng rộng rãi trong các hệ thống nhúng, giúp dễ dàng áp dụng vào nhiều tình huống thực tế. Bên cạnh đó, ARM Cortex-A9 còn cung cấp nền tảng vững chắc cho việc tìm hiểu và triển khai các hệ thống SoC trên FPGA, đặc biệt là trong các ứng dụng yêu cầu hiệu năng ổn định và khả năng xử lý đa nhiệm.



Hình 2.13: Hệ thống xử lý với đa CPU lõi cứng Arm Cortex-A9 trên Altera và Xilinx FPGA.

Hình 2.13 mô tả kiến trúc tổng quát SoC của FPGA khi nhìn hệ thống dưới dạng khía cạnh: hệ thống xử lý (PS) và mảng lập trình được (PL). Trong hình, PS bao gồm các bộ điều khiển xử lý như CPU ARM, bộ nhớ và các bộ điều khiển ngoại vi. PL là phần FPGA, có khả năng tái cấu hình linh hoạt, thực hiện các phép toán song song mạnh mẽ. Ví dụ ánh trắc PS của Cyclone V FPGA với vi xử lý đa lõi cùng ARM Cortex-A9 gồm hai CPU ARM Cortex-A9, mỗi CPU có 32KB bộ đệm lệnh và 32KB bộ đệm dữ liệu, giúp xử lý nhanh chóng và hiệu quả các tác vụ tính toán. Các CPU này được kết nối với bộ bộ nhớ L2 (512 KB), bộ ROM khởi động (64 KB), và bộ RAM trên chip (64 KB). Hệ thống này cũng tích hợp bộ điều khiển SDRAM/DDR với ECC tùy chọn, giúp đảm bảo tính toàn vẹn dữ liệu trong các ứng dụng yêu cầu xử lý bộ nhớ phức tạp. Các bộ điều khiển giao tiếp như Công truy cập gỡ lỗi, Bộ định tuyến theo dõi, Bộ điều khiển SD/MMC, Đầu cảm USB, Bộ điều khiển NAND Flash, và Bộ điều khiển DMA giúp kết nối và giao tiếp với các thiết bị ngoại vi trong hệ thống, đảm bảo hệ thống có khả năng tương tác linh hoạt và hiệu quả. Bên cạnh đó, các ngoại vi như UART, Bộ đếm thời gian, I2C, GPIO, SPI, Quản lý xung clock, Quản lý khởi động lại, Quản lý quét, Quản lý hệ thống, và Bộ điều khiển Flash SPI giúp giao tiếp và điều khiển các thiết bị ngoài, phục vụ cho các ứng dụng nhúng và hệ thống phức tạp.

### Đa CPU lõi cứng

Vi xử lý ARM Cortex-A9 là một vi xử lý đa lõi 32-bit, được thiết kế để tối ưu hóa hiệu suất và tiết kiệm năng lượng trong các ứng dụng yêu cầu tính toán mạnh mẽ, đồng thời giảm tiêu thụ năng lượng. Việc tích hợp ARM Cortex-A9 vào các hệ thống FPGA như Cyclone V hay Zynq-7000 mang lại sự kết hợp giữa khả năng xử lý mạnh mẽ của CPU và tính linh hoạt của mảng logic lập trình được (PL).

- Vi xử lý ARM Cortex-A9 là vi xử lý đa lõi, với 2 lõi xử lý và hỗ trợ tần số tối đa lên đến 800 MHz, giúp tối ưu hóa hiệu suất cho các ứng dụng yêu cầu tính toán mạnh mẽ.
- Vi xử lý hỗ trợ xử lý đồng bộ (SMP) và xử lý không đồng bộ (AMP), cho phép chia sẻ tải công việc và tối ưu hóa hiệu suất trong các hệ thống đa nhiệm.

- Mỗi lõi xử lý của ARM Cortex-A9 bao gồm 32KB bộ đệm lệnh và 32KB bộ đệm dữ liệu, giúp tăng tốc độ truy cập dữ liệu và lệnh trong quá trình xử lý.
- Vi xử lý này sử dụng công nghệ xử lý đa phương tiện NEON, hỗ trợ xử lý đồng thời 128-bit SIMD cho các ứng dụng như xử lý video và xử lý đồ họa.
- Vi xử lý ARM Cortex-A9 có đơn vị dấu phẩy động đơn và kép (single- or double-precision floating-point unit), hỗ trợ các phép toán dấu phẩy động chuẩn IEEE với độ chính xác cao.
- Đơn vị quản lý bộ nhớ (MMU) trong vi xử lý giúp quản lý bộ nhớ hiệu quả, đặc biệt trong các hệ thống yêu cầu khả năng quản lý bộ nhớ phức tạp.
- Vi xử lý này còn trang bị bộ đếm thời gian riêng và bộ đếm thời gian giám sát riêng, giúp quản lý thời gian và giám sát các tác vụ hệ thống.
- Bộ đệm cấp 2 (L2 cache) 512KB được chia sẻ giữa các lõi, giúp tăng cường hiệu suất truy cập bộ nhớ và giảm tắc nghẽn trong quá trình xử lý.
- SCU đảm bảo tính nhất quán bộ nhớ (cache coherency), giúp đồng bộ hóa bộ đệm của các lõi trong hệ thống.
- Cổng điều khiển nhất quán tăng tốc giúp tăng tốc hiệu suất giao tiếp giữa hệ thống xử lý và mảng logic lập trình được (PL) trong FPGA.
- Bộ hẹn giờ toàn cầu giúp đồng bộ hóa các tác vụ trong hệ thống, đảm bảo sự chính xác trong các phép tính và đồng bộ hóa thời gian.
- Bộ điều khiển ngắt chung quản lý các ngắt trong hệ thống, đảm bảo sự phân phối ngắt hợp lý giữa các bộ xử lý và các thiết bị ngoại vi.
- Theo dõi lệnh CoreSight hỗ trợ giám sát và phân tích các lệnh thực thi trong hệ thống, hỗ trợ phát triển phần mềm và kiểm tra.

Việc sử dụng ARM Cortex-A9 trong các hệ thống FPGA giúp giảm bớt số lượng linh kiện ngoài, đồng thời tiết kiệm không gian và chi phí, mang lại hiệu suất tối ưu cho các thiết bị nhúng và viễn thông, thiết bị di động, và các ứng dụng đòi hỏi tính toán cao.

### **Bộ điều khiển SDRAM đa công với ECC tùy chọn**

Hệ thống điều khiển SDRAM ở PS bao gồm một bộ điều khiển SDRAM đa cổng và bộ nhớ DDR vật lý được chia sẻ giữa PL (qua giao diện PL-đến-PS SDRAM), bộ nhớ đệm L2, và hệ thống kết nối L3. Giao diện PL-đến-PS SDRAM hỗ trợ các tiêu chuẩn giao diện AMBA AXI và Avalon, đồng thời cung cấp tối đa sáu cổng riêng biệt để truy cập các bộ xử lý chính được triển khai trong PL. Để tối đa hóa hiệu suất bộ nhớ, bộ điều khiển SDRAM hỗ trợ tái sắp xếp lệnh và dữ liệu, phân phối tài nguyên theo phương pháp round-robin với sự ưu tiên cao, và các tính năng bổ sung cho các yêu cầu quan trọng. Bộ điều khiển SDRAM hỗ trợ các thiết bị DDR2, DDR3, hoặc LPDDR2 có dung lượng lên tới 4 Gb, hoạt động ở tần số tối đa 400 MHz (tốc độ dữ liệu 800 Mbps).

### Các bộ điều khiển giao tiếp và các ngoại vi

Các bộ điều khiển giao tiếp đóng vai trò quan trọng trong việc kết nối và giao tiếp giữa vi xử lý và các thiết bị ngoại vi. Các bộ điều khiển này bao gồm Cổng truy cập gỡ lỗi, Bộ định tuyến theo dõi, Bộ điều khiển SD/MMC, Đầu cảm USB, Bộ điều khiển NAND Flash, và Bộ điều khiển DMA. Cổng truy cập gỡ lỗi giúp kết nối với các công cụ gỡ lỗi để theo dõi và phân tích hoạt động của hệ thống, trong khi Bộ định tuyến theo dõi giúp ghi lại và giám sát các tín hiệu theo dõi trong quá trình xử lý. Bộ điều khiển SD/MMC cho phép giao tiếp với các thiết bị lưu trữ như thẻ SD, trong khi Đầu cảm USB hỗ trợ giao tiếp với các thiết bị ngoại vi qua chuẩn USB. Bộ điều khiển NAND Flash quản lý giao tiếp với bộ nhớ NAND Flash, và Bộ điều khiển DMA giúp giảm tải cho CPU khi truyền dữ liệu trực tiếp giữa bộ nhớ và các thiết bị ngoại vi mà không cần can thiệp của bộ xử lý.

Các ngoại vi trong hệ thống cũng đóng vai trò quan trọng trong việc giao tiếp và điều khiển các thiết bị ngoại vi. Một số ngoại vi quan trọng bao gồm UART, Bộ đếm thời gian, I2C, GPIO, SPI, Quản lý xung clock, Quản lý khởi động lại, Quản lý quét, Quản lý hệ thống, và Bộ điều khiển Flash SPI. UART là giao thức truyền thông nối tiếp, giúp truyền dữ liệu giữa vi xử lý và các thiết bị ngoại vi, trong khi Bộ đếm thời gian giúp quản lý thời gian trong hệ thống. I2C và SPI là các giao thức truyền thông nối tiếp giúp kết nối vi xử lý với các thiết bị ngoại vi như cảm biến và màn hình. GPIO cho phép giao tiếp với các thiết bị qua các chân vào/ra có thể cấu hình. Quản lý xung clock và Quản lý khởi động lại giúp

duy trì sự ổn định của hệ thống, trong khi Quản lý quét và Quản lý hệ thống đảm bảo hoạt động hiệu quả của toàn bộ hệ thống. Bộ điều khiển Flash SPI giúp giao tiếp với bộ nhớ Flash qua giao thức SPI, phục vụ cho các ứng dụng nhúng.

### Cầu giao tiếp giữa hệ thống xử lý và mảng lập trình được

Các cổng PS-đến-PL (hệ thống xử lý sang mảng lập trình được) hoặc PL-đến-PS (mảng lập trình được sang hệ thống xử lý) cho phép kết nối giữa PS và PL để thực hiện các tác vụ tính toán, xử lý tín hiệu hoặc quản lý bộ nhớ. Cổng PS-đến-PL giúp truyền tải dữ liệu từ các bộ xử lý ARM sang mảng logic của FPGA, trong khi cổng PL-đến-PS giúp mảng FPGA giao tiếp với hệ thống xử lý, mang lại khả năng mở rộng và tùy chỉnh cho hệ thống. Các cầu giao tiếp giữa các cổng PS-đến-PL hoặc PL-đến-PS thông qua đường bus, bao gồm các cầu như sau:

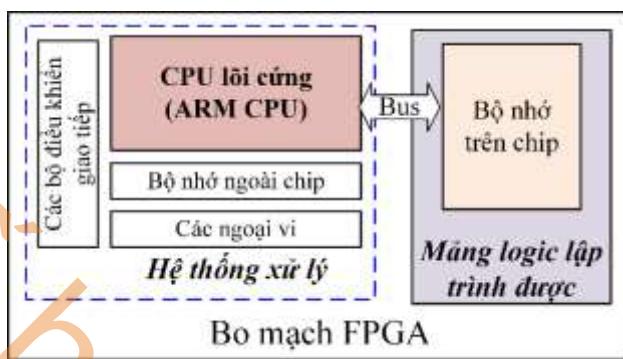
- Cầu FPGA-đến-PS: Là cầu bus hiệu suất cao hỗ trợ băng thông dữ liệu 32, 64 và 128 bit, cho phép mảng PL phát hành các giao dịch tới các thiết bị con trong PS.
- Cầu PS-đến-FPGA: Là cầu bus hiệu suất cao hỗ trợ băng thông dữ liệu 32, 64 và 128 bit, cho phép PS phát hành các giao dịch tới các thiết bị con trong mảng PL.
- Cầu PS-đến-FPGA nhẹ: Là cầu bus với độ trễ thấp 32 bit giúp PS phát hành giao dịch tới các thiết bị con trong mảng PL. Cầu này chủ yếu được sử dụng cho việc truy cập vào các thanh ghi điều khiển và trạng thái (CSR) của các ngoại vi trong mảng PL.

Các cầu PS-PL cho phép các chủ trong PL giao tiếp với các tớ trong PS, và ngược lại. Ví dụ, cầu PS-đến-FPGA cho phép PS chia sẻ bộ nhớ được cài đặt trong PL với một hoặc cả hai vi xử lý trong PS, trong khi cầu FPGA-đến-PS cho phép logic trong PL truy cập bộ nhớ và các ngoại vi trong PS. Mỗi cầu PS-PL cũng cung cấp chuyển mạch đồng hồ không đồng bộ cho dữ liệu được truyền giữa PL và PS, giúp duy trì sự chính xác và hiệu suất khi dữ liệu được chia sẻ giữa hai phần của hệ thống.

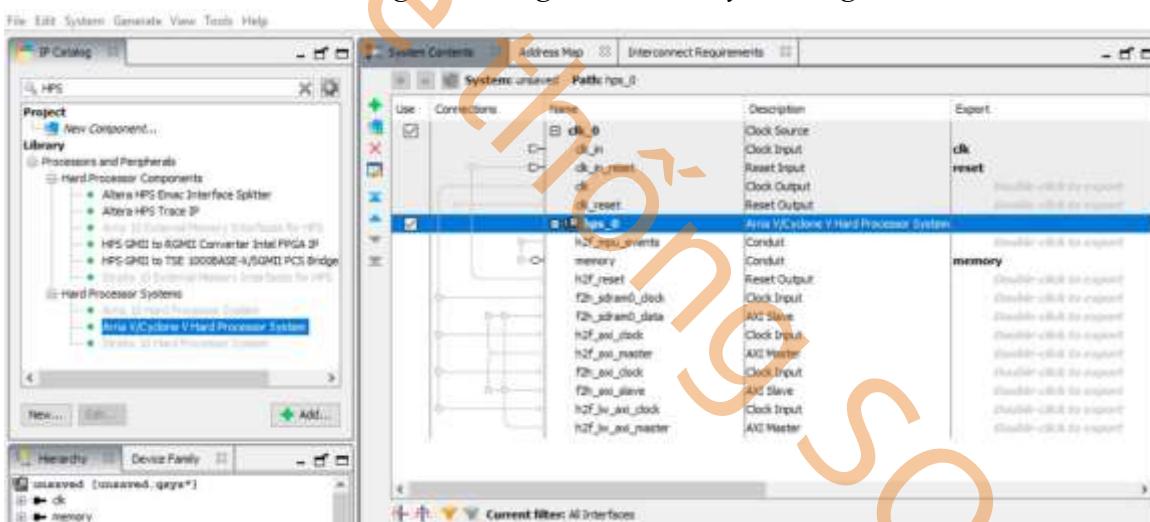
#### 2.3.3. Cách kết nối vi xử lý cứng vào trong hệ thống SoC

Phần này giới thiệu cách kết nối vi xử lý cứng vào trong hệ thống SoC trên FPGA, với một ví dụ minh họa qua hình 2.14. Hệ thống SoC đơn giản này gồm vi xử lý ARM Cortex-

A9 làm CPU và bộ nhớ trên chip cùng các bộ điều khiển giao tiếp kết nối với các thiết bị ngoại vi. Vi xử lý ARM Cortex-A9 được tích hợp trong mạch FPGA, kết nối với mảng logic lập trình được (PL) thông qua bus, cho phép hệ thống thực hiện các tác vụ tính toán mạnh mẽ với khả năng mở rộng cao. Bộ nhớ trên chip hỗ trợ vi xử lý trong việc lưu trữ dữ liệu và tối ưu hóa các phép toán tính toán.



Hình 2.14: Hệ thống SoC đơn giản với vi xử lý lõi cứng trên FPGA.



Hình 2.15: Hệ thống SoC tích hợp CPU lõi cứng ARM Cortex-A9.

Hình 2.15 mô tả giao diện phần mềm khi tích hợp CPU lõi cứng ARM Cortex-A9 vào hệ thống SoC. Trong hình, ta có thể thấy việc sử dụng công cụ phần mềm để cấu hình hệ thống xử lý (HPS - Hard Processor System) và kết nối các thành phần trong hệ thống. Cụ thể, bộ xử lý cứng ARM Cortex-A9 được cấu hình trong môi trường phần mềm với các kết nối và tín hiệu đầu vào/đầu ra cần thiết, bao gồm các tín hiệu như đồng hồ, bộ nhớ, và các kết nối bus cho việc giao tiếp với các thành phần khác của hệ thống. Việc sử dụng các thành phần như "Arria V/ Cyclone V Hard Processor System" cho phép thiết lập một hệ

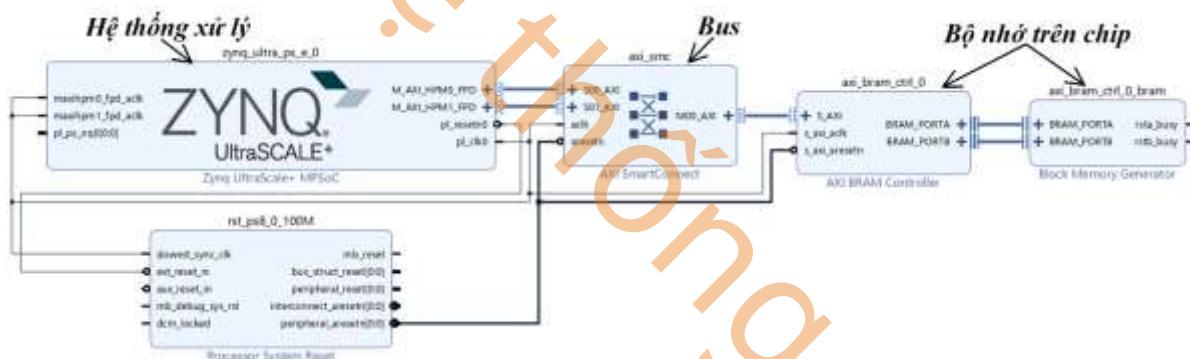
thống hoàn chỉnh với CPU ARM kết hợp với các thành phần FPGA, tạo ra một môi trường xử lý mạnh mẽ, linh hoạt và có thể tái cấu hình. Các tín hiệu này giúp đảm bảo khả năng giao tiếp hiệu quả giữa phần hệ thống xử lý (HPS) và phần mang lập trình được (PL) trong FPGA.

```
#include <stdio.h>

void main(){
    printf("Bai 1 thuc hanh SoC\n");
}
```

Hình 2.16: Đoạn mã đơn giản dùng chạy cho CPU lõi cứng ARM Cortex-A9.

Để kiểm tra xem CPU trong hệ thống HPS đã hoạt động hay chưa, ta có thể thiết lập một phần mềm đơn giản như trong hình. Đoạn mã phần mềm này chỉ thực hiện tương tác với CPU và hệ thống SoC thông qua lệnh in ra câu thông báo "Bài 1 thực hành SoC" trên cửa sổ phần mềm, xác nhận rằng hệ thống HPS đang hoạt động bình thường, như mô tả ở Hình 2.16.

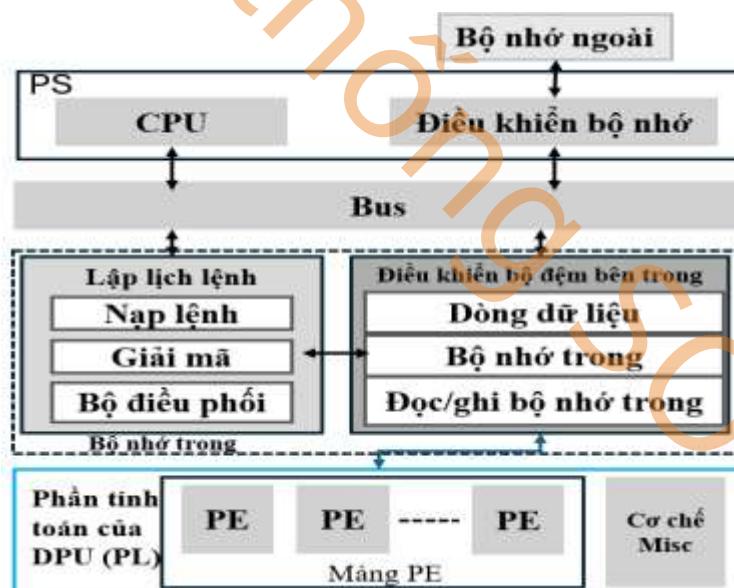


Hình 2.17: Giải đồ hệ thống SoC tích hợp CPU lõi cùng ARM Cortex-A9.

Hình 2.17 minh họa quá trình tạo một hệ thống SoC tích hợp CPU lõi cứng ARM Cortex-A9 trên Xilinx FPGA. Trong đó, hệ thống xử lý (PS) được kết nối với các thành phần khác như bộ nhớ trên chip (BRAM) và các thiết bị ngoại vi thông qua bus. Trong thiết kế này, ZYNQ UltraScale+ MPSoC IP, được xem là PS, được sử dụng, với các kết nối vào bộ điều khiển AXI SmartConnect, giúp giao tiếp hiệu quả với các thành phần như bộ nhớ và các thiết bị ngoại vi. PS của Xilinx FPGA điều khiển các thành phần như bộ nhớ trên chip (BRAM) thông qua bộ điều khiển AXI BRAM, mang lại khả năng tương tác linh hoạt giữa CPU và bộ nhớ, đồng thời tối ưu hóa hiệu suất hệ thống.

## 2.4. Vi xử lý chuyên dụng cho học sâu trên SoC FPGA

Đơn vị vi xử lý học sâu (DPU- Deep Learning Processing Unit) là một IP cứng có kiến trúc hoạt động giống một bộ xử lý chuyên dụng được thiết kế để tăng tốc các tác vụ AI, đặc biệt là các mô hình học sâu (Deep Learning) trên FPGA. Kiến trúc của DPU được tối ưu hóa cho các phép toán tensor, hỗ trợ các định dạng dữ liệu như INT8 nhằm tăng tốc độ xử lý trong khi vẫn duy trì độ chính xác cao. DPU thường bao gồm nhiều phần tử xử lý (PE - Processing Elements) hoạt động song song, bộ nhớ cache hiệu suất cao, và các đơn vị điều khiển chuyên biệt để tăng tốc các phép toán tích chập (Convolution), kích hoạt (Activation), và xử lý tensor. Trên các nền tảng FPGA của Xilinx, DPU được tích hợp trong các dòng Zynq UltraScale+ MPSoC và Versal ACAP, hỗ trợ thông qua công cụ Vitis AI. Trong khi đó, Intel (Altera) cung cấp các giải pháp tương tự dựa trên OpenVINO và sử dụng các bộ gia tốc AI trong các FPGA như Stratix 10 hoặc Agilex. Việc tích hợp DPU trên FPGA giúp cân bằng giữa hiệu suất, mức tiêu thụ năng lượng thấp, và khả năng tùy chỉnh linh hoạt theo từng ứng dụng AI, từ xử lý ảnh, nhận dạng giọng nói, đến phân tích dữ liệu lớn.



Hình 2.18: Vị trí IP DPU trong một board mạch FPGA

Hình 2.18 mô tả vị trí của IP DPU chuyên dụng được tích hợp vào một hệ thống SoC trên FPGA. Hệ thống SoC này bao gồm CPU cứng như ARM Cortex, hệ thống bus kết nối, bộ nhớ trong (BRAM) và bộ nhớ ngoài (DRAM). Trong đó, DPU đóng vai trò là phần tử tính toán chuyên dụng, thường được tích hợp trong phần logic lập trình (PL). DPU được

kết nối trực tiếp với CPU thông qua hệ thống bus, cho phép CPU thực hiện các tác vụ như lập lịch lệnh, nạp lệnh, giải mã và điều phối dữ liệu đến DPU để xử lý hiệu quả.

Mặc dù DPU có tập lệnh riêng, nhưng chúng thường được điều khiển bởi CPU lõi mềm hoặc cứng, giúp phối hợp và quản lý các tác vụ phức tạp trong hệ thống. Việc phát triển vi xử lý chuyên dụng như DPU nhằm đáp ứng nhu cầu xử lý các tác vụ đặc thù mà vi xử lý đa năng không thể thực hiện hiệu quả. Trong các ứng dụng AI và ML, việc xử lý song song hàng triệu phép toán là cần thiết, và DPU được thiết kế để thực hiện các phép toán này một cách nhanh chóng và hiệu quả hơn so với CPU hoặc GPU truyền thống. Điều này giúp giảm độ trễ, tiết kiệm năng lượng và tăng hiệu suất tổng thể của hệ thống. Vi xử lý chuyên dụng trên SoC FPGA có một số ưu điểm và nhược điểm như sau:

#### **Ưu điểm:**

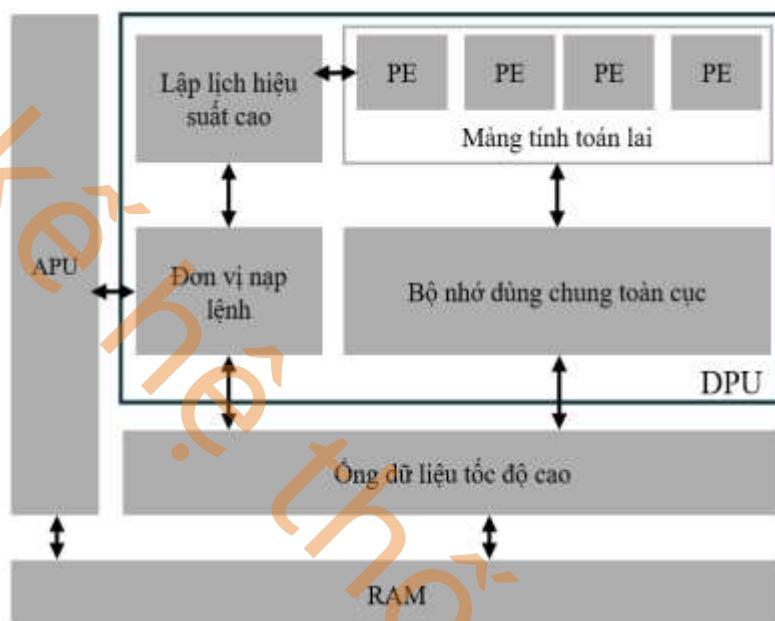
- ✓ **Tăng tốc AI/ML:** DPU được thiết kế để tăng tốc các tác vụ AI và ML, giúp giảm thời gian xử lý và tăng hiệu suất cho các ứng dụng như nhận diện hình ảnh, phân tích dữ liệu và các tác vụ học sâu khác.
- ✓ **Tiết kiệm năng lượng:** So với việc sử dụng CPU hoặc GPU truyền thống, DPU tiêu thụ ít năng lượng hơn khi xử lý các tác vụ AI/ML, giúp tiết kiệm chi phí vận hành và giảm thiểu tác động đến môi trường.
- ✓ **Khả năng mở rộng và linh hoạt:** DPU có thể được tích hợp vào các hệ thống SoC FPGA, cho phép mở rộng và tùy chỉnh theo nhu cầu cụ thể của ứng dụng, mang lại sự linh hoạt cao trong thiết kế hệ thống.

#### **Nhược điểm:**

- ❖ **Khả năng tùy chỉnh phần cứng hạn chế:** Mặc dù DPU cung cấp khả năng tăng tốc cho các tác vụ AI/ML, nhưng khả năng tùy chỉnh phần cứng để đáp ứng các yêu cầu đặc thù có thể bị hạn chế so với việc sử dụng các giải pháp phần cứng tùy chỉnh hoàn toàn.
- ❖ **Chi phí phát triển và tích hợp:** Việc tích hợp DPU vào hệ thống có thể đòi hỏi chi phí phát triển và tích hợp cao hơn, đặc biệt đối với các ứng dụng yêu cầu hiệu suất cao và tính năng đặc biệt.

- ❖ Tiêu tốn tài nguyên phần cứng: DPU chiếm dụng một phần tài nguyên logic và bộ nhớ của FPGA, điều này có thể ảnh hưởng đến khả năng triển khai các chức năng khác trên cùng một FPGA. Việc sử dụng DPU sẽ giảm bớt tài nguyên còn lại cho các IP tự thiết kế khác.

#### 2.4.1. Kiến trúc của DPU



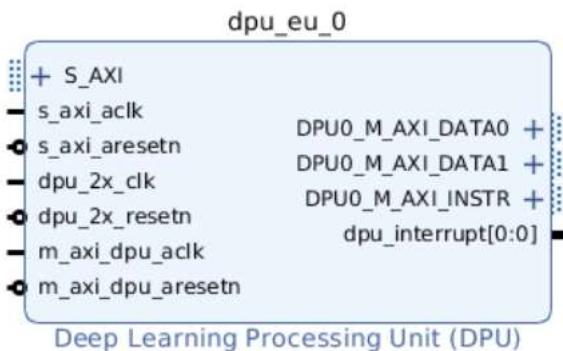
Hình 2.19: Kiến trúc chi tiết của DPU tích hợp trên board FPGA [21]

DPU có cấu trúc bên trong bao gồm nhiều khối chức năng tối ưu cho xử lý AI như mô tả trong Hình 2.19. Trong đó, đơn vị xử lý ứng dụng (APU - Application Processing Unit) chịu trách nhiệm giao tiếp với các thành phần bên ngoài và điều phối quá trình thực thi. DPU có một đơn vị nạp lệnh (Instruction Fetch Unit), giúp lấy lệnh từ bộ nhớ và gửi đến các đơn vị xử lý thích hợp. Bộ lập lịch hiệu năng cao (High Performance Scheduler) đóng vai trò lập lịch, tối ưu hóa thứ tự thực thi các tác vụ để đảm bảo hiệu suất cao nhất. Thành phần quan trọng nhất là mảng tính toán lai (Hybrid Computing Array), bao gồm nhiều PEs hoạt động song song để thực hiện các phép toán tensor và ma trận trong học sâu. Các PE này được kết nối với một bộ nhớ dùng chung toàn cục (Global Memory Pool), giúp lưu trữ tạm thời dữ liệu và giảm độ trễ trong quá trình tính toán. Để đảm bảo băng thông cao giữa bộ nhớ chính và DPU, một ống dữ liệu tốc độ cao (High Speed Data Tube) được sử dụng để truyền dữ liệu giữa RAM và DPU, giúp tăng tốc độ xử lý mô hình AI.

Các đặc tính chính của DPU bao gồm:

- Giao diện AXI linh hoạt:
  - DPU hỗ trợ một giao diện AXI phụ (slave) để truy cập các thanh ghi cấu hình và trạng thái, cho phép điều khiển và giám sát hoạt động của DPU dễ dàng.
  - Bên cạnh đó, DPU tích hợp một giao diện AXI chính (master) để truy cập các lệnh xử lý, giúp tối ưu hóa khả năng trao đổi dữ liệu với bộ nhớ hệ thống.
  - Giao diện AXI chính có thể cấu hình với băng thông 64 hoặc 128 bit, tùy thuộc vào thiết bị mục tiêu, giúp tối ưu hóa tốc độ truyền tải dữ liệu.
- Khả năng cấu hình riêng lẻ: DPU cho phép cấu hình từng kênh xử lý một cách độc lập, giúp tăng tính linh hoạt trong việc tối ưu hóa các tác vụ xử lý khác nhau.
- Hỗ trợ yêu cầu ngắt (Interrupt Request): DPU hỗ trợ tính năng tạo các yêu cầu ngắt tùy chọn, giúp tăng cường khả năng quản lý và kiểm soát các sự kiện trong quá trình xử lý dữ liệu.
- Các tính năng nổi bật về chức năng:
  - Kiến trúc phần cứng có thể cấu hình: DPU hỗ trợ nhiều đơn vị tính toán (computational units) hoặc số lượng phép nhân cộng tích lũy (MACs -Multiply-Accumulate Units) khác nhau như B512, B800, B1024, B1152, B1600, B2304, B3136, và B4096. Điều này cho phép tối ưu hóa hiệu năng dựa trên yêu cầu của từng ứng dụng cụ thể.
  - Hỗ trợ tối đa bốn lõi đồng nhất (homogeneous cores): Giúp tăng cường khả năng xử lý song song, phù hợp với các bài toán AI phức tạp.
  - Các phép toán xử lý chuyên dụng: DPU hỗ trợ các phép toán phổ biến trong mạng học sâu như tích chập (convolution), phép tích chập ngược (deconvolution), và tích chập theo chiều sâu (depthwise convolution).
  - Kỹ thuật pooling: Bao gồm cả max pooling và average pooling để giảm kích thước dữ liệu và tăng hiệu quả tính toán.
  - Hàm kích hoạt đa dạng: DPU hỗ trợ các hàm kích hoạt phổ biến như ReLU, ReLU6, và Leaky ReLU, phù hợp với nhiều kiến trúc mạng nơ-ron khác nhau.

- Các phép toán cơ bản khác: Bao gồm nối dữ liệu (concat) và cộng từng phần tử (elementwise-sum) để xử lý dữ liệu hiệu quả trong các tầng mạng phức tạp.



Hình 2.20: Bảng đóng gói của IP DPU được cung cấp bởi Xilinx

Hình 2.20 là bảng đóng gói IP DPU được tích hợp sẵn trong thư viện của Xilinx. Bảng 2.3 mô tả chi tiết ý nghĩa từng tín hiệu của IP để người dùng dễ tiếp cận và sử dụng cho mục đích chuyên dụng riêng.

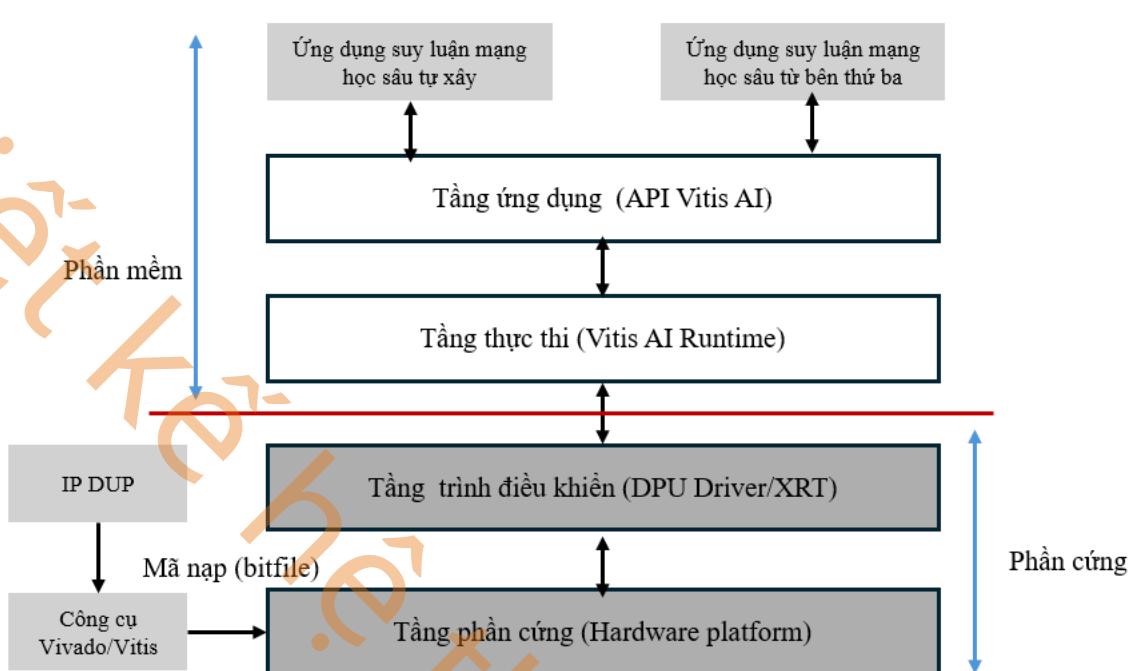
### Bảng 2.3: Chi tiết ý nghĩa của các chân tín hiệu trong IP DPU

Tên Tín Hiệu	Loại Giao Tiếp	Độ Rộng	I/O	Mô Tả
S_AXI	Giao tiếp AXI slave ánh xạ bộ nhớ	32	I/O	Giao tiếp AXI 32-bit ánh xạ bộ nhớ cho các thanh ghi.
s_axi_aclk	Tín hiệu xung nhịp (Clock)	1	I	Tín hiệu xung nhịp đầu vào cho S_AXI.
s_axi_aresetn	Tín hiệu reset	1	I	Tín hiệu reset mức thấp hoạt động (Active-Low) cho S_AXI.
dpu_2x_clk	Tín hiệu xung nhịp (Clock)	1	I	Tín hiệu xung nhịp đầu vào dùng cho các khối DSP trong DPU. Tần số gấp đôi m_axi_dpu_aclk.
dpu_2x_resetn	Tín hiệu reset	1	I	Tín hiệu reset mức thấp hoạt động cho các khối DSP.

m_axi_dpu_aclk	Tín hiệu xung nhịp (Clock)	1	I	Tín hiệu xung nhịp đầu vào dùng cho logic tổng quát của DPU.
m_axi_dpu_ares etn	Tín hiệu reset	1	I	Tín hiệu reset mức thấp hoạt động cho logic tổng quát của DPU.
DPUx_M_AXI_INSTR	Giao tiếp AXI master ánh xạ bộ nhớ	32	I/O	Giao tiếp AXI 32-bit ánh xạ bộ nhớ cho các lệnh DPU.
DPUx_M_AXI_DATA0	Giao tiếp AXI master ánh xạ bộ nhớ	64 hoặc 128	I/O	Giao tiếp AXI 64-bit cho dòng Zynq7000 hoặc 128-bit cho dòng MPSoC.
DPUx_M_AXI_DATA1	Giao tiếp AXI master ánh xạ bộ nhớ	64 hoặc 128	I/O	Giao tiếp AXI 64-bit cho dòng Zynq7000 hoặc 128-bit cho dòng Zynq MP.
dpu_interrupt	Ngắt (Interrupt)	1~4	O	Tín hiệu ngắt mức cao hoạt động (Active-High) từ DPU. Độ rộng dữ liệu phụ thuộc vào số lỗi.
SFM_M_AXI (tùy chọn)	Giao tiếp AXI master ánh xạ bộ nhớ	128	I/O	Giao tiếp AXI 128-bit ánh xạ bộ nhớ cho module softmax (tùy chọn).
sfm_interrupt (tùy chọn)	Ngắt (Interrupt)	1	O	Tín hiệu ngắt mức cao hoạt động từ module softmax (tùy chọn).

Hiện nay, DPU (Deep Processing Unit) đang được ứng dụng và phát triển mạnh mẽ trên các board FPGA của Xilinx, đặc biệt là các dòng SoC và MPSoC. Trong phần tiếp theo,

tác giả sẽ trình bày tổng quan về cách tích hợp và vận hành DPU trên nền tảng công cụ hỗ trợ Vitis AI của Xilinx.



Hình 2.21: Quy trình hiện thực một ứng dụng trên DPU bằng công cụ Vitis AI [21]

Hình 2.21 minh họa cấu trúc phân lớp của các thành phần phần cứng (HW) và phần mềm (SW) cho hệ thống sử dụng DPU trên nền tảng FPGA của Xilinx. Kiến trúc này thể hiện quy trình hoạt động tổng thể từ phần mềm đến phần cứng, nhằm tối ưu hóa việc triển khai các ứng dụng AI. Hoạt động của hệ thống có thể được chia thành các lớp chính như sau:

**Tầng Ứng dụng (Vitis AI APIs):** Cung cấp các giao diện lập trình ứng dụng giúp các nhà phát triển dễ dàng tích hợp các mô hình AI vào ứng dụng của mình. Các ứng dụng gồm: Các ứng dụng mẫu được tối ưu hóa sẵn (Example NN Inference Application) để minh họa cách sử dụng DPU hoặc các ứng dụng AI từ bên thứ ba (Third-Party NN Inference Application) cũng có thể tích hợp dễ dàng thông qua các API của Vitis AI.

**Tầng thực thi (Vitis AI Runtime (VART)):** Đây là môi trường thực thi phần mềm, đóng vai trò quan trọng trong việc: quản lý luồng dữ liệu giữa các ứng dụng AI và phần cứng DPU, đảm bảo sự tối ưu hóa hiệu năng khi chạy các mô hình học sâu, tương thích với các thư viện và công cụ phát triển AI của Xilinx.

**Tầng trình điều khiển (DPU Driver/XRT (Xilinx Runtime)):** Đây là tầng trung gian giữa phần cứng và phần mềm. Trình điều khiển này chịu trách nhiệm: quản lý tài nguyên phần cứng, điều phối dữ liệu giữa phần mềm và phần cứng, tối ưu hóa hiệu năng khi thực thi các tác vụ AI.

### Tầng phần cứng (Hardware layer):

- ✓ DPU IP (Intellectual Property): Đây là khối xử lý phần cứng chuyên dụng cho các tác vụ AI, được tích hợp vào FPGA thông qua các công cụ thiết kế phần cứng như Vivado hoặc Vitis.
- ✓ Vivado/Vitis: Các công cụ này được sử dụng để thiết kế, cấu hình và tạo file cấu hình phần cứng (bitstream/bitfile).
- ✓ Nền tảng phần cứng (Hardware Platform): Sau khi bitfile được tạo ra, nó sẽ được nạp vào nền tảng phần cứng FPGA để cấu hình các tài nguyên phần cứng, bao gồm cả DPU.

### Quy trình hoạt động tổng thể

- ✓ Phát triển và triển khai ứng dụng AI thông qua các API của Vitis AI để chạy các tác vụ suy luận AI trên DPU.
- ✓ Sử dụng Vitis AI Runtime để quản lý việc thực thi các mô hình AI.
- ✓ Cài đặt DPU Driver/XRT để đảm bảo giao tiếp giữa phần mềm và phần cứng
- ✓ Thiết kế phần cứng DPU IP bằng Vivado/Vitis và tạo bitfile.
- ✓ Nạp bitfile vào nền tảng phần cứng FPGA để cấu hình DPU.

**DPU** là một kiến trúc phần cứng chuyên dụng, được tích hợp sẵn trên các board FPGA hiện nay, đặc biệt là dòng **Zynq UltraScale+ MPSoC** của Xilinx. DPU được thiết kế dưới dạng **IP cứng**, vì vậy nó chỉ hỗ trợ cho từng loại bảng mạch FPGA cụ thể và tối ưu hóa cho các kiến trúc mạng học sâu có trong thư viện đi kèm. Điểm mạnh của DPU là khả năng tối ưu hóa hiệu năng cao nhờ vào sự tích hợp chặt chẽ giữa phần cứng và phần mềm, đặc biệt hiệu quả khi xử lý các ứng dụng AI đã được hỗ trợ sẵn trong thư viện. Tuy nhiên, với những ứng dụng có kiến trúc mạng nơ-ron khác biệt hoặc chưa được tối ưu hóa sẵn, DPU có thể không khai thác được tối đa hiệu suất. Do đó, người dùng cần thực hiện

các phép đo, kiểm tra và đánh giá hiệu năng để xác định xem DPU có đáp ứng tốt các yêu cầu của ứng dụng đang triển khai hay không. Việc này giúp đảm bảo hệ thống hoạt động hiệu quả và phù hợp với từng bài toán cụ thể.

## 2.5. So sánh các loại vi xử lý

Trong phần này, chúng ta sẽ so sánh ba loại vi xử lý chính trên SoC FPGA: vi xử lý đa năng lõi cứng, vi xử lý đa năng lõi mềm, và vi xử lý chuyên dụng (DPU). Bảng 2.4 trình bày các tiêu chí so sánh giữa ba loại vi xử lý này, bao gồm tần số hoạt động, tiêu tốn tài nguyên phần cứng FPGA, khả năng chạy hệ điều hành Linux, công dụng, khả năng tùy chỉnh phần cứng và tiêu thụ năng lượng.

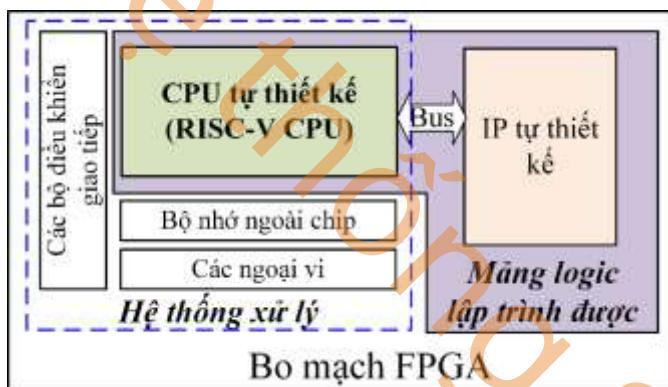
**Bảng 2.4: So sánh 3 loại vi xử lý trên SoC FPGA**

Tiêu chí	Vi xử lý đa năng lõi cứng	Vi xử lý đa năng lõi mềm	Vi xử lý chuyên dụng (DPU)
Tần số hoạt động	Thường cao, có thể lên đến 1.3 GHz hoặc hơn.	Thường thấp, khoảng 100-200 MHz.	Thường trung bình, khoảng 300 MHz.
Tiêu tốn tài nguyên phần cứng FPGA	Không chiếm dụng tài nguyên FPGA.	Tiêu tốn vừa phải tài nguyên FPGA.	Tiêu tốn nhiều tài nguyên FPGA
Khả năng chạy hệ điều hành Linux	Có thể chạy Linux.	Có thể chạy Linux.	Thường không chạy Linux.
Công dụng	Thực hiện các tác vụ tính toán phức tạp, hỗ trợ đa nhiệm.	Tùy chỉnh cho các ứng dụng cụ thể, linh hoạt.	Tăng tốc các tác vụ AI/ML, xử lý dữ liệu.
Khả năng tùy chỉnh phần cứng	Không thể tùy chỉnh.	Cao, có thể tùy chỉnh theo yêu cầu.	Hạn chế.
Tiêu thụ năng lượng	Trung bình	Trung bình đến cao	Cao

Tần số hoạt động của vi xử lý đa năng lõi cứng thường cao, có thể lên đến 1.3 GHz hoặc hơn, giúp thực hiện các tác vụ tính toán phức tạp nhanh chóng và hiệu quả. Vi xử lý đa năng lõi mềm có tần số hoạt động thấp hơn, thường dao động trong khoảng 100-200 MHz, và vi xử lý chuyên dụng (DPU) có tần số hoạt động trung bình khoảng 300 MHz. Tuy nhiên, DPU được tối ưu hóa để xử lý các tác vụ AI/ML, do đó tần số hoạt động không phải là yếu tố quan trọng nhất trong trường hợp này. Về tiêu tốn tài nguyên phần cứng FPGA, vi xử lý đa năng lõi cứng không chiếm dụng tài nguyên FPGA vì nó đã được tích hợp sẵn trên chip. Ngược lại, vi xử lý đa năng lõi mềm tiêu tốn tài nguyên FPGA vừa phải, trong khi DPU chiếm nhiều tài nguyên hơn, vì nó được thiết kế đặc biệt để xử lý các tác vụ AI/ML và yêu cầu nhiều tài nguyên phần cứng. Về khả năng chạy hệ điều hành Linux, cả vi xử lý đa năng lõi cứng và vi xử lý đa năng lõi mềm đều có thể chạy Linux, điều này mở ra khả năng phát triển các ứng dụng phức tạp. Tuy nhiên, DPU thường không chạy Linux, vì nó được tối ưu hóa cho các tác vụ chuyên biệt như xử lý AI/ML và không cần hệ điều hành đầy đủ. Công dụng của các loại vi xử lý này cũng rất khác nhau. Vi xử lý đa năng lõi cứng thực hiện các tác vụ tính toán phức tạp và hỗ trợ đa nhiệm. Vi xử lý đa năng lõi mềm có thể được tùy chỉnh cho các ứng dụng cụ thể và linh hoạt, trong khi DPU chuyên biệt cho việc tăng tốc các tác vụ AI/ML, giúp xử lý dữ liệu và mô hình học sâu hiệu quả hơn. Về khả năng tùy chỉnh phần cứng, vi xử lý đa năng lõi cứng không thể tùy chỉnh, vì cấu trúc của nó đã được xác định sẵn, trong khi vi xử lý đa năng lõi mềm có khả năng tùy chỉnh cao, cho phép thay đổi cấu trúc và tính năng. DPU, mặc dù rất hiệu quả cho các tác vụ AI/ML, lại có khả năng tùy chỉnh hạn chế so với các giải pháp phần cứng tùy chỉnh hoàn toàn. Cuối cùng, về tiêu thụ năng lượng, vi xử lý đa năng lõi mềm và DPU thường tiết kiệm năng lượng hơn so với vi xử lý đa năng lõi cứng, đặc biệt là trong các ứng dụng chuyên biệt như xử lý AI/ML, nhờ vào khả năng tối ưu hóa phần cứng cho các tác vụ cụ thể. Tóm lại, việc lựa chọn loại vi xử lý nào phụ thuộc vào yêu cầu cụ thể của ứng dụng, bao gồm hiệu suất, khả năng tùy chỉnh, tiêu thụ năng lượng và chi phí.

## 2.6. Tích hợp vi xử lý tự thiết kế vào hệ thống SoC

Do có rất nhiều ứng dụng đòi hỏi các tính toán khác nhau, mỗi loại vi xử lý (vi xử lý đa năng lõi cứng, vi xử lý đa năng lõi mềm và vi xử lý chuyên dụng) thường do các nhà phát triển FPGA như Altera và Xilinx cung cấp. Tuy nhiên, những vi xử lý này thường có kiến trúc cố định, được tối ưu hóa cho các ứng dụng chung và không linh hoạt để điều chỉnh theo các yêu cầu tính toán đặc thù của người dùng. Điều này là do các vi xử lý do các nhà cung cấp FPGA phát triển thường được thiết kế như một hộp đen, nơi người dùng không thể truy cập vào mã nguồn hoặc mã Verilog để thay đổi hoặc tối ưu hóa. Chúng cung cấp một tập hợp các tính năng và hiệu suất tiêu chuẩn nhưng không thể tùy chỉnh để đáp ứng yêu cầu của từng tác vụ cụ thể trong một hệ thống. Điều này dẫn đến hạn chế trong việc tối ưu hóa hiệu suất cho các ứng dụng đặc biệt hoặc các tác vụ tính toán phức tạp. Để khắc phục hạn chế này, vi xử lý tự thiết kế đã trở thành một giải pháp hiệu quả, cho phép người thiết kế chủ động xây dựng cấu trúc vi xử lý phù hợp nhất với yêu cầu của ứng dụng.



Hình 2.22: Cấu trúc hệ thống SoC FPGA với vi xử lý tự thiết kế.

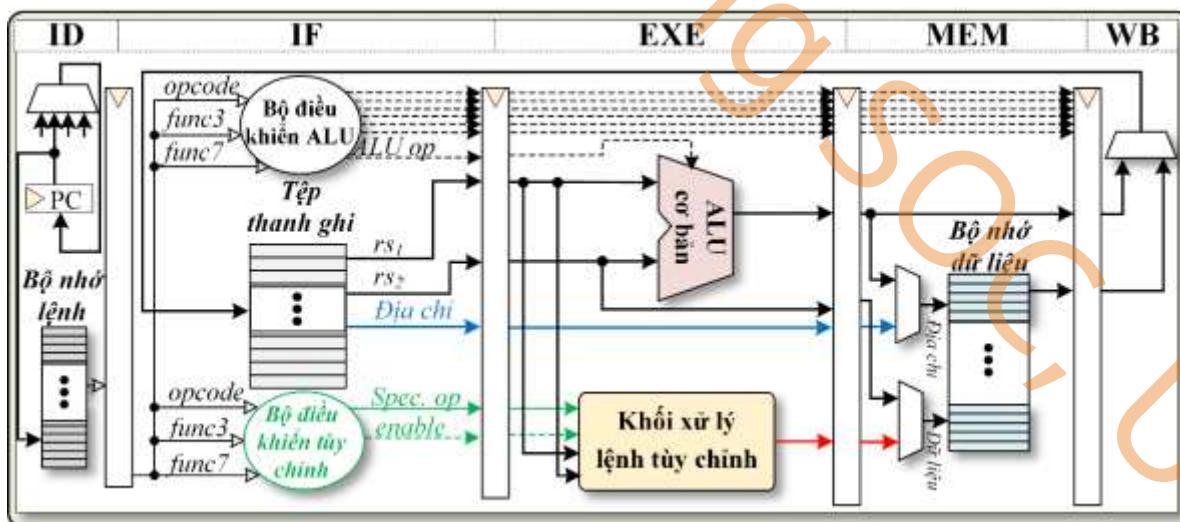
Hình 2.22 mô tả cấu trúc hệ thống SoC FPGA với vi xử lý tự thiết kế. Vi xử lý tự thiết kế có thể kết hợp đặc tính của cả vi xử lý mềm và vi xử lý chuyên dụng. Việc tự thiết kế vi xử lý mang lại sự linh hoạt cao cho người dùng trong việc tạo ra các vi xử lý phù hợp với yêu cầu của hệ thống, từ việc tối ưu hóa hiệu suất đến tiết kiệm tài nguyên phần cứng. Vi xử lý tự thiết kế có thể được tích hợp vào SoC FPGA tương tự như vi xử lý đa năng mềm và vi xử lý chuyên dụng, vì chúng thực chất cũng là một dạng IP tự thiết kế với chức năng giống như vi xử lý. Việc sử dụng vi xử lý tự thiết kế có thể giúp tối ưu hóa hiệu suất hệ thống trong các ứng dụng đặc thù, đồng thời giảm thiểu kích thước và chi phí hệ thống vì các chức năng tính toán và điều khiển có thể được tinh chỉnh và tích hợp một cách hiệu

qua vào mảng logic lập trình được của FPGA. Tuy nhiên, do có rất nhiều yêu cầu khác nhau về vi xử lý tự thiết kế, chương này chỉ giới hạn vào việc giới thiệu vi xử lý tự thiết kế dựa trên RISC-V để tiết kiệm năng lượng trong các hệ thống FPGA. RISC-V là một kiến trúc mở, linh hoạt và có khả năng tối ưu hóa cao cho các ứng dụng nhúng, đặc biệt là trong các hệ thống yêu cầu hiệu suất cao nhưng vẫn tiết kiệm năng lượng. Việc sử dụng RISC-V giúp dễ dàng điều chỉnh và tùy chỉnh vi xử lý theo các yêu cầu cụ thể của ứng dụng mà không gặp phải những hạn chế của các vi xử lý thương mại sẵn có.

Phần này sẽ giới thiệu về thiết kế vi xử lý tự thiết kế dựa trên kiến trúc RISC-V. Chúng ta chọn RISC-V vì tính chất mã nguồn mở và sự phát triển rộng rãi hiện nay, đã trở thành lựa chọn phổ biến cho việc thiết kế vi xử lý, đặc biệt trong các ứng dụng nhúng. Kiến trúc này cho phép người thiết kế tự do điều chỉnh và tối ưu hóa các đặc tính của vi xử lý để đáp ứng nhu cầu của từng ứng dụng cụ thể. Lý do chọn RISC-V là vì tính linh hoạt trong việc tùy chỉnh bộ lệnh, khả năng mở rộng linh hoạt và cộng đồng phát triển mạnh mẽ hỗ trợ. RISC-V được đề xuất lần đầu tiên bởi nhóm nghiên cứu tại Đại học California, Berkeley vào năm 2010, và kể từ đó đã phát triển mạnh mẽ nhờ vào sự hỗ trợ của cộng đồng mã nguồn mở. Điểm mạnh của RISC-V so với ARM là tính linh hoạt và khả năng mở rộng dễ dàng, giúp giảm thiểu chi phí bản quyền vì không bị ràng buộc bởi các giấy phép độc quyền như ARM. Cộng đồng mã nguồn mở RISC-V liên tục phát triển và cung cấp nhiều công cụ và tài nguyên hỗ trợ, tạo ra cơ hội lớn cho việc áp dụng RISC-V trong các ứng dụng từ đơn giản đến phức tạp.

Hình 2.23 minh họa kiến trúc vi xử lý RISC-V cơ bản với các khối xử lý lệnh tùy chỉnh, giúp hỗ trợ các tính toán chuyên biệt phù hợp với yêu cầu của từng ứng dụng cụ thể. Trong hệ thống này, các khối như ALU, bộ điều khiển ALU, bộ nhớ, và các thanh ghi được sử dụng để thực hiện các phép toán tính toán. Cấu trúc này cho phép người thiết kế linh hoạt trong việc tùy chỉnh các lệnh và chức năng của vi xử lý, nhờ vào khả năng thêm bớt các khối xử lý hoặc điều chỉnh các tham số trong từng giai đoạn của chu kỳ lệnh (IF, ID, EXE, MEM, WB). Các chu kỳ lệnh IF, ID, EXE, MEM và WB trong kiến trúc vi xử lý RISC-V lần lượt đại diện cho các giai đoạn trong pipeline của quá trình xử lý lệnh:

- ✓ IF (Instruction Fetch): Đây là giai đoạn lấy lệnh. Trong giai đoạn này, bộ vi xử lý sẽ lấy lệnh tiếp theo từ bộ nhớ chương trình vào thanh ghi lệnh, sử dụng địa chỉ lệnh hiện tại (PC - Program Counter). Sau đó, PC được cập nhật để trỏ đến lệnh tiếp theo.
- ✓ ID (Instruction Decode): Giai đoạn này là nơi giải mã lệnh đã được lấy. Tại đây, bộ điều khiển sẽ phân tích opcode và các tham số liên quan của lệnh để xác định các thanh ghi và các tham số cần thiết cho các phép toán. Các tín hiệu như func3, func7, rs1, rs2 sẽ được giải mã để chuẩn bị cho giai đoạn tiếp theo.
- ✓ EXE (Execute): Trong giai đoạn này, các phép toán thực tế được thực hiện. ALU (Arithmetic Logic Unit) sẽ thực hiện phép toán và tính toán kết quả. Nếu lệnh là một phép toán số học hoặc logic, ALU sẽ thực hiện phép toán tương ứng.
- ✓ MEM (Memory Access): Giai đoạn này xử lý truy cập bộ nhớ nếu lệnh yêu cầu, chẳng hạn như đối với các lệnh load (lấy dữ liệu từ bộ nhớ) và store (ghi dữ liệu vào bộ nhớ). Bộ điều khiển sẽ quyết định nếu lệnh cần phải truy xuất bộ nhớ hoặc chỉ đơn giản là truyền kết quả từ ALU.
- ✓ WB (Write Back): Đây là giai đoạn cuối cùng, trong đó kết quả tính toán từ ALU hoặc dữ liệu lấy từ bộ nhớ sẽ được ghi trở lại vào thanh ghi, để hệ thống có thể sử dụng kết quả này trong các chu kỳ lệnh tiếp theo.



Hình 2.23: Kiến trúc vi xử lý RISC-V cơ bản với khối xử lý lệnh tùy chỉnh để hỗ trợ các tính toán chuyên biệt phù hợp với yêu cầu của từng ứng dụng cụ thể.

Các chu kỳ này là những bước cơ bản trong pipeline của vi xử lý, giúp tối ưu hóa hiệu suất xử lý bằng cách cho phép thực hiện nhiều lệnh đồng thời trong các giai đoạn khác nhau.

RISC-V có một hệ thống bộ lệnh (ISA - Instruction Set Architecture - Kiến trúc Bộ lệnh Vi xử lý) chia thành nhiều loại khác nhau, phục vụ cho các nhu cầu khác nhau trong thiết kế vi xử lý. Các loại bộ lệnh chính của RISC-V bao gồm các tập lệnh cơ bản như I (Integer - Số nguyên), E (Embedded - Nhúng), V (Vector - Vecto), C (Compressed - Nén), M (Multiplication and Division - Nhân và Chia), A (Atomic - Tập lệnh Hạt nhân), F (Single-Precision Floating-Point - Dấu phẩy động độ chính xác đơn), D (Double-Precision Floating-Point - Dấu phẩy động độ chính xác kép), và Z (Standard Extensions - Các Mở rộng Tiêu chuẩn). Các loại RISC-V chủ yếu bao gồm:

➤ RV32I và RV64I (Base Integer Instruction Sets):

- RV32I: Hệ thống bộ lệnh 32-bit, hỗ trợ các lệnh cơ bản cho các ứng dụng sử dụng bộ xử lý 32-bit. RV32I bao gồm các lệnh cơ bản như toán học số học, thao tác bit, chuyển dữ liệu, v.v. Tuy nhiên, RV32I không có các tính năng nâng cao như hỗ trợ dấu phẩy động hay tính toán vectơ, điều này làm cho nó không thể chạy các hệ điều hành như Linux.
- RV64I: Là phiên bản mở rộng của RV32I, hỗ trợ bộ xử lý 64-bit, cho phép xử lý dữ liệu lớn hơn và tăng cường khả năng tính toán cho các ứng dụng yêu cầu hiệu suất cao. RV64I là sự lựa chọn phổ biến cho các hệ thống có yêu cầu phần cứng 64-bit, như máy tính hoặc các ứng dụng lớn.

➤ RV32E và RV64E:

- RV32E: Là một phiên bản thu gọn của RV32I, được thiết kế cho các ứng dụng yêu cầu ít bộ lệnh và tiêu thụ ít tài nguyên hơn. Đây là lựa chọn lý tưởng cho các vi xử lý sử dụng trong các thiết bị nhúng có yêu cầu phần cứng tiết kiệm.
- RV64E: Tương tự như RV32E nhưng với bộ lệnh 64-bit, phù hợp với các ứng dụng sử dụng bộ vi xử lý 64-bit với tài nguyên thấp.

- RV32M và RV64M (M-Extensions): Các mở rộng M cho phép vi xử lý hỗ trợ các phép toán số học dấu phẩy động (multiplication và division) cho các bộ xử lý 32-bit (RV32M) và 64-bit (RV64M). Điều này giúp tăng khả năng xử lý các phép toán phức tạp hơn mà không cần đến các phần mềm hỗ trợ từ bên ngoài.
- RV32F và RV64F (F-Extensions): Các mở rộng F hỗ trợ các phép toán dấu phẩy động đơn (single-precision floating point). Đây là những bộ lệnh được sử dụng cho các ứng dụng yêu cầu tính toán với số thực có độ chính xác cao.
- RV32D và RV64D (D-Extensions): D-Extensions hỗ trợ các phép toán dấu phẩy động kép (double-precision floating point). Đây là lựa chọn cho các ứng dụng yêu cầu tính toán dấu phẩy động với độ chính xác cực cao.
- RV32A và RV64A (A-Extensions): A-Extensions cung cấp hỗ trợ cho các phép toán liên quan đến quản lý bộ nhớ như các lệnh cho phép xác định và thao tác với các địa chỉ bộ nhớ theo cách tối ưu.
- RV32V và RV64V (V-Extensions): V-Extensions mở rộng khả năng hỗ trợ xử lý vectơ cho các ứng dụng cần tính toán song song, chẳng hạn như các ứng dụng AI hoặc máy học.
- RV32C và RV64C (C-Extensions): C-Extensions là các mở rộng giúp giảm kích thước mã lệnh, thích hợp cho các ứng dụng yêu cầu tiết kiệm bộ nhớ và tài nguyên phần cứng.
- RV32G và RV64G (General Extensions): G-Extensions là sự kết hợp của tất cả các mở rộng như M, A, F, D, và C, cho phép một vi xử lý RISC-V có thể thực hiện tất cả các loại phép toán từ cơ bản đến phức tạp. Việc sử dụng các mở rộng này làm cho RISC-V trở thành một kiến trúc rất linh hoạt và mạnh mẽ cho các ứng dụng đa dạng.

Mỗi loại RISC-V phục vụ một mục đích cụ thể, từ việc tối ưu hóa tài nguyên cho các hệ thống nhúng đến cung cấp khả năng tính toán mạnh mẽ cho các ứng dụng yêu cầu hiệu suất cao như máy tính, xử lý ảnh, và các hệ thống tính toán khoa học.

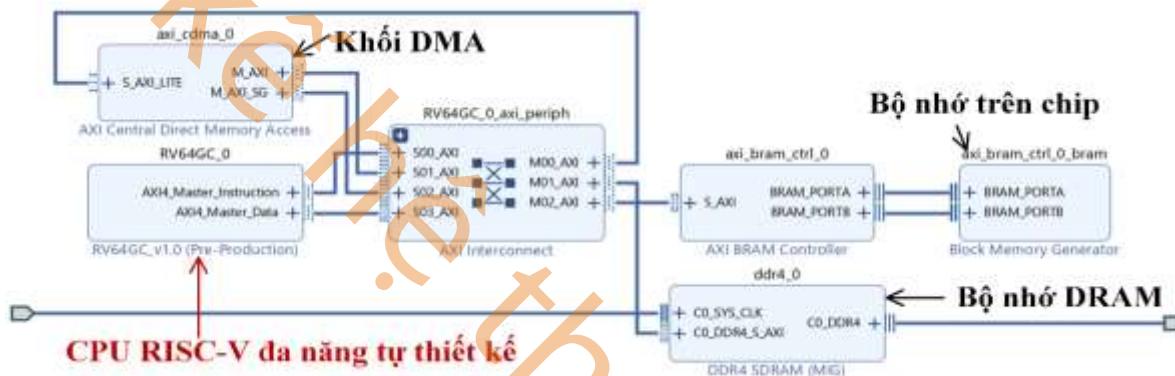
			31	imme[11:0]	20 19	rs1	funct3	rd	7 6	opcode
I-Type	FLW	FLW rd, offset(rs1)	F	imme[11:0]		rs1	010	rd	0000011	
	LB	LB rd, offset(rs1)	I	imme[11:0]		rs1	000	rd	00000011	
	LH	LH rd, offset(rs1)	I	imme[11:0]		rs1	001	rd	00000011	
	LW	LW rd, offset(rs1)	I	imme[11:0]		rs1	010	rd	00000011	
	LBU	LBU rd, offset(rs1)	I	imme[11:0]		rs1	100	rd	00000011	
	LHU	LHU rd, offset(rs1)	I	imme[11:0]		rs1	101	rd	00000011	
	ADDI	ADDI rd, rs1, imme	I	imme[11:0]		rs1	000	rd	0010011	
	SLTI	SLTI rd, rs1, imme	I	imme[11:0]		rs1	010	rd	0010011	
	SLTIU	SLTIU rd, rs1, imme	I	imme[11:0]		rs1	011	rd	0010011	
	XORI	XORI rd, rs1, imme	I	imme[11:0]		rs1	100	rd	0010011	
	ORI	ORI rd, rs1, imme	I	imme[11:0]		rs1	110	rd	0010011	
	ANDI	ANDI rd, rs1, imme	I	imme[11:0]		rs1	111	rd	0010011	
	JALR	jalr rd, offset(rs1)	I	imme[11:0]		rs1	000	rd	1100111	
	NOP	NOP	I	0000000000000000		00000	000	00000	0010011	
					est	rs1	funct3	rd		opcode
	CSRWR	CSRWR rd, csr, rs1	Zicsr	csr		rs1	001	rd	1110011	
	CSRRS	CSRRS rd, csr, rs1	Zicsr	csr		rs1	010	rd	11100011	
	CSRRC	CSRRC rd, csr, rs1	Zicsr	csr		rs1	011	rd	1110011	
					est	imme[4:0]	funct3	rd		opcode
	CSRRWI	CSRRWI rd, csr, ulimm	Zicsr	csr	ulimm	rs1	101	rd	1110011	
	CSRWSI	CSRWSI rd, csr, ulimm	Zicsr	csr	ulimm	rs1	110	rd	1110011	
	CSRRCI	CSRRCI rd, csr, ulimm	Zicsr	csr	ulimm	rs1	111	rd	1110011	
					funct?	shamt	rs1	funct3	rd	opcode
	SLLI	SLLI rd, rs1, imme	I	0000000		shamt	rs1	001	rd	0010011
	SRLI	SRLI rd, rs1, imme	I	0000000		shamt	rs1	101	rd	0010011
	SRAI	SRAI rd, rs1, imme	I	0100000		shamt	rs1	101	rd	0010011
S-Type				imme[11:5]	rs2	rs1	funct3	imme[4:0]		opcode
	FSW	FSW rs2, offset(rs1)	F	imme[11:5]	frs2	rs1	010	imme[4:0]	0100111	
	SB	SB rs2, offset(rs1)	I	imme[11:5]	rs2	rs1	000	imme[4:0]	0100011	
	SH	SH rs2, offset(rs1)	I	imme[11:5]	rs2	rs1	001	imme[4:0]	0100011	
	SW	SW rs2, offset(rs1)	I	imme[11:5]	rs2	rs1	010	imme[4:0]	0100011	

Hình 2.24: Kiến trúc bộ tập lệnh I.

Hình 2.24 minh họa kiến trúc bộ lệnh I của RISC-V, trong đó các lệnh cơ bản được phân loại theo kiểu I-Type và S-Type. Các trường trong lệnh bao gồm các phần như opcode, func3, func7, rs1, rs2, rd, và immed. Cụ thể, các lệnh như FLW, LB, LH, LW, LBU, LHU, và ADDI thuộc kiểu I-Type, nơi opcode xác định loại lệnh (ví dụ, 0000011 cho lệnh tải dữ liệu như FLW hay LB), func3 và func7 chỉ định các kiểu phép toán hoặc hành động cụ thể, rs1 và rd chỉ ra các thanh ghi nguồn và đích, và immed chứa giá trị tức thời (immediate) cần thiết cho tính toán hoặc địa chỉ bộ nhớ. Các lệnh như SLLI, SRLI, SRAI cũng thuộc loại này, phục vụ cho các phép toán dịch chuyển bit. Bên cạnh đó, kiểu S-Type được sử dụng cho các lệnh như FSW, SB, SH, và SW liên quan đến việc lưu trữ dữ liệu vào bộ nhớ, với các trường tương tự để chỉ định vị trí bộ nhớ và các tham số cần thiết cho thao tác lưu trữ.

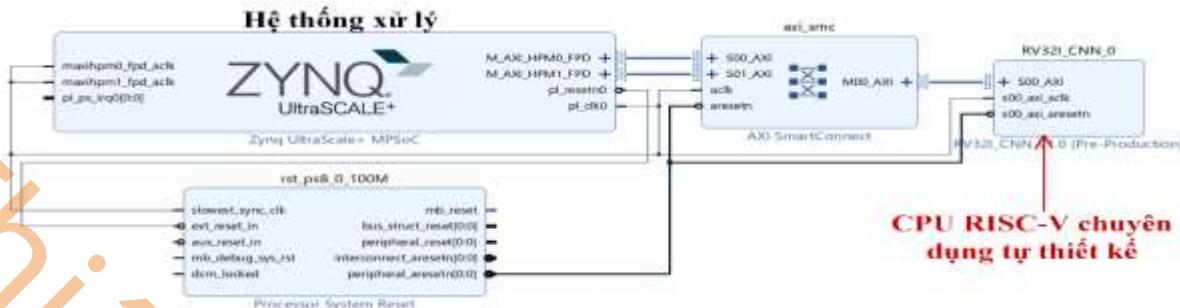
Tương tự như bộ lệnh I-Type và S-Type, các bộ lệnh khác của RISC-V cũng có các quy định và cấu trúc lệnh phù hợp với từng loại tính toán cụ thể. Các bộ lệnh như M, A, F, D, Z, V, C đều có các trường lệnh được thiết kế riêng biệt để hỗ trợ các phép toán hoặc tính toán chuyên biệt.

Việc thiết kế CPU RISC-V có thể chia thành hai loại chính: CPU RISC-V đa năng và CPU RISC-V chuyên dụng. Các vi xử lý đa năng, chẳng hạn như RV64GC, có thể chạy được hệ điều hành Linux nhờ vào khả năng xử lý các tác vụ phức tạp và khả năng mở rộng bộ lệnh, trong khi các vi xử lý chuyên dụng, ví dụ như RV32I, không hỗ trợ hệ điều hành như Linux và chỉ có thể thực hiện các tác vụ cụ thể hoặc tính toán đơn giản. Việc lựa chọn loại CPU RISC-V phù hợp phụ thuộc vào yêu cầu của ứng dụng, với CPU đa năng thích hợp cho các ứng dụng đòi hỏi tính toán phức tạp và chạy hệ điều hành, trong khi CPU chuyên dụng được sử dụng cho các tác vụ nhúng có yêu cầu đơn giản và hiệu suất tối ưu.



Hình 2.25: Giản đồ hệ thống SoC tích hợp CPU RISC-V đa năng tự thiết kế.

Hình 2.25 minh họa một hệ thống SoC tích hợp CPU RISC-V đa năng tự thiết kế, trong đó CPU RISC-V RV64GC kết hợp với các khối DMA, bộ nhớ trên chip và bộ nhớ DRAM. Hệ thống này cho phép thực hiện các tác vụ tính toán phức tạp và giao tiếp với bộ nhớ thông qua các kết nối AXI, giúp tối ưu hóa việc truy xuất và xử lý dữ liệu. Các khối DMA (Direct Memory Access) và bộ điều khiển bộ nhớ DRAM cung cấp khả năng truyền tải dữ liệu nhanh chóng và hiệu quả, trong khi bộ nhớ trên chip (BRAM) đảm bảo hiệu suất cao cho các tác vụ yêu cầu bộ nhớ nhanh. CPU RISC-V đa năng này có khả năng hỗ trợ các ứng dụng phức tạp và chạy hệ điều hành như Linux, mang lại sự linh hoạt và hiệu suất cao cho các hệ thống SoC hiện đại. Ngoài ra, hệ thống SoC này còn cho phép thêm vào các lệnh tính toán tại khối xử lý lệnh tùy chỉnh, giúp tối ưu hóa CPU RISC-V đa năng. Tùy thuộc vào yêu cầu của ứng dụng, các lệnh tính toán có thể được tích hợp để hỗ trợ các tác vụ tính toán đặc thù, từ đó vừa đảm bảo CPU có thể chạy hệ điều hành như Linux, vừa mang lại khả năng tính toán hiệu quả hơn cho các ứng dụng đòi hỏi hiệu suất cao.



Hình 2.26: Giản đồ hệ thống SoC tích hợp CPU RISC-V tùy chỉnh tự thiết kế.

Hình 2.26 minh họa một hệ thống SoC tích hợp CPU RISC-V chuyên dụng tự thiết kế, tối ưu hóa cho các ứng dụng mạng nơ-ron nhân tạo (CNN). Để hỗ trợ hiệu quả cho CNN, phần xử lý lệnh tùy chỉnh trong CPU có thể thực hiện các phép tính nhân ma trận, max pooling, fully connected layer trong các mạng CNN, giúp tính toán nhanh chóng và hiệu quả. Tuy nhiên, CPU RISC-V chuyên dụng này không thể chạy hệ điều hành Linux, vì nó được thiết kế đặc biệt cho các tác vụ tính toán chuyên biệt. Để có thể chạy hệ điều hành Linux, cần có hệ thống xử lý với CPU lõi mềm hoặc lõi cứng đa dụng, thậm chí là CPU RISC-V đa năng (ví dụ RV64GC), để cung cấp khả năng điều khiển và hỗ trợ các tác vụ phức tạp hơn. Hệ thống này kết hợp với các khối DMA, bộ điều khiển bộ nhớ DRAM và các kết nối thông qua AXI SmartConnect, giúp giao tiếp hiệu quả giữa các thành phần trong hệ thống, tối ưu hóa hiệu suất cho các ứng dụng yêu cầu tính toán phức tạp như mạng CNN.

## 2.7. Tóm tắt

Chương 2 cung cấp cái nhìn tổng quan về kiến trúc và vai trò của vi xử lý trên SoC FPGA, một giải pháp tích hợp giữa tính linh hoạt của FPGA và khả năng xử lý mạnh mẽ của vi xử lý. Các loại vi xử lý trên SoC FPGA bao gồm: Vi xử lý đa năng lõi mềm, được tích hợp bằng cách sử dụng tài nguyên logic trên FPGA, mang lại sự linh hoạt cao trong thiết kế và tùy chỉnh, ví dụ như MicroBlaze của Xilinx và Nios II của Altera (Intel); Vi xử lý đa năng lõi cứng, là CPU vật lý được tích hợp trực tiếp trên chip FPGA, cung cấp hiệu suất xử lý cao, khả năng tiết kiệm năng lượng và hỗ trợ các hệ điều hành phức tạp như Linux, với các dòng FPGA phổ biến sử dụng vi xử lý lõi cứng bao gồm Cyclone V và Zynq-7000 với ARM Cortex-A9, cũng như Zynq UltraScale+ MPSoC và Versal sử dụng

Cortex-A53 và Cortex-A72; Vi xử lý chuyên dụng (DPU), được tối ưu hóa cho các tác vụ AI và học máy, đặc biệt là các phép toán tensor và ma trận, giúp tăng tốc các tác vụ tính toán phức tạp trong các ứng dụng AI, xử lý ảnh và xử lý ngôn ngữ tự nhiên; và Vi xử lý tự thiết kế, cho phép các nhà phát triển tự tạo ra kiến trúc vi xử lý phù hợp với yêu cầu cụ thể của hệ thống, mang lại khả năng tùy chỉnh và tối ưu hóa cao hơn so với các vi xử lý thương mại có sẵn. Chương này cũng giới thiệu cách tích hợp vi xử lý vào hệ thống SoC trên Altera và Xilinx FPGA sử dụng các phần mềm hỗ trợ như Quartus Prime và Vivado, giúp thiết kế và cấu hình hệ thống một cách linh hoạt và hiệu quả.

## 2.8. Câu hỏi và bài tập

1. Phân biệt vi xử lý đa năng lõi mềm và vi xử lý đa năng lõi cứng trên SoC FPGA.
2. Nêu ưu điểm và nhược điểm của vi xử lý đa năng lõi mềm và cứng trên FPGA.
3. Vi xử lý chuyên dụng (DPU) trên SoC FPGA được tối ưu hóa cho các tác vụ nào?
4. Tại sao vi xử lý tự thiết kế được sử dụng trên SoC FPGA và ưu điểm của nó so với vi xử lý thương mại?
5. Lý do nào khiến ARM Cortex-A9 phổ biến trên các dòng FPGA Cyclone V và Zynq-7000?
6. So sánh ARM Cortex-A9 với ARM Cortex-A53 về hiệu suất và ứng dụng trên FPGA.
7. Nêu sự khác biệt giữa kiến trúc ARM Cortex-A và ARM Cortex-R trong ứng dụng trên FPGA.
8. Các dòng ARM Cortex-A và Cortex-R nào được tích hợp trên Zynq Ultrascale+ MPSoC của Xilinx và ứng dụng của chúng?
9. Giải thích cách sử dụng cầu PS-đến-PL và PL-đến-PS trong việc giao tiếp giữa CPU lõi cứng và mảng logic lập trình được trên SoC FPGA.
10. Tại sao vi xử lý tự thiết kế trên SoC FPGA thường ưu tiên chọn kiến trúc RISC-V thay vì các kiến trúc vi xử lý khác?
11. Vi xử lý tự thiết kế có hỗ trợ các hệ điều hành phức tạp như Linux không? Nếu có, yêu cầu nào cần đáp ứng?

## CHƯƠNG 3: THIẾT KẾ BỘ NHỚ

### 3.1. Giới thiệu

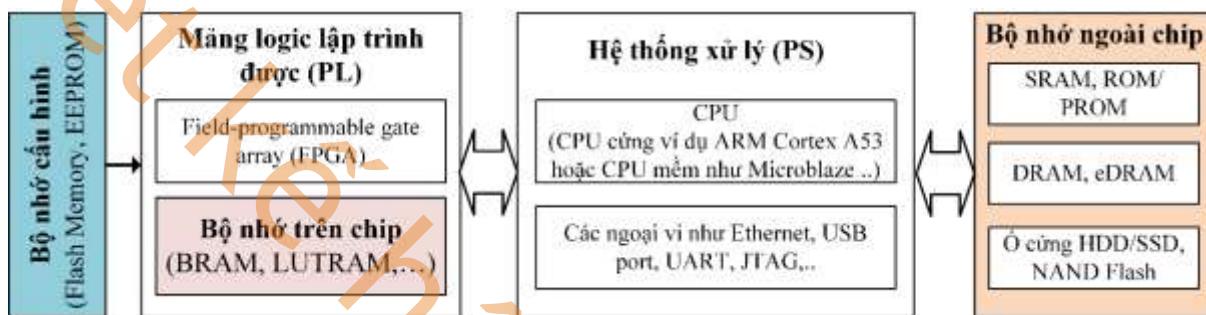
Bộ nhớ là một thành phần quan trọng trong thiết kế hệ thống SoC trên FPGA, đóng vai trò quyết định trong việc tối ưu hóa hiệu suất và khả năng mở rộng của hệ thống. Trong các ứng dụng tính toán hiệu năng cao, việc quản lý bộ nhớ hiệu quả giúp đảm bảo tốc độ truy xuất dữ liệu nhanh chóng và giảm thiểu độ trễ trong quá trình xử lý. FPGA cung cấp khả năng lập trình lại sau khi sản xuất, cho phép thiết kế các giải pháp bộ nhớ tùy chỉnh phù hợp với yêu cầu của từng ứng dụng.



Hình 3.1: Một số bộ nhớ và vị trí của chúng trên bảng mạch ZCU102.

Bộ nhớ không chỉ ảnh hưởng đến tốc độ và hiệu suất của hệ thống mà còn tác động đến năng lượng tiêu thụ và khả năng đáp ứng với các tác vụ xử lý phức tạp. Do đó, trong thiết kế SoC trên FPGA, việc lựa chọn kiến trúc bộ nhớ phù hợp là yếu tố quan trọng để đảm bảo rằng hệ thống hoạt động hiệu quả và tối ưu cho các ứng dụng yêu cầu tính toán mạnh mẽ. Như được thể hiện trong Hình 3.1, ví dụ minh họa về các loại bộ nhớ trên ZCU102 FPGA, bao gồm bộ nhớ trên chip (on-chip memory), bộ nhớ ngoài chip (off-chip memory) và bộ nhớ cấu hình (configuration memory), được sử dụng trong các hệ thống SoC FPGA để tối ưu hóa hiệu suất và đáp ứng các yêu cầu ứng dụng đa dạng. Bộ nhớ trên chip như BRAM (Block RAM) và URAM (Ultra RAM) giúp lưu trữ dữ liệu tạm thời trong

các tác vụ tính toán trực tiếp, trong khi bộ nhớ ngoài chip như PS 4GB DDR4 64-bit SODIMM hỗ trợ lưu trữ dữ liệu lâu dài và có thể mở rộng. Bộ nhớ cấu hình, như Dual 64MB Quad SPI Flash, đóng vai trò quan trọng trong việc nạp và cấu hình phần mềm cho các phần cứng có thể lập trình được. Các hệ thống SoC FPGA sử dụng những bộ nhớ này để quản lý dữ liệu, tối ưu hóa hoạt động và đảm bảo khả năng tùy chỉnh phần cứng linh hoạt.



Hình 3.2: Kiến trúc tổng quan một SoC trên FPGA với bộ nhớ trên chip (on-chip memory), bộ nhớ ngoài chip (off-chip memory), và bộ nhớ cấu hình (configuration memory).

Như đã được giới thiệu, bộ nhớ trong các hệ thống SoC FPGA được chia thành ba loại chính: on-chip memory (bộ nhớ trên chip), off-chip memory (bộ nhớ ngoài chip), và configuration memory (bộ nhớ cấu hình), như mô tả trong Hình 3.2. Các loại bộ nhớ này có những đặc điểm riêng biệt, đáp ứng các nhu cầu khác nhau của hệ thống, từ việc xử lý dữ liệu tạm thời đến việc lưu trữ dữ liệu lâu dài và cấu hình phần cứng:

- ✓ Bộ nhớ trên chip là các bộ nhớ được tích hợp trực tiếp trong mảng logic lập trình được của FPGA, chẳng hạn như BRAM, LUTRAM, và URAM. Các bộ nhớ này cung cấp khả năng truy cập dữ liệu cực nhanh với độ trễ thấp vì chúng nằm trực tiếp trên mạch logic của FPGA. Bộ nhớ trên chip được sử dụng cho các tác vụ đòi hỏi tốc độ xử lý cao và thời gian thực, chẳng hạn như xử lý tín hiệu số (DSP), mã hóa/giải mã dữ liệu, và các thuật toán trí tuệ nhân tạo đòi hỏi độ trễ thấp. Các bộ nhớ này giúp giảm thời gian trễ giao tiếp và tăng hiệu suất nhờ vào tốc độ truy cập dữ liệu nhanh chóng và băng thông cao.
- ✓ Bộ nhớ ngoài chip bao gồm các loại bộ nhớ nằm ngoài FPGA, chẳng hạn như SRAM, DRAM, eDRAM, ROM/PROM, và các thiết bị lưu trữ như HDD/SSD và NAND Flash. Những bộ nhớ này có dung lượng lớn hơn và thường được kết nối

với FPGA thông qua các giao diện như AXI hoặc PCIe. Bộ nhớ ngoài chip được sử dụng để lưu trữ các dữ liệu khối lớn, chương trình thực thi, và các tập dữ liệu cần xử lý. Mặc dù có độ trễ cao hơn do phải qua các giao diện giao tiếp, bộ nhớ này lại cung cấp dung lượng lớn hơn nhiều so với bộ nhớ trên chip và rất linh hoạt trong việc lưu trữ dữ liệu cho các ứng dụng như học máy và phân tích dữ liệu lớn.

- ✓ Bộ nhớ cấu hình (configuration memory) dùng để lưu trữ bitstream cho FPGA, giúp cấu hình lại mảng logic lập trình được (PL) mỗi khi FPGA khởi động lại hoặc khi có thay đổi trong thiết kế. Bộ nhớ cấu hình, thường là Flash Memory (ví dụ Quad SPI Flash trên Zynq FPGA) hoặc EEPROM, đảm bảo rằng FPGA có thể tải lại các thiết kế phần cứng cần thiết khi được bật hoặc tái cấu hình.

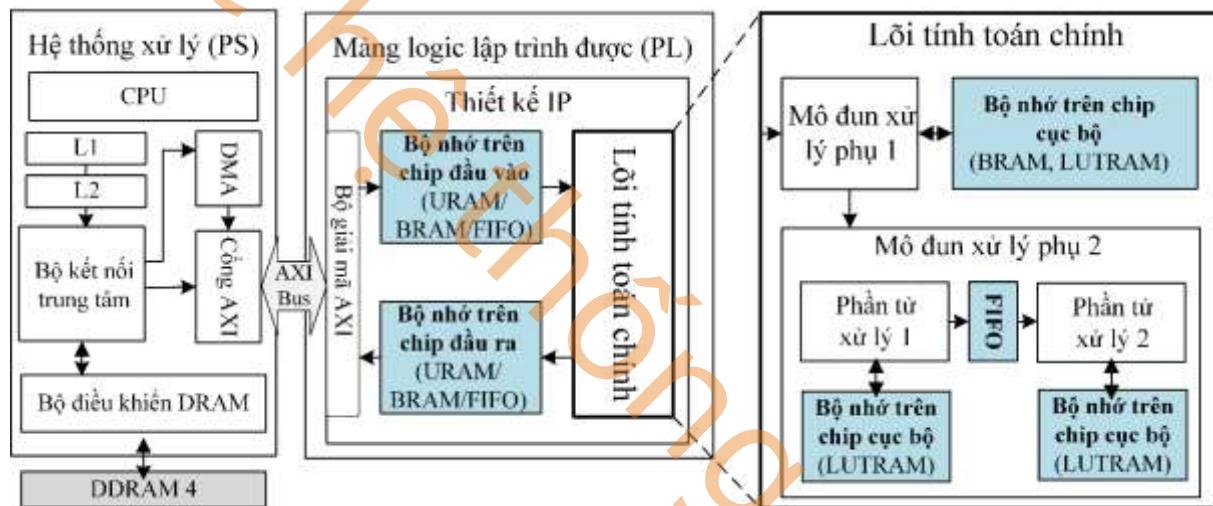
Lưu ý rằng trong bối cảnh sử dụng FPGA, sự phân loại giữa bộ nhớ trên chip và bộ nhớ ngoài chip được xác định dựa trên vị trí của bộ nhớ liên quan đến mảng logic lập trình được của FPGA. Do FPGA không phải là một phần của SoC cứng thuần túy, bất kỳ bộ nhớ nào nằm ngoài mảng logic lập trình được của FPGA đều được xem là bộ nhớ ngoài chip. Trong một hệ thống SoC, các bộ nhớ đệm như L1, L2, và L3 Cache thường được tích hợp trực tiếp vào CPU lõi cứng và được xem là bộ nhớ trên chip đối với CPU. Tuy nhiên, từ quan điểm của FPGA, những bộ nhớ đệm này được coi là bộ nhớ ngoài chip vì chúng nằm ngoài mảng logic lập trình được (PL) của FPGA. Điều này có nghĩa là, mặc dù các bộ nhớ đệm này là bộ nhớ trên chip đối với CPU, nhưng đối với FPGA, chúng được xem là bộ nhớ ngoài chip. Trong khi đó, nếu các bộ nhớ đệm L1, L2, và L3 này là của CPU lõi mềm hoặc CPU chuyên dụng tự thiết kế được sử dụng tài nguyên từ mảng logic lập trình được (PL), chẳng hạn như BRAM hoặc URAM, thì chúng được xem là on-chip memory. Điều này cũng áp dụng cho các loại bộ nhớ như DRAM, eDRAM, NAND Flash, và HDD/SSD, những bộ nhớ này thường được coi là off-chip memory khi chúng không nằm trong mảng logic lập trình được của FPGA. Nhìn chung, trong sách này, bộ nhớ được phân loại dựa trên vị trí và chức năng của chúng so với mảng logic lập trình được (PL) của FPGA. Cụ thể, bộ nhớ trên chip là loại bộ nhớ được tích hợp trực tiếp vào mảng PL của FPGA, bộ

nhớ ngoài chip là loại bộ nhớ nằm bên ngoài mảng PL của FPGA, và bộ nhớ cấu hình được sử dụng để lưu trữ tập tin bitstream nhằm cấu hình cho mảng PL của FPGA.

Các phần tiếp theo của chương này sẽ tập trung phân tích sâu về đặc điểm và cách thiết kế cho ba loại bộ nhớ này, bắt đầu với bộ nhớ trên chip, sau đó là bộ nhớ ngoài chip, và cuối cùng là bộ nhớ cấu hình. Sự sắp xếp này được thực hiện dựa trên mức độ ưu tiên thiết kế và điều chỉnh các loại bộ nhớ sao cho phù hợp với yêu cầu hiệu suất và khả năng mở rộng của hệ thống SoC trên FPGA.

## 3.2. Bộ nhớ trên chip (On-chip Memory)

### 3.2.1. Giới thiệu



Hình 3.3: Một SoC đơn giản với thiết kế IP trên mảng logic lập trình được sử dụng nhiều loại bộ nhớ trên chip để lưu trữ dữ liệu tính toán.

Bộ nhớ trên chip đóng vai trò quan trọng trong các hệ thống tính toán tích hợp, đặc biệt là trong các thiết kế SoC. Bộ nhớ này được tích hợp trực tiếp trên mạch của FPGA để đảm bảo hiệu suất cao, băng thông lớn và độ trễ thấp trong quá trình xử lý dữ liệu. Các loại bộ nhớ trên chip phổ biến trên FPGA bao gồm BRAM, URAM, LUTRAM, và FIFO (First In First Out). Mỗi loại bộ nhớ này có đặc điểm và ứng dụng khác nhau, phù hợp với các yêu cầu xử lý và lưu trữ cụ thể trong hệ thống. Bộ nhớ trên chip trong FPGA không chỉ bao gồm các loại bộ nhớ tích hợp sẵn như BRAM, URAM, và LUTRAM mà còn có thể bao gồm bộ nhớ tự thiết kế (custom memory), giống như SRAM. Bộ nhớ tự thiết kế hiệu

quả cho lưu trữ dữ liệu nhỏ và truy xuất một lần nhiều dữ liệu cùng lúc do sử dụng tài nguyên LUT và FF. Tuy nhiên, việc sử dụng quá nhiều tài nguyên này có thể ảnh hưởng đến hiệu suất và kích thước thiết kế, vì vậy nên hạn chế sử dụng trong các ứng dụng yêu cầu lưu trữ dữ liệu lớn.

Hình 3.3 minh họa một thiết kế SoC đơn giản với hệ thống xử lý (PS) và mảng logic lập trình được (PL). PS bao gồm CPU, bộ nhớ đệm L1, L2 và bộ điều khiển DRAM ngoài chip, với dữ liệu từ DDRAM4 được chuyển vào phần PL qua cổng AXI và DMA, sử dụng băng thông cao của bộ nhớ ngoài chip để tăng tốc độ xử lý. Lưu ý: PS không có bộ nhớ trên chip. Trong thiết kế SoC này, các loại bộ nhớ trên chip như BRAM, URAM, LUTRAM, và FIFO được sử dụng trong các thiết kế IP ở phần PL để lưu trữ dữ liệu cho các quá trình tính toán. BRAM và URAM được dùng để lưu trữ dữ liệu đầu vào từ hệ thống PS hoặc bộ nhớ ngoài, cung cấp không gian lưu trữ tạm thời cho dữ liệu cần xử lý. Bộ nhớ đầu ra sử dụng FIFO để đảm bảo đồng bộ hóa và lưu trữ dữ liệu trước khi truyền ra ngoài hệ thống. Các lõi tính toán trong thiết kế IP có thể sử dụng BRAM hoặc LUTRAM để lưu trữ dữ liệu tạm thời, giúp giảm thời gian truy xuất và tăng tốc độ xử lý, trong khi FIFO được sử dụng để đồng bộ hóa dữ liệu giữa các phần tử xử lý, đảm bảo dữ liệu được truyền đi đúng thứ tự và không bị mất.

Nhìn chung, trong thiết kế SoC, các loại bộ nhớ trên chip như BRAM, URAM, LUTRAM, và FIFO được sử dụng để lưu trữ dữ liệu từ bộ nhớ ngoài chip (như DRAM) và dữ liệu nội bộ cần thiết cho các IP để tính toán. Mỗi loại bộ nhớ có ưu điểm riêng:

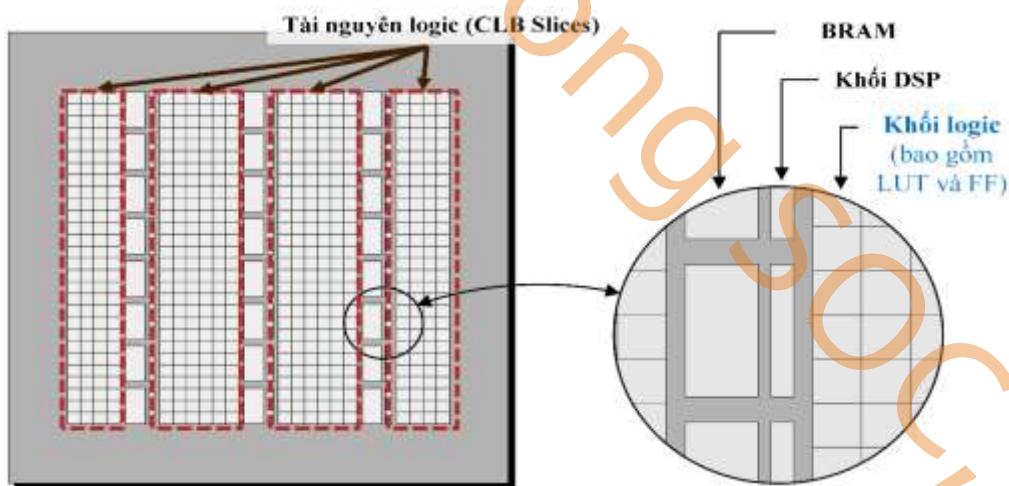
- ✓ URAM: Cung cấp dung lượng lớn hơn BRAM và phù hợp với các ứng dụng yêu cầu lưu trữ dữ liệu lớn với tốc độ cao.
- ✓ BRAM: Cung cấp dung lượng vừa phải với độ trễ thấp, thích hợp cho các bộ nhớ đệm và lưu trữ tạm thời.
- ✓ LUTRAM: Linh hoạt trong việc cấu hình và phù hợp với các ứng dụng yêu cầu lưu trữ nhỏ với độ trễ cực thấp.

- ✓ FIFO: Cung cấp khả năng đồng bộ hóa và lưu trữ dữ liệu tuần tự, giúp quản lý dữ liệu giữa các phần của hệ thống có tốc độ xử lý khác nhau, và đảm bảo rằng dữ liệu đầu vào và đầu ra được xử lý đúng thứ tự.

Phản tiếp theo sẽ tập trung mô tả chi tiết về BRAM, LUTRAM, và FIFO trong FPGA. Trong khi đó, URAM, mặc dù có sự tương đồng về bản chất với BRAM nhưng được hỗ trợ ở các FPGA đời mới và cung cấp dung lượng lớn hơn, sẽ được đề cập một cách đơn giản do có tính năng tương tự BRAM nhưng với khả năng lưu trữ lớn hơn.

### 3.2.2. LUTRAM hay Distributed RAM

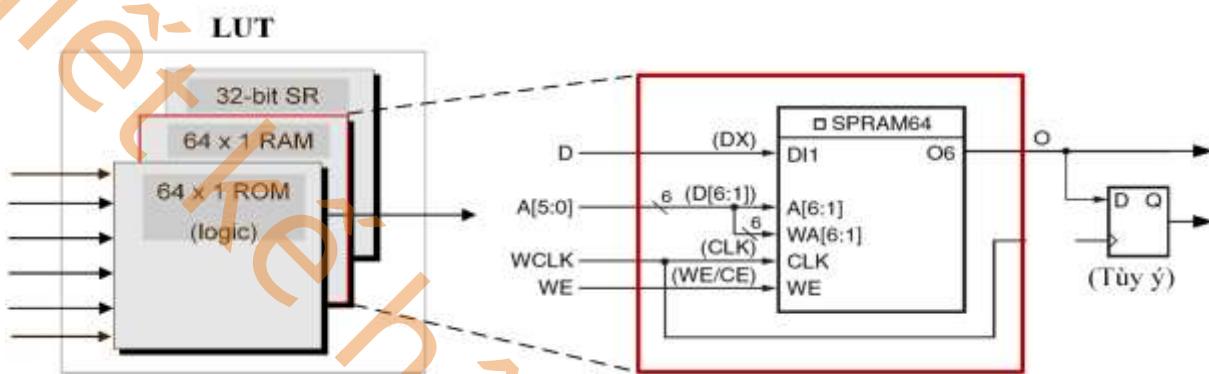
LUTRAM, hay còn gọi là Distributed RAM, là một loại bộ nhớ được cấu hình từ các Look-Up Table (LUT) có sẵn trong các khối logic của FPGA. LUTRAM được dùng chủ yếu để lưu trữ các dữ liệu nhỏ và có thể truy cập với độ trễ rất thấp. Đặc tính này phù hợp cho các ứng dụng đòi hỏi bộ nhớ nhanh, dung lượng nhỏ như các bộ đệm (buffers) hoặc các bộ nhớ tạm thời (scratchpad memory). LUTRAM cũng thường được sử dụng trong các máy trạng thái (state machine) để lưu trữ trạng thái hoặc trong các ứng dụng cần lưu trữ dữ liệu nhỏ với tốc độ truy xuất nhanh.



Hình 3.4: Vị trí LUT trong khối logic FPGA có thể tạo thành LUTRAM [13].

Hình 3.4 mô tả vị trí của các LUT bên trong các khối logic của FPGA, nơi mà LUT có thể được cấu hình thành bộ nhớ LUTRAM. Các LUT này nằm trong các tài nguyên logic CLB (Configurable Logic Blocks) và có thể được sử dụng để tạo ra các bộ nhớ nhỏ có độ trễ thấp. Trong các FPGA của Xilinx, bộ nhớ được tạo từ các khối logic được gọi là

LUTRAM. Tuy nhiên, trong các FPGA của Altera (nay là Intel), thuật ngữ LUTRAM không được sử dụng, nhưng chúng có thể thực hiện chức năng tương tự bằng cách sử dụng các thanh ghi trong Phần tử Logic (Logic Element - LE) như các ô RAM. Về bản chất, LUTRAM ở Xilinx FPGA và Distributed RAM bằng LE ở Altera FPGA có nguyên lý giống nhau, nên những phần sau sẽ chỉ dùng LUTRAM cho đơn giản và đồng bộ.



Hình 3.5 Cấu trúc bên trong LUT khi được cấu hình thành bộ nhớ LUTRAM [13].

Hình 3.5 thể hiện cấu trúc bên trong của một LUT khi nó được cấu hình thành bộ nhớ LUTRAM. Trong trường hợp này, LUT có thể lưu trữ dữ liệu ở dạng RAM ( $64 \times 1$  RAM) thay vì dạng mặc định là ROM (Read-Only Memory) hoặc các mạch logic khác. Khi LUT được cấu hình thành RAM, nó có thể lưu trữ và đọc dữ liệu theo cách thức tương tự như một bộ nhớ RAM nhỏ.

```

module LUTRAM (CLK, WE, A, Di, O);
    input CLK;
    input WE;
    input [5:0] A;
    input [15:0] Di;
    output [15:0] O;

    reg [15:0] ram [63:0];

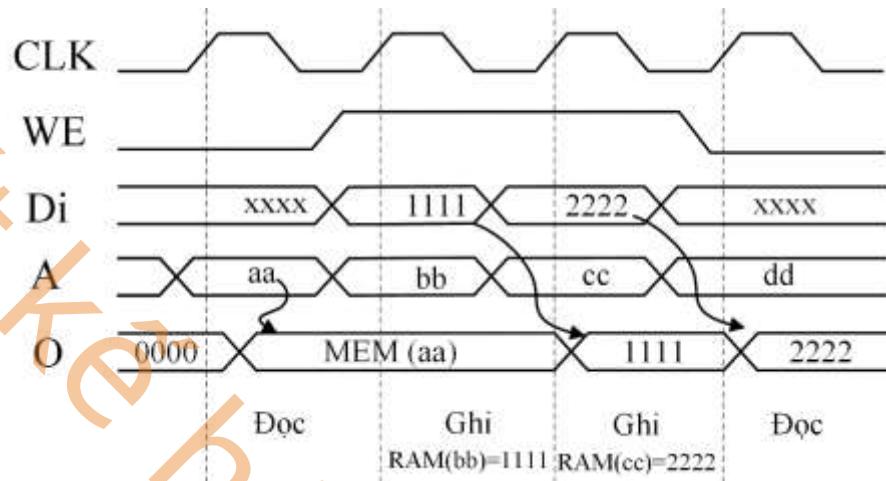
    always @(posedge CLK)
    begin
        if (WE)
            ram[A] <= Di;
    end

    assign O = ram[A];
endmodule

```

Hình 3.6: Mô tả LUTRAM bằng Verilog.

Hình 3.6 cung cấp code Verilog mô tả cách tạo LUTRAM từ các LUT trong FPGA. Code Verilog này cho phép thiết kế bộ nhớ RAM 64 x 16 bits, sử dụng LUT để tạo ra bộ nhớ có thể đọc ghi được.

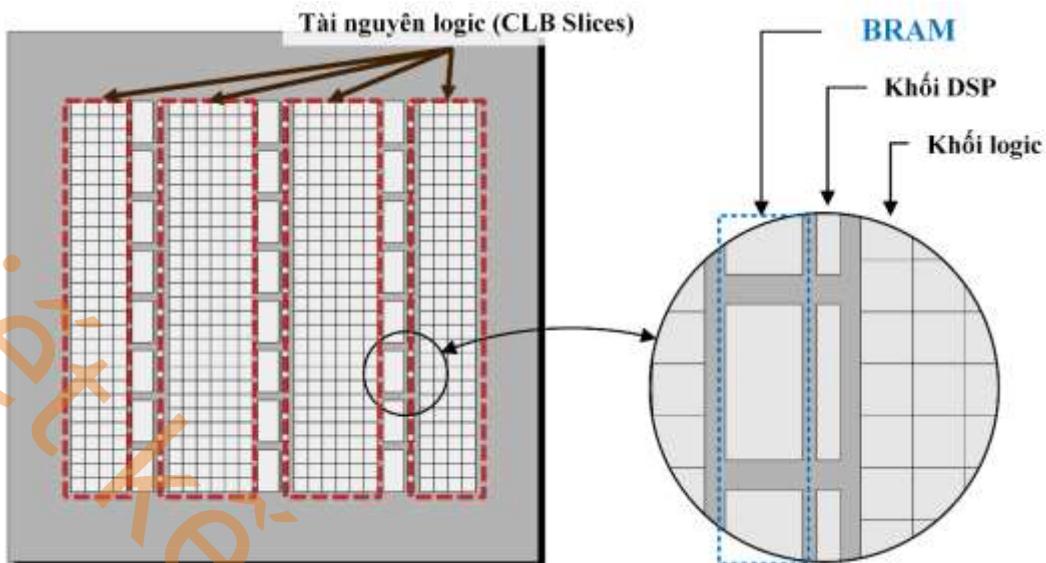


Hình 3.7: Sơ đồ thời gian chi tiết của tín hiệu trong LUT cho việc đọc và ghi.

Hình 3.7 trình bày sơ đồ thời gian chi tiết của các tín hiệu cho việc đọc và ghi trong một LUT được cấu hình thành LUTRAM. Các tín hiệu chính bao gồm: CLK (Clock), là tín hiệu xung nhịp đồng bộ hóa toàn bộ hoạt động; WE (Write Enable), cho phép ghi dữ liệu vào LUTRAM khi ở mức cao và chỉ cho phép đọc khi ở mức thấp; Di (Data In), là tín hiệu đầu vào dữ liệu cần ghi vào bộ nhớ khi tín hiệu WE kích hoạt; A (Address), tín hiệu địa chỉ xác định vị trí trong LUTRAM để đọc hoặc ghi; và O (Output), là tín hiệu đầu ra nơi dữ liệu được đọc từ địa chỉ A xuất hiện. Trong quá trình hoạt động, khi tín hiệu WE ở mức thấp, LUTRAM thực hiện đọc, ví dụ như địa chỉ "aa" được chọn và giá trị tương ứng MEM(aa) được xuất ra tại O. Khi WE chuyển sang mức cao, LUTRAM cho phép ghi dữ liệu, ví dụ, dữ liệu "1111" tại tín hiệu Di được ghi vào địa chỉ "bb" và lưu trữ tại RAM(bb). Tương tự, dữ liệu "2222" sẽ được ghi vào địa chỉ "cc" và lưu trữ tại RAM(cc).

Nhìn chung, LUTRAM là một giải pháp bộ nhớ nhỏ gọn, nhanh chóng và linh hoạt trên FPGA, thích hợp cho các ứng dụng yêu cầu độ trễ thấp và dung lượng bộ nhớ không lớn. So với các loại bộ nhớ khác như BRAM hay URAM, LUTRAM có lợi thế về độ trễ, nhưng hạn chế về dung lượng.

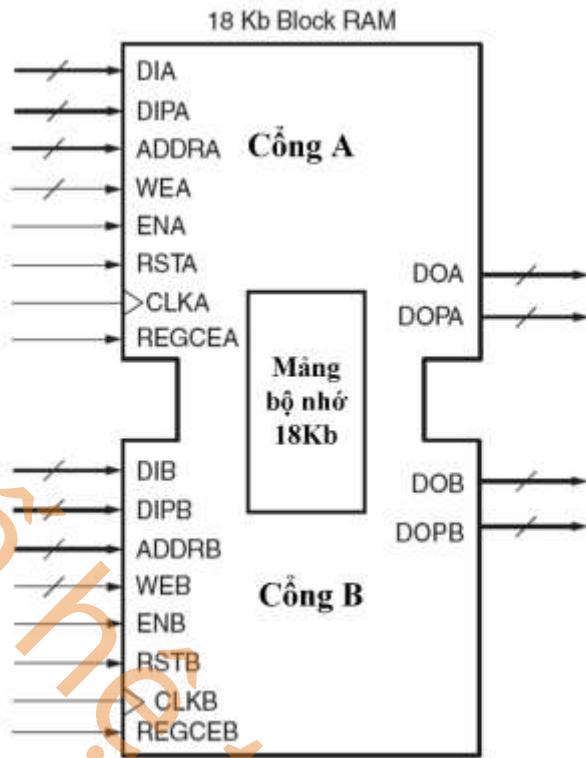
### 3.2.3. Block RAM



Hình 3.8: Vị trí BRAM trong FPGA [13].

Bộ nhớ Block RAM (BRAM) là một thành phần quan trọng trong FPGA, cung cấp khả năng lưu trữ dữ liệu trực tiếp trên chip với độ trễ thấp và băng thông cao. BRAM có kích thước lớn hơn so với LUTRAM (RAM tạo thành từ các LUT), vì vậy nó thường được sử dụng để lưu trữ các khối dữ liệu lớn hơn, hỗ trợ các ứng dụng đòi hỏi dung lượng bộ nhớ cao hơn. Được cấu trúc dưới dạng các khối bộ nhớ riêng biệt và độc lập, BRAM cho phép thiết kế linh hoạt trong việc lưu trữ và truy xuất dữ liệu, đặc biệt hữu ích cho các ứng dụng cần tốc độ xử lý cao như xử lý tín hiệu số, điều khiển hệ thống, và xử lý dữ liệu trong các hệ thống SoC. Trong thiết kế IP, BRAM thường được sử dụng như bộ nhớ toàn cục (global memory) để lưu trữ các dữ liệu lớn cần được truy cập bởi nhiều thành phần xử lý khác nhau, hoặc như bộ nhớ cục bộ (local memory) để lưu trữ dữ liệu tạm thời và trạng thái trong các lõi xử lý cụ thể. Sự đa dụng này giúp BRAM trở thành một lựa chọn phổ biến trong các thiết kế FPGA phức tạp.

Hình 3.8 cho thấy vị trí của các khối BRAM trong một FPGA. BRAM được phân bổ dọc theo chiều dọc của FPGA, xen kẽ với các khối logic và DSP, cho phép kết nối nhanh chóng với các thành phần logic khác. Điều này giúp tối ưu hóa đường truyền dữ liệu và giảm thiểu độ trễ trong quá trình truy xuất bộ nhớ.



Hình 3.9: Giao diện bộ nhớ BRAM [14].

Hình 3.9 mô tả giao diện của một khối BRAM có dung lượng 18 Kb. Bộ nhớ BRAM hỗ trợ giao diện hai cổng (cổng A và cổng B), cho phép đọc và ghi độc lập. Mỗi cổng bao gồm các tín hiệu dữ liệu đầu vào (DIA/DIB), địa chỉ (ADDRA/ADDRB), xung nhịp (CLKA/CLKB), tín hiệu điều khiển ghi (WEA/WEB), và tín hiệu điều khiển đầu ra (DOA/DOB). Bảng 3.1 cung cấp mô tả chi tiết cho các tín hiệu này, giúp người thiết kế dễ dàng hiểu và sử dụng.

**Bảng 3.1: Các tín hiệu của bộ nhớ BRAM**

Tên tín hiệu của cổng A và B	Mô tả
DIA/B	Tín hiệu đầu vào dữ liệu
DIPA/B	Tín hiệu đầu vào kiểm tra chẵn lẻ dữ liệu
ADDRA/B	Tín hiệu địa chỉ
WEA/B	Cho phép ghi theo chiều rộng byte

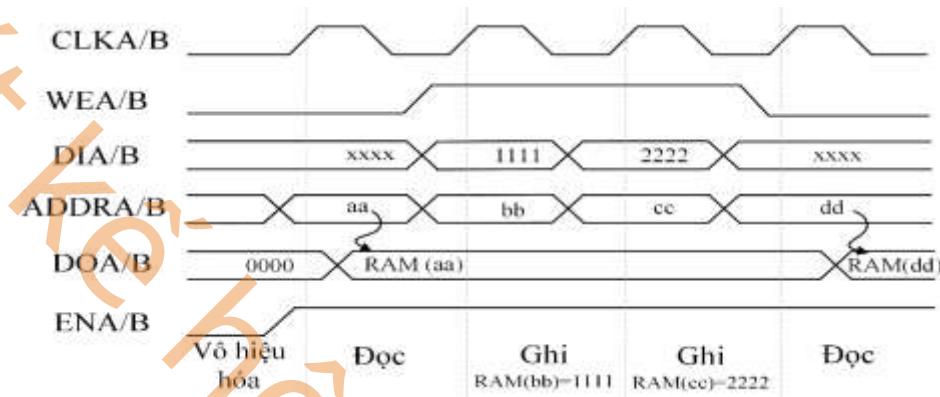
ENA/B	Khi không hoạt động, không có dữ liệu nào được ghi vào RAM khỏi và bus đầu ra vẫn giữ nguyên trạng thái trước đó.
RSTA/B	Đặt lại/Thiết lập đồng bộ các thanh ghi đầu ra (DO_REG = 1).
CLKA/B	Đầu vào xung nhịp
DOA/B	Tín hiệu đầu ra dữ liệu
DOPA/B	Tín hiệu đầu ra kiểm tra chẵn lẻ dữ liệu
REGCEA/B	Cho phép xung nhịp thanh ghi đầu ra

Bảng 3.1 liệt kê các tín hiệu đầu vào và đầu ra quan trọng của bộ nhớ BRAM trong FPGA. Mỗi cổng của BRAM, A và B, đều có các tín hiệu riêng để điều khiển và truyền dữ liệu. Tín hiệu đầu vào dữ liệu (DIA/B) và tín hiệu đầu vào kiểm tra chẵn lẻ (DIPA/B) cung cấp dữ liệu và kiểm tra tính toàn vẹn của dữ liệu được ghi vào bộ nhớ. Tín hiệu địa chỉ (ADDRA/B) xác định vị trí dữ liệu cần truy xuất hoặc ghi vào. Tín hiệu WEA/B là tín hiệu cho phép ghi theo chiều rộng byte, quyết định việc ghi dữ liệu vào bộ nhớ. ENA/B là tín hiệu cho phép hoạt động của cổng; khi không hoạt động, không có dữ liệu nào được ghi vào BRAM và tín hiệu đầu ra vẫn giữ nguyên trạng thái trước đó. Tín hiệu RSTA/B được sử dụng để thiết lập lại hoặc đặt lại đồng bộ các thanh ghi đầu ra khi DO\_REG = 1. CLKA/B là tín hiệu đầu vào xung nhịp cho cổng A hoặc B, điều khiển tốc độ hoạt động của bộ nhớ. Tín hiệu đầu ra dữ liệu (DOA/B) và tín hiệu đầu ra kiểm tra chẵn lẻ (DOPA/B) cung cấp dữ liệu và thông tin kiểm tra chẵn lẻ từ bộ nhớ. Cuối cùng, REGCEA/B là tín hiệu cho phép xung nhịp thanh ghi đầu ra, điều khiển hoạt động của các thanh ghi này.

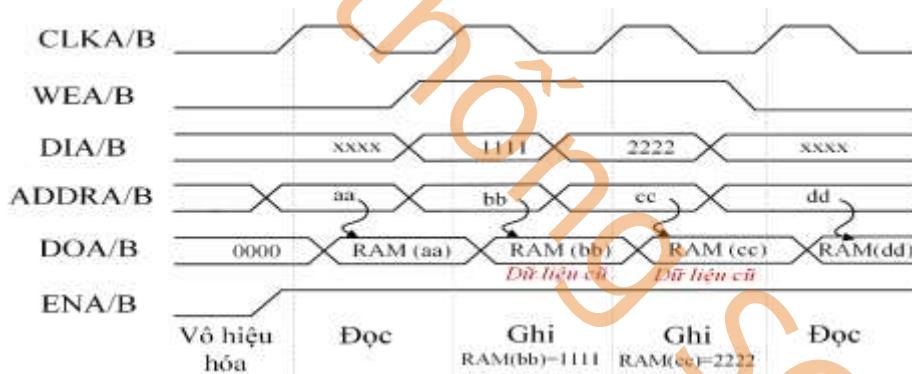
BRAM có thể hoạt động ở nhiều chế độ khác nhau tùy thuộc vào cấu hình và yêu cầu của ứng dụng:

- ✓ **Chế độ NO\_CHANGE:** Ở chế độ này, giá trị đầu ra không thay đổi trong quá trình ghi. Sơ đồ thời gian của tín hiệu trong chế độ NO\_CHANGE được mô tả trong Hình 3.10. Trong trường hợp này, khi có truy cập đồng thời đọc và ghi, tín hiệu dữ liệu đầu ra (DOA/DOA) sẽ giữ nguyên giá trị trước đó trong suốt quá trình ghi. Điều này có nghĩa là dữ liệu đầu ra sẽ không bị cập nhật cho đến khi việc ghi hoàn tất và phép đọc sẽ phản ánh dữ liệu cũ trong suốt quá trình ghi. Các tín hiệu DIA/B (dữ

liệu đầu vào), WEA/B (tín hiệu ghi cho phép), và ADDRA/B (địa chỉ của bộ nhớ) điều khiển việc ghi và đọc dữ liệu từ bộ nhớ. Khi WEA/B được kích hoạt, dữ liệu mới sẽ được ghi vào bộ nhớ tại địa chỉ tương ứng (ADDRA/B), nhưng DOA/B vẫn giữ giá trị cũ cho đến khi việc ghi hoàn tất. Chế độ NO\_CHANGE được sử dụng khi muốn tránh việc thay đổi giá trị đầu ra trong khi quá trình ghi đang diễn ra.



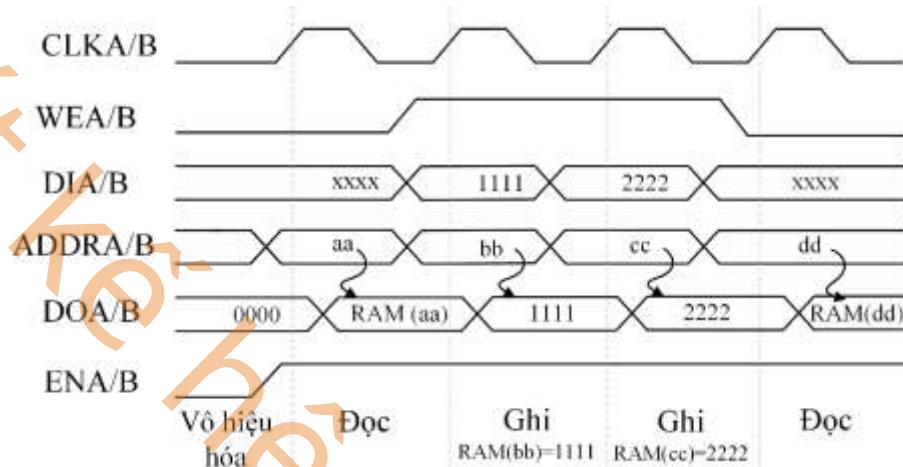
Hình 3.10: Sơ đồ thời gian chi tiết của tín hiệu trong BRAM cho việc đọc và ghi dữ liệu ở chế độ NO\_CHANGE [14].



Hình 3.11: Sơ đồ thời gian chi tiết của tín hiệu trong BRAM cho việc đọc và ghi dữ liệu ở chế độ READ\_FIRST [14].

- ✓ **Chế độ READ\_FIRST:** Ở chế độ READ\_FIRST, khi có truy cập đồng thời đọc và ghi, việc đọc dữ liệu sẽ được thực hiện trước khi ghi dữ liệu vào bộ nhớ. Điều này có nghĩa là dữ liệu sẽ được đọc từ bộ nhớ tại thời điểm truy xuất, và sau đó dữ liệu mới sẽ được ghi vào bộ nhớ. Do đó, dữ liệu đầu ra sẽ luôn phản ánh giá trị của bộ nhớ trước khi cập nhật. Hình 3.11 minh họa sơ đồ thời gian cho chế độ READ\_FIRST. Trong chế độ này, các tín hiệu điều khiển như DIA/B (dữ liệu đầu vào), WEA/B (tín hiệu ghi cho phép), và ADDRA/B (địa chỉ bộ nhớ) sẽ điều khiển

quá trình đọc và ghi. Khi có tín hiệu WEA/B kích hoạt, bộ nhớ sẽ thực hiện việc ghi tại địa chỉ ADDRA/B, nhưng DIA/B sẽ chỉ được ghi vào bộ nhớ sau khi việc đọc hoàn tất. Chế độ READ\_FIRST thường được sử dụng khi cần đảm bảo rằng dữ liệu được đọc chính xác trước khi có bất kỳ thay đổi nào được thực hiện.



Hình 3.12: Sơ đồ thời gian chi tiết của tín hiệu trong BRAM cho việc đọc và ghi dữ liệu ở chế độ WRITE\_FIRST [14].

- ✓ **Chế độ WRITE\_FIRST:** Ở chế độ WRITE\_FIRST, dữ liệu mới sẽ được ghi trực tiếp vào bộ nhớ và xuất hiện ngay lập tức tại đầu ra, bất chấp có truy cập đọc đồng thời hay không. Hình 3.12 minh họa sơ đồ thời gian cho chế độ này. Trong chế độ WRITE\_FIRST, khi có tín hiệu ghi vào bộ nhớ, các tín hiệu như DIA/B (dữ liệu đầu vào), WEA/B (tín hiệu ghi cho phép), và ADDRA/B (địa chỉ bộ nhớ) sẽ điều khiển việc ghi dữ liệu vào bộ nhớ tại các địa chỉ ADDRA/B. Dữ liệu mới được ghi vào bộ nhớ và sẽ được cập nhật ngay lập tức tại đầu ra (DOA/B), ngay lập tức thay vì đợi đến khi kết thúc một quá trình đọc. Chế độ này thường được sử dụng khi cần cập nhật dữ liệu nhanh chóng, đặc biệt trong các ứng dụng yêu cầu dữ liệu mới phải được phản ánh ngay lập tức.

Có hai cách chính để sử dụng BRAM trong thiết kế IP bằng ngôn ngữ Verilog: một là viết code Verilog theo chuẩn để tự nhận BRAM, và hai là gọi BRAM thông qua mô đun IP. Cụ thể, cách đầu tiên là thiết kế mô đun BRAM bằng code Verilog, trong đó người thiết

kết khai báo bộ nhớ trực tiếp trong code và thực hiện truy xuất bộ nhớ thông qua các tín hiệu điều khiển như địa chỉ, dữ liệu và tín hiệu ghi/đọc như sau.

```
module BRAM #(  
    parameter AWIDTH = 10, // độ rộng địa chỉ  
    parameter DWIDTH = 32 // độ rộng dữ liệu  
(input clka, // xung nhịp (clock)  
// *** Cổng A ***//  
input ena, // tín hiệu cho phép đọc cổng A  
input wea, // tín hiệu cho phép ghi cổng A  
input [AWIDTH-1:0] addra, // địa chỉ cổng A  
input [DWIDTH-1:0] dina, // dữ liệu cổng A  
output reg [DWIDTH-1:0] douta, // dữ liệu đầu ra cổng A  
// *** Cổng B ***//  
input clkb, // xung nhịp (clock)  
input enb, // tín hiệu cho phép đọc cổng B  
input web, // tín hiệu cho phép ghi cổng B  
input [AWIDTH-1:0] addrb, // địa chỉ cổng B  
input [DWIDTH-1:0] dinb, // dữ liệu cổng B  
output reg [DWIDTH-1:0] doutb ); // dữ liệu đầu ra cổng B  
// Khai báo bộ nhớ BRAM với kiểu "block",  
// sử dụng kích thước bộ nhớ theo độ rộng địa chỉ AWIDTH và độ rộng dữ liệu DWIDTH.  
(* ram_style = "block" *) reg [DWIDTH-1:0] mem [2**AWIDTH-1:0];  
  
always @(posedge clka) begin  
// *** Cổng A ***//  
if (ena) begin  
    if (wea) begin  
        mem[addra] <= dina; // ghi dữ liệu vào bộ nhớ tại địa chỉ addra  
    end  
    douta <= mem[addra]; // đọc dữ liệu từ bộ nhớ tại địa chỉ addra  
end  
end  
  
always @(posedge clkb) begin  
// *** Cổng B ***//  
if (enb) begin  
    if (web) begin  
        mem[addrb] <= dinb; // ghi dữ liệu vào bộ nhớ tại địa chỉ addrb  
    end  
    doutb <= mem[addrb]; // đọc dữ liệu từ bộ nhớ tại địa chỉ addrb  
end  
end  
endmodule
```

Hình 3.13: Code Verilog mô tả Dual-Port BRAM với hai cổng đọc và ghi, sử dụng bộ nhớ BRAM trong thiết kế IP.

Hình 3.13 mô tả một đoạn code Verilog cho Dual-Port BRAM với hai cổng đọc và ghi, sử dụng BRAM trong thiết kế IP. Code này cho phép truy xuất độc lập giữa hai cổng (Port A và Port B) để đọc và ghi dữ liệu vào bộ nhớ. Cổng A và cổng B đều có tín hiệu

điều khiển đọc (ena, enb) và tín hiệu điều khiển ghi (wea, web). Bộ nhớ BRAM được khai báo với kiểu "block" và có kích thước xác định bởi độ rộng địa chỉ AWIDTH và độ rộng dữ liệu DWIDTH. Các thao tác ghi và đọc được thực hiện đồng thời ở cả hai cổng, với dữ liệu được ghi vào bộ nhớ tại các địa chỉ addra và addrb, và đọc dữ liệu từ bộ nhớ tại các địa chỉ tương ứng. Khi gọi đoạn code này và tổng hợp (Synthesis), triển khai (Implementation), hoặc tạo Bitstream, phần mềm chuyên tổng hợp SoC như Vivado sẽ tự động chuyển đổi mô đun BRAM này thành tài nguyên BRAM trong FPGA.

```

/* 1. Chế độ NO_CHANGE:*/
always @(posedge clka) begin
    // *** Cổng A ***
    if (ena) begin
        if (wea) begin
            mem[addra] <= dina; // ghi dữ liệu vào bộ nhớ tại địa chỉ addra
        end
        douta <= douta; // giữ nguyên giá trị của douta trong suốt quá trình ghi
    end
end

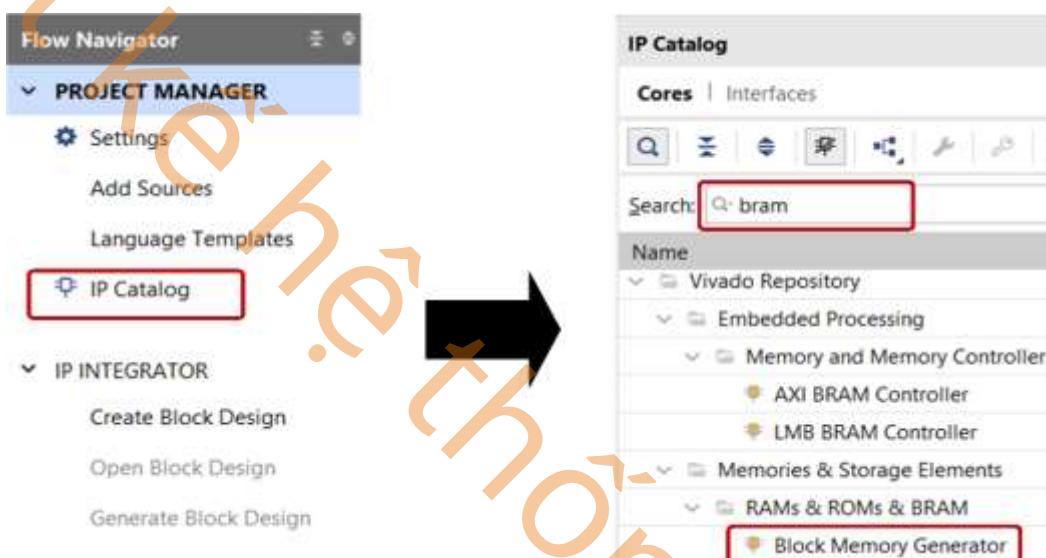
/* 2. Chế độ READ_FIRST:*/
always @(posedge clka) begin
    // *** Cổng A ***
    if (ena) begin
        douta <= mem[addra]; // đọc dữ liệu từ bộ nhớ tại địa chỉ addra
        if (wea) begin
            mem[addra] <= dina; // ghi dữ liệu vào bộ nhớ tại địa chỉ addra
        end
    end
end

/* 3. Chế độ WRITE_FIRST:*/
always @(posedge clka) begin
    // *** Cổng A ***
    if (ena) begin
        if (wea) begin
            mem[addra] <= dina; // ghi dữ liệu vào bộ nhớ tại địa chỉ addra
            douta <= dina; // cập nhật ngay lập tức dữ liệu đầu ra sau khi ghi
        end
        else begin
            douta <= mem[addra]; // đọc dữ liệu từ bộ nhớ tại địa chỉ addra
        end
    end
end

```

Hình 3.14: Code Verilog để điều chỉnh mô đun BRAM với các chế độ NO\_CHANGE, READ\_FIRST, và WRITE\_FIRST ở Cổng A, cổng B sẽ tương tự hoàn toàn.

Để thay đổi chế độ hoạt động của BRAM trong thiết kế Verilog, chế độ có thể được điều chỉnh thông qua đoạn mã như minh họa trong Hình 3.14. Hình này mô tả cách thay đổi chế độ của BRAM với các chế độ NO\_CHANGE, READ\_FIRST, và WRITE\_FIRST. Các tín hiệu điều khiển trong khối always của Công A cần được điều chỉnh, và Công B sẽ có cấu trúc tương tự như Công A. Các chế độ hoạt động này không có loại nào là tối ưu nhất, mà chỉ là lựa chọn phù hợp với bối cảnh thuật toán yêu cầu để có thể tối ưu hóa xử lý.

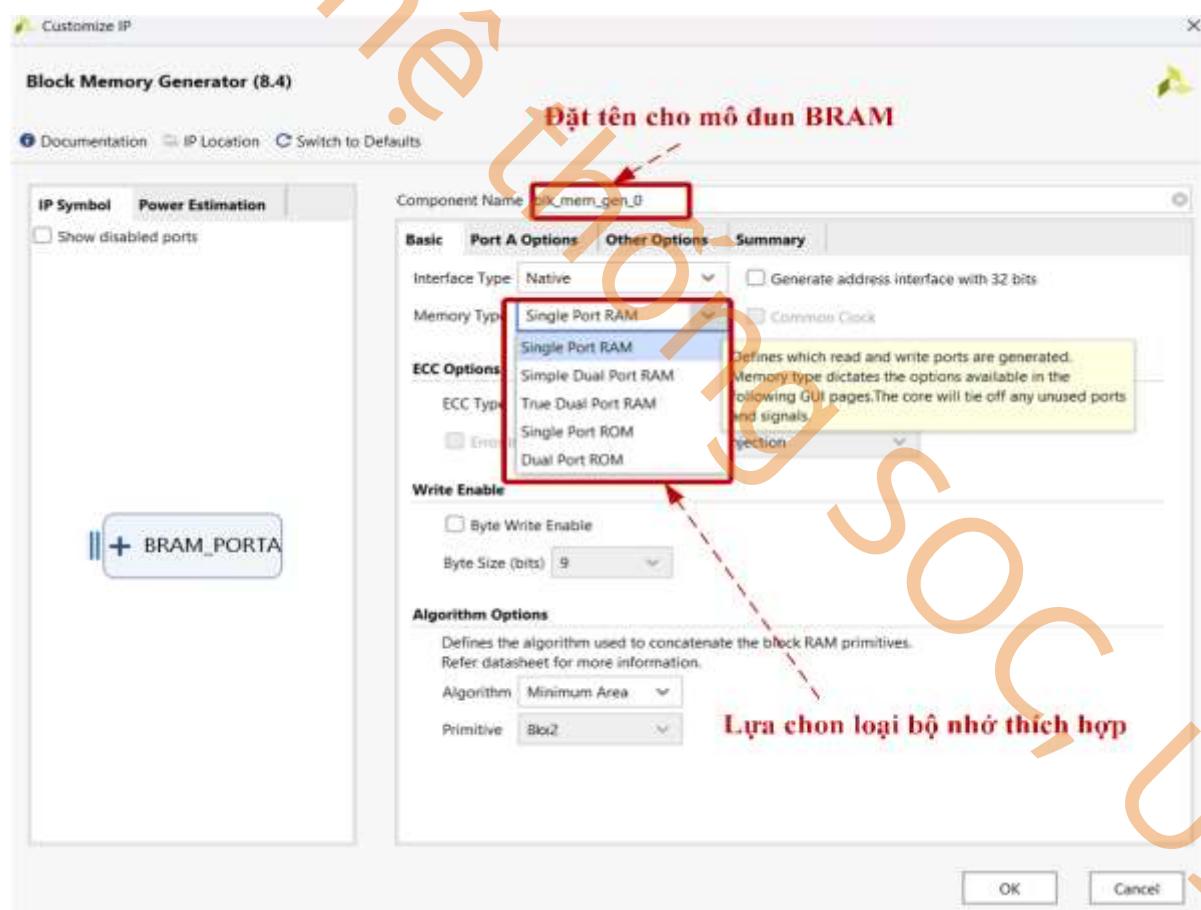


Hình 3.15: Giao diện IP Catalog trong Vivado, nơi tìm kiếm và chọn mô-đun Block Memory Generator để tạo và cấu hình BRAM trong thiết kế IP ở FPGA.

Cách thứ hai để sử dụng BRAM trong thiết kế FPGA là gọi mô-đun IP có sẵn trong các phần mềm thiết kế phần cứng trên FPGA, ví dụ phần mềm Vivado cho các bo FPGA dòng Xilinx. Khi sử dụng phương pháp này, người thiết kế không cần phải viết mã từ đầu mà có thể tận dụng các mô-đun BRAM đã được tối ưu hóa sẵn. Trong Vivado, người thiết kế có thể mở IP Catalog, tìm kiếm và chọn mô-đun Block Memory Generator, sau đó cấu hình các tham số của BRAM như kích thước bộ nhớ, chế độ đọc và ghi, và số cổng cần thiết cho ứng dụng cụ thể. Hình 3.15 minh họa quá trình này, cho thấy cách thức tìm kiếm và chọn mô-đun Block Memory Generator trong IP Catalog để sử dụng trong thiết kế phần cứng trên FPGA.

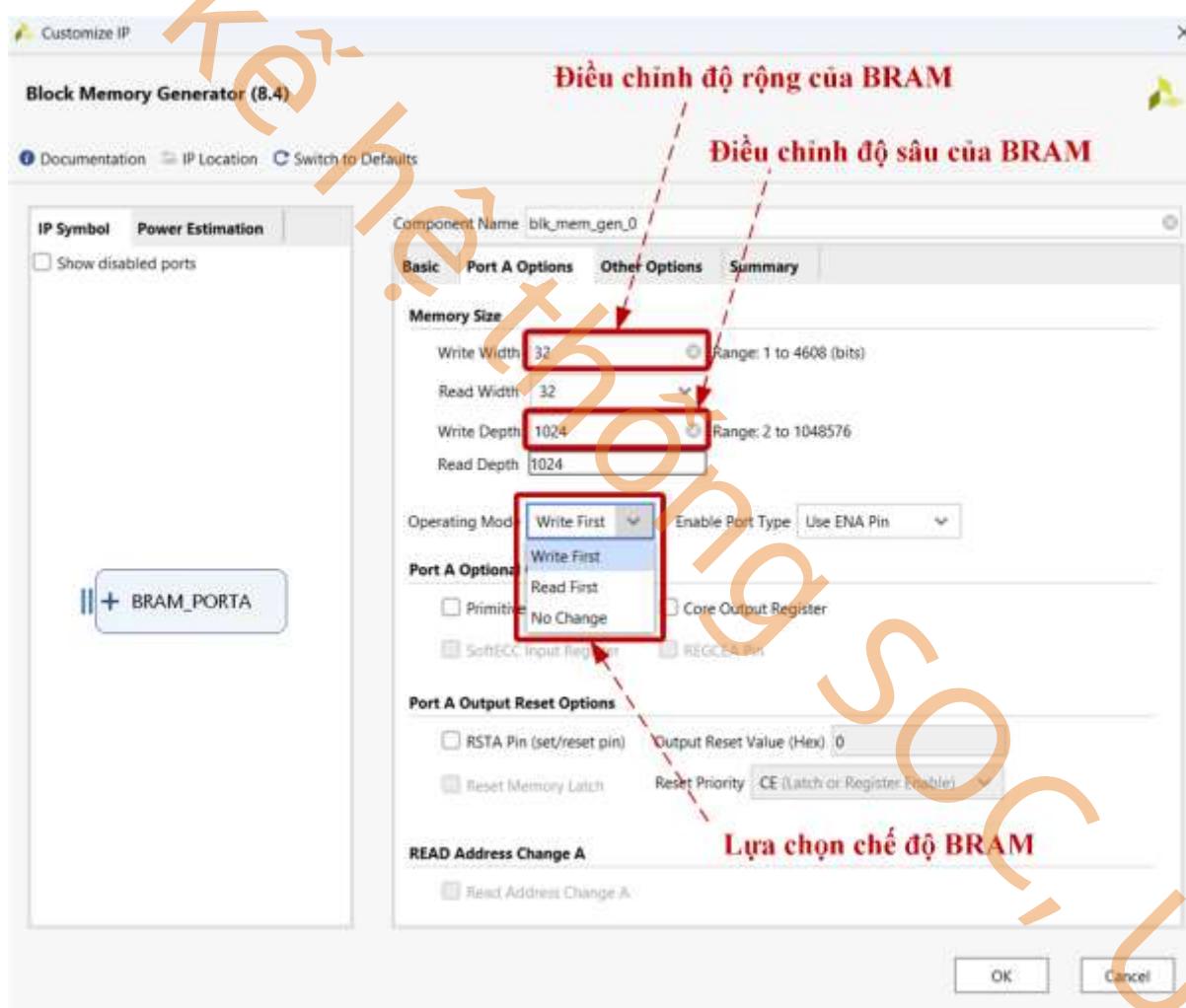
Sau khi chọn Block Memory Generator trong IP Catalog, sẽ có giao diện như hình 3.16, nơi người thiết kế có thể đặt tên cho mô-đun BRAM và lựa chọn loại bộ nhớ phù hợp.

Trong phần Memory Type, có 5 loại bộ nhớ để lựa chọn: Single Port RAM, bộ nhớ chỉ có một cổng đọc và ghi, không thể thực hiện đồng thời; Simple Dual Port RAM, với hai cổng nhưng không thể đọc và ghi đồng thời; True Dual Port RAM, cho phép đọc và ghi đồng thời từ hai cổng độc lập, phù hợp với các ứng dụng yêu cầu băng thông cao; Single Port ROM, bộ nhớ chỉ có một cổng đọc và không có khả năng ghi, thích hợp cho dữ liệu tĩnh như bảng tra cứu hoặc mã lệnh cố định; và Dual Port ROM, bộ nhớ chỉ có chức năng đọc đồng thời từ hai cổng, hữu ích trong các ứng dụng cần truy xuất dữ liệu tĩnh đồng thời từ nhiều nguồn. Điều đáng lưu ý là các loại RAM có thể thực hiện cả đọc và ghi, trong khi các loại ROM chỉ có thể thực hiện đọc. Các loại bộ nhớ này có số lượng cổng và đặc tính khác nhau, vì vậy cần được lựa chọn phù hợp với yêu cầu lưu trữ của thuật toán hoặc mô-đun.



Hình 3.16: Giao diện cấu hình mô-đun Block Memory Generator trong Vivado, nơi người thiết kế có thể đặt tên cho mô-đun BRAM và lựa chọn loại bộ nhớ thích hợp.

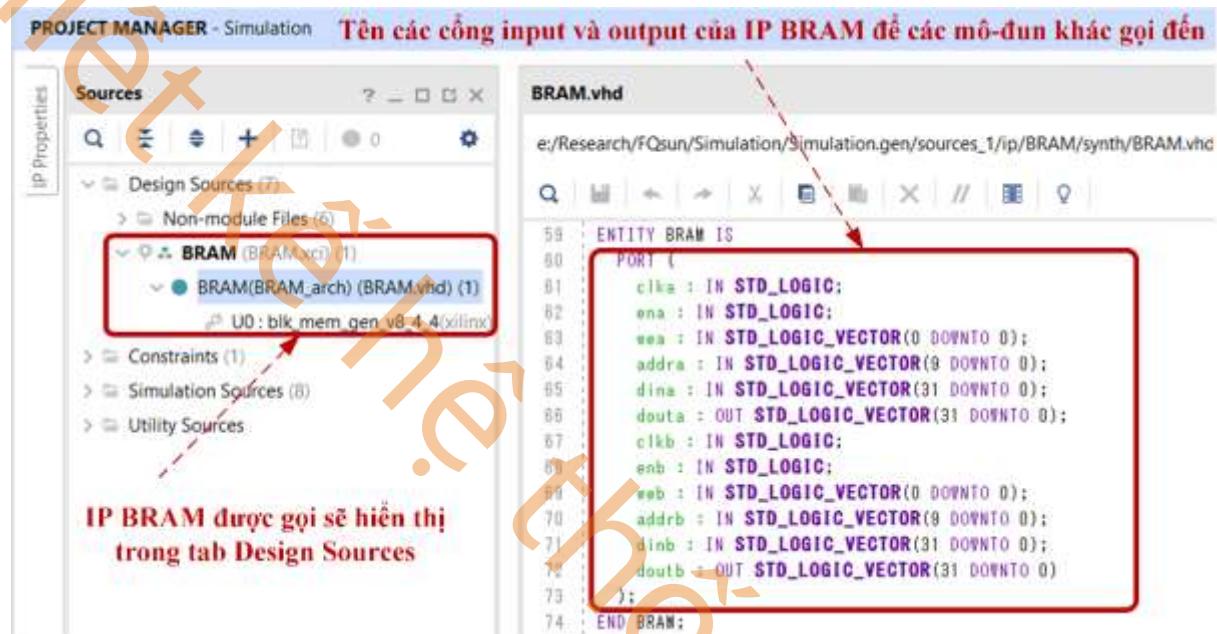
Ở tab Port Options, người thiết kế có thể điều chỉnh cấu hình chính cho các cổng của BRAM, bao gồm độ rộng, độ sâu bộ nhớ và chế độ hoạt động như WRITE\_FIRST, READ\_FIRST hoặc NO\_CHANGE. Hình 3.17 minh họa giao diện cấu hình mô-đun Block Memory Generator trong Vivado, nơi người thiết kế có thể điều chỉnh độ rộng và độ sâu của bộ nhớ BRAM và chọn chế độ hoạt động phù hợp. Để tối ưu hóa số lượng BRAM, độ rộng (width) và độ sâu (depth) của bộ nhớ nên được cấu hình sao cho là bội số của 16,384, ví dụ như width 8 và depth 2048, width 16 và depth 1024, width 32 và depth 512, vì Vivado tính toán số lượng BRAM dựa trên bội số này, với mỗi bội số tương ứng với 0.5 BRAM.



Hình 3.17: Giao diện cấu hình mô-đun Block Memory Generator trong Vivado, nơi người thiết kế điều chỉnh độ rộng, độ sâu bộ nhớ BRAM và chọn chế độ hoạt động.

Sau khi hoàn tất gọi và cấu hình cho IP BRAM, mô-đun này sẽ hiển thị trong tab Design Sources. Hình 3.18 minh họa giao diện hiển thị các cổng input và output của

BRAM, bao gồm các tín hiệu như clka, ena, addra, dina, douta (cho cổng A) và clkb, enb, addrb, dinb, doutb (cho cổng B), được khai báo trong mã VHDL của mô-đun này. Các tên cổng input và output này tương tự như những gì được sử dụng trong Hình 3.13, nơi mô tả code Verilog cho Dual-Port BRAM với hai cổng đọc và ghi, sử dụng bộ nhớ BRAM trong thiết kế IP.

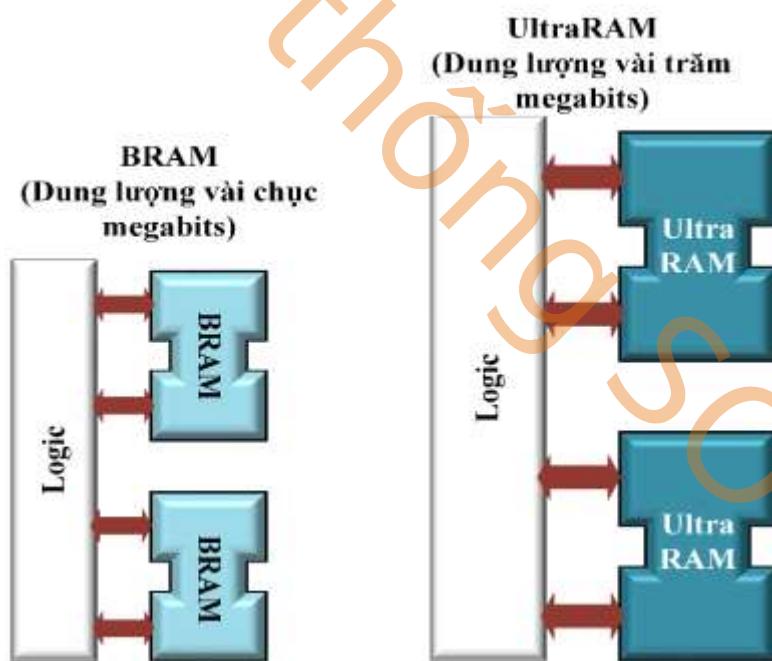


Hình 3.18: Giao diện hiển thị các cổng input và output của IP BRAM trong tab Design Sources, nơi mô-đun BRAM được gọi để các mô-đun khác có thể kết nối và sử dụng.

Nhìn chung, BRAM trong FPGA cung cấp một phương pháp lưu trữ dữ liệu hiệu quả và linh hoạt, đáp ứng tốt các yêu cầu về hiệu suất và độ trễ thấp trong nhiều ứng dụng khác nhau. Với các chế độ hoạt động đa dạng và khả năng cấu hình linh hoạt, BRAM là một thành phần không thể thiếu trong các thiết kế FPGA hiện đại. Việc lựa chọn và sử dụng BRAM phù hợp có thể tối ưu hóa hiệu suất của hệ thống, đặc biệt trong các ứng dụng đòi hỏi tốc độ cao và xử lý dữ liệu phức tạp. Có hai cách gọi BRAM để sử dụng: một là sử dụng mô-đun IP có sẵn trong Vivado, và hai là gọi BRAM thông qua code Verilog. Cách gọi BRAM bằng code Verilog sẽ có lợi thế hơn nếu thiết kế IP muốn triển khai trên nhiều FPGA khác nhau, vì sẽ không tốn thời gian gọi lại IP BRAM, giúp tiết kiệm tài nguyên và tăng tính linh hoạt trong việc triển khai.

### 3.2.4. Ultra RAM

Sự khác biệt chính giữa BRAM và UltraRAM (URAM) là dung lượng bộ nhớ và khả năng tối ưu hóa hiệu suất. URAM có dung lượng lớn hơn nhiều so với BRAM, và có thể cung cấp băng thông cao hơn với độ trễ thấp. Hình 3.19 minh họa sự khác biệt về dung lượng giữa BRAM và URAM, cho thấy URAM cung cấp dung lượng bộ nhớ lớn hơn nhiều, phù hợp với các yêu cầu bộ nhớ cao trong thiết kế FPGA. BRAM có thể hỗ trợ các chế độ độc lập cho các cổng như chế độ đọc-ghi đồng thời (TDP) và chế độ đọc-ghi tuần tự (SDP), trong khi URAM không hỗ trợ các chế độ này, mà chỉ hỗ trợ một thao tác đọc hoặc ghi tại mỗi chu kỳ. URAM không có khả năng điều chỉnh chế độ đọc (read-first, write-first) như BRAM và không hỗ trợ các bộ điều khiển đầu vào/ra động (dynamic cascade). URAM có khả năng tiết kiệm năng lượng tốt hơn với tính năng "auto sleep", tự động chuyển sang chế độ ngủ khi không có hoạt động, giúp tiết kiệm năng lượng mà không làm hỏng dữ liệu. URAM cũng không gặp phải vấn đề va chạm địa chỉ (address collision) như BRAM, giúp việc xử lý dữ liệu trở nên ổn định hơn.



Hình 3.19: So sánh dung lượng giữa BRAM và URAM trong FPGA.

URAM chỉ được hỗ trợ ở các FPGA đời mới, đặc biệt là các dòng Virtex UltraScale+, Kintex UltraScale+, Alveo, và Versal của Xilinx. Ví dụ, các FPGA như Virtex UltraScale+ VU9P, Kintex UltraScale+ KU115, và Alveo U280 đều hỗ trợ URAM. Trong khi đó, các

FPGA đại trà như Zynq UltraScale+ MPSoC lại không hỗ trợ URAM mà thay vào đó sử dụng BRAM. Do đó, phần này sẽ phân tích ngắn gọn về URAM và cách chúng được áp dụng trong các thiết kế yêu cầu bộ nhớ lớn.

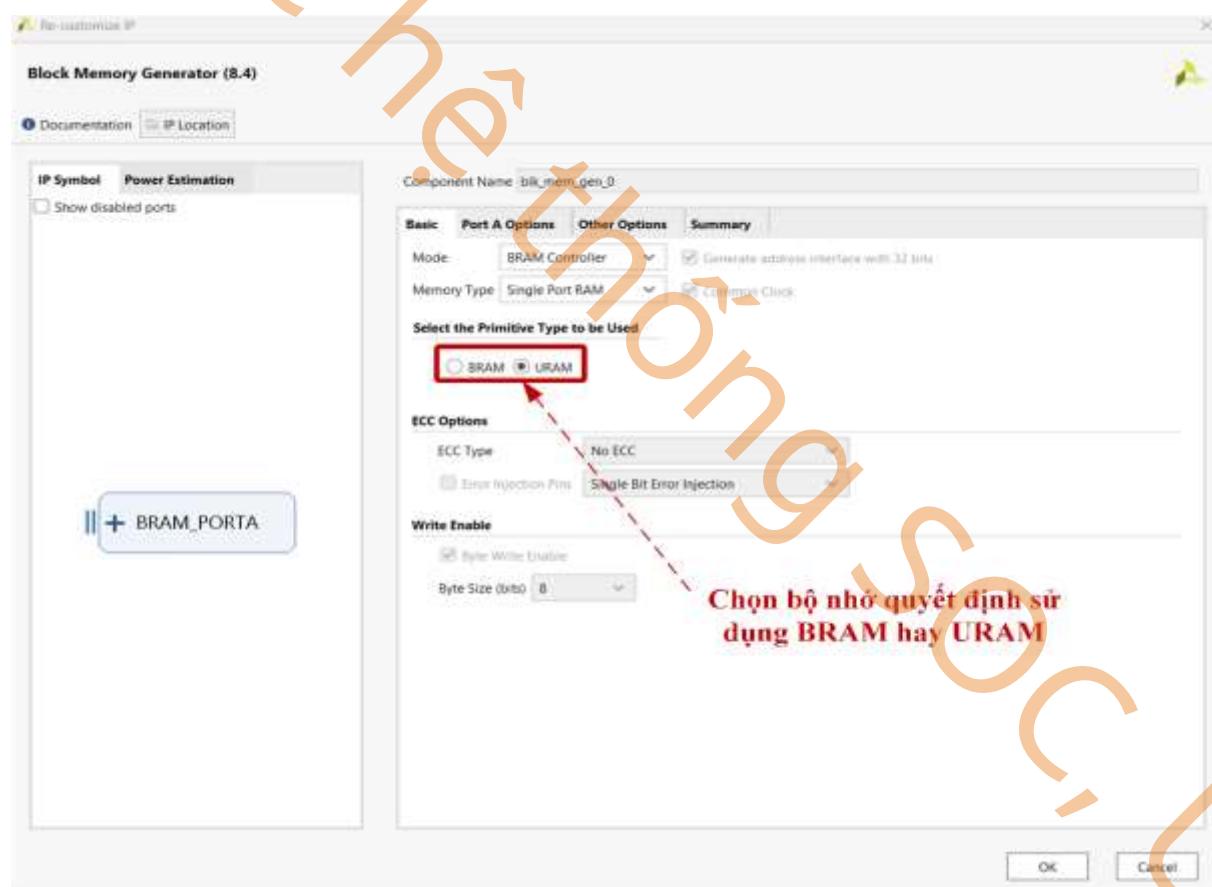
Tương tự như BRAM, URAM cũng có hai phương thức sử dụng chính trong thiết kế IP. Phương pháp đầu tiên là viết mã Verilog để thiết kế mô-đun URAM, trong đó người thiết kế khai báo bộ nhớ URAM trực tiếp trong code và thực hiện truy xuất bộ nhớ thông qua các tín hiệu điều khiển như địa chỉ, dữ liệu và tín hiệu ghi/đọc. Phương pháp thứ hai là sử dụng mô-đun URAM có sẵn trong các công cụ thiết kế FPGA như Vivado, giúp người thiết kế dễ dàng tích hợp URAM vào hệ thống mà không cần phải khai báo chi tiết trong mã nguồn.

```
module URAM #(
    parameter AWIDTH = 12, // độ rộng địa chỉ
    parameter DWIDTH = 72 // độ rộng dữ liệu
)(  
    input clk, // clock
    input ena, // Tín hiệu cho phép đọc cổng A
    input wea, // Tín hiệu cho phép ghi cổng A
    input [AWIDTH-1:0] addra, // Địa chỉ cổng A
    input [DWIDTH-1:0] dina, // Dữ liệu cổng A
    output reg [DWIDTH-1:0] douta // Dữ liệu đầu ra cổng A
);  
  
    // Khai báo bộ nhớ URAM với kiểu "ultra",
    // sử dụng kích thước bộ nhớ theo độ rộng địa chỉ AWIDTH và độ rộng dữ liệu DWIDTH.  
    (* ram_style = "ultra" *) reg [DWIDTH-1:0] mem [2**AWIDTH-1:0];  
  
    // Việc ghi và đọc dữ liệu từ bộ nhớ sẽ được thực hiện với độ trễ là một chu kỳ xung nhịp  
    always @ (posedge clk) begin  
        if (ena) begin  
            if (wea) begin  
                mem[addra] <= dina; // Ghi dữ liệu vào bộ nhớ tại địa chỉ addra  
            end  
            douta <= mem[addra]; // Đọc dữ liệu từ bộ nhớ tại địa chỉ addra  
        end  
    end  
  
endmodule
```

Hình 3.20: Code Verilog mô tả URAM với một cổng đọc và ghi trong thiết kế IP.

Hình 3.20 minh họa mã Verilog mô tả URAM với một cổng đọc và ghi trong thiết kế IP. Mô-đun này sử dụng một cổng A để thực hiện các thao tác đọc và ghi dữ liệu vào bộ nhớ URAM. Bộ nhớ URAM được khai báo với thuộc tính (`* ram_style = "ultra" *`), với

độ rộng địa chỉ AWIDTH và độ rộng dữ liệu DWIDTH được xác định qua các tham số. Để sử dụng bộ nhớ URAM một cách hiệu quả, độ rộng dữ liệu DWIDTH phải là 72 và độ rộng địa chỉ AWIDTH ít nhất phải là 12, vì điều này sẽ đảm bảo rằng mỗi URAM được tiêu thụ và sử dụng tối ưu. Khi tín hiệu ena (enable) được kích hoạt, và nếu tín hiệu wea (write enable) cũng được kích hoạt, dữ liệu từ dina sẽ được ghi vào bộ nhớ tại địa chỉ addra. Nếu tín hiệu wea không kích hoạt, dữ liệu tại địa chỉ addra sẽ được đọc và xuất ra douta. Mô-đun này là một ví dụ đơn giản về cách sử dụng URAM với một cổng đọc và ghi trong thiết kế IP. Tương tự, đối với thiết kế hai cổng, quá trình sử dụng URAM sẽ tương tự nhưng với sự bổ sung của cổng thứ hai, cho phép thao tác đọc và ghi độc lập tại một địa chỉ khác trong bộ nhớ.



Hình 3.21: Giao diện cấu hình mô-đun Block Memory Generator trong Vivado, nơi người thiết kế điều chỉnh bộ nhớ sử dụng BRAM hay URAM.

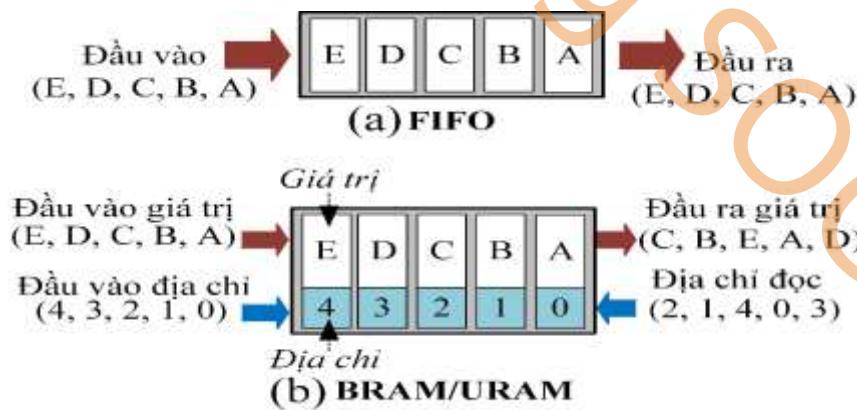
Cách thứ hai để sử dụng bộ nhớ trong thiết kế FPGA là gọi mô-đun IP có sẵn trong các phần mềm thiết kế phần cứng trên FPGA, ví dụ phần mềm Vivado cho các bo FPGA

dòng Xilinx. Hình 3.21 minh họa giao diện mô-đun Block Memory Generator trong Vivado, nơi người thiết kế có thể cấu hình bộ nhớ và chọn giữa BRAM và URAM.

Nhìn chung, URAM mặc dù có lợi thế về dung lượng lớn hơn BRAM, nhưng lại bị giới hạn bởi yêu cầu độ rộng dữ liệu phải là 72 bit và độ rộng địa chỉ phải từ  $2^{12}$  (hoặc 12 bit) trở lên để tối ưu hiệu suất sử dụng. URAM chỉ được hỗ trợ trên các FPGA tiên tiến, và không phải tất cả các dòng FPGA đều hỗ trợ loại bộ nhớ này. Tương tự như BRAM, URAM cũng có hai cách gọi: một là sử dụng mô-đun IP có sẵn trong Vivado, và hai là gọi URAM thông qua code Verilog. Việc lựa chọn URAM và BRAM tùy thuộc vào độ lớn của dữ liệu cần lưu trữ, với URAM phù hợp cho các ứng dụng yêu cầu bộ nhớ lớn hơn, trong khi BRAM thích hợp cho các ứng dụng có yêu cầu về hiệu suất cao và độ trễ thấp với dung lượng bộ nhớ nhỏ hơn.

### 3.2.5. FIFO

FIFO (First In, First Out) là một cấu trúc bộ nhớ được sử dụng để lưu trữ dữ liệu theo nguyên tắc "vào trước, ra trước". Điều này có nghĩa là dữ liệu được ghi vào bộ nhớ FIFO sẽ được đọc ra theo thứ tự mà chúng được ghi vào. FIFO thường được sử dụng trong các hệ thống cần đồng bộ hóa giữa các tác vụ hoặc xử lý dữ liệu theo một luồng liên tục, chẳng hạn như trong các bộ đệm giữa các hệ thống không đồng bộ hoặc khi cần truyền tải dữ liệu giữa các mô-đun với tốc độ khác nhau.



Hình 3.22: So sánh cách lưu trữ và truy xuất dữ liệu giữa (a) FIFO và (b) BRAM/URAM.

Hình 3.22 (a) minh họa cách thức hoạt động của FIFO, với dữ liệu vào và ra theo thứ tự xác định mà không thay đổi vị trí trong bộ nhớ. Dữ liệu được ghi vào FIFO theo thứ tự

"vào trước, ra trước", đảm bảo dữ liệu được xử lý theo đúng trình tự. FIFO thường được sử dụng trong các ứng dụng yêu cầu xử lý dữ liệu theo chuỗi, như trong các bộ đệm tạm thời, để duy trì tính liên tục trong việc xử lý các luồng dữ liệu. Một đặc điểm quan trọng của FIFO là dữ liệu ra luôn theo thứ tự ghi vào mà không thay đổi vị trí trong bộ nhớ. Ngược lại, Hình 3.22 (b) mô tả BRAM/URAM (Block RAM/Ultra RAM), loại bộ nhớ có khả năng truy xuất dữ liệu ngẫu nhiên thông qua địa chỉ. Đây là bộ nhớ linh hoạt hơn, vì dữ liệu có thể được ghi vào và đọc ra từ bất kỳ địa chỉ nào trong bộ nhớ. Việc này giúp tối ưu hóa các ứng dụng cần truy xuất dữ liệu không theo thứ tự và cần linh hoạt trong việc quản lý bộ nhớ, chẳng hạn như trong các ứng dụng xử lý tín hiệu số hoặc các tác vụ yêu cầu truy cập vào nhiều địa chỉ bộ nhớ trong thời gian ngắn. So với BRAM/URAM, FIFO có những lợi và hại sau:

#### Lợi thế của FIFO:

- ✓ **Quản lý thứ tự dữ liệu dễ dàng:** FIFO xử lý dữ liệu theo thứ tự "vào trước, ra trước", rất hiệu quả cho các ứng dụng yêu cầu xử lý dữ liệu theo chuỗi, chẳng hạn như bộ đệm giữa các hệ thống không đồng bộ hoặc trong các hệ thống truyền tải dữ liệu với tốc độ khác nhau. Bên cạnh đó, việc không cần quản lý địa chỉ cũng giúp đơn giản hóa thiết kế và tiết kiệm tài nguyên phần cứng. BRAM/URAM, ngược lại, yêu cầu quản lý địa chỉ đọc và ghi, điều này làm cho thiết kế phức tạp hơn và có thể tốn thêm chi phí cho việc triển khai các mạch điều khiển địa chỉ phù hợp với yêu cầu của thuật toán.
- ✓ **Đơn giản và hiệu quả diện tích:** FIFO có cấu trúc đơn giản, giúp tiết kiệm tài nguyên bộ nhớ khi chỉ cần lưu trữ một lượng dữ liệu nhỏ với độ trễ thấp.
- ✓ **Quản lý bộ đệm hiệu quả:** FIFO thường được sử dụng làm bộ đệm giữa các hệ thống không đồng bộ, đặc biệt trong các ứng dụng yêu cầu điều phối dữ liệu từ các nguồn khác nhau với tốc độ xử lý khác nhau. FIFO giúp tránh tình trạng tràn bộ đệm hoặc mất mát dữ liệu do tốc độ truyền tải không đồng đều, điều mà URAM không thể đảm bảo một cách trực tiếp.

#### Hạn chế của FIFO:

- ✓ **Truy xuất dữ liệu ngẫu nhiên không linh hoạt:** FIFO không hỗ trợ truy xuất dữ liệu ngẫu nhiên, điều này hạn chế khả năng sử dụng khi cần truy cập hoặc thay đổi dữ liệu ở các vị trí khác nhau trong bộ nhớ.
- ✓ **Dung lượng bộ nhớ hạn chế:** FIFO thường chỉ có dung lượng bộ nhớ nhỏ hơn so với BRAM/URAM, và không thích hợp cho các ứng dụng yêu cầu bộ nhớ với dung lượng lớn hoặc khả năng truy cập đồng thời nhiều dữ liệu.
- ✓ **Không hỗ trợ nhiều cổng đồng thời:** FIFO chỉ hỗ trợ một cổng đọc và một cổng ghi, điều này không thuận lợi cho các ứng dụng yêu cầu truy cập đồng thời vào bộ nhớ.

```

module FIFO #(parameter AWIDTH = 10, DWIDTH = 32) (
    input clk, srst, // Tín hiệu clock và reset
    input wr_en, rd_en, // Tín hiệu ghi và đọc
    input [DWIDTH-1:0] din, // Dữ liệu đầu vào
    output reg [DWIDTH-1:0] dout, // Dữ liệu đầu ra
    output full, empty, // Trạng thái đầy và trống
    output wr_rst_busy, rd_rst_busy // Trạng thái reset của ghi và đọc
);
    // Con trỏ đọc và viết, số bit của con trỏ là AWIDTH
    reg [AWIDTH-1:0] w_ptr, r_ptr; // sử dụng AWIDTH để xác định số bit của con trỏ
    // FIFO sử dụng mảng 1 chiều với số phần tử DEPTH = 2^AWIDTH (với AWIDTH = 10)
    reg [DWIDTH-1:0] fifo[(1 << AWIDTH) - 1:0];

    always @ (posedge clk or negedge srst) begin
        if (!srst) begin
            w_ptr <= 0;
            r_ptr <= 0;
            dout <= 0;
        end else begin
            // Ghi dữ liệu vào FIFO
            if (wr_en && !full) begin
                fifo[w_ptr] <= din;
                w_ptr <= w_ptr + 1;
            end

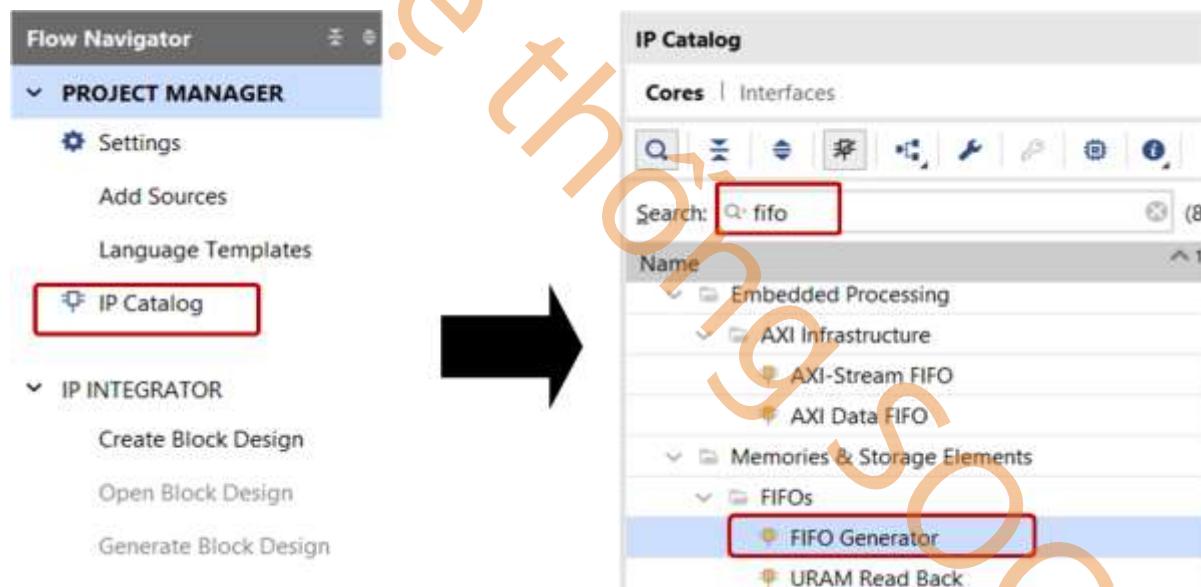
            // Đọc dữ liệu từ FIFO
            if (rd_en && !empty) begin
                dout <= fifo[r_ptr];
                r_ptr <= r_ptr + 1;
            end
        end
    end
    // Tính toán trạng thái đầy và trống
    assign full = ((w_ptr + 1) == r_ptr); // FIFO đầy khi chỉ số ghi kế tiếp trùng với chỉ số đọc
    assign empty = (w_ptr == r_ptr); // FIFO trống khi con trỏ ghi và con trỏ đọc bằng nhau
    // Tính toán trạng thái reset của ghi và đọc
    assign wr_rst_busy = (w_ptr != r_ptr); // Kiểm tra nếu FIFO đang bịt ghi
    assign rd_rst_busy = (r_ptr != w_ptr); // Kiểm tra nếu FIFO đang bịt đọc
endmodule

```

Hình 3.23: Mã Verilog mô tả FIFO với độ rộng địa chỉ 10 bit và dữ liệu 32 bit.

Giống như BRAM và URAM, FIFO có hai phương thức sử dụng trong thiết kế IP. Phương pháp đầu tiên là viết mã Verilog để thiết kế FIFO, khai báo bộ nhớ và truy xuất qua các tín hiệu điều khiển. Phương pháp thứ hai là sử dụng mô-đun FIFO có sẵn trong các công cụ thiết kế FPGA như Vivado, giúp tích hợp FIFO dễ dàng mà không cần khai báo chi tiết trong mã nguồn.

Hình 3.23 minh họa mã Verilog mô tả FIFO với độ rộng địa chỉ 10 bit và dữ liệu 32 bit. Trong mô-đun này, các tham số như độ rộng địa chỉ (AWIDTH) và độ rộng dữ liệu (DWIDTH) có thể được điều chỉnh tùy thuộc vào yêu cầu của thuật toán hoặc ứng dụng cụ thể. FIFO được thiết kế để xử lý dữ liệu theo thứ tự vào ra, với con trỏ đọc và ghi để đảm bảo các thao tác đọc/ghi chỉ diễn ra khi bộ nhớ không đầy hoặc không trống. Trạng thái đầy và trống của FIFO được tính toán dựa trên con trỏ đọc và ghi, giúp tối ưu hóa việc quản lý bộ nhớ.

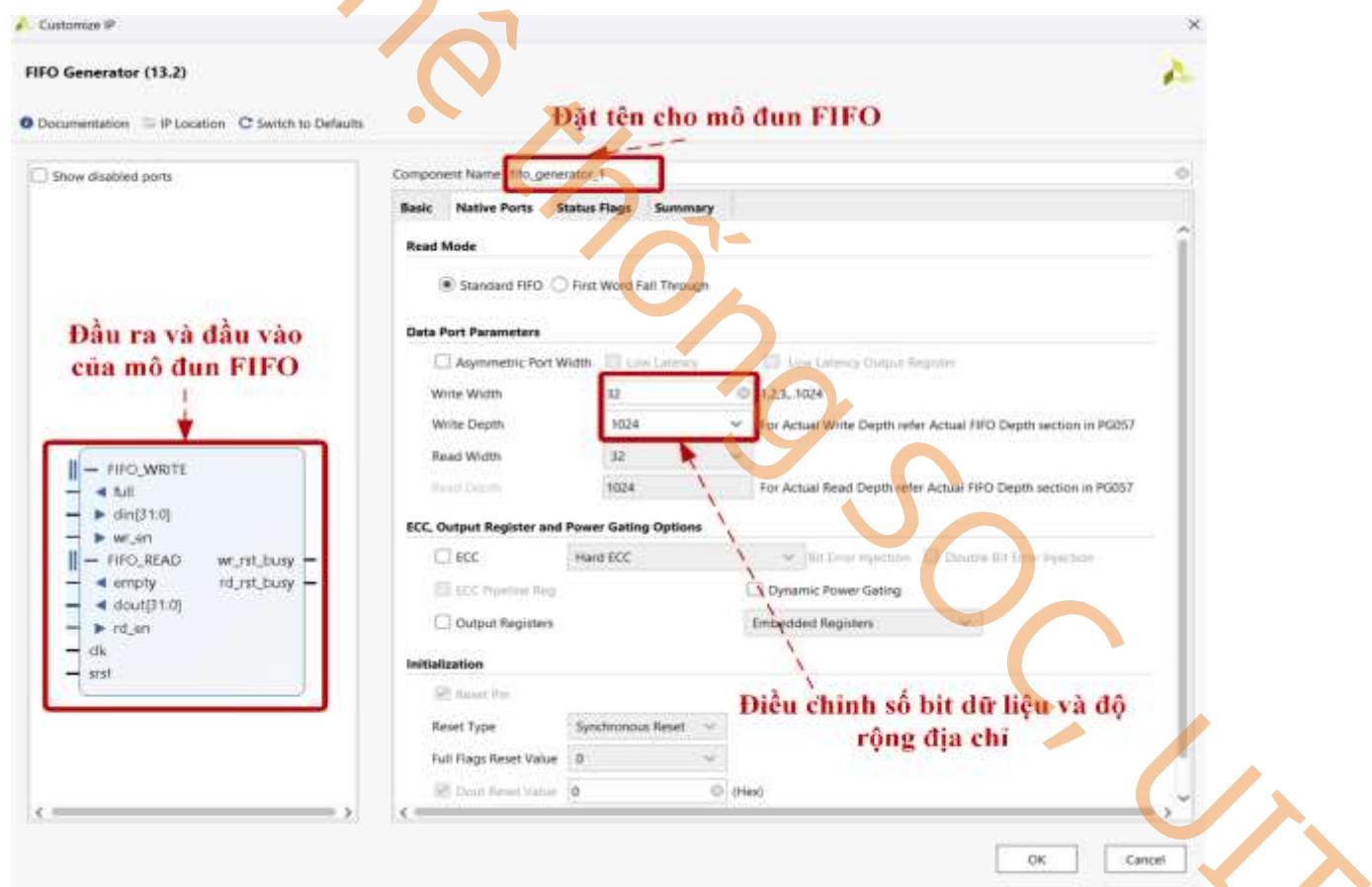


Hình 3.24: Giao diện IP Catalog trong Vivado, nơi tìm kiếm và chọn mô-đun FIFO Generator để tạo và cấu hình FIFO trong thiết kế IP ở FPGA.

Cách thứ hai để sử dụng FIFO trong thiết kế FPGA là gọi mô-đun IP có sẵn trong các phần mềm thiết kế phần cứng trên FPGA, ví dụ phần mềm Vivado cho các bo FPGA dòng Xilinx. Khi sử dụng phương pháp này, người thiết kế không cần phải viết mã từ đầu mà có thể tận dụng các mô-đun FIFO đã được tối ưu hóa sẵn. Trong Vivado, người thiết kế có thể mở IP Catalog, tìm kiếm và chọn mô-đun FIFO Generator, sau đó cấu hình các tham số

của FIFO như kích thước bộ đệm, chế độ đọc và ghi, và số cổng cần thiết cho ứng dụng cụ thể. Hình 3.24 minh họa quá trình này, cho thấy cách thức tìm kiếm và chọn mô-đun FIFO Generator trong IP Catalog để sử dụng trong thiết kế phần cứng trên FPGA.

Sau khi chọn FIFO Generator trong IP Catalog, sẽ có giao diện như hình 3.25, nơi người thiết kế có thể đặt tên cho mô-đun FIFO và cấu hình các tham số phù hợp với yêu cầu ứng dụng. Trong phần Data Port Parameters, người thiết kế có thể lựa chọn các tham số quan trọng như Write Width (độ rộng ghi), Write Depth (độ sâu ghi), Read Width (độ rộng đọc), và Read Depth (độ sâu đọc). Đặc biệt, có thể điều chỉnh độ rộng dữ liệu và độ sâu bộ đệm của FIFO để phù hợp với băng thông và yêu cầu về hiệu suất của hệ thống. Trong giao diện cấu hình này, người thiết kế cũng có thể chọn các chế độ đọc/ghi cho FIFO như Low Latency để tối ưu hóa hiệu suất hệ thống.



Hình 3.25: Giao diện cấu hình mô-đun FIFO Generator trong Vivado, nơi người thiết kế điều chỉnh độ rộng, độ sâu bộ nhớ FIFO và chọn các tham số liên quan đến chế độ đọc/ghi.

Nhìn chung, FIFO trong FPGA cung cấp một phương pháp lưu trữ dữ liệu đơn giản nhưng hiệu quả, đặc biệt trong các ứng dụng yêu cầu xử lý dữ liệu theo nguyên tắc "vào trước, ra trước". FIFO giúp đồng bộ hóa các hệ thống hoặc tác vụ với tốc độ khác nhau, đồng thời đảm bảo không mất mát dữ liệu và tránh tình trạng tràn bộ đệm. Mặc dù có cơ chế hoạt động khác với các loại bộ nhớ như BRAM và URAM, FIFO vẫn có thể được coi là một dạng bộ nhớ và tiêu thụ tài nguyên BRAM trên FPGA. Với khả năng cấu hình linh hoạt và các chế độ đọc/ghi tùy chỉnh, FIFO là một thành phần quan trọng trong các thiết kế FPGA hiện đại, đặc biệt là trong các ứng dụng truyền tải dữ liệu hoặc xử lý tín hiệu. Giống như BRAM/URAM, FIFO trong thiết kế FPGA có hai cách gọi: một là sử dụng môđun IP có sẵn trong Vivado, và hai là gọi FIFO thông qua code Verilog.

### 3.3. Bộ nhớ ngoài chip (Off-chip Memory)

#### 3.3.1. Giới thiệu

Trong thiết kế SoC trên FPGA, việc lựa chọn và sử dụng các bộ nhớ ngoài chip là yếu tố quan trọng để tối ưu hóa hiệu suất hệ thống. Thay vì thiết kế bộ nhớ từ đầu, các hệ thống SoC trên FPGA thường tận dụng các bộ nhớ có sẵn như DDR, NAND Flash, hoặc SRAM để đáp ứng yêu cầu về dung lượng, tốc độ truy cập và chi phí. Hiểu rõ đặc tính về dung lượng và tốc độ truy cập của các loại bộ nhớ này giúp các nhà thiết kế chọn lựa bộ nhớ phù hợp, tối ưu hóa hiệu suất hệ thống và giảm thiểu độ trễ, đồng thời tiết kiệm tài nguyên.



Hình 3.26: Hệ thống phân cấp bộ nhớ ngoài chip dựa trên kích thước, tốc độ, và giá.

Hình 3.26 minh họa hệ thống phân cấp các bộ nhớ ngoài chip dựa trên kích thước, tốc độ, và giá thành. Các loại bộ nhớ được sử dụng trong hệ thống máy tính và thiết bị nhúng có sự khác biệt rõ rệt về tốc độ truy cập, dung lượng và chi phí. Bộ nhớ cache L1, L2, L3 (SRAM) cung cấp tốc độ truy cập nhanh nhất nhưng có dung lượng nhỏ và chi phí cao. Các bộ nhớ như eDRAM và ROM có dung lượng lớn hơn cache và tốc độ truy cập ở mức trung bình, với chi phí giảm dần. DRAM là bộ nhớ chính, cung cấp dung lượng lớn hơn và chi phí hợp lý, nhưng tốc độ truy cập chậm hơn so với SRAM và eDRAM. NAND Flash và HDD cung cấp dung lượng lưu trữ lớn nhất với chi phí thấp, nhưng tốc độ truy cập chậm nhất, thích hợp cho lưu trữ dữ liệu dài hạn và các ứng dụng ít yêu cầu về tốc độ.

**Bảng 3.2: So sánh giữa các loại bộ nhớ khác nhau**

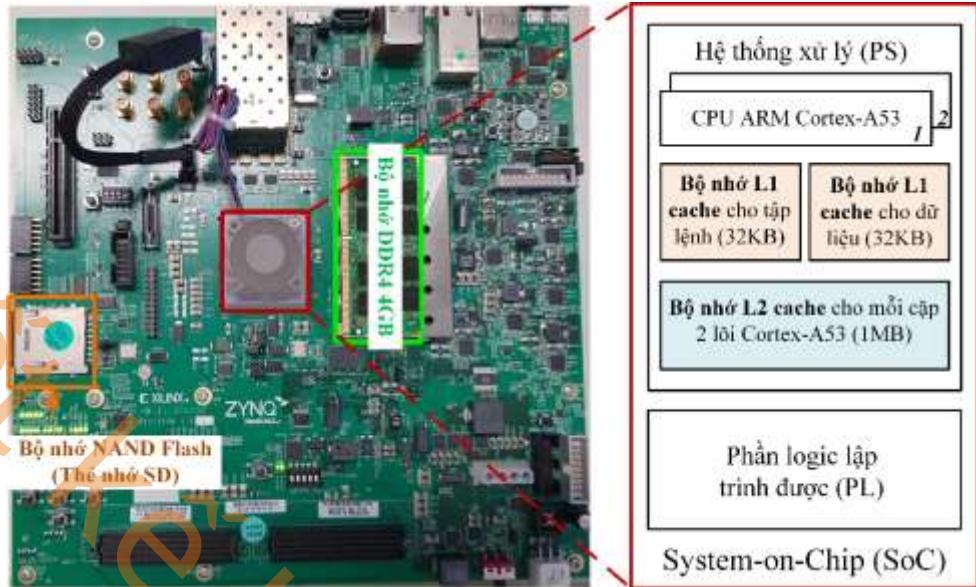
Loại bộ nhớ	Thời gian truy cập (chu kỳ)	Dung lượng	Quản lý bởi
L1 Cache (SRAM)	3~4	64KB	Phần cứng
L2 Cache (SRAM)	10~30	256KB	Phần cứng
L3 Cache (SRAM)	30~60	2~8MB	Phần cứng
eDRAM	50~100	64~256MB	Phần cứng
ROM	100~500	1~16MB	Phần cứng
DRAM	100~300	512MB~32GB	Hệ điều hành
NAND Flash	5K~10K	8GB~32GB	Hệ điều hành
HDD	10M~20M	>1TB	Hệ điều hành

Bảng 3.2 so sánh các loại bộ nhớ khác nhau dựa trên ba tiêu chí chính: thời gian truy cập, dung lượng, và quản lý bởi phần cứng hoặc phần mềm. Các loại bộ nhớ này được sử dụng trong các hệ thống máy tính và thiết bị nhúng, mỗi loại có ưu và nhược điểm riêng, phù hợp cho các ứng dụng và mục đích khác nhau như sau:

- ✓ **L1, L2, L3 Cache (SRAM):** L1 Cache có thời gian truy cập nhanh nhất, từ 3 đến 4 chu kỳ, với dung lượng nhỏ, khoảng 64KB, giúp tăng tốc độ truy cập dữ liệu cho CPU. L2 Cache (256KB) và L3 Cache (2~8MB) có dung lượng lớn hơn nhưng tốc độ truy cập chậm hơn (từ 10 đến 60 chu kỳ). Các bộ nhớ cache này được sử dụng

để lưu trữ tạm thời các dữ liệu và lệnh thường xuyên truy cập, giúp giảm độ trễ và cải thiện hiệu suất hệ thống.

- ✓ **eDRAM (Embedded DRAM)**: eDRAM có thời gian truy cập từ 50 đến 100 chu kỳ và dung lượng từ 64MB đến 256MB. Nó thường được sử dụng làm bộ nhớ cache tốc độ cao hoặc bộ nhớ tạm thời trong các ứng dụng cần dung lượng lớn hơn cache nhưng vẫn đòi hỏi tốc độ truy cập nhanh. eDRAM cung cấp khả năng truy cập nhanh hơn so với DRAM tiêu chuẩn và được dùng trong các bộ nhớ cache lớn hoặc bộ nhớ chính trong một số hệ thống nhúng.
- ✓ **ROM (Read-Only Memory)**: ROM có thời gian truy cập từ 100 đến 500 chu kỳ và dung lượng từ 1MB đến 16MB. ROM được sử dụng để lưu trữ các dữ liệu cố định, như firmware, BIOS, hoặc mã khởi động, vì dữ liệu trong ROM không thể bị thay đổi sau khi sản xuất. Nó thích hợp cho việc lưu trữ những thông tin cần được bảo vệ và không thay đổi trong suốt vòng đời sản phẩm.
- ✓ **DRAM (Dynamic RAM)**: DRAM có thời gian truy cập từ 100 đến 300 chu kỳ và dung lượng từ 512MB đến 32GB. Đây là bộ nhớ chính của hệ thống, được quản lý bởi hệ điều hành và được sử dụng để lưu trữ các dữ liệu và chương trình đang chạy. DRAM cung cấp dung lượng lớn với chi phí hợp lý, phù hợp cho các ứng dụng cần lưu trữ nhiều dữ liệu và cần truy cập thường xuyên.
- ✓ **NAND Flash**: NAND Flash có thời gian truy cập từ 5.000 đến 10.000 chu kỳ và dung lượng từ 8GB đến 32GB. Đây là bộ nhớ không bay hơi, thường được sử dụng trong các thiết bị lưu trữ như USB, SSD, và các thiết bị nhúng. NAND Flash có chi phí thấp hơn so với DRAM và thích hợp cho việc lưu trữ dữ liệu dài hạn và dữ liệu có tính di động cao, dù tốc độ truy cập chậm hơn.
- ✓ **HDD (Hard Disk Drive)**: HDD có thời gian truy cập chậm nhất, từ 10 triệu đến 20 triệu chu kỳ, nhưng dung lượng rất lớn, thường lớn hơn 1TB (Terabyte). HDD là lựa chọn phổ biến cho việc lưu trữ dữ liệu dài hạn với chi phí thấp, phù hợp cho các ứng dụng lưu trữ dữ liệu lớn như cơ sở dữ liệu, lưu trữ dữ liệu sao lưu, và các thư viện số.



Hình 3.27: Các bộ nhớ ngoài chip trên FPGA ZCU102, bao gồm bộ nhớ DDR4 4GB, bộ nhớ NAND Flash, và các bộ nhớ cache L1, L2 của CPU ARM Cortex-A53 trong hệ thống SoC.

Bộ nhớ ngoài chip được sử dụng với nhiều loại và chức năng khác nhau, tùy thuộc vào yêu cầu của hệ thống. Trong hình 3.27, bộ nhớ NAND Flash (thẻ nhớ SD) được sử dụng để lưu trữ hệ điều hành Petalinux, giúp hệ thống khởi động và chạy các ứng dụng nhúng. Bộ nhớ này có dung lượng cao và chi phí thấp, phù hợp cho việc lưu trữ hệ điều hành và các tệp dữ liệu. Bên cạnh đó, bộ nhớ DDR4 4GB cung cấp bộ nhớ chính cho các tác vụ tính toán, trong khi bộ nhớ cache L1 và L2 của CPU ARM Cortex-A53 tăng tốc độ truy xuất dữ liệu, hỗ trợ quá trình thực thi hệ điều hành và các ứng dụng đi kèm.

Ở những phần kế tiếp, chúng tôi sẽ giới thiệu sơ về cách sử dụng các bộ nhớ ngoài một cách hiệu quả, vì bộ nhớ ngoài chip thường đã được thiết kế và có sẵn trong các hệ thống FPGA. Do đó, việc tận dụng các bộ nhớ ngoài như L1/L2 Cache, DDRAM, hoặc NAND Flash sẽ giúp tối ưu hóa hiệu suất hệ thống mà không cần phải thiết kế bộ nhớ từ đầu.

### 3.3.2. Bộ nhớ đệm (cache) L1, L2, L3

L1, L2 và L3 Cache là các bộ nhớ tạm thời (cache memory) giúp tăng tốc độ truy cập dữ liệu cho CPU trong hệ thống máy tính, giảm độ trễ và nâng cao hiệu suất xử lý. L1 Cache, với dung lượng nhỏ (khoảng 32KB), có thời gian truy cập nhanh nhất, từ 3 đến 4 chu kỳ, và chủ yếu lưu trữ các dữ liệu và lệnh được CPU sử dụng thường xuyên. L2 Cache,

với dung lượng lớn hơn (khoảng 256KB), có tốc độ truy cập chậm hơn một chút (10 đến 30 chu kỳ), được sử dụng để lưu trữ dữ liệu không có trong L1 Cache nhưng vẫn cần truy xuất nhanh. L3 Cache, thường có dung lượng lớn hơn (từ vài MB đến hàng chục MB), là bộ nhớ cache chung cho tất cả các lõi của CPU, có thời gian truy cập chậm nhất trong ba loại cache nhưng vẫn nhanh hơn so với bộ nhớ chính. L3 Cache giúp giảm tải cho L1 và L2 Cache, đồng thời cung cấp bộ nhớ cho các dữ liệu ít được truy xuất nhưng vẫn cần thiết. Cả ba loại cache này đều được quản lý bởi phần cứng và đóng vai trò quan trọng trong việc tối ưu hóa hiệu suất hệ thống. Để sử dụng hiệu quả L1, L2 và L3 Cache, việc tối ưu mã nguồn, đặc biệt trong code C, là rất quan trọng, nhằm tận dụng tối đa khả năng truy xuất nhanh của cache và giảm thiểu độ trễ truy cập bộ nhớ chính (RAM).

Cache lưu trữ dữ liệu theo các đơn vị gọi là "cache lines", mỗi cache line có độ dài cố định và phụ thuộc vào phần cứng. Ví dụ, trong trường hợp này, mỗi cache line có độ dài 32 byte và được căn chỉnh trên biên độ 32 byte. Khi CPU truy cập một địa chỉ bộ nhớ (đọc hoặc ghi), toàn bộ cache line chứa địa chỉ đó sẽ được kéo vào bộ nhớ cache, giúp tăng tốc độ truy cập dữ liệu. Khi dữ liệu được tìm thấy trong bộ nhớ cache, đó gọi là cache hit. Khi xảy ra cache hit, dữ liệu đã được lưu trữ sẵn trong cache và có thể truy cập ngay lập tức, giúp tiết kiệm thời gian truy cập bộ nhớ chính. Ngược lại, khi dữ liệu không có trong cache, gọi là cache miss, CPU sẽ phải truy xuất dữ liệu từ bộ nhớ chính và tải nó vào cache. Trong quá trình này, bộ nhớ cache sẽ nạp một vùng bộ nhớ liên kết, được gọi là một line, mà chứa một hoặc nhiều từ vật lý (physical words). Từ vật lý là đơn vị cơ bản của việc truy cập bộ nhớ, được sử dụng để chuyển dữ liệu giữa bộ xử lý và bộ nhớ cache. Giả sử một bộ vi xử lý có kích thước dòng bộ nhớ cache là 32 byte. Khi truy cập địa chỉ 0x3a40, bộ nhớ cache sẽ tải các địa chỉ từ 0x3a40 đến 0x3a5f vào bộ nhớ cache. Tiếp theo, khi truy cập địa chỉ 0x3a94, bộ nhớ cache sẽ tải các địa chỉ từ 0x3a94 đến 0x3ab3. Khi truy cập địa chỉ 0x3a48, vì bộ nhớ cache đã tải từ 0x3a94 đến 0x3ab3, nên sẽ xảy ra một lỗi cache vì địa chỉ 0x3a48 không nằm trong phạm vi của dòng bộ nhớ cache hiện tại. Cuối cùng, khi truy cập 4 từ 32-bit liên tiếp từ 0x8000 đến 0x801c, mỗi từ sẽ được truy cập vào bộ nhớ cache trong các dòng cache khác nhau. Tùy thuộc vào việc các dòng này đã được tải vào bộ nhớ cache hay

chưa, sẽ có một số lần lỗi cache và hit cache. Tổng cộng, sẽ có ít nhất 3 lần lỗi cache (cho 0x3a48 và các địa chỉ trong dải 0x8000 đến 0x801c) và các lần hit cache phụ thuộc vào việc các dòng bộ nhớ cache đã có sẵn khi truy cập các địa chỉ đó.

Để giảm tỷ lệ lỗi của cache và tăng hiệu suất, nguyên lý của tính cục bộ (locality) cho rằng các chương trình có xu hướng tái sử dụng dữ liệu và lệnh mà chúng đã sử dụng gần đây, hoặc các dữ liệu và lệnh có địa chỉ gần các tham chiếu gần đây. Nguyên lý này được chia thành hai loại: tính cục bộ theo thời gian (temporal locality) và tính cục bộ theo không gian (spatial locality). Tính cục bộ theo thời gian ám chỉ khả năng các mục dữ liệu vừa được truy xuất sẽ được truy xuất lại trong tương lai gần, trong khi tính cục bộ theo không gian ám chỉ các mục có địa chỉ gần nhau thường xuyên được truy xuất gần nhau trong thời gian. Ví dụ về tính cục bộ theo không gian là việc tham chiếu các phần tử mảng liên tiếp, chẳng hạn như trong mô hình tham chiếu stride-1. Tính cục bộ theo thời gian thể hiện qua việc tham chiếu một biến, ví dụ như sum, trong mỗi vòng lặp. Ví dụ, đoạn mã với vòng lặp qua các phần tử mảng thể hiện cả tính cục bộ theo không gian (trong việc truy cập mảng) và tính cục bộ theo thời gian (trong việc tham chiếu biến sum). Tương tự, các lệnh được thực thi liên tiếp đại diện cho tính cục bộ theo không gian, trong khi việc lặp lại vòng lặp minh họa tính cục bộ theo thời gian.

#### Giả sử bộ nhớ cache có kích thước 4 byte và mỗi block cache chứa 7 từ (words)

```
int sumarrayrows (int a[M][N])
{
    int i, j, sum = 0;
    for ( i = 0; i < M; i++)
        for ( j = 0; j < N; j++)
            sum += a[ i ][ j ];
    return sum;
}
```

Tỷ lệ lỗi cache là =  $1/8 = 12.5\%$

(a)

```
int sumarrayrows (int a[M][N])
{
    int i, j, sum = 0;
    for ( j = 0; j < N; j++)
        for ( i = 0; i < M; i++)
            sum += a[ i ][ j ];
    return sum;
}
```

Tỷ lệ lỗi cache là =  $100\%$

(b)

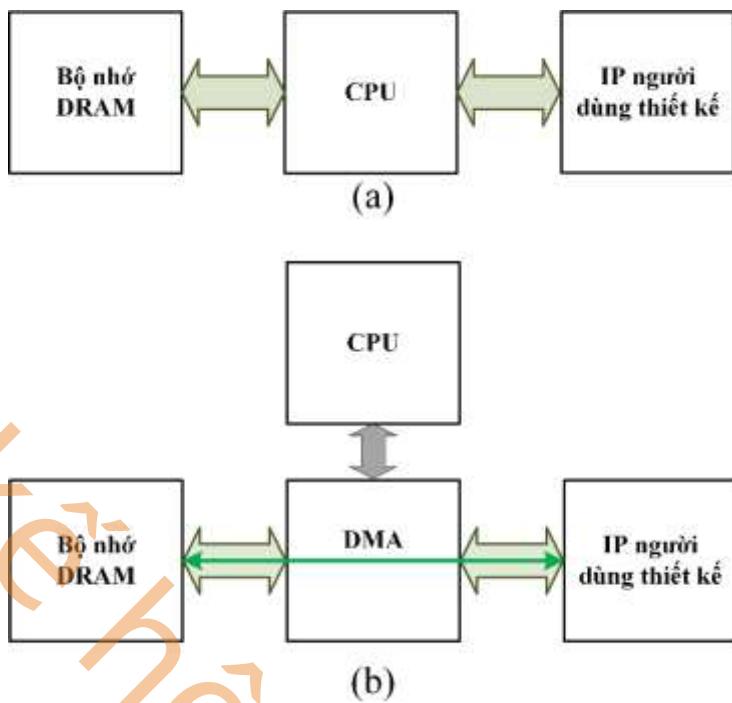
Hình 3.28: Tỷ lệ lỗi cache khi truy cập mảng theo thứ tự khác nhau với bộ nhớ cache 4 byte và mỗi block chứa 7 từ. (a) Lỗi cache 12.5%, (b) Lỗi cache 100%.

Hình 3.28 minh họa tỷ lệ lỗi cache khi truy cập mảng theo thứ tự khác nhau với bộ nhớ cache 4 byte và mỗi block cache chứa 7 từ. Ví dụ mã nguồn được trình bày với hai cách duyệt mảng: trong sumarrayrows (bên trái), dữ liệu được truy cập theo chiều dọc (thứ tự cột trước, sau đó mới đến hàng), giúp tận dụng tính cục bộ theo không gian (spatial locality) và dẫn đến tỷ lệ lỗi cache thấp (12.5%). Tỷ lệ này được tính như sau: vì mỗi block trong bộ nhớ cache chứa 7 từ (words), mà bộ nhớ cache có thể lưu trữ tối đa 7 từ mỗi lần, khi tiếp cận một phần tử mảng khác ngoài 7 từ đầu tiên trong block đó, bộ nhớ cache sẽ bị "miss" (lỗi cache). Do đó, với các kích thước bộ nhớ cache và cách duyệt này, tổng số lỗi cache xảy ra sẽ là 1 trong số 8 lần truy cập (1/8), tương đương với tỷ lệ lỗi cache là 12.5%. Ngược lại, trong sumarraycols (bên phải), việc duyệt qua các cột trước, sau đó mới đến các hàng khiến các địa chỉ bộ nhớ không liên tiếp và không tối ưu cho bộ nhớ cache, dẫn đến tỷ lệ lỗi cache cao (100%). Do đó, cách tiếp cận tối ưu là duyệt qua các hàng (như trong sumarrayrows) thay vì các cột (sumarraycols), giúp tối ưu hóa việc sử dụng bộ nhớ cache và giảm độ trễ do lỗi cache.

Tóm lại, việc tối ưu hóa sử dụng bộ nhớ cache L1, L2 và L3 chủ yếu được thực hiện thông qua việc tối ưu hóa mã nguồn, đặc biệt là tối ưu hóa cách thức truy cập bộ nhớ, sử dụng mảng và cải thiện tính cục bộ theo thời gian và không gian trong phần mềm.

### 3.3.3. Bộ nhớ DRAM

Bộ nhớ truy cập ngẫu nhiên động DRAM (Dynamic Random Access Memory) là một loại bộ nhớ có khả năng lưu trữ dữ liệu với dung lượng lớn, thường từ vài GB đến hàng chục GB, tốc độ truy xuất chậm hơn so với các bộ nhớ cache như L1, L2, L3, nhưng vẫn đóng vai trò quan trọng trong hệ thống SoC FPGA. Tuy DRAM có tốc độ truy xuất thấp hơn bộ nhớ cache, nhưng bộ nhớ cache L1, L2, L3 thực chất cũng sẽ truy cập và lấy dữ liệu từ DRAM khi bộ nhớ cache không có dữ liệu cần thiết, giúp tối ưu hóa tốc độ xử lý và hiệu suất hệ thống. DRAM thường được sử dụng để lưu trữ dữ liệu tạm thời và các chương trình lớn, giúp cung cấp không gian bộ nhớ linh hoạt cho các ứng dụng phức tạp.

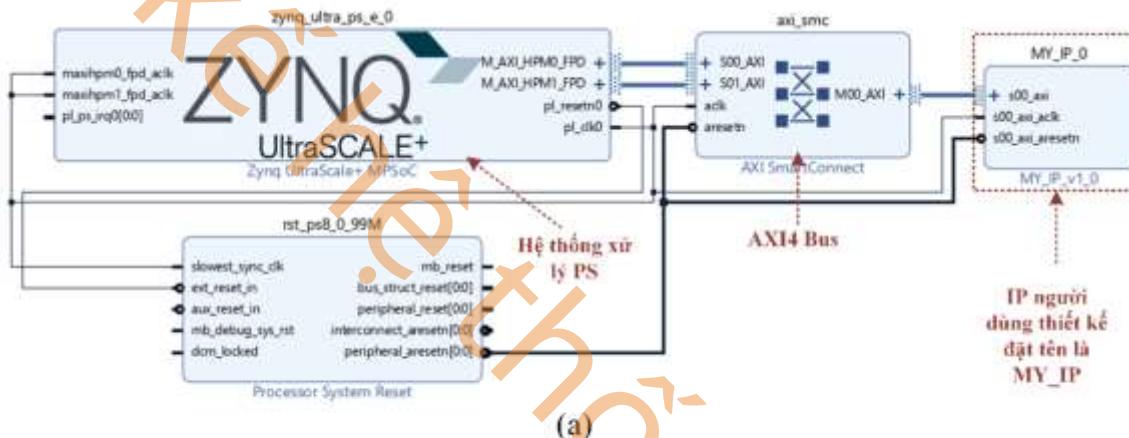


Hình 3.29: Các phương thức truyền dữ liệu giữa bộ nhớ DRAM và IP người dùng thiết kế (a) PIO, (b) DMA.

Trong các hệ thống SoC FPGA, DRAM thường được kết nối với các IP người dùng thiết kế để cung cấp dung lượng lưu trữ cao, đồng thời hỗ trợ truy xuất dữ liệu nhanh chóng cho các phép toán tính toán hoặc xử lý tín hiệu. DRAM lưu trữ các dữ liệu tạm thời cho các IP người dùng thiết kế. Một vài ứng dụng điển hình của DRAM lưu trữ dữ liệu tạm thời có thể kể đến là trong ứng dụng AI, nơi DRAM lưu trữ các tham số huấn luyện trước (pre-training parameters) như trọng số (weight) và độ lệch (bias), có số lượng lớn, việc lưu trữ toàn bộ ở bộ nhớ on-chip sẽ quá tốn kém. Hoặc trong các ứng dụng xử lý ảnh, trong đó số lượng lớn các ảnh đầu vào và đầu ra có thể được lưu trữ tạm thời trong DRAM và truyền liên tiếp vào IP để tính toán tuần tự. Việc sử dụng DRAM trong những trường hợp này giúp tiết kiệm chi phí bộ nhớ và nâng cao hiệu suất xử lý.

Cách thức sử dụng DRAM trong hệ thống SoC FPGA có thể thực hiện qua hai phương thức chính: PIO (Programmed I/O-Đầu ra vào được lập trình) và DMA (Direct Memory Access- Truy cập bộ nhớ trực tiếp). Trong phương thức PIO, CPU trực tiếp điều khiển việc truyền tải dữ liệu giữa DRAM và các IP thông qua các lệnh I/O, như được mô tả trong Hình 3.29 (a). Phương thức này dễ triển khai nhưng có thể gây tốn năng cho CPU vì CPU

phải xử lý tất cả các thao tác truyền nhận dữ liệu. Ngược lại, DMA cho phép truyền tải dữ liệu giữa DRAM và các IP mà không cần sự can thiệp của CPU, giúp giảm tải cho CPU và tối ưu hóa băng thông, như mô tả trong Hình 3.29 (b). DMA hoạt động độc lập với CPU, truyền tải dữ liệu nhanh chóng và hiệu quả mà không cần phải chờ đợi các thao tác xử lý của CPU. Tùy vào yêu cầu của hệ thống, PIO hoặc DMA có thể được sử dụng. Ví dụ, với các tín hiệu điều khiển cơ bản cho IP, PIO là phương thức dễ dàng triển khai và quản lý, trong khi DMA được ưu tiên khi cần truyền nhận dữ liệu lớn giữa DRAM và các bộ nhớ trên chip (URAM, BRAM, FIFO) của IP, giúp giảm độ trễ và tối ưu hóa hiệu suất hệ thống

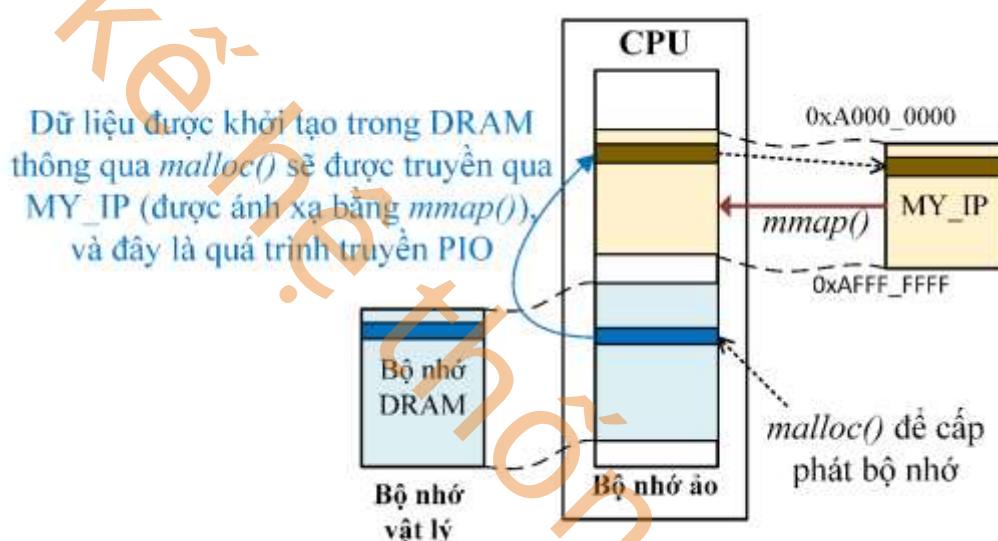


Hình 3.30: (a) Mô tả hệ thống SoC đơn giản với PS và IP thiết kế với tên MY\_IP. (b) Địa chỉ được chỉ định của MY\_IP trên SoC.

Để mô tả cách thức truyền tải dữ liệu giữa DRAM và IP trong hệ thống SoC, chúng tôi thiết kế một SoC đơn giản bao gồm hệ thống xử lý PS (Processing System) và một IP thiết kế với tên gọi MY\_IP, như Hình 3.30 (a). Trong hệ thống này, PS và MY\_IP được kết nối thông qua AXI4 bus, với địa chỉ của MY\_IP được chỉ định trong phạm vi từ 0xA000\_0000 đến 0xAFFF\_FFFF, như thể hiện trong Hình 3.30 (b). Cách thức này cho phép PS và

MY\_IP giao tiếp trực tiếp với nhau thông qua bus AXI4, với MY\_IP có thể truy xuất các dữ liệu từ DRAM hoặc truyền dữ liệu ra ngoài thông qua các lệnh được cung cấp từ PS. Trong quá trình này, PS sẽ cấp phát và điều khiển dữ liệu giữa DRAM và MY\_IP thông qua các cơ chế truyền như PIO hoặc DMA, tùy thuộc vào yêu cầu về hiệu suất và băng thông của hệ thống.

**Truyền PIO:** Trong phương thức này, CPU trực tiếp điều khiển việc truyền tải dữ liệu giữa DRAM và IP thông qua các lệnh I/O, với PS cung cấp tín hiệu điều khiển để truy xuất và truyền dữ liệu.



Hình 3.31: Quá trình truyền PIO từ DRAM vào MY\_IP thông qua ánh xạ bộ nhớ `mmap()`.

Hình 3.31 minh họa quá trình truyền dữ liệu được khởi tạo trong DRAM thông qua `malloc()` và sau đó truyền vào MY\_IP, nơi MY\_IP được ánh xạ vào bộ nhớ ảo thông qua `mmap()`. Quá trình này thể hiện việc sử dụng bộ nhớ ảo để cấp phát bộ nhớ, sau đó truyền dữ liệu từ bộ nhớ DRAM vào MY\_IP. Đây là một ví dụ về quá trình truyền PIO (Programmed I/O), trong đó CPU điều khiển việc truyền tải dữ liệu giữa DRAM và MY\_IP thông qua các lệnh I/O. Trong quá trình này, MY\_IP sử dụng bộ nhớ ảo đã được ánh xạ từ vùng bộ nhớ vật lý của DRAM, giúp dữ liệu được truy cập và truyền gián tiếp từ DRAM vào MY\_IP.

```

#define MY_IP_BASE_ADDRESS 0xA8000000 // Địa chỉ của MY_IP trong bộ nhớ
#define MY_IP_SIZE 0x1000 // Kích thước vùng bộ nhớ của MY_IP
#define MY_IP_REGISTER_OFFSET 0x04 // Địa chỉ offset cho register của MY_IP
#define DRAM_SIZE 0x1000 // Kích thước vùng bộ nhớ DRAM

int main() {
    // Cấp phát bộ nhớ cho dữ liệu trong DRAM bằng malloc()
    uint32_t* dram_data = (uint32_t*) malloc(DRAM_SIZE);
    if (!dram_data) { perror("malloc() failed"); exit(1); } // Kiểm tra xem malloc có thành công không

    // Khởi tạo dữ liệu trong DRAM
    for (int i = 0; i < DRAM_SIZE / sizeof(uint32_t); i++) {
        dram_data[i] = 0x1000 + i; // Dữ liệu mẫu
    }

    // Mở /dev/mem và ánh xạ MY_IP vào bộ nhớ người dùng bằng mmap()
    int mem_fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (mem_fd == -1) {
        perror("Failed to open /dev/mem"); exit(1); // Kiểm tra việc mở /dev/mem có thành công không
    }

    // Ánh xạ MY_IP vào bộ nhớ ảo
    volatile uint32_t* my_ip_base = (volatile uint32_t*) mmap(NULL, MY_IP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, mem_fd, MY_IP_BASE_ADDRESS);
    close(mem_fd); // Đóng tệp /dev/mem

    // Truyền dữ liệu từ DRAM vào MY_IP
    for (int i = 0; i < DRAM_SIZE / sizeof(uint32_t); i++) {
        my_ip_base[i] = dram_data[i]; // Truyền dữ liệu từ DRAM vào MY_IP
    }

    // Đọc lại dữ liệu từ MY_IP (để kiểm tra)
    for (int i = 0; i < DRAM_SIZE / sizeof(uint32_t); i++) {
        dram_data[i] = my_ip_base[i]; // Lấy dữ liệu từ MY_IP và lưu vào DRAM
    }

    free(dram_data); // Giải phóng bộ nhớ đã cấp phát cho DRAM
    munmap((void*)my_ip_base, MY_IP_SIZE); // Giải phóng bộ nhớ đã ánh xạ cho MY_IP
    return 0;
}

```

Dịnh nghĩa vùng địa chỉ và kích thước vùng bộ nhớ của MY\_IP

Cấp phát và khởi tạo dữ liệu trong bộ nhớ ảo, và dữ liệu này có thể được ánh xạ vào bộ nhớ vật lý (như DRAM)

Mở /dev/mem và ánh xạ MY\_IP vào nhớ ảo người dùng bằng mmap()

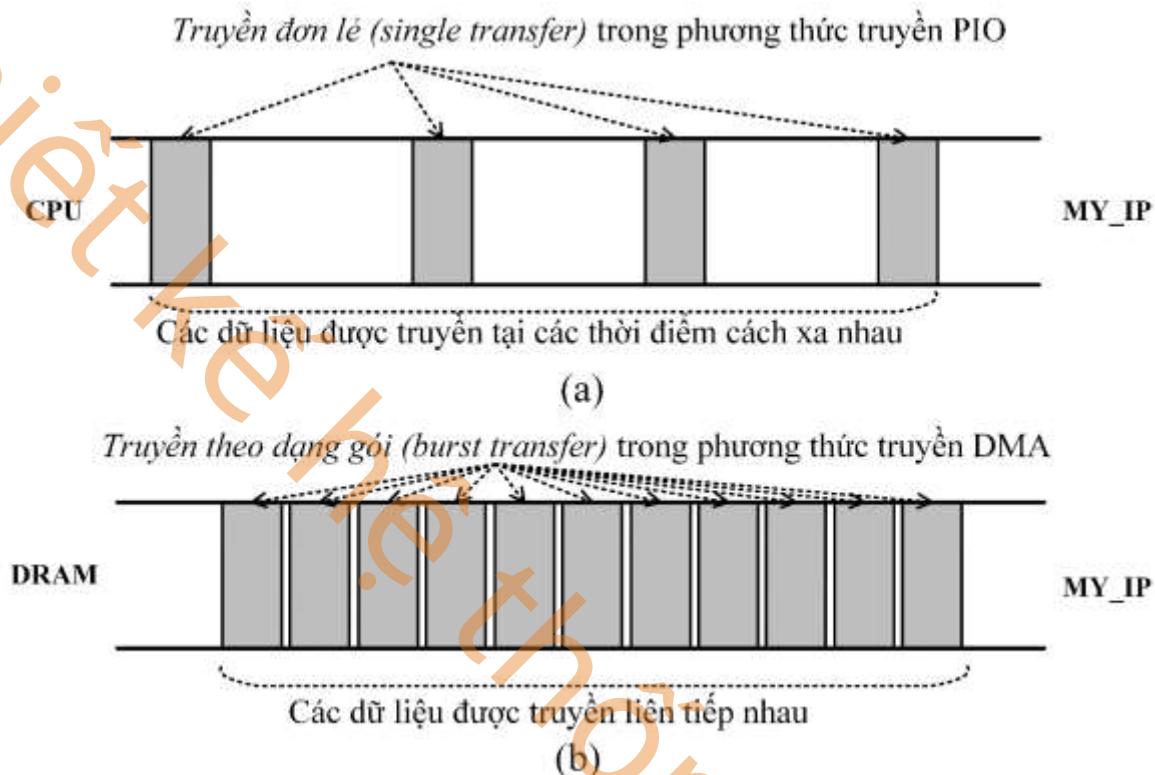
Ghi và đọc dữ liệu cho MY\_IP từ vùng nhớ ảo được ánh xạ vào bộ nhớ vật lý (DRAM)

Hình 3.32: Code C mô tả quá trình truyền PIO từ DRAM vào MY\_IP thông qua ánh xạ bộ nhớ mmap().

Hình 3.32 mô tả quá trình truyền dữ liệu từ bộ nhớ DRAM vào MY\_IP sử dụng phương thức PIO (Programmed I/O) thông qua ánh xạ bộ nhớ mmap(). Đầu tiên, bộ nhớ cho dữ liệu trong DRAM được cấp phát bằng malloc(), và dữ liệu mẫu được khởi tạo trong vùng bộ nhớ này. Sau đó, hệ thống mở tệp /dev/mem để ánh xạ vùng bộ nhớ của MY\_IP vào bộ nhớ ảo thông qua mmap(), cho phép CPU truy cập trực tiếp MY\_IP. Tiếp theo, dữ liệu từ DRAM được truyền vào MY\_IP qua các lệnh I/O, tương tự như quá trình truyền PIO. Sau khi dữ liệu được truyền, nó có thể được đọc lại từ MY\_IP và lưu vào DRAM để kiểm tra tính chính xác của quá trình truyền. Cuối cùng, bộ nhớ được giải phóng bằng cách gọi free() cho DRAM và munmap() cho bộ nhớ đã ánh xạ.

**Truyền DMA:** Truyền DMA là một phương thức cho phép truyền tải dữ liệu trực tiếp giữa các bộ nhớ trong hệ thống mà không cần sự can thiệp của CPU. Với DMA, dữ liệu có thể được truyền từ DRAM tới các IP hoặc ngược lại mà không cần CPU xử lý từng lệnh

truy xuất, giúp giảm tải cho CPU và cải thiện hiệu suất hệ thống. Phương thức này đặc biệt hữu ích trong các ứng dụng cần băng thông lớn hoặc khi cần truyền tải khối lượng dữ liệu lớn một cách nhanh chóng.

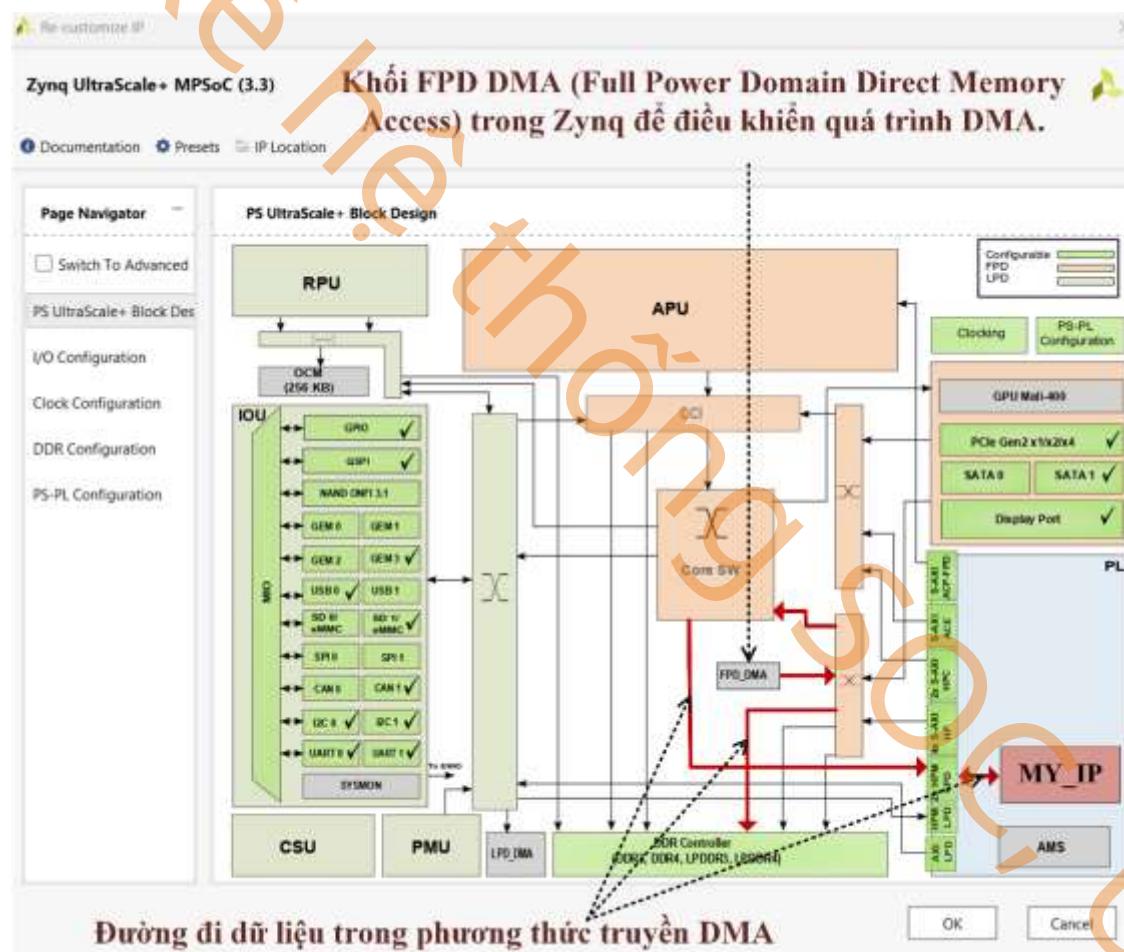


Hình 3.33: (a) Truyền đơn lẻ (single transfer) trong phương thức truyền PIO và (b) truyền theo dạng gói (burst transfer) trong phương thức truyền DMA.

Hình 3.33 minh họa sự khác biệt giữa phương thức truyền PIO và DMA. Trong phương thức truyền PIO, như thể hiện ở Hình 3.33 (a), truyền đơn lẻ (single transfer) yêu cầu CPU điều khiển từng lần truyền dữ liệu giữa DRAM và MY\_IP, với mỗi đơn vị dữ liệu được xử lý riêng biệt. Phương thức này cần sự can thiệp của CPU vào mỗi thao tác truyền tải, với CPU phải gửi lệnh và xử lý từng đơn vị dữ liệu, điều này làm giảm hiệu suất và hiệu quả băng thông vì CPU phải thực hiện quá nhiều thao tác cho mỗi đơn vị dữ liệu. Ngược lại, trong phương thức truyền DMA, như thể hiện ở Hình 3.33 (b), dữ liệu được truyền theo dạng gói (burst transfer), cho phép dữ liệu được truyền liên tục từ DRAM vào MY\_IP mà không cần sự can thiệp của CPU. Dữ liệu trong DMA có thể được truyền liên tục hoặc cách xa nhau tùy theo yêu cầu của ứng dụng. Việc truyền liên tục dữ liệu trong các gói giúp tận

dụng tối đa băng thông và giảm thiểu độ trễ, vì DMA có khả năng truyền tải dữ liệu từ các vùng bộ nhớ liền kề mà không cần sự gián đoạn.

Để sử dụng DMA, hệ thống SoC trên FPGA như ZCU102 hỗ trợ sẵn khối DMA, giúp truyền tải dữ liệu giữa các bộ nhớ mà không cần CPU can thiệp. Hình 3.34 mô tả hệ thống xử lý ZYNQ UltraScale+ MPSoC với khối truy cập bộ nhớ trực tiếp miền công suất đầy đủ FPD DMA (Full Power Domain Direct Memory Access), tính năng được tích hợp sẵn trong hệ thống này. FPD DMA cho phép truyền DMA từ DRAM sang MY\_IP một cách trực tiếp, tương tự như việc gọi khói DMA Controller ở phần PL ở rất nhiều hệ thống SoC trên FPGA khác.



Hình 3.34: Khối FPD DMA trong hệ thống xử lý (ZYNQ PS) hỗ trợ truyền DMA từ DRAM sang MY\_IP.

Hình 3.35 minh họa mã C mô tả quá trình truyền DMA giữa DRAM và MY\_IP thông qua FPD DMA trên Zynq. Dữ liệu trong DRAM được khởi tạo thông qua việc ánh xạ vùng

bộ nhớ DRAM lên bộ nhớ ảo bằng cách sử dụng hàm mmap(). Sau khi ánh xạ thành công, DRAM trở thành vùng bộ nhớ có thể truy cập trực tiếp từ chương trình, và dữ liệu có thể được khởi tạo hoặc thay đổi tại đây. Để thực hiện việc truyền DMA, hệ thống tiếp tục ánh xạ khỏi DMA vào bộ nhớ ảo thông qua mmap(), cho phép cấu hình và điều khiển quá trình DMA. Trong quá trình DMA, các thông số như ZDMA\_CH\_SRC\_DSCR\_WORD0 và ZDMA\_CH\_DST\_DSCR\_WORD0 được sử dụng để cấu hình địa chỉ nguồn (DRAM) và địa chỉ đích (MY\_IP). Sau khi cấu hình xong, DMA được kích hoạt bằng cách gán giá trị cho ZDMA\_CH\_CTRL2. Hệ thống sẽ tiếp tục chờ cho đến khi trạng thái DMA (ZDMA\_CH\_STATUS) cho biết quá trình truyền dữ liệu đã hoàn tất

```

#define MY_IP_BASE_ADDRESS 0x00000000 // Địa chỉ của MY_IP
#define MY_IP_SIZE 0x1000 // Kích thước vùng bộ nhớ của MY_IP
#define MY_IP_REGISTER_OFFSET 0x004 // Địa chỉ offset cho register MY_IP
#define DRAM_SIZE 0x1000 // Kích thước vùng bộ nhớ DRAM
#define DMA_BASE_PHYS 0x0000000000000000 // Địa chỉ DMA
#define DMA_MMAP_SIZE 0x0000000000000000 // Kích thước bộ nhớ ánh xạ DMA

int main() {
    // Ánh xạ DRAM vào bộ nhớ ảo
    int fd_ddr = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd_ddr == -1) { perror("Failed to open /dev/mem"); exit(1); }
    uint32_t* dram_data = (uint32_t*) mmap(NULL, DRAM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd_ddr, DMA_BASE_PHYS);
    if (dram_data == MAP_FAILED) { perror("mmap() failed"); close(fd_ddr); exit(1); }
    close(fd_ddr);

    // Khai tạo dữ liệu trong DRAM
    for (int i = 0; i < DRAM_SIZE / sizeof(uint32_t); i++) {
        dram_data[i] = 0x1000 + i; // Dữ liệu mẫu
    }

    // Mở /dev/mem và ánh xạ DMA vào bộ nhớ ảo
    int mem_fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (mem_fd == -1) { perror("Failed to open /dev/mem"); exit(1); }
    // Ánh xạ DMA vào bộ nhớ ảo
    volatile uint32_t* dma_base = (volatile uint32_t*) mmap(NULL, DMA_MMAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, mem_fd, DMA_BASE_PHYS);
    close(mem_fd); // Đóng file /dev/mem

    // Truyền dữ liệu từ DRAM vào MY_IP
    dma_base->ZDMA_CH_SRC_DSCR_WORD0 = (uint64_t) dram_data; // Địa chỉ nguồn (DRAM)
    dma_base->ZDMA_CH_SRC_DSCR_WORD2 = DRAM_SIZE; // Kích thước dữ liệu nguồn
    dma_base->ZDMA_CH_DST_DSCR_WORD0 = MY_IP_BASE_ADDRESS + MY_IP_REGISTER_OFFSET; // Địa chỉ đích (MY_IP)
    dma_base->ZDMA_CH_DST_DSCR_WORD2 = DRAM_SIZE; // Kích thước dữ liệu đích
    dma_base->ZDMA_CH_CTRL2 = 1; // Bật DMA để truyền dữ liệu
    while ((dma_base->ZDMA_CH_STATUS & 1) == 0); // Chờ cho quá trình DMA truyền dữ liệu từ DRAM vào MY_IP hoàn tất

    // Truyền dữ liệu từ MY_IP vào DRAM
    dma_base->ZDMA_CH_SRC_DSCR_WORD0 = MY_IP_BASE_ADDRESS + MY_IP_REGISTER_OFFSET; // Địa chỉ nguồn (MY_IP)
    dma_base->ZDMA_CH_SRC_DSCR_WORD2 = DRAM_SIZE; // Kích thước dữ liệu nguồn
    dma_base->ZDMA_CH_DST_DSCR_WORD0 = (uint64_t) dram_data; // Địa chỉ đích (DRAM)
    dma_base->ZDMA_CH_DST_DSCR_WORD2 = DRAM_SIZE; // Kích thước dữ liệu đích
    dma_base->ZDMA_CH_CTRL2 = 1; // Bật DMA để truyền dữ liệu
    while ((dma_base->ZDMA_CH_STATUS & 3) == 0); // Chờ cho quá trình DMA truyền dữ liệu từ MY_IP về DRAM hoàn tất
}

return 0;

```

*Định nghĩa vùng địa chỉ và kích thước địa chỉ của MY\_IP*

*Cấp phát và khởi tạo dữ liệu trong bộ nhớ ảo thông qua mmap(), và dữ liệu này được ánh xạ vào bộ nhớ vật lý DRAM*

*Mở /dev/mem và ánh xạ DMA vào nhớ nhô người dùng bằng mmap()*

*Truyền DMA từ DRAM vào MY\_IP hoặc đọc dữ liệu từ MY\_IP vào DRAM*

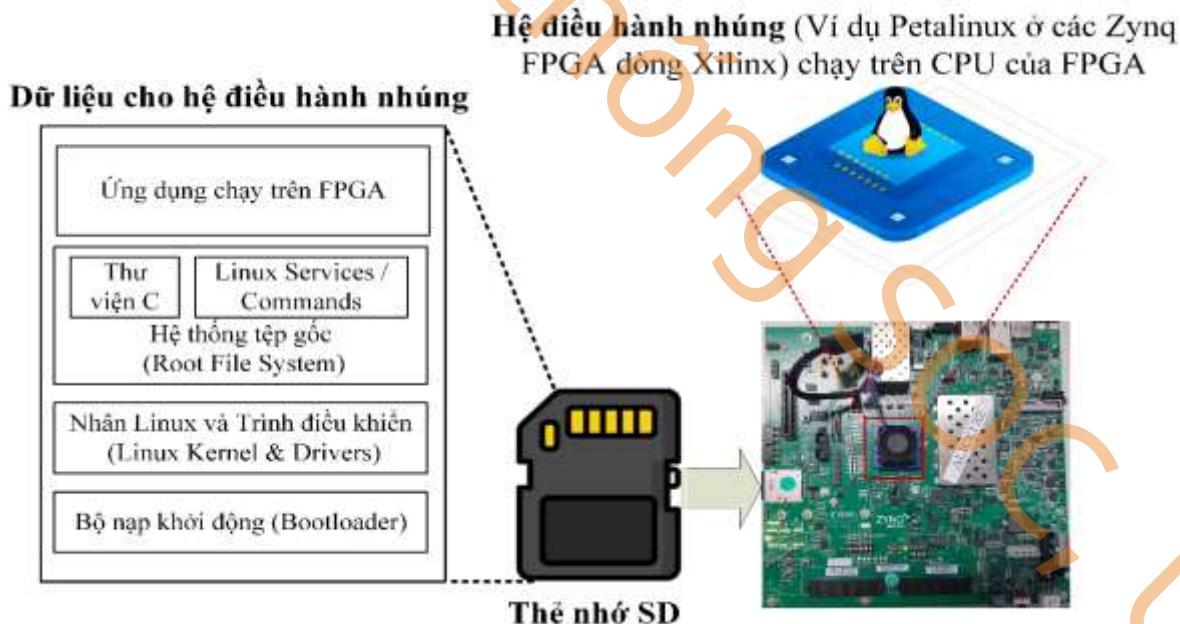
Hình 3.35: Code C mô tả quá trình truyền DMA từ DRAM vào MY\_IP thông qua FPD DMA.

Nhìn chung, có hai cách ghi dữ liệu từ DRAM vào IP thiết kế trong hệ thống SoC: PIO (Programmed I/O) và DMA (Direct Memory Access). Phương thức PIO yêu cầu CPU điều khiển từng thao tác truyền tải dữ liệu giữa DRAM và IP thông qua các lệnh I/O, trong khi

DMA cho phép truyền tải dữ liệu trực tiếp giữa DRAM và IP mà không cần sự can thiệp của CPU.

### 3.3.4. Bộ nhớ NAND Flash

Bộ nhớ NAND Flash, đặc biệt là thẻ nhớ an toàn kỹ thuật số (SD, Secure Digital), ngày càng trở thành một phần quan trọng trong các ứng dụng FPGA. NAND Flash được triển khai trong các thẻ nhớ như SD, Mini SD và Micro SD, với khả năng thay thế ổ đĩa, mang lại lợi ích về mật độ lưu trữ cao và tốc độ truyền tải nhanh. Thẻ nhớ SD với NAND Flash cung cấp mật độ bit lớn hơn so với các công nghệ Flash khác như NOR, giúp lưu trữ nhiều dữ liệu hơn trong không gian vật lý nhỏ gọn. Với tốc độ đọc/ghi khoảng từ 22 MBps (đọc) đến 18 MBps (ghi), NAND Flash rất phù hợp cho các ứng dụng như lưu trữ dữ liệu trong FPGA, đặc biệt là khi yêu cầu dung lượng lớn và tốc độ truyền tải nhanh. Mặc dù có giới hạn về số lần ghi, các phương pháp phát hiện và sửa lỗi giúp đảm bảo tính toàn vẹn của dữ liệu, làm cho NAND Flash vẫn là lựa chọn phổ biến trong các ứng dụng lưu trữ.



Hình 3.36: NAND Flash lưu trữ dữ liệu cho hệ điều hành nhúng trên SoC FPGA như một ổ cứng trên các máy tính thông thường.

Hình 3.36 mô tả cách bộ nhớ NAND Flash (thường là thẻ nhớ SD) được sử dụng để lưu trữ các dữ liệu cần thiết cho hệ điều hành nhúng trên SoC FPGA, giống như một ổ

cứng. Bộ nhớ này lưu trữ các thành phần quan trọng như ứng dụng chạy trên FPGA, thư viện C, dịch vụ và lệnh Linux, hệ thống tệp gốc (root file system), nhân Linux và các trình điều khiển, cũng như bộ nạp khởi động. Thẻ nhớ SD cung cấp không gian lưu trữ cho các dữ liệu hệ thống của hệ điều hành nhúng như Petalinux, cho phép khởi động và vận hành hệ thống trên SoC FPGA.

Ứng dụng thẻ nhớ SD trong hệ thống SoC, như thể hiện trong Hình 3.37, cho phép lưu trữ các dữ liệu quan trọng cho các IP thiết kế. Dữ liệu được lưu trữ trong các tệp trên thẻ nhớ SD và sau đó được đọc vào bộ nhớ DRAM thông qua các lệnh truy xuất tiêu chuẩn như `open("file")`. Sau khi dữ liệu đã có mặt trong DRAM, quá trình truyền dữ liệu từ DRAM đến MY\_IP được thực hiện thông qua DMA, giúp giảm tải cho CPU và tối ưu hóa băng thông. Quá trình này không yêu cầu sự can thiệp của CPU và đảm bảo hiệu suất truyền tải dữ liệu nhanh chóng và hiệu quả giữa bộ nhớ và các IP xử lý.



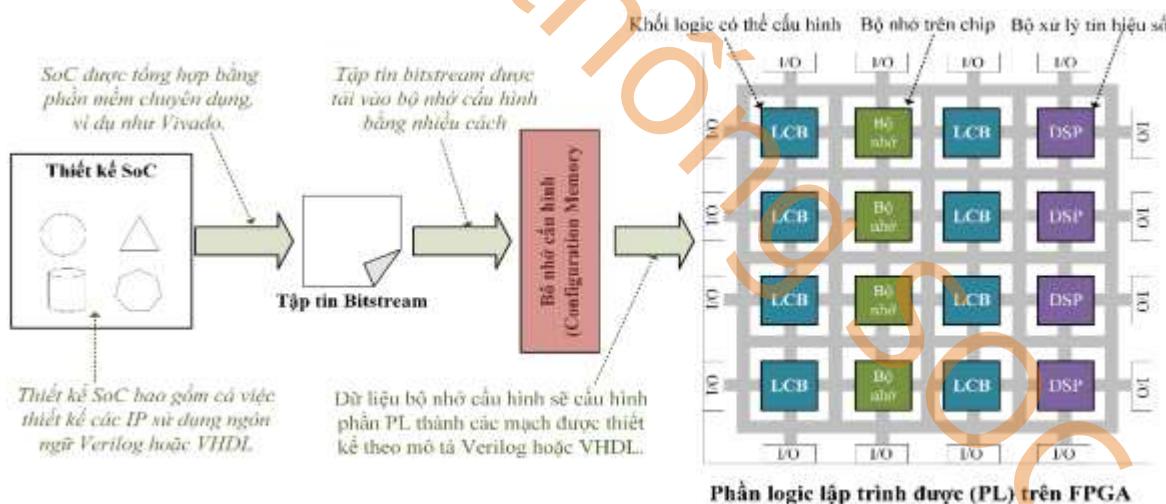
Hình 3.37: *Ứng dụng của thẻ nhớ SD trong việc lưu trữ dữ liệu cho IP thiết kế trong SoC và quy trình truyền dữ liệu từ thẻ nhớ SD lên DRAM và MY\_IP.*

Nhìn chung, bộ nhớ NAND Flash, đặc biệt là thẻ nhớ SD, đóng vai trò quan trọng trong các ứng dụng FPGA nhờ mật độ lưu trữ cao và tốc độ truyền tải nhanh. Với khả năng thay thế ổ đĩa, NAND Flash có mật độ bit lớn hơn so với công nghệ Flash NOR, giúp lưu trữ dữ liệu lớn trong không gian nhỏ gọn. Thẻ nhớ SD, sử dụng NAND Flash, rất phù hợp cho các ứng dụng đòi hỏi dung lượng lưu trữ lớn và tốc độ cao, chẳng hạn như lưu trữ hệ điều hành nhúng trên SoC FPGA. Mặc dù có giới hạn về số lần ghi, NAND Flash vẫn được sử dụng rộng rãi nhờ các phương pháp phát hiện và sửa lỗi, giúp bảo vệ tính toàn vẹn của dữ liệu.

### 3.4. Bộ nhớ cấu hình

Bộ nhớ cấu hình (Configuration Memory) trong FPGA là một phần quan trọng giúp định hình và cấu hình lại phần logic có thể lập trình (PL) trên FPGA. Khi một SoC được thiết kế và tổng hợp, các mô-đun IP được mô tả bằng ngôn ngữ Verilog hoặc VHDL, và một tập tin bitstream sẽ được tạo ra từ các mô tả này. Tập tin bitstream này chứa thông tin cấu hình cần thiết để lập trình phần PL của FPGA. Trong khi bộ nhớ cấu hình lưu trữ thông tin cấu hình, việc tải dữ liệu từ bộ nhớ này vào FPGA sẽ định hình lại các khối logic trong chip, giúp FPGA thực hiện các nhiệm vụ theo yêu cầu của hệ thống.

Hình 3.38 mô tả quy trình cấu hình SoC sử dụng bộ nhớ cấu hình trên FPGA. Tập tin bitstream được tạo ra thông qua phần mềm chuyên dụng (như Vivado) và sau đó được tải vào bộ nhớ cấu hình của FPGA, nơi nó cấu hình lại phần logic có thể lập trình (PL). Việc tải bitstream vào FPGA có thể được thực hiện qua nhiều phương thức, bao gồm việc nạp trực tiếp từ thẻ nhớ, qua giao diện JTAG, hoặc qua các kết nối bộ nhớ ngoài. Các phương pháp này giúp cấu hình lại các khối logic của FPGA một cách linh hoạt và dễ dàng, đáp ứng các yêu cầu của hệ thống.



Hình 3.38: Quy trình cấu hình SoC dùng bộ nhớ cấu hình trên FPGA.

Nhìn chung, bộ nhớ cấu hình là thành phần quan trọng trong việc thiết lập các mạch logic của FPGA. Quá trình cấu hình từ ngôn ngữ Verilog hoặc VHDL thành mạch trên FPGA qua bộ nhớ cấu hình ít khi bị tác động và thường là quy trình mặc định trong hầu hết các FPGA. Do đó, bộ nhớ cấu hình ít có sự điều chỉnh trong các hệ thống SoC, vì nó

được sử dụng để định hình phần logic của FPGA theo các yêu cầu đã được xác định từ trước.

### 3.5. Tóm tắt

Chương 3 đã cung cấp một cái nhìn tổng quan về thiết kế bộ nhớ trong các hệ thống sử dụng FPGA và SoC, đặc biệt nhấn mạnh vai trò quan trọng của bộ nhớ trong việc tối ưu hóa hiệu suất hệ thống. Bộ nhớ đóng vai trò quyết định trong việc quản lý và xử lý dữ liệu, đặc biệt trong các ứng dụng đòi hỏi tốc độ truy cập cao và dung lượng lớn. Chương đã phân tích chi tiết các loại bộ nhớ trong hệ thống SoC, bao gồm bộ nhớ trên chip, bộ nhớ ngoài chip và bộ nhớ cấu hình. Bộ nhớ trên chip (on-chip memory), như BRAM, URAM và LUTRAM, được tích hợp trực tiếp vào FPGA, mang lại tốc độ truy cập rất nhanh và băng thông cao, phù hợp cho các tác vụ cần xử lý ngay lập tức. Bộ nhớ ngoài chip (off-chip memory), như DRAM và NAND Flash, cung cấp dung lượng lưu trữ lớn nhưng có độ trễ cao hơn, được sử dụng chủ yếu để lưu trữ dữ liệu lớn hoặc khi yêu cầu dung lượng vượt quá khả năng của bộ nhớ trên chip. Bộ nhớ cấu hình (configuration memory) đóng vai trò quan trọng trong việc định hình các mạch logic trên FPGA, lưu trữ bitstream và cấu hình lại phần logic có thể lập trình (PL), giúp chuyển đổi các mô hình thiết kế từ Verilog hoặc VHDL thành các mạch logic thực tế trên FPGA.

### 3.6. Câu hỏi và bài tập

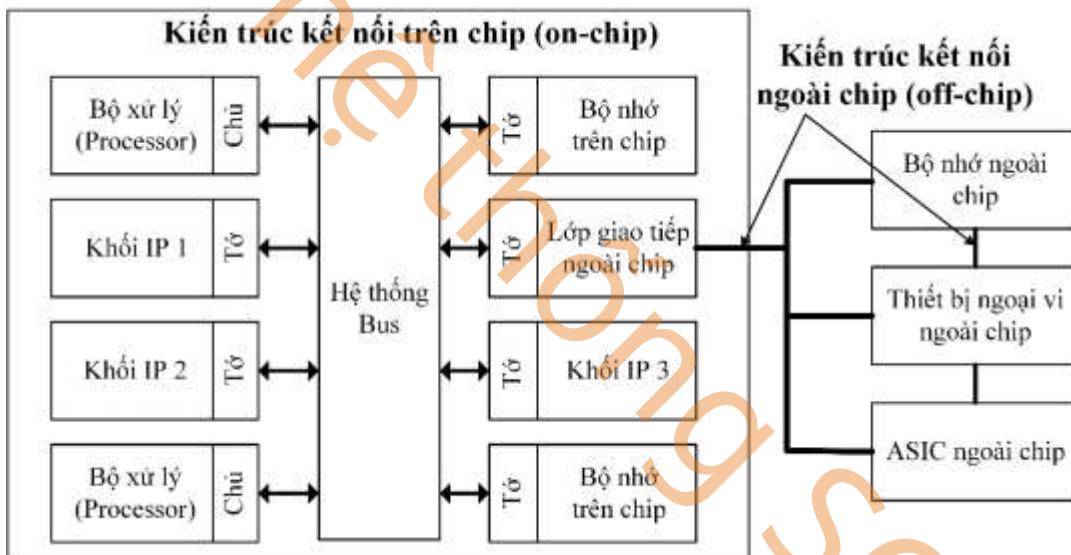
1. Bộ nhớ trên chip trong FPGA là gì và tại sao lại quan trọng đối với hiệu suất của hệ thống?
2. So sánh bộ nhớ BRAM, URAM, LUTRAM và FIFO trong FPGA về dung lượng, tốc độ truy cập và ứng dụng điển hình của mỗi loại.
3. Tại sao bộ nhớ BRAM thường được sử dụng cho các ứng dụng yêu cầu độ trễ thấp và tốc độ xử lý cao? Giả sử bạn cần lưu trữ 8MB dữ liệu trong FPGA, mỗi khối BRAM có dung lượng 18Kb. Tính số lượng BRAM cần thiết để lưu trữ 8MB dữ liệu.

4. Giải thích tại sao LUTRAM thích hợp cho các ứng dụng yêu cầu bộ nhớ nhỏ và độ trễ cực thấp.
5. Bộ nhớ FIFO có gì đặc biệt và khi nào nên sử dụng FIFO trong thiết kế SoC trên FPGA?
6. Mô phỏng quy trình truy xuất bộ nhớ BRAM trong FPGA bằng cách sử dụng mã Verilog để đọc và ghi dữ liệu.
7. Trong một thiết kế FPGA, bạn cần lưu trữ dữ liệu tạm thời trong một ứng dụng xử lý tín hiệu. Bạn sẽ chọn sử dụng loại bộ nhớ nào từ BRAM, URAM, LUTRAM, hoặc FIFO? Giải thích lý do của bạn.
8. Trình bày sự khác biệt giữa các chế độ NO\_CHANGE, READ\_FIRST và WRITE\_FIRST trong BRAM. Hãy giải thích khi nào bạn sẽ sử dụng mỗi chế độ này trong thiết kế FPGA.
9. So sánh các loại bộ nhớ ngoài chip như DRAM, NAND Flash và HDD về tốc độ truy cập, dung lượng, và ứng dụng điển hình của chúng trong các hệ thống SoC trên FPGA.
10. Tại sao bộ nhớ NAND Flash lại phù hợp cho lưu trữ hệ điều hành và các tệp dữ liệu trong các hệ thống nhúng trên SoC FPGA?
11. Giải thích cách thức sử dụng bộ nhớ DRAM trong hệ thống SoC FPGA để hỗ trợ các ứng dụng tính toán và xử lý tín hiệu? Trong hệ thống SoC có tốc độ xung nhịp 330 MHz và truyền DMA dữ liệu từ DRAM với tốc độ 40 Gbps. Độ rộng dữ liệu mỗi chu kỳ là 128 bit. Nếu bạn cần truyền 256MB dữ liệu từ DRAM, tính thời gian truyền nhận dữ liệu từ DRAM tới các IP.
12. Khi nào nên sử dụng phương thức DMA thay vì PIO trong việc truyền tải dữ liệu giữa DRAM và các IP trên SoC FPGA?
13. Bộ nhớ cấu hình trong FPGA có vai trò gì và tại sao nó lại quan trọng trong quá trình lập trình lại phần logic có thể lập trình (PL) trên FPGA?

## CHƯƠNG 4: HỆ THỐNG KẾT NỐI

### 4.1. Giới Thiệu Tổng Quan Kiến Trúc Kết Nối

Hệ thống kết nối trong SoC FPGA là mạng lưới các giao diện và bus kết nối các thành phần trong chip, bao gồm bộ vi xử lý, bộ nhớ, các khối logic và thiết bị ngoại vi. Hệ thống kết nối đóng vai trò quan trọng trong việc đảm bảo sự tương tác và giao tiếp mượt mà giữa các khối xử lý khác nhau, từ đó tối ưu hóa hiệu suất tổng thể của SoC. Nó giúp giảm thiểu độ trễ và tăng cường băng thông cho các ứng dụng yêu cầu truyền tải dữ liệu với tốc độ cao, như trong các thiết bị di động, viễn thông, và các hệ thống nhúng phức tạp. Việc thiết kế một hệ thống kết nối mạnh mẽ và linh hoạt là cần thiết để đảm bảo SoC hoạt động hiệu quả.



Hình 4.1: Sơ đồ mô tả kiến trúc kết nối trên chip và ngoài chip trong hệ thống SoC [1].

Hệ thống kết nối trong một SoC bao gồm các phần kết nối trên chip (on-chip) và kết nối ngoài chip (off-chip), đóng vai trò quan trọng trong việc truyền tải dữ liệu và điều khiển giữa các thành phần của hệ thống. Trong kiến trúc kết nối này, chủ (master) và tớ (slave) là các thuật ngữ được sử dụng để chỉ vai trò của các thành phần trong giao tiếp, trong đó thiết bị chủ điều khiển các giao dịch, còn thiết bị tớ thực hiện các yêu cầu từ chủ. Kết nối trên chip bao gồm các giao tiếp giữa bộ xử lý, bộ nhớ và các khối IP thông qua các bus nội bộ, trong khi kết nối ngoài chip đảm nhận việc giao tiếp với các thiết bị ngoại vi, bộ nhớ ngoài và các hệ thống khác ngoài chip. Kiến trúc kết nối trên chip, như thể hiện trong Hình

4.1, sử dụng một hệ thống bus để kết nối bộ xử lý (chủ), các khối IP và bộ nhớ trên chip (tớ). Bus trong kiến trúc kết nối này là hệ thống giao tiếp, truyền tải dữ liệu, tín hiệu điều khiển và các thông tin giữa các thành phần, giúp điều phối các giao dịch giữa thiết bị chủ và thiết bị tớ. Trong đó, các bộ xử lý đóng vai trò là chủ, còn các khối IP và bộ nhớ là tớ. Hệ thống bus trên chip đảm nhiệm vai trò truyền tải dữ liệu và tín hiệu điều khiển giữa các thành phần này, tối ưu hóa khả năng giao tiếp và truyền tải dữ liệu. Ngoài ra, một lớp giao tiếp ngoài chip được sử dụng để kết nối các thành phần trên chip với các thiết bị ngoại vi và bộ nhớ ngoài chip. Các kết nối ngoài chip này thường sử dụng các giao thức phức tạp hơn và có băng thông cao để đáp ứng yêu cầu truyền tải dữ liệu lớn giữa hệ thống SoC và các thiết bị ngoại vi. Hệ thống bus và các lớp giao tiếp ngoài chip giúp mở rộng khả năng kết nối và tương tác của hệ thống, cho phép kết nối với nhiều loại thiết bị ngoại vi và bộ nhớ khác nhau.

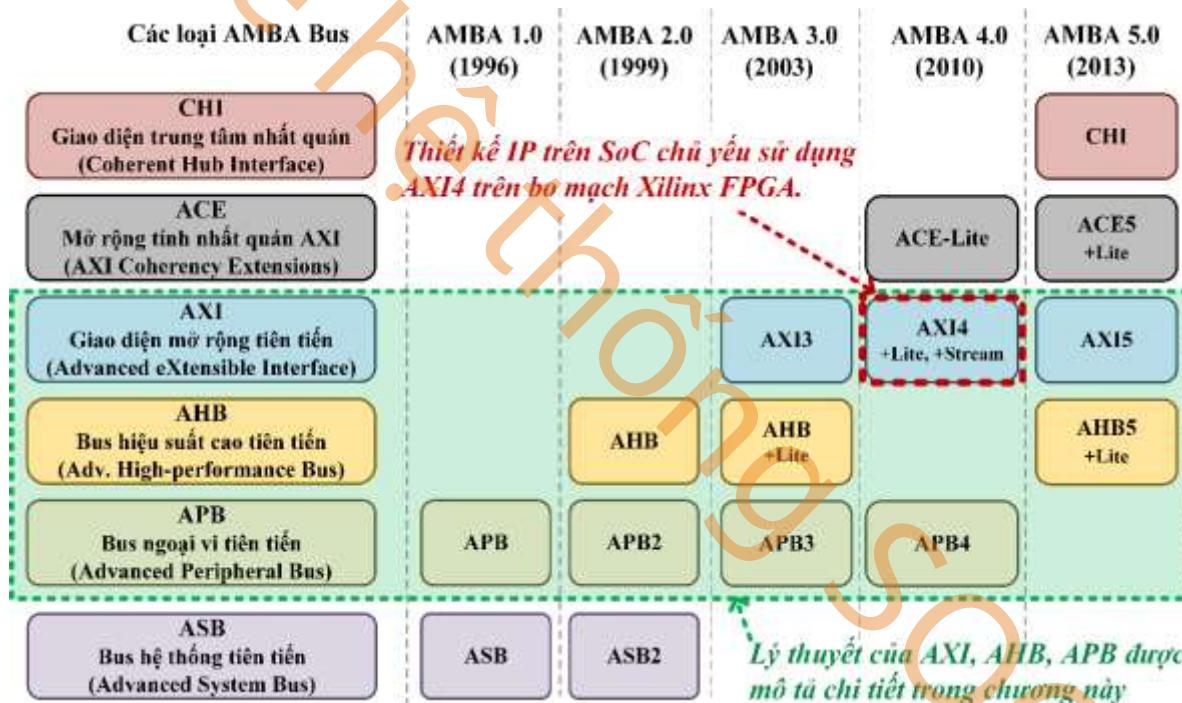
Mặc dù có nhiều tiêu chuẩn bus tồn tại, chúng chủ yếu tập trung vào kết nối ở mức bảng mạch và chừa chi phí phụ để hỗ trợ ghép kênh tín hiệu, phát hiện cấu hình, và xử lý các đặc tính điện, nhưng những chi phí này không áp dụng cho môi trường SoC. Đồng thời, cần có một tiêu chuẩn mở để cho phép tái sử dụng thiết kế tốt hơn và cho phép các nhà cung cấp IP phát triển các thiết bị ngoại vi cho nền tảng ARM. Kết quả là, ARM đã công bố đặc tả giao thức kiến trúc bus vi điều khiển nâng cao (AMBA) như một tiêu chuẩn mở (không tính phí bản quyền), và đây đã trở thành tiêu chuẩn giao diện bộ xử lý phổ biến nhất cho các bộ xử lý nhúng 32-bit. Do tính chất mở và đơn giản của nó, AMBA đã trở thành tiêu chuẩn thực tế cho giao diện bus trong kiến trúc hệ thống trên chip, với đặc tính chi phí thấp và độ trễ thấp, rất cần thiết cho các hệ thống nhúng tốc độ cao. Ngoài AMBA bus, Avalon Bus là một giao thức bus nổi bật trong các hệ thống SoC của Intel (trước đây là Altera), được thiết kế đặc biệt cho các hệ thống nhúng, giúp kết nối các bộ xử lý và các thiết bị ngoại vi một cách hiệu quả. Avalon Bus hỗ trợ các giao dịch giữa các thiết bị chủ và tớ thông qua các chu kỳ đồng bộ, tối ưu hóa việc truyền tải dữ liệu và điều phối các giao dịch, đồng thời duy trì hiệu suất cao trong các hệ thống yêu cầu băng thông lớn. Giao thức này đơn giản, dễ triển khai và được sử dụng rộng rãi trong các FPGA của Intel, đặc biệt là

trong các dòng FPGA như Cyclone, Arria và Stratix, trở thành lựa chọn phổ biến cho các hệ thống SoC của Intel nhờ cung cấp một giải pháp bus đồng bộ hiệu quả.

Chương này sẽ giới thiệu các bus trong chuẩn AMBA dành cho các Xilinx FPGA, đồng thời cũng phân tích Avalon Bus trong các hệ thống SoC của Altera FPGA.

## 4.2. Giao thức AMBA

Giao thức AMBA là một tập hợp các giao thức interconnect dùng để giao tiếp giữa các khối hoặc IP khác nhau trong thiết kế SoC. AMBA được ARM giới thiệu vào năm 1996 và các đặc tả của nó được ARM cung cấp miễn phí như một tiêu chuẩn cho việc giao tiếp trên chip. Các giao tiếp trong AMBA dựa trên giao thức chủ-tớ.



Hình 4.2: Các phiên bản và đặc tả chính của kiến trúc bus AMBA qua các thế hệ [15].

Một SoC điển hình được sử dụng trong laptop hoặc máy tính để bàn có thể bao gồm nhiều thành phần như CPU, GPU, bộ nhớ, quản lý nguồn, âm thanh, video, DSP và các bộ điều khiển được tích hợp trên một chip duy nhất. Tiêu chuẩn AMBA cho phép giao tiếp hiệu quả giữa các thành phần này, hỗ trợ phát triển thiết kế chính xác ngay từ đầu, thiết kế mô-đun, khả năng tái sử dụng, tương thích và khả năng mở rộng của một IP. Điều này giúp giảm thời gian đưa sản phẩm ra thị trường và tránh các thiết kế lại tốn kém. Ngoài ra, có

rất nhiều giải pháp IP AMBA sẵn sàng sử dụng từ các nhà cung cấp IP như Synopsys và ARM. Các giải pháp xác minh IP (VIP) có tính linh hoạt và cấu hình cao rất hiệu quả cho việc xác minh chức năng toàn diện. Giải pháp Synopsys VC AutoTestbench cho phép tích hợp và cấu hình dễ dàng, nhanh chóng cho hàng trăm cổng AMBA nhất quán và không nhất quán cũng như các VIP tương ứng.

Kể từ năm 1996, AMBA đã phát triển và hiện nay đã ở thế hệ thứ 5, như mô tả ở Hình 4.2. APB (Bus ngoại vi tiên tiến) và ASB (Bus hệ thống tiên tiến) là những giao thức bus đầu tiên của AMBA. Phiên bản AMBA 2 vào năm 1999 đã giới thiệu AHB (Bus hiệu suất cao tiên tiến). AMBA 3 được giới thiệu vào năm 2003, bao gồm giao thức AXI (Giao diện mở rộng tiên tiến). AMBA 4 đã giới thiệu giao thức ACE (Mở rộng tính nhất quán AXI) vào năm 2010 và AMBA 5 giới thiệu CHI (Giao diện trung tâm nhất quán) vào năm 2013.

#### Các Loại Bus trong AMBA:

- ✓ **ASB:** Hỗ trợ các tính năng cho các hệ thống hiệu suất cao như truyền dữ liệu theo gói (hay còn gọi là truyền theo Burst), hoạt động truyền theo đường ống và nhiều bus thiết bị chủ.
- ✓ **APB:** Giao thức băng thông thấp, tối ưu hóa cho công suất thấp và độ phức tạp thấp, dùng để hỗ trợ các thiết bị ngoại vi. APB thường được sử dụng làm giao diện giá rẻ cho các thiết bị ngoại vi không yêu cầu hiệu suất cao. Mỗi lần chuyển dữ liệu mất ít nhất 2 chu kỳ.
- ✓ **AHB:** Được thiết kế để kết nối các bộ xử lý nhúng, như lõi bộ xử lý ARM, với các thiết bị ngoại vi hiệu năng cao, bộ điều khiển truy cập trực tiếp bộ nhớ (DMA), các bộ điều khiển trên chip, và các giao diện. Đây là một bus tốc độ cao, băng thông cao với kiến trúc tách biệt các bus đọc, ghi và bus rộng. Một bus rộng 32-bit được đề xuất trong tiêu chuẩn và chiều rộng dữ liệu có thể mở rộng tới 1024 bit. Các giao dịch đồng thời nhiều chủ/tớ được hỗ trợ. Nó cũng hỗ trợ các giao dịch chế độ Burst và các giao dịch tách biệt. Tất cả các giao dịch trên AHB được tham chiếu tới một clock đơn lẻ, giúp thiết kế cấp hệ thống dễ hiểu.

- ✓ **AXI:** Giao thức interconnect điểm-điểm vượt qua những hạn chế của các giao thức bus chia sẻ. Giao thức này nhắm đến các hệ thống hiệu suất cao và tần số cao với các tính năng chính bao gồm hỗ trợ nhiều giao dịch chờ đồng thời và khả năng hoàn thành dữ liệu không theo thứ tự. Nó cho phép thực hiện các giao dịch dựa trên Burst với chỉ địa chỉ bắt đầu được phát hành, đồng thời hỗ trợ truyền dữ liệu không cần chỉnh bằng cách sử dụng strobes. Giao thức này còn cho phép các giao dịch đọc và ghi diễn ra đồng thời, cùng với việc sử dụng kết nối đường ống để hoạt động ở tốc độ cao.
- ✓ **ACE:** ACE mở rộng giao thức AXI4 với các bộ nhớ đệm phần cứng nhất quán. Giao thức ACE đảm bảo tất cả các chủ đều thấy dữ liệu chính xác cho bất kỳ vị trí địa chỉ nào của nó.
- ✓ **CHI:** CHI định nghĩa các giao diện cho kết nối các bộ xử lý hoàn toàn nhất quán. Đây là một giao thức liên lạc dạng gói với các lớp Protocol, Link và Network, hỗ trợ tần số cao và truyền dữ liệu không chặn giữa các bộ xử lý.

Chương này tập trung vào ba giao thức APB, AHB, AXI vì chúng là những giao thức phổ biến nhất trong thiết kế SoC trên FPGA. Các giao thức này thuộc kiến trúc AMBA của ARM, cung cấp giải pháp linh hoạt và tối ưu cho việc kết nối các thành phần IP trên cùng một chip, từ các bộ xử lý đến các thiết bị ngoại vi. APB được sử dụng cho các thiết bị ngoại vi yêu cầu băng thông thấp, trong khi AHB phù hợp cho các thành phần cần băng thông cao và độ trễ thấp như DMA và bộ nhớ. AXI mang lại hiệu suất cao với khả năng xử lý nhiều giao dịch đồng thời, làm cho nó lý tưởng cho các ứng dụng yêu cầu hiệu suất tính toán cao. Đối với các nhà thiết kế FPGA, hiểu rõ các giao thức này giúp tối ưu hóa hiệu năng và khả năng mở rộng của SoC.

### 4.3. Hệ thống bus APB

#### 4.3.1. Giới thiệu về hệ thống bus APB

APB là một bus đơn giản chủ yếu được sử dụng cho kết nối các thiết bị ngoại vi. APB được giới thiệu như một phần của đặc tả AMBA 2, và các chức năng của nó đã được mở rộng trong AMBA 3 và AMBA 4 để hỗ trợ các trạng thái chờ, phản hồi lỗi và các thuộc

tính truyền dữ liệu bổ sung (bao gồm hỗ trợ TrustZone). Hầu hết các hệ thống APB đều có độ rộng bus 32-bit. Mặc dù giao thức bus không có giới hạn về độ rộng bus, thông lệ chung cho các hệ thống dựa trên Arm là sử dụng bus ngoại vi 32-bit.

Mặc dù có thể kết nối trực tiếp một thiết bị ngoại vi với AHB, việc tách kết nối ngoại vi bằng APB có nhiều ưu điểm:

- ✓ Nhiều thiết kế hệ thống SoC chứa một số lượng lớn các thiết bị ngoại vi. Nếu chúng được kết nối với bus hệ thống AHB, chúng có thể làm giảm tần số tối đa của hệ thống do độ rộng tín hiệu cao và logic giải mã địa chỉ phức tạp. Việc nhóm các kết nối ngoại vi trong APB có thể giảm thiểu tác động đến hiệu suất của AHB.
- ✓ Một hệ thống con ngoại vi có thể hoạt động ở một tần số xung nhịp khác, hoặc được tắt nguồn mà không ảnh hưởng đến AHB.
- ✓ Các giao diện APB sử dụng một giao thức bus đơn giản hơn, giúp đơn giản hóa thiết kế ngoại vi cũng như giảm công sức xác minh.
- ✓ Hầu hết các thiết bị ngoại vi được thiết kế cho các bộ xử lý truyền thống có thể dễ dàng kết nối với APB vì các giao dịch APB không hoạt động theo kiểu đường ống.

#### 4.3.2. Tín hiệu và kết nối APB

**Bảng 4.1: Các tín hiệu điển hình của APB**

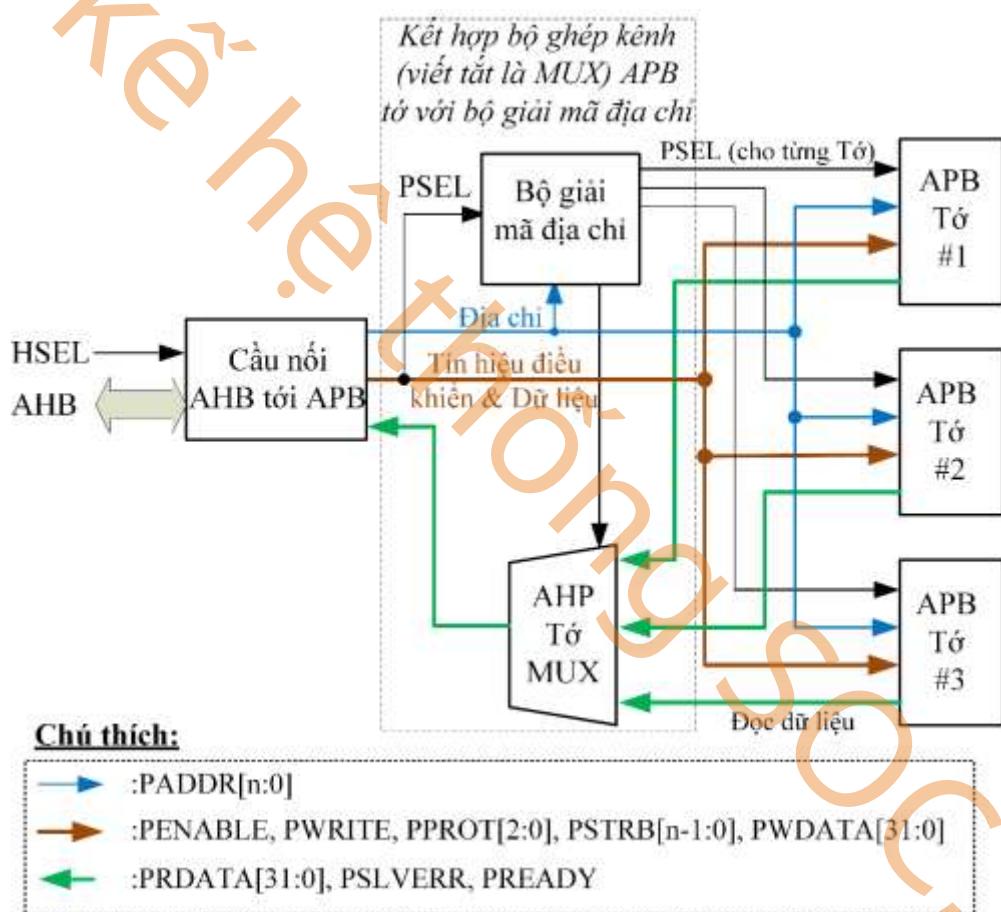
Tín hiệu	Chiều tín hiệu	Mô tả
PCLK	Nguồn xung nhịp → tất cả các khối APB	Tín hiệu xung nhịp chung
RESETn	Nguồn reset → tất cả các khối APB	Tín hiệu reset mức thấp chung
PSEL	Bộ giải mã địa chỉ → Tớ	Chọn thiết bị tớ
PADDR[n:0]	Chủ → Tớ	Bus địa chỉ (thường nhỏ hơn 32 bits)
PENABLE	Chủ → Tớ	Kiểm soát truyền dữ liệu
PWRITE	Chủ → Tớ	Kiểm soát Bảo vệ truyền dữ liệu

PPROT[2:0]	Chủ → Tớ	Điều khiển việc bảo vệ truyền dữ liệu ( <b>chỉ có ở AMBA 4</b> )
PSTRB[n-1:0]	Chủ → Tớ	Tín hiệu byte strobe cho các hoạt động ghi ( <b>chỉ có ở AMBA 4</b> )
PWDATA[31:0]	Chủ → Tớ	Dữ liệu ghi
PRDATA[31:0]	Chủ → Tớ	Dữ liệu đọc
PSLVERR	Tớ → Chủ	Phản hồi từ Tớ ( <b>từ AMBA 3 trở đi</b> )
PREADY	Tớ → Chủ	Tớ sẵn sàng ( <b>từ AMBA 3 trở đi</b> )

Trong một hệ thống APB điển hình, các tín hiệu sử dụng được mô tả trong Bảng 4.1. Cụ thể, một hệ thống APB hoạt động với một tín hiệu xung nhịp gọi là PCLK. Tín hiệu này được chia sẻ chung cho bus chủ (thường là một cầu nối AHB tới APB), bus tớ, và các khối hạ tầng bus. Tất cả các thanh ghi trên APB sẽ kích hoạt ở các cạnh lên của PCLK. Cũng có một tín hiệu reset mức thấp gọi là PRESETn. Khi tín hiệu này ở mức thấp, nó sẽ reset hệ thống APB ngay lập tức (reset không đồng bộ). Điều này cho phép hệ thống được reset ngay cả khi xung nhịp bị dừng. Tín hiệu PRESETn cần được đồng bộ hóa với PCLK để tránh các điều kiện thực hiện. Các tín hiệu còn lại như PSEL, PADDR, PENABLE, PWRITE, PWDATA, PRDATA, PSLVERR, và PREADY được sử dụng để quản lý quá trình truyền dữ liệu giữa chủ và tớ. Theo đó, khi PSEL được kích hoạt, nó chọn một thiết bị ngoại vi cụ thể để giao tiếp. Tín hiệu PADDR là địa chỉ của thiết bị ngoại vi được chọn và được gửi từ thiết bị chủ đến thiết bị tớ. Trong quá trình truyền dữ liệu, tín hiệu PENABLE được sử dụng để kiểm soát và xác nhận sự bắt đầu của quá trình truyền. Tín hiệu PWRITE xác định hướng truyền dữ liệu; nếu PWRITE ở mức cao (1), dữ liệu sẽ được ghi vào tớ, nếu PWRITE ở mức thấp (0), tớ sẽ đọc dữ liệu từ chủ. PWDATA mang dữ liệu từ chủ đến tớ trong các hoạt động ghi, trong khi PRDATA mang dữ liệu từ tớ đến chủ trong các hoạt động đọc. Các tín hiệu bổ sung như PSTRB trong AMBA 4 cung cấp thêm khả năng kiểm soát byte trong quá trình ghi dữ liệu, cho phép chỉ ghi vào các byte cụ thể thay vì toàn bộ từ dữ liệu. PPROT cung cấp các tính năng bảo vệ truyền dữ liệu, chẳng hạn như quyền truy cập và các thuộc tính bảo vệ khác. Nếu có lỗi trong quá trình truyền, PSLVERR

sẽ được kích hoạt để thông báo cho chủ về lỗi đó. Khi tớ đã sẵn sàng và quá trình truyền dữ liệu hoàn tất, tín hiệu PREADY sẽ được sử dụng để báo hiệu rằng tớ đã hoàn tất chu kỳ truyền, đảm bảo sự đồng bộ trong hệ thống.

Thông thường, APB chỉ chiếm một phần nhỏ không gian bộ nhớ. Do đó, bus địa chỉ của hệ thống APB (PADDR[n:0]) thường không quá 32 bit ( $n < 32$ ). Bên cạnh đó, không tín hiệu có điều khiển kích thước truyền trên APB. Tất cả các lần truyền được coi là 32 bit và thông thường, hai bit cuối có giá trị nhỏ nhất (bit 1 và bit 0) của PADDR không được sử dụng vì phải căn chỉnh một lần truyền từ trên APB.

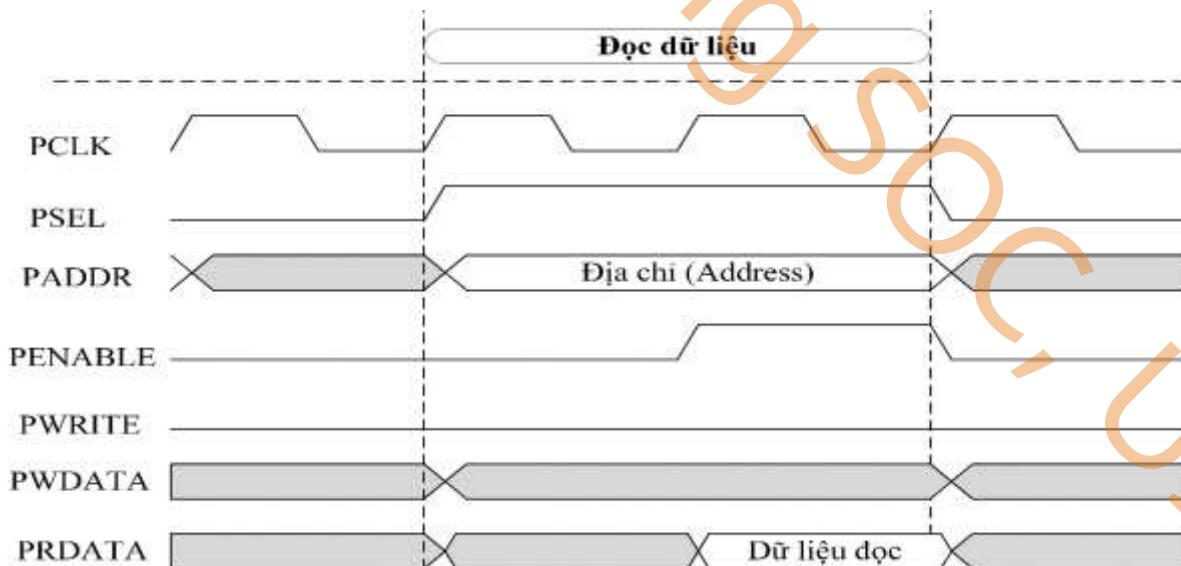


Hình 4.3: Một ví dụ về hệ thống APB [16].

Trong hầu hết các trường hợp, hệ thống APB có một cầu nối bus làm bus chủ kết nối APB với bus bộ xử lý chính (thường là AHB), như mô tả ở Hình 4.3. Ngoài ra, cần có bộ ghép kênh tớ APB và bộ giải mã địa chỉ. Bộ giải mã địa chỉ xác định thiết bị ngoại vi APB tớ nào được chọn thông qua tín hiệu PSEL, trong khi bộ ghép kênh (viết tắt là MUX) APB

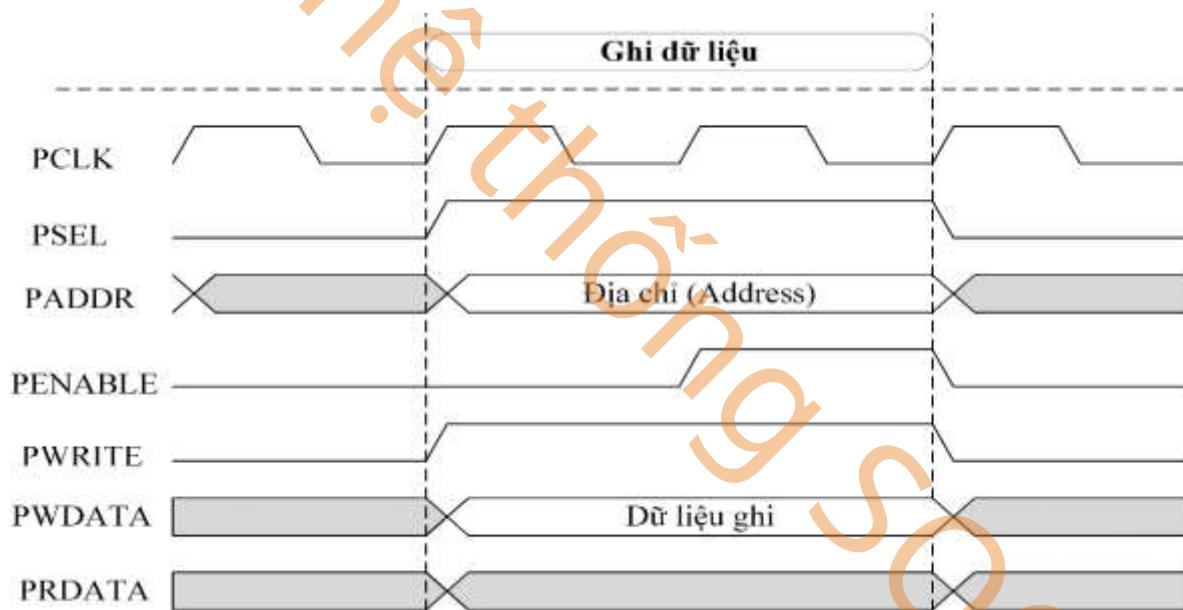
tổ quản lý việc truyền và nhận dữ liệu giữa AHB và các APB từ. Điều này giúp hệ thống quản lý kết nối nhiều thiết bị ngoại vi một cách hiệu quả, tránh xung đột. Lưu ý rằng trong các hệ thống đơn giản có cả AHB và APB, PCLK thường lấy từ cùng một nguồn xung nhịp với HCLK, và PRESETn lấy từ cùng một nguồn reset với HRESETn. Tuy nhiên, cũng có những hệ thống sử dụng các tần số HCLK và PCLK riêng biệt. Trong trường hợp đó, thiết kế cầu nối bus từ AHB đến APB cần phải có khả năng xử lý các giao dịch dữ liệu qua các tần số xung nhịp khác nhau hoặc các miền xung nhịp khác nhau. Ngoài PCLK, PRESETn, và PSEL, các **tín hiệu điều khiển & dữ liệu** đóng vai trò quan trọng trong việc truyền tải thông tin giữa AHB và các APB từ thông qua cầu nối và bộ ghép kenh. Các tín hiệu như PENABLE, PWRITE, PPROT, PSTRB, và PWDATA được sử dụng để điều khiển hoạt động ghi và quản lý dữ liệu từ chủ đến tớ. PENABLE và PWRITE xác định quá trình ghi hoặc đọc, trong khi PWDATA mang dữ liệu cần ghi từ chủ đến tớ. Khi thực hiện một hoạt động đọc, dữ liệu được các APB tớ gửi lại qua tín hiệu PRDATA, trong khi các tín hiệu như PSLVERR báo lỗi và PREADY báo hiệu khi tớ đã sẵn sàng cho một giao dịch mới. Các tín hiệu này được truyền qua AHB tớ MUX, đảm bảo dữ liệu được đọc chính xác và gửi đến AHB một cách hiệu quả.

#### **4.3.3. Chi tiết tín hiệu của APB trong việc ghi đọc dữ liệu**



Hình 4.4: So đồ thời gian chi tiết của tín hiệu đọc trong APB theo chuẩn AMBA 2.

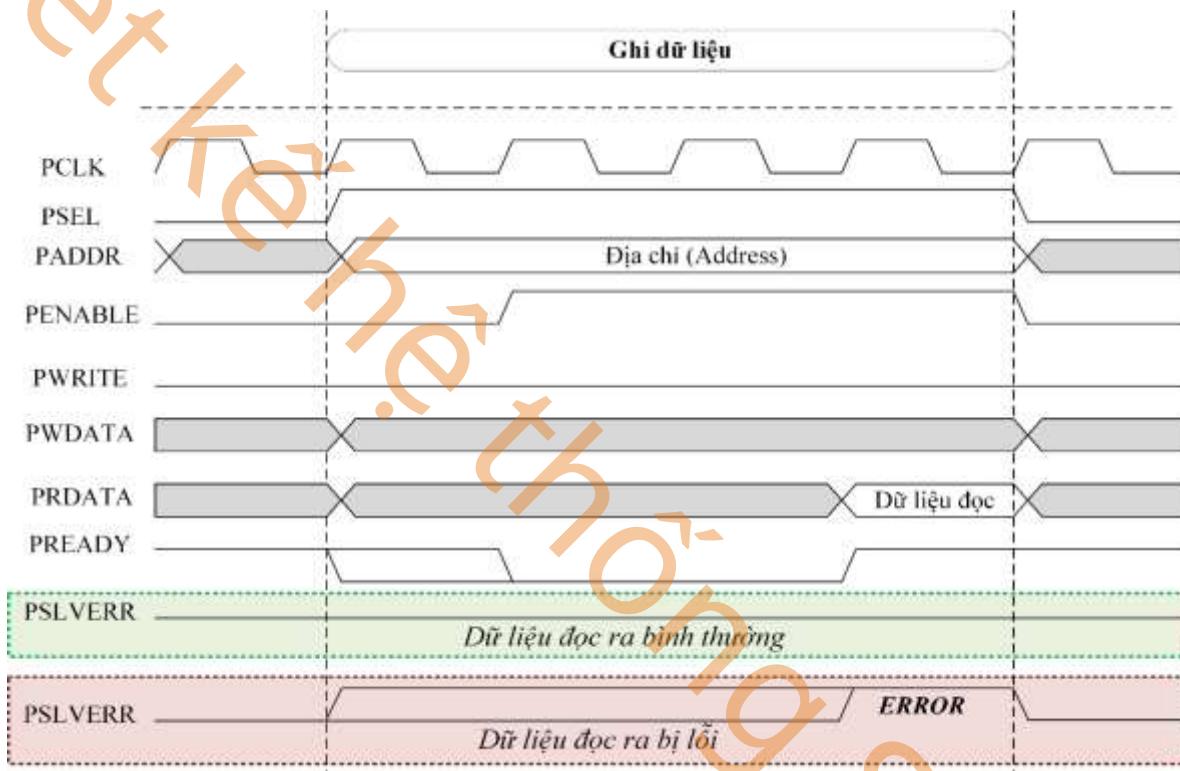
Trong AMBA 2, mỗi lần truyền dữ liệu của APB cần tối thiểu hai chu kỳ xung nhịp. Đối với các thao tác đọc, dữ liệu cần phải sẵn sàng và hợp lệ ít nhất vào cuối chu kỳ xung nhịp thứ hai, như minh họa trong Hình 4.4. Tín hiệu PCLK là xung nhịp chính điều phối toàn bộ hoạt động của APB, và tất cả các thay đổi tín hiệu đều diễn ra tại cạnh lên của PCLK. Khi PSEL được kích hoạt (mức cao), nó chọn thiết bị ngoại vi để bắt đầu giao tiếp. Sau khi PSEL được thiết lập, bus địa chỉ PADDR sẽ chỉ định địa chỉ của thiết bị ngoại vi cần truy cập. Tín hiệu PENABLE được đặt ở mức cao sau chu kỳ xung nhịp đầu tiên để xác nhận rằng quá trình truyền dữ liệu sẽ được thực hiện. Đối với hoạt động đọc, PWRITE được giữ ở mức thấp (0), chỉ định đây là một thao tác đọc. Trong trường hợp này, PRDATA mang dữ liệu từ tổ về chủ, và đến cuối chu kỳ xung nhịp thứ hai, PRDATA phải chứa giá trị hợp lệ để hoàn tất việc đọc dữ liệu.



Hình 4.5: Sơ đồ thời gian chi tiết của tín hiệu ghi trong APB theo chuẩn AMBA 2.

Trong quá trình truyền dữ liệu ghi trên APB ở AMBA2, thao tác ghi thực tế ở tổ có thể xảy ra trong chu kỳ xung nhịp đầu tiên hoặc chu kỳ thứ hai, tùy thuộc vào cách triển khai cụ thể. Do đó, chủ của APB phải đảm bảo rằng dữ liệu ghi (PWDATA) là hợp lệ trong cả hai chu kỳ xung nhịp. Ngoài ra, giữa hai lần truyền dữ liệu trên APB có thể có một số chu kỳ xung nhịp tùy ý, cho phép linh hoạt trong việc sắp xếp và điều phối các giao dịch dữ liệu. Trong sơ đồ thời gian ghi dữ liệu ở Hình 4.5, khi tín hiệu PSEL được kích hoạt, bus

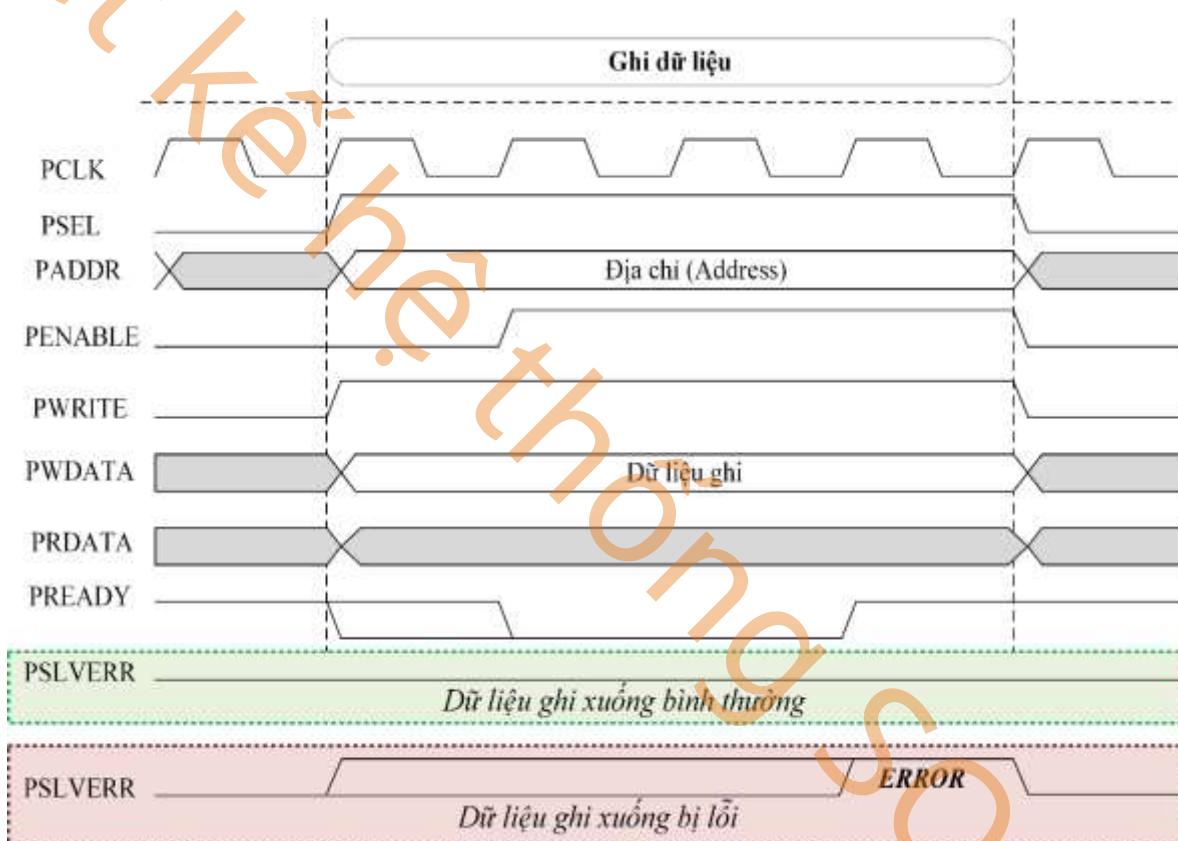
địa chỉ PADDR được thiết lập để chọn địa chỉ thiết bị ngoại vi muốn ghi. Sau khi PENABLE được kích hoạt sau chu kỳ xung nhịp đầu tiên, quá trình truyền dữ liệu sẽ diễn ra. PWRITE được đặt ở mức cao (giá trị 1) để xác định rằng đây là một hoạt động ghi. Tín hiệu PWDATA mang dữ liệu ghi từ chủ đến tớ, và dữ liệu này phải được giữ ổn định và hợp lệ trong suốt quá trình thực hiện ghi. Điều này đảm bảo rằng dữ liệu được ghi chính xác bắt kể thời điểm thực hiện ghi trong các chu kỳ xung nhịp được xác định.



Hình 4.6: Sơ đồ thời gian chi tiết của tín hiệu đọc trong APB với 3 trạng thái chờ và tín hiệu phản hồi theo chuẩn AMBA 3.

Trong chuẩn AMBA 3, mỗi APB tớ có thể kéo dài thời gian truyền dữ liệu bằng cách hủy kích hoạt tín hiệu đầu ra PREADY hoặc phản hồi với một lỗi bằng tín hiệu PSLVERR. Chẳng hạn, nếu một APB tớ cần 4 chu kỳ xung nhịp để hoàn thành một lần truyền đọc (tương đương với 3 trạng thái chờ), dạng sóng hoạt động đọc sẽ giống như trong Hình 4.6. Kết thúc quá trình truyền đọc được chỉ định khi tín hiệu PREADY được kích hoạt. Số chu kỳ tối thiểu cho một lần truyền dữ liệu APB vẫn là hai chu kỳ (tương tự như AMBA 2), và giá trị của PREADY trong chu kỳ đầu tiên của quá trình truyền sẽ bị bỏ qua. Điều này có nghĩa là, ngay cả khi giá trị của PREADY là logic 1, quá trình truyền vẫn phải mất ít nhất

hai chu kỳ. Trong ví dụ được đề cập, AHB trả phản hồi với trạng thái OKAY (được chỉ định bằng giá trị logic 0 trên PSLVERR khi PREADY là 1). Khi phản hồi lỗi được tạo ra, dữ liệu đọc từ AHB trả có thể không chứa bất kỳ thông tin hữu ích nào và có thể bị loại bỏ. Giá trị của PSLVERR chỉ hợp lệ khi PREADY ở mức cao và không phải trong chu kỳ đầu tiên của quá trình truyền. Ví dụ, nếu PSLVERR và PREADY đều ở mức cao trong chu kỳ đầu tiên của quá trình truyền, thì nó không được coi là phản hồi lỗi vì quá trình truyền APB phải có ít nhất hai chu kỳ xung nhịp.



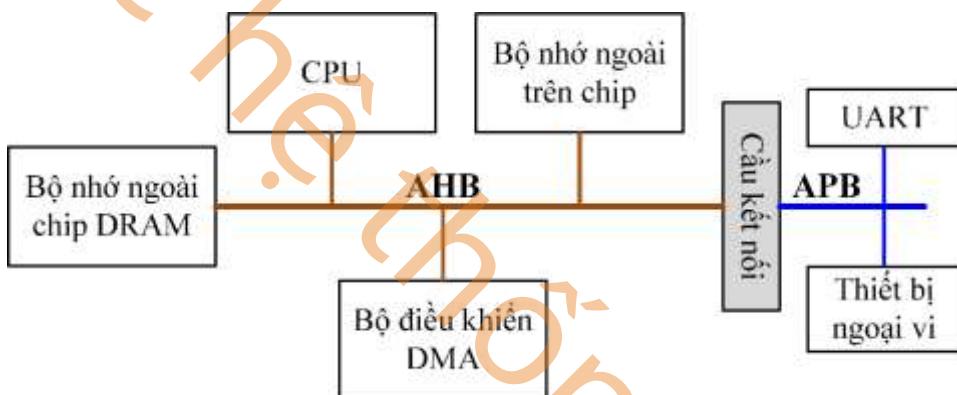
Hình 4.7: Sơ đồ thời gian chi tiết của tín hiệu ghi trong APB với 3 trạng thái chờ và tín hiệu phản hồi theo chuẩn AMBA 3.

Tương tự như quá trình đọc, trong quá trình ghi, PREADY và PSLVERR đóng vai trò quyết định trạng thái ghi thành công hay thất bại như mô tả ở Hình 4.7. Khi PREADY được kích hoạt, tín hiệu PSLVERR xác định nếu dữ liệu ghi xuống là hợp lệ hoặc gấp lỗi (ERROR). Nếu PSLVERR ở mức thấp, dữ liệu được ghi thành công; nếu PSLVERR ở mức cao, ghi dữ liệu thất bại và có lỗi xảy ra.

Trong chuẩn AMBA 4, APB hoạt động tương tự như trong AMBA 3, nhưng được bổ sung thêm một số tín hiệu để cải thiện tính năng và độ an toàn của truyền dữ liệu. Cụ thể, tín hiệu PPROT[2:0] được thêm vào để điều khiển việc bảo vệ truyền dữ liệu, đảm bảo rằng chỉ các giao dịch có quyền hợp lệ mới được thực hiện. Bên cạnh đó, tín hiệu PSTRB[n-1:0] cũng được giới thiệu để cung cấp tín hiệu byte strobe cho các hoạt động ghi, cho phép kiểm soát chi tiết hơn đối với các byte được ghi vào thiết bị ngoại vi, giúp tối ưu hóa và linh hoạt hơn trong việc xử lý dữ liệu.

#### 4.4. Hệ thống bus AHB

##### 4.4.1. Giới thiệu về hệ thống bus AHB



Hình 4.8: Một hệ thống điển hình dựa trên bus AMBA dùng AHB và APB [16].

Hầu hết các hệ thống dựa trên AHB đều sử dụng bus 32-bit, nhưng giao thức này được thiết kế để hỗ trợ các kích thước bus khác nhau. Độ rộng bus điển hình cho các hệ thống AHB là 32 hoặc 64-bit. Hình 4.8 mô tả một hệ thống điển hình sử dụng kiến trúc bus AMBA. Theo đó, AHB hình thành xương sống của hệ thống trên đó bộ xử lý ARM, giao diện bộ nhớ băng thông cao và bộ nhớ truy cập ngẫu nhiên (RAM), và bộ điều khiển truy cập bộ nhớ trực tiếp (DMA). Giao diện giữa bus AHB và bus APB chậm hơn là thông qua một module cầu kết nối. Không giống như các kiến trúc bus được thiết kế cho các hệ thống dựa trên PCB, AMBA AHB tránh việc triển khai tristate bằng cách sử dụng một sơ đồ kết nối chuyển mạch trung tâm. Phương pháp kết nối này cung cấp hiệu suất cao hơn và công suất thấp hơn so với việc sử dụng các bộ đệm tristate. Tất cả các chủ xác nhận các tín hiệu địa chỉ và điều khiển, chỉ ra loại truyền mà mỗi chủ yêu cầu. Một bộ giải mã trung tâm xác

định chủ nào có địa chỉ và tín hiệu điều khiển định tuyến tới tất cả các thiết bị tớ. Một mạch giải mã trung tâm chọn dữ liệu đọc thích hợp và tín hiệu phản hồi xác nhận từ tớ có liên quan tham gia vào giao dịch.

Tiêu chuẩn AHB đã trải qua nhiều lần phát hành và giai đoạn khác nhau:

- ✓ AMBA 2 AHB - phiên bản đầu tiên: Phiên bản này sử dụng một cặp tín hiệu bắt tay để phân giải xung đột giữa nhiều bus chủ.
- ✓ Thiết kế AHB đa lớp: Sản phẩm bộ thiết kế AMBA từ ARM đã giới thiệu một thành phần kết nối AHB gọi là ma trận AHB Bus. Thiết kế này cho phép truyền dữ liệu đồng thời trên bus trong các hệ thống nhiều chủ để tăng băng thông, loại bỏ sự cần thiết của các tín hiệu yêu cầu và cấp quyền bus, và không chính thức được gọi là AHB Lite.
- ✓ Trong đặc tả AMBA 3, AHB Lite trở thành tên chính thức. Phiên bản này đã loại bỏ các tín hiệu Bus Request và Bus Granted và đơn giản hóa một số khía cạnh khác của giao thức AHB. Nó được sử dụng rộng rãi trong nhiều hệ thống vi xử lý ARM.
- ✓ Trong đặc tả AMBA 5, AHB đã được cập nhật để hỗ trợ TrustZone cho Armv8-M và bổ sung hỗ trợ chính thức cho các tín hiệu sideband truy cập độc quyền. Nó cũng giới thiệu một số cải tiến khác, bao gồm hỗ trợ thêm cho các thuộc tính bộ nhớ cache và các giải thích chi tiết hơn.

AHB 5 (hay còn gọi là AHB5) là phiên bản phát hành mới nhất của đặc tả AHB. Trong nhiều trường hợp, nó rất tương thích với phiên bản tiền nhiệm của nó và các thiết kế bus tớ hiện có được thiết kế cho AHB Lite có thể được tái sử dụng trong các hệ thống AHB5.

#### 4.4.2. Tín hiệu và kết nối AHB

Hệ thống AHB hoạt động với tín hiệu xung nhịp gọi là HCLK. Tín hiệu này là chung cho tất cả các bus chủ, bus tớ và các khối cơ sở hạ tầng bus trong một phân đoạn bus. Tất cả các thanh ghi trên AHB kích hoạt tại các cạnh tăng của HCLK. Ngoài ra còn có một tín hiệu đặt lại hoạt động ở mức thấp gọi là HRESETn. Khi tín hiệu này ở mức thấp, nó sẽ đặt lại hệ thống AHB ngay lập tức (đặt lại không đồng bộ). Điều này cho phép hệ thống được đặt lại ngay cả khi xung nhịp đã dừng. Để hoạt động chính xác, bắn thân tín hiệu HRESETn

phải được đồng bộ hóa với HCLK để có thể tránh được tình trạng chạy đua. Mặt khác, nếu HRESETn hủy xác nhận vào cùng thời điểm với cạnh tăng của HCLK, bạn có thể thấy rằng một số phần của thanh ghi vẫn được đặt lại tại cạnh xung nhịp và một số thì không.

**Bảng 4.2: Các tín hiệu điển hình của AHB**

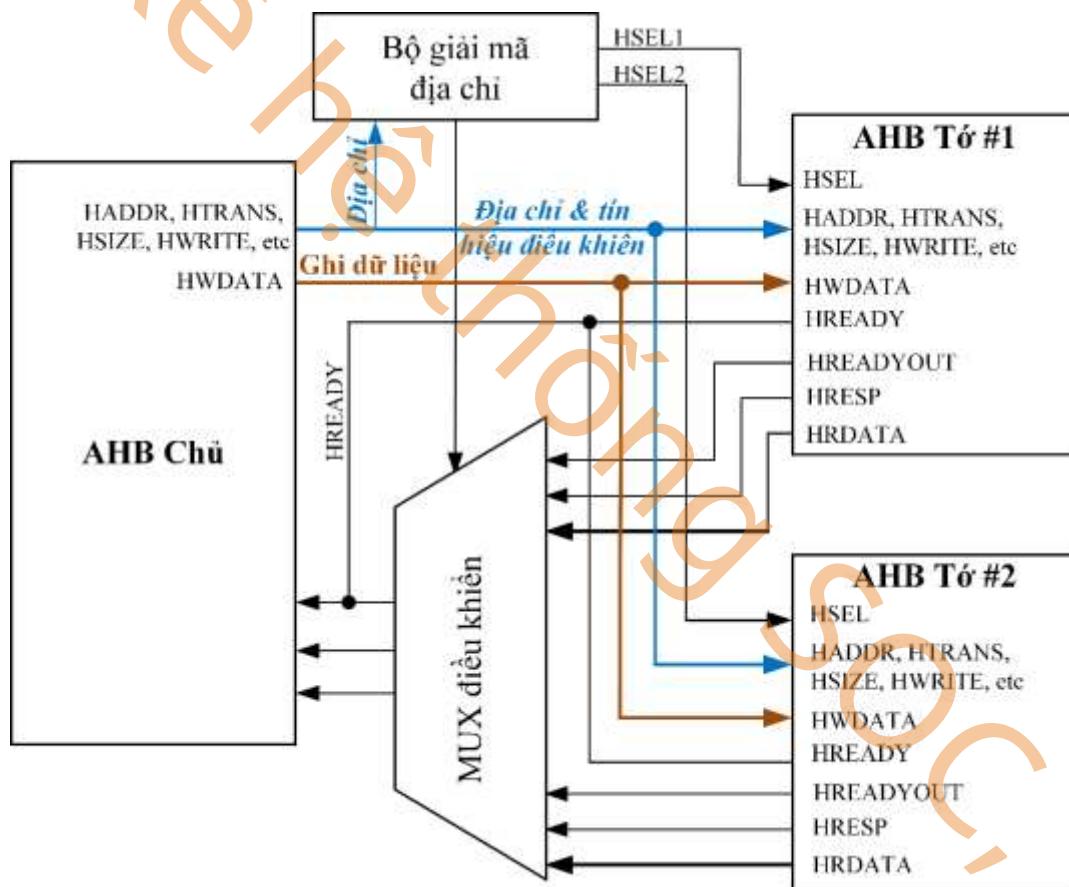
Tín hiệu	Chiều tín hiệu	Mô tả
HCLK	Nguồn xung nhịp → tất cả các khối AHB	Tín hiệu xung nhịp chung
HRESETn	Nguồn reset → tất cả các khối AHB	Tín hiệu reset mức thấp chung
HSEL	Bộ giải mã địa chỉ → Tớ	Chọn thiết bị tớ
HADDR[31:0]	Chủ → Tớ	Bus địa chỉ
HTRANS[1:0]	Chủ → Tớ	Kiểm soát truyền dữ liệu
HWRITE	Chủ → Tớ	Kiểm soát ghi (1=Ghi, 0=Đọc)
HSIZE[2:0]	Chủ → Tớ	Kiểm soát kích thước truyền dữ liệu
HBURST[2:0]	Chủ → Tớ	Kiểm soát truyền dữ liệu kiểu Burst
HPROT[3:0] / HPROT[6:0]	Chủ → Tớ	Kiểm soát bảo vệ truyền dữ liệu (4 bit trong AHB-Lite và 7 bit trong AHB5)
HMASTLOCK	Chủ → Tớ	Kiểm soát khóa truyền dữ liệu
HMASTER[3:0]	Thành phần bus → Bus tớ	Xác định bus chủ hiện tại
HWDATA[31:0]	Chủ → Tớ	Dữ liệu ghi (thường là 32-bit, nhưng có thể là 64-bit trên hệ thống 64-bit)
HRDATA[31:0]	Tớ → Chủ	Dữ liệu đọc (thường là 32-bit, nhưng có thể là 64-bit trên hệ thống 64-bit)
HRESP[n:0] / HRESP	Tớ → Chủ	Phản hồi từ tớ (độ rộng 2 bit trong AMBA 2, 1 bit trong AHB-Lite/AHB5)

HREADY (HREADYOUT)	Tớ → Chủ (HREADYOUT), Thành phần bus → các tớ khác (HREADY)	Tớ sẵn sàng (truyền dữ liệu hoàn tất). Tớ được chọn hiện tại sẽ truyền tín hiệu HREADY đến chủ và tắt cả các AHB tớ khác. Do đó, một AHB tớ có đầu vào HREADY và đầu ra HREADYOUT.
-----------------------	--	--

Đối với một hệ thống AHB thông thường, ta có thể tìm thấy hầu hết các tín hiệu được mô tả trong Bảng 4.2. Trong hệ thống AHB, các tín hiệu điều khiển và dữ liệu được sử dụng để quản lý quá trình truyền dữ liệu giữa chủ và tớ. HADDR[31:0] là bus địa chỉ, được gửi từ chủ đến tớ để chỉ định địa chỉ mục tiêu trong quá trình truyền dữ liệu. Tín hiệu HTRANS[1:0] được sử dụng để kiểm soát việc truyền dữ liệu, xác định loại giao dịch (như IDLE, BUSY, NONSEQUENTIAL, SEQUENTIAL), trong khi HWRITE xác định hướng truyền dữ liệu (1 cho ghi và 0 cho đọc). HSIZE[2:0] chỉ định kích thước của mỗi lần truyền (byte, halfword, word, v.v.), và HBURST[2:0] kiểm soát loại Burst trong quá trình truyền dữ liệu, cho phép xử lý hiệu quả các chuỗi truyền dữ liệu liên tiếp. HPROT là tín hiệu bảo vệ, cung cấp thông tin về loại truy cập và mức độ bảo vệ dữ liệu, giúp tăng cường tính bảo mật và toàn vẹn dữ liệu, đặc biệt với các ứng dụng yêu cầu bảo mật cao. HMASTER[3:0] chỉ ra chủ hiện tại đang kiểm soát bus và tham gia vào giao dịch, giúp quản lý việc truy cập bus trong hệ thống đa chủ. Trong khi đó, HREADY và HREADYOUT được sử dụng để chỉ ra rằng tớ đã sẵn sàng và việc truyền dữ liệu đã hoàn tất, cho phép chủ biết khi nào có thể bắt đầu giao dịch tiếp theo. Các tín hiệu này cùng hoạt động để đảm bảo tính đồng bộ và hiệu quả trong các giao dịch trên bus AHB. Cuối cùng, các tín hiệu HWDATA[31:0] và HRDATA[31:0] được sử dụng để truyền dữ liệu ghi và đọc giữa chủ và tớ, trong khi HRESP báo cáo trạng thái của giao dịch (OKAY, ERROR, RETRY, SPLIT), cung cấp thông tin phản hồi về kết quả của các giao dịch để xử lý lỗi kịp thời.

Thiết kế đơn giản với một bus chủ (ví dụ: bộ xử lý Cortex-M) và nhiều bus tớ được mô tả như Hình 4.9. Trong hệ thống này, AHB Chủ gửi các tín hiệu địa chỉ và điều khiển như HADDR, HTRANS, HSIZE, HWRITE, và tín hiệu dữ liệu ghi HWDATA đến bộ giải mã địa chỉ và bộ MUX điều khiển. Bộ giải mã địa chỉ nhận tín hiệu địa chỉ từ chủ và giải

mã để xác định từ nào sẽ được chọn cho giao dịch hiện tại bằng cách tạo ra các tín hiệu chọn từ như HSEL1, HSEL2. Các tín hiệu địa chỉ và điều khiển này sau đó được định tuyến đến các từ thông qua bộ MUX điều khiển. Hai AHB Tờ trong hệ thống (Tờ #1 và Tờ #2) nhận các tín hiệu đã giải mã, bao gồm các tín hiệu địa chỉ, điều khiển, và dữ liệu từ AHB chủ. Mỗi tờ sẽ phản hồi bằng cách gửi dữ liệu đọc (HRDATA) và phản hồi trạng thái (HRESP, HREADY) trở lại qua bộ MUX điều khiển. Bộ MUX điều khiển sau đó sẽ đã hợp các phản hồi từ các tờ và gửi chúng về cho chủ. Thiết kế này cho phép AHB chủ giao tiếp với nhiều AHB tờ trên cùng một bus, quản lý giao dịch thông qua các tín hiệu địa chỉ, điều khiển, và dữ liệu, đồng thời xử lý các phản hồi từ các tờ một cách hiệu quả.



Hình 4.9: Hệ thống AHB đơn giản với một AHB chủ và hai AHB tờ [16].

Các tín hiệu trong hệ thống AHB được chia thành hai nhóm: tín hiệu pha địa chỉ và tín hiệu pha dữ liệu.

#### Tín hiệu Pha Địa Chỉ:

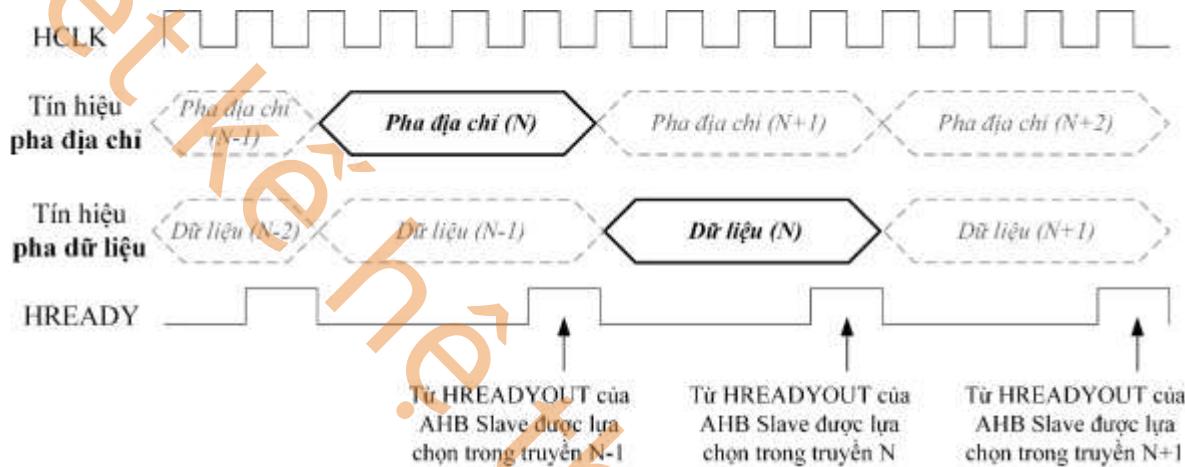
- ❖ HADDR, HTRANS, HSEL, HWRITE, HSIZE: Đây là các tín hiệu bắt buộc dùng để điều khiển quá trình truyền dữ liệu và địa chỉ trên bus AHB.
- ❖ Tùy chọn:
  - ✓ HPROT: Tín hiệu bảo vệ giao dịch, cung cấp thông tin về tính chất của giao dịch như quyền ưu tiên, người dùng không đặc quyền, và bảo vệ bộ nhớ.
  - ✓ HBURST: Tín hiệu chỉ định kiểu truyền Burst, được sử dụng để tối ưu hóa hiệu suất cho các giao dịch dữ liệu liên tiếp.
  - ✓ HMASTLOCK: Tín hiệu kiểm soát khóa truyền dữ liệu, chỉ có trong AMBA 2 AHB, cho phép khóa bus cho một chủ nhất định trong suốt quá trình truyền.
  - ✓ HEXCL: Tín hiệu chỉ ra rằng giao dịch hiện tại là quyền truy cập độc quyền, chỉ có trong AMBA 5 AHB, được sử dụng trong các hệ thống có nhiều chủ để kiểm soát quyền truy cập duy nhất.
  - ✓ HAUSER: Dải tần người dùng tùy chọn cho tín hiệu pha địa chỉ, chỉ có trong AMBA 5 AHB, cho phép truyền các thông tin bổ sung do người dùng định nghĩa.

#### **Tín hiệu Pha Dữ Liệu:**

- ❖ HWDATA, HRDATA, HRESP, HREADY, HREADYOUT: Đây là các tín hiệu chính được sử dụng trong pha dữ liệu của giao dịch. HWDATA và HRDATA truyền dữ liệu viết và đọc, trong khi HRESP, HREADY, và HREADYOUT xác định trạng thái và độ sẵn sàng của các giao dịch.
- ❖ Tùy chọn:
  - ✓ HEXOKAY: Tín hiệu phản hồi thành công truy cập độc quyền, chỉ có trong AMBA 5 AHB, dùng để xác nhận rằng giao dịch độc quyền đã được thực hiện thành công.
  - ✓ HWUSER/HRUSER: Dải tần người dùng tùy chọn cho tín hiệu pha dữ liệu, chỉ có trong AMBA 5 AHB, cho phép truyền thông tin bổ sung tùy theo yêu cầu của người dùng trong suốt quá trình truyền dữ liệu.

Mỗi lần truyền dữ liệu trong AHB bao gồm hai pha: pha địa chỉ và pha dữ liệu. Các lần truyền này được thực hiện theo kiểu pipelined, cho phép pha địa chỉ của một lần truyền

có thể chồng lên pha dữ liệu của lần truyền trước đó. Mỗi pha được kết thúc khi tín hiệu HREADYOUT (HREADY) từ AHB tách hiện tại trong pha dữ liệu được kích hoạt. Tín hiệu HREADYOUT từ các AHB tách được ghép kênh bởi bộ ghép kênh tách, tạo thành tín hiệu HREADY cho toàn hệ thống. Bộ ghép kênh này hoạt động trong pha dữ liệu của mỗi lần truyền. Việc điều khiển bộ ghép kênh có thể được tạo ra từ bộ giải mã AHB, hoặc từ các tín hiệu HSEL và HREADY.



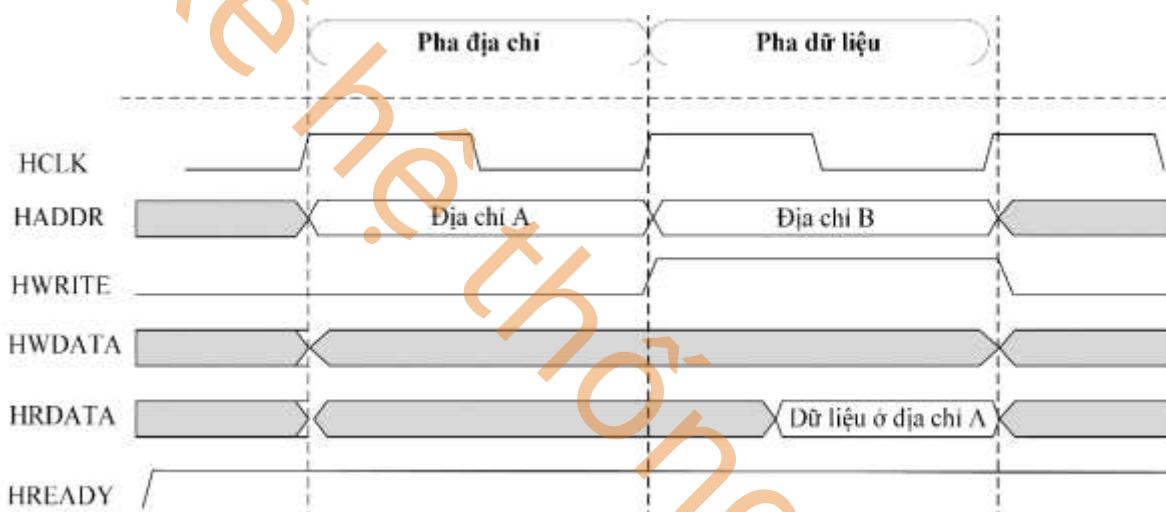
Hình 4.10: Phân chia quá trình truyền dữ liệu thành giai đoạn địa chỉ và dữ liệu.

Trong Hình 4.10, ta thấy rõ quá trình phân chia và phối hợp giữa pha địa chỉ và pha dữ liệu trong các lần truyền dữ liệu. Trong khi pha địa chỉ của một lần truyền mới (N) bắt đầu, pha dữ liệu của lần truyền trước đó (N-1) vẫn đang diễn ra. Mỗi lần truyền dữ liệu được kết thúc bằng tín hiệu HREADYOUT từ AHB Tách được chọn, báo hiệu kết thúc pha dữ liệu và cho phép hệ thống bắt đầu pha địa chỉ của lần truyền tiếp theo.

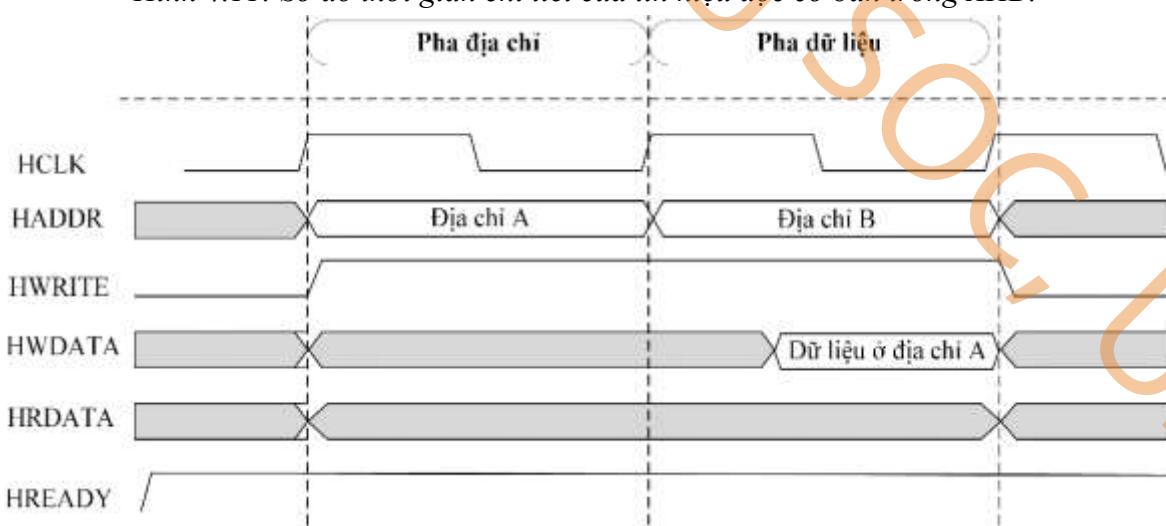
#### 4.4.3. Chi tiết tín hiệu của AHB trong việc ghi đọc dữ liệu cơ bản

Phần này trình bày chi tiết về các tín hiệu của AHB trong quá trình ghi và đọc dữ liệu cơ bản, bao gồm cách thức hoạt động và sự phối hợp giữa các tín hiệu chính như HCLK, HADDR, HWRITE, HWDATA, HRDATA, và HREADY, như mô tả ở Hình 4.11 và 5.12. Quá trình truyền dữ liệu trong AHB bao gồm hai pha: pha địa chỉ và pha dữ liệu. Pha địa chỉ kéo dài trong một chu kỳ của HCLK, trừ khi bị kéo dài bởi lần truyền trước đó, trong khi pha dữ liệu có thể yêu cầu nhiều chu kỳ HCLK. Tín hiệu HREADY được sử dụng để kiểm soát số chu kỳ cần thiết để hoàn thành quá trình truyền dữ liệu. Tín hiệu HWRITE

điều khiển hướng truyền dữ liệu đến hoặc từ Chủ; khi HWRITE ở mức cao, đó là lần truyền ghi và Chủ sẽ gửi dữ liệu lên bus dữ liệu ghi, HWDATA, còn khi HWRITE ở mức thấp, đó là lần truyền đọc và Tớ phải tạo dữ liệu trên bus dữ liệu đọc HRDATA. Quá trình truyền đơn giản nhất là không có trạng thái chờ, bao gồm một chu kỳ địa chỉ và một chu kỳ dữ liệu. Ví dụ này cho thấy cách các pha địa chỉ và dữ liệu của quá trình truyền xảy ra trong các chu kỳ xung nhịp khác nhau, với pha địa chỉ của bất kỳ quá trình truyền nào cũng diễn ra trong pha dữ liệu của lần truyền trước đó. Sự chồng chéo này cho phép hoạt động hiệu suất cao trong cấu trúc đường ống của bus, đồng thời cung cấp đủ thời gian để Tớ đưa ra phản hồi cho một lần truyền.

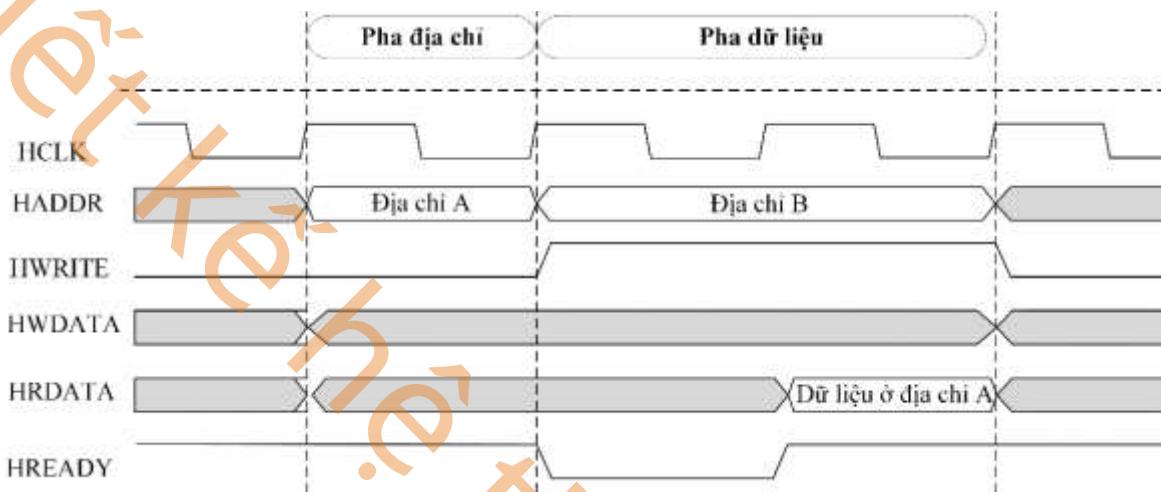


Hình 4.11: Sơ đồ thời gian chi tiết của tín hiệu đọc cơ bản trong AHB.



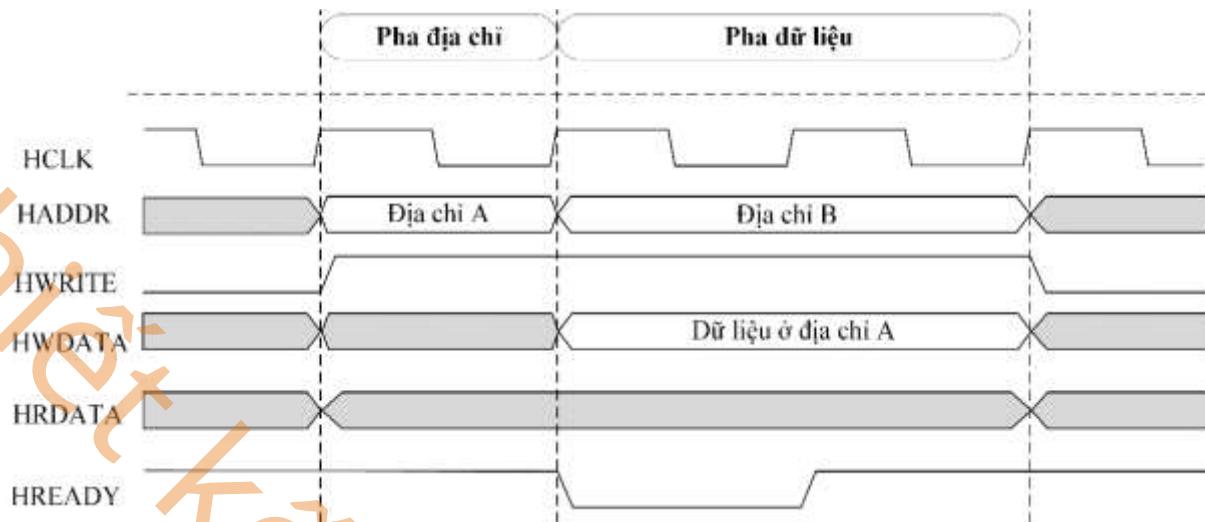
Hình 4.12: Sơ đồ thời gian chi tiết của tín hiệu ghi cơ bản trong AHB.

Một Tờ có thể chèn các trạng thái chờ vào bất kỳ lần truyền dữ liệu nào để có thêm thời gian hoàn thành. Mỗi Tờ có một tín hiệu HREADYOUT mà nó điều khiển trong pha dữ liệu của một lần truyền. Kết nối liên lạc chịu trách nhiệm kết hợp các tín hiệu HREADYOUT từ tất cả các Tờ để tạo ra một tín hiệu HREADY duy nhất dùng để kiểm soát tiến trình tổng thể.

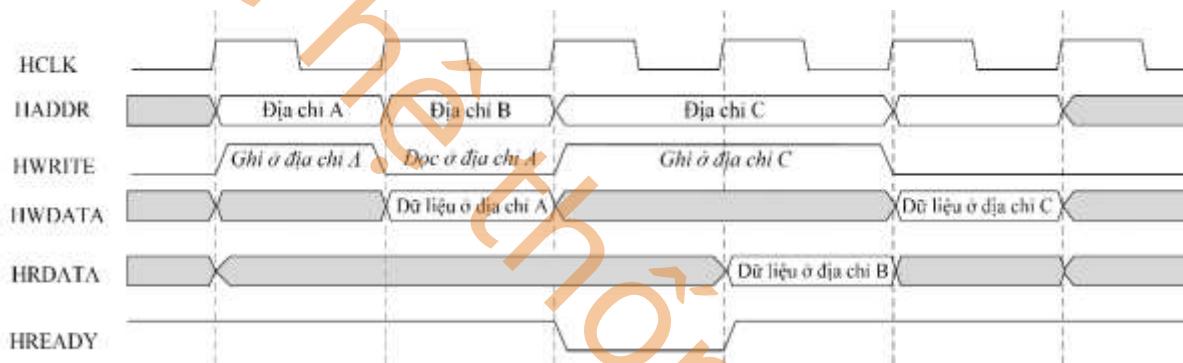


Hình 4.13: Sơ đồ thời gian chi tiết của tín hiệu đọc cơ bản với hai trạng thái chờ.

Trong Hình 4.13 và Hình 4.14, sơ đồ thời gian minh họa chi tiết các tín hiệu trong quá trình đọc và ghi dữ liệu cơ bản trên bus AHB khi Tờ chèn hai trạng thái chờ. Trong Hình 4.13, quá trình đọc dữ liệu diễn ra như sau: HADDR chỉ định địa chỉ của dữ liệu cần đọc trong pha địa chỉ (địa chỉ A). Tín hiệu HWRITE ở mức thấp cho thấy đây là một quá trình đọc. Sau đó, tín hiệu HREADY ở mức thấp trong hai chu kỳ xung nhịp, biểu thị hai trạng thái chờ. Trong thời gian này, Tờ chưa sẵn sàng cung cấp dữ liệu. Khi HREADY lên mức cao, HRDATA sẽ chứa dữ liệu tại địa chỉ A, hoàn thành quá trình đọc. Tương tự, Hình 4.14 mô tả quá trình ghi dữ liệu với hai trạng thái chờ. Trong pha địa chỉ, HADDR xác định địa chỉ đích (địa chỉ A) và HWRITE được đặt ở mức cao để chỉ định quá trình ghi. Dữ liệu cần ghi sẽ được chuẩn bị trên HWDATA trong pha dữ liệu. Tín hiệu HREADY cũng được đặt ở mức thấp trong hai chu kỳ xung nhịp, biểu thị hai trạng thái chờ. Sau khi HREADY lên mức cao, dữ liệu tại HWDATA sẽ được ghi vào địa chỉ A, kết thúc quá trình ghi.



Hình 4.14: Sơ đồ thời gian chi tiết của tín hiệu ghi cơ bản với hai trạng thái chờ.



Hình 4.15: Sơ đồ thời gian chi tiết của tín hiệu kết hợp đọc và ghi cơ bản trong AHB.

Khi một quá trình truyền dữ liệu được kéo dài, nó sẽ kéo dài pha địa chỉ của lần truyền tiếp theo. Hình 4.15 minh họa ba lần truyền đến ba địa chỉ không liên quan: A, B, và C, với pha địa chỉ mở rộng cho địa chỉ C. Trong sơ đồ này, các lần truyền đến địa chỉ A và C không có trạng thái chờ (zero wait state), trong khi lần truyền đến địa chỉ B có một trạng thái chờ. Việc kéo dài pha dữ liệu của lần truyền đến địa chỉ B làm kéo dài pha địa chỉ của lần truyền đến địa chỉ C. Điều này được thể hiện rõ ràng khi pha địa chỉ của địa chỉ C bắt đầu sau khi pha dữ liệu của địa chỉ B kết thúc. Điều này minh họa cách mà tín hiệu HREADY và các trạng thái chờ có thể ảnh hưởng đến các pha địa chỉ và dữ liệu liên tiếp trong hệ thống bus AHB.

#### 4.4.4. Chi tiết tín hiệu của AHB trong việc ghi đọc dữ liệu với Burst

Truyền dữ liệu cơ bản trong AHB thường đơn giản trong thiết kế và dễ dàng triển khai cho các ứng dụng không yêu cầu băng thông cao. Tuy nhiên, nhược điểm của nó là không hiệu quả khi cần truyền một lượng lớn dữ liệu liên tiếp, vì mỗi lần truyền dữ liệu yêu cầu một chu kỳ địa chỉ và một chu kỳ dữ liệu riêng biệt, dẫn đến thời gian chờ giữa các lần truyền dài và sử dụng băng thông kém hiệu quả. Để khắc phục những hạn chế này, truyền Burst được sử dụng, cho phép truyền nhiều dữ liệu liên tiếp trong một chuỗi mà chỉ cần thiết lập địa chỉ một lần. Điều này giúp tối ưu hóa băng thông và tăng hiệu suất truyền dữ liệu, đặc biệt là trong các ứng dụng yêu cầu tốc độ cao như truyền dữ liệu từ bộ nhớ đến bộ xử lý hoặc các thiết bị ngoại vi tốc độ cao.

**Bảng 4.3: Mã hóa tín hiệu Burst**

HBURST[2:0]	Loại	Mô tả
000	SINGLE	Truyền đơn Burst
001	INCR	Truyền Burst gia tăng với độ dài không xác định
010	WRAP4	Truyền Burst vòng 4 nhịp
011	INCR4	Truyền Burst tăng dần 4 nhịp
100	WRAP8	Truyền Burst vòng 8 nhịp
101	INCR8	Truyền Burst gia tăng 8 nhịp
110	WRAP16	Truyền Burst vòng 16 nhịp
111	INCR16	Truyền Burst gia tăng 16 nhịp

Truyền Burst cho phép truyền nhiều dữ liệu liên tiếp trong một chuỗi, giúp tối ưu hóa băng thông và hiệu suất truyền dữ liệu trên bus. Trong giao thức AHB, các Burst có độ dài 4, 8, 16 nhịp, các Burst có độ dài không xác định, và các lần truyền đơn lẻ đều được định nghĩa. Giao thức này hỗ trợ các Burst gia tăng (incrementing) và các Burst vòng (wrapping). Các Burst gia tăng truy cập các vị trí địa chỉ liên tiếp, và địa chỉ của mỗi lần truyền trong Burst là một phần tăng của địa chỉ trước đó. Trong khi đó, Burst vòng sẽ quay lại khi chúng vượt qua một ranh giới địa chỉ. Ranh giới địa chỉ được tính toán như là tích của số lượng nhịp trong một Burst và kích thước của lần truyền. Số lượng nhịp được kiểm soát bởi tín hiệu HBURST và kích thước truyền dữ liệu được kiểm soát bởi HSIZE. Ví dụ,

một Burst vòng có 4 nhịp, mỗi nhịp là một từ (4 byte), sẽ vòng lại tại các ranh giới 16 byte. Do đó, nếu địa chỉ bắt đầu của Burst là 0x34, nó sẽ bao gồm bốn lần truyền đến các địa chỉ 0x34, 0x38, 0x3C và 0x30. Tín hiệu HBURST[2:0] điều khiển loại Burst, như được liệt kê trong Bảng 4.3.

Các Chủ không nên khởi đầu một Burst gia tăng vượt qua ranh giới địa chỉ 1KB. Chủ có thể thực hiện các lần truyền đơn lẻ bằng cách sử dụng SINGLE transfer Burst hoặc Burst có độ dài không xác định nhưng chỉ có một nhịp. Kích thước Burst cho biết số nhịp trong Burst chứ không phải số byte được truyền. Để tính tổng lượng dữ liệu được truyền trong một Burst, ta nhân số nhịp với lượng dữ liệu trong mỗi nhịp, như được chỉ định bởi HSIZE[2:0]. Tất cả các lần truyền trong một Burst phải được căn chỉnh với ranh giới địa chỉ bằng với kích thước của lần truyền. Ví dụ, các lần truyền từ phải căn chỉnh với các ranh giới từ (HADDR[1:0] = 0b00), và các lần truyền halfword phải căn chỉnh với các ranh giới halfword (HADDR[0] = 0).



Hình 4.16: Sơ đồ thời gian chi tiết của tín hiệu kết hợp đọc với Burst vòng 4 nhịp.

Hình 4.16 mô tả một lần truyền ghi sử dụng Burst vòng 4 nhịp với một trạng thái chờ được thêm vào cho lần truyền đầu tiên. Do đây là một Burst vòng 4 nhịp cho các lần truyền

dữ liệu kiểu từ, địa chỉ sẽ vòng lại tại các biên 16 byte. Cụ thể, lần truyền đến địa chỉ 0x3C sẽ được sau bởi lần truyền đến địa chỉ 0x30. Các trạng thái của HTRANS[1:0] được mô tả chi tiết trong Bảng 4.4.

**Bảng 4.4: Mã hóa loại truyền dữ liệu**

HTRANS[1:0]	Loại	Mô tả
00	IDLE	Chỉ ra rằng không cần truyền dữ liệu. Manager sử dụng truyền IDLE khi nó không muốn thực hiện truyền dữ liệu. Nên kết thúc một truyền bị khóa bằng truyền IDLE. Subordinate luôn phải cung cấp phản hồi OKAY trạng thái chờ bằng không (zero wait state) cho truyền IDLE và bỏ qua truyền này.
01	BUSY	Loại truyền BUSY cho phép Manager chèn các chu kỳ nhàn rỗi vào giữa một Burst. Loại truyền này chỉ ra rằng Manager đang tiếp tục với một Burst nhưng lần truyền tiếp theo không thể diễn ra ngay lập tức. Khi Manager sử dụng loại truyền BUSY, các tín hiệu địa chỉ và điều khiển phải phản ánh lần truyền tiếp theo trong Burst. Chỉ những Burst có độ dài không xác định mới có thể có truyền BUSY làm chu kỳ cuối của Burst. Subordinate luôn phải cung cấp phản hồi OKAY trạng thái chờ bằng không cho các truyền BUSY và bỏ qua truyền này.
10	NONSEQ	Chỉ ra một truyền đơn lẻ hoặc lần truyền đầu tiên của một Burst. Các tín hiệu địa chỉ và điều khiển không liên quan đến lần truyền trước đó. Các lần truyền đơn lẻ trên bus được coi là Burst có độ dài một nhịp và do đó loại truyền này là NONSEQUENTIAL.
11	SEQ	Các lần truyền còn lại trong một Burst là <b>SEQUENTIAL</b> và địa chỉ có liên quan đến lần truyền

trước đó. Thông tin điều khiển giống hệt với lần truyền trước. Địa chỉ bằng với địa chỉ của lần truyền trước đó cộng với kích thước truyền, tính bằng byte, với kích thước truyền được báo hiệu bằng tín hiệu **HSIZE[2:0]**. Trong trường hợp Burst vòng, địa chỉ của truyền sẽ vòng lại ở ranh giới địa chỉ.



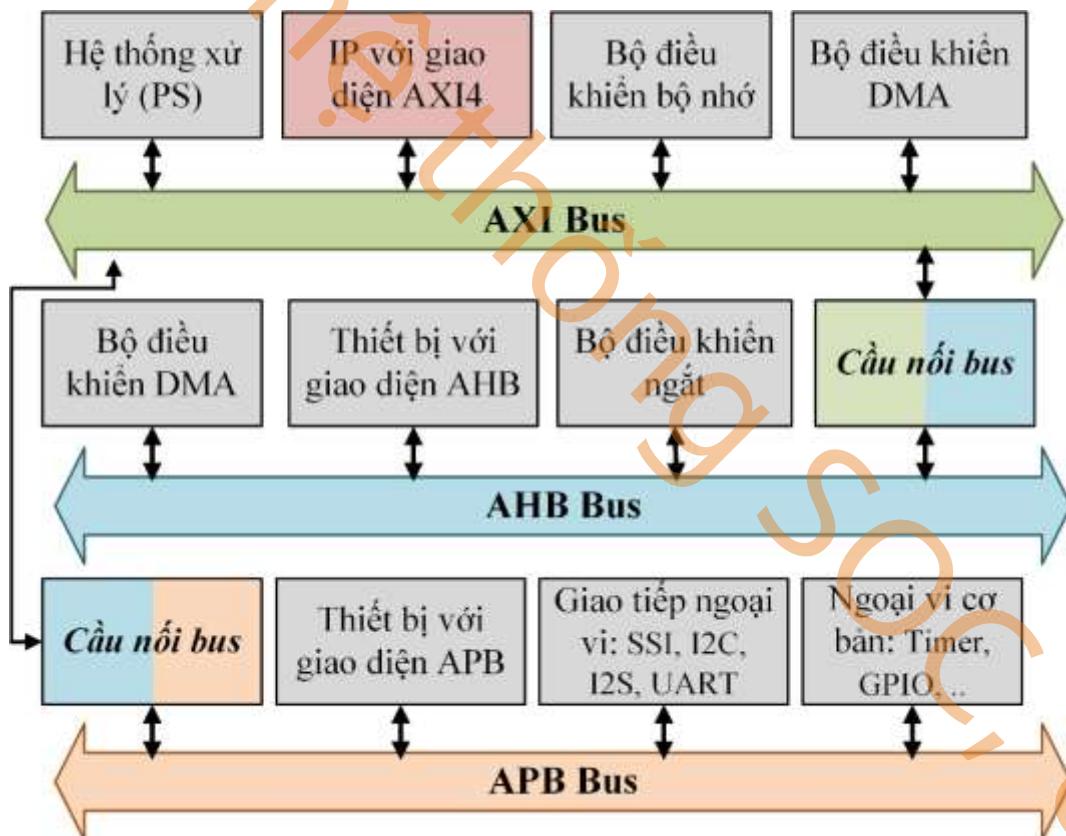
Hình 4.17: Sơ đồ thời gian chi tiết của tín hiệu kết hợp đọc với Burst gia tăng 4 nhịp.

Hình 4.17 minh họa quá trình truyền dữ liệu đọc bằng cách sử dụng một Burst gia tăng 4 nhịp (INCR4) với một trạng thái chờ được thêm vào cho lần truyền đầu tiên. Trong trường hợp này, địa chỉ không quay vòng tại ranh giới 16 byte. Địa chỉ bắt đầu từ 0x38 và tiếp theo là 0x3C, sau đó đến 0x40 và 0x44. Điều này được thể hiện qua tín hiệu HTRANS[1:0] với trạng thái NONSEQ cho lần truyền đầu tiên và SEQ cho các lần truyền tiếp theo. Các tín hiệu dữ liệu HWDATA[31:0] hiển thị dữ liệu được đọc lần lượt tại các địa chỉ tương ứng, bắt đầu từ 0x38 và kết thúc tại 0x44. Việc truyền dữ liệu theo kiểu Burst gia tăng giúp cải thiện hiệu suất truy cập bộ nhớ trong hệ thống nhúng, giảm thiểu độ trễ bằng cách duy trì các giao dịch liên tiếp mà không cần khởi động lại quy trình truyền.

Việc ghi dữ liệu trong các trường hợp Burst vòng 4 nhịp (WRAP4) và Burst gia tăng 4 nhịp (INCR4) cũng tuân theo nguyên tắc tương tự như quá trình đọc dữ liệu đã mô tả. Với Burst vòng 4 nhịp, địa chỉ sẽ quay vòng lại khi vượt qua ranh giới 16 byte, trong khi với Burst gia tăng 4 nhịp, địa chỉ sẽ tiếp tục tăng mà không quay vòng. Tương tự, các loại Burst khác như Burst vòng 8 nhịp (WRAP8), Burst gia tăng 8 nhịp (INCR8), Burst vòng 16 nhịp (WRAP16), và Burst gia tăng 16 nhịp (INCR16) đều hoạt động dựa trên nguyên tắc này. Sự khác biệt chính nằm ở số lần nhịp trong mỗi Burst và việc địa chỉ có quay vòng hay không.

## 4.5. Hệ thống bus AXI

### 4.5.1. Giới thiệu về hệ thống bus AXI



Hình 4.18: Hệ thống kết hợp các bus AXI, AHB, và APB trên nhiều Xilinx FPGA [17].

AMBA AXI có băng thông cao hơn AHB và APB nhờ thiết kế pipeline tối ưu, cho phép sử dụng hiệu quả bus khi chia sẻ giữa nhiều lõi CPU và các bộ điều khiển truy cập bộ nhớ trực tiếp DMA. AXI hỗ trợ các cấu trúc bus phức tạp, cho phép chia sẻ các điểm cuối

tổ giữa nhiều bus chủ với nhiều giao dịch cùng lúc. Trong nhiều hệ thống trên nhiều FPGA, nhiều loại bus như AXI, AHB, và APB được kết hợp lại với nhau như mô tả ở Hình 4.18. Ví dụ, phần hệ thống xử lý (PS) của Zynq FPGA sử dụng cả ba loại bus AXI, AHB và APB. Kết nối chính của hệ thống dựa trên AXI, kết nối lõi xử lý với SRAM trên chip, bộ điều khiển bộ nhớ DDR ngoài, và với FPGA. Các cổng AXI của FPGA bao gồm cả giao diện tớ và chủ. Các cổng tớ cho phép bộ xử lý truy cập các tài nguyên FPGA và các cổng chủ cho phép logic của FPGA truy cập các tài nguyên của PS. Các thiết bị ngoại vi phần cứng của Zynq FPGA được kết nối với APB thông qua cầu nối đến kết nối chính AXI. Các bộ điều khiển USB và Ethernet trong PS của Zynq FPGA có giao diện AHB Chủ, nhưng chúng được kết nối với AXI để có thể truy cập bộ nhớ của bộ xử lý, điều này được gọi là truy cập bộ nhớ trực tiếp.

Tiêu chuẩn AXI đã trải qua nhiều lần phát triển và mở rộng qua các thế hệ AMBA khác nhau để đáp ứng nhu cầu truyền nhận dữ liệu các hệ thống nhúng phức tạp khác nhau theo từng yêu cầu của hệ thống:

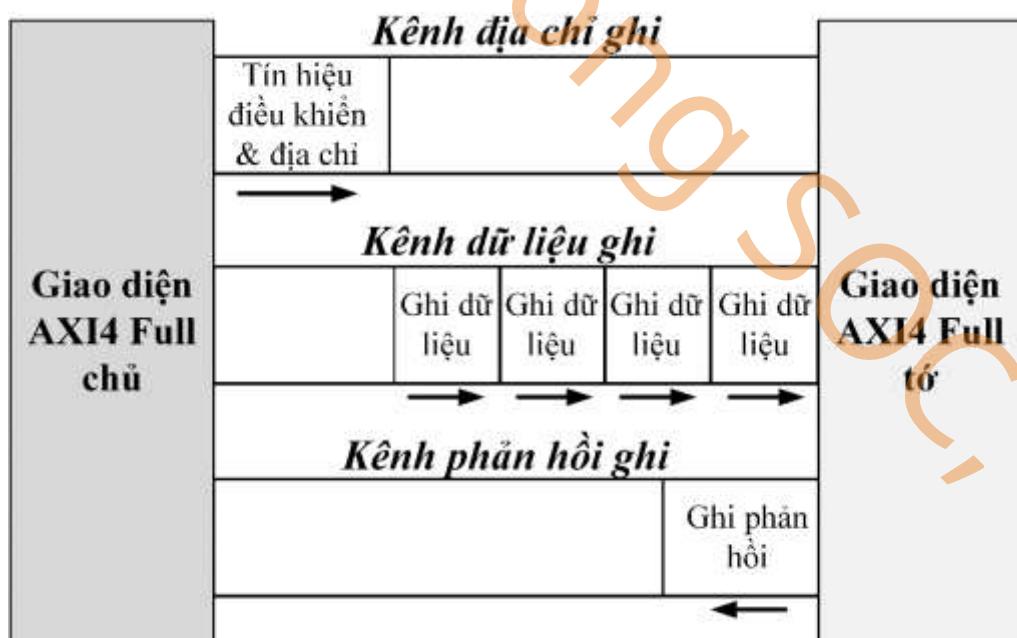
- ✓ AXI3: AXI3 là phiên bản đầu tiên của giao thức AXI được giới thiệu trong đặc tả AMBA 3. Phiên bản này hỗ trợ các giao dịch đọc và ghi không đồng bộ với các kênh dữ liệu, địa chỉ, và phản hồi độc lập, cho phép các giao dịch có thể diễn ra song song mà không bị chờ đợi. Điều này giúp cải thiện băng thông và tối ưu hóa việc sử dụng bus trong các hệ thống có nhiều chủ.
- ✓ AXI4: Phiên bản này mở rộng tính năng của AXI3 bằng cách hỗ trợ các Burst dài hơn (lên đến 256 nhịp), tăng cường hiệu suất truyền tải dữ liệu trong các hệ thống đòi hỏi băng thông cao. AXI4 cũng hỗ trợ các thiết kế đa chủ với nhiều tớ và các giao dịch phức tạp. AXI4 có bả biến thể bao gồm AXI Full, AXI-Lite, và AXI-Stream. Cụ thể, AXI4 Full hỗ trợ truyền Burst với độ dài và kích thước dữ liệu đa dạng. AXI4-Lite, đơn giản hóa chỉ với các giao dịch một từ, giảm thiểu số lượng tín hiệu và đơn giản hóa logic cần thiết cho các thiết kế không cần băng thông lớn. AXI4-Stream được tối ưu hóa cho các ứng dụng truyền dữ liệu tốc độ cao mà không cần truyền địa chỉ, chẳng hạn như truyền video, âm thanh, hoặc dữ liệu từ DMA.

Giao thức này không sử dụng các kênh địa chỉ, chỉ tập trung vào kênh dữ liệu với tốc độ cao.

- ✓ AXI5: Trong đặc tả AMBA 5, AXI5 được cập nhật để hỗ trợ TrustZone và các ứng dụng bảo mật cao trên nền tảng Armv8-M. Nó bổ sung các tính năng kiểm soát giao dịch tốt hơn, hỗ trợ các thuộc tính bảo mật nâng cao và cải thiện khả năng quản lý băng thông trong các hệ thống có nhiều bus chủ và tớ.

Hầu hết các Xilinx FPGA hiện đại đều sử dụng chuẩn giao tiếp AXI4 cho việc thiết kế các IP và kết nối các khối chức năng trong hệ thống. AXI4 là một chuẩn giao tiếp linh hoạt và mạnh mẽ, cho phép truyền dữ liệu hiệu quả với băng thông cao, hỗ trợ nhiều loại Burst và khả năng truyền song song. Với ba biến thể chính là AXI4 Full, AXI4-Lite và AXI4-Stream, chuẩn AXI4 phù hợp với nhiều loại ứng dụng khác nhau từ điều khiển ngoại vi đơn giản đến truyền dữ liệu tốc độ cao. Do đó, phần này sẽ tập trung vào việc giới thiệu chi tiết về AXI4 để giúp hiểu rõ hơn cách thức hoạt động, thiết kế, và triển khai các IP trên Xilinx FPGA, nhằm tối ưu hóa hiệu suất và khả năng mở rộng của hệ thống.

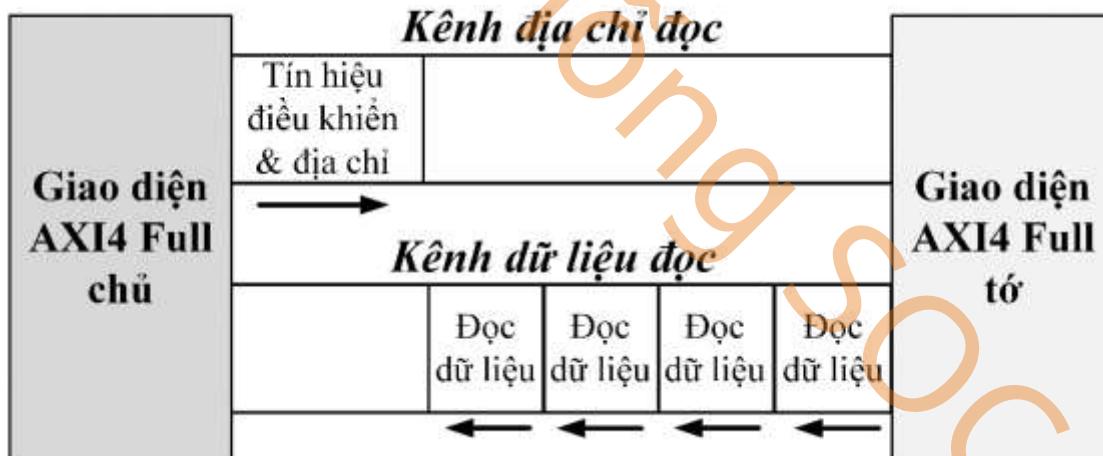
#### 4.5.2. Tín hiệu và kết nối AXI4 Full



Hình 4.19: Quá trình ghi dữ liệu trên AMBA AXI4 Full sử dụng ba kênh chính: kênh địa chỉ ghi, kênh dữ liệu ghi, và kênh phản hồi ghi [18].

Trong phần này, chúng ta sẽ tìm hiểu về tín hiệu và kết nối của giao thức AXI4 Full trong hệ thống AMBA. Cụ thể, AXI4 Full hỗ trợ truyền tải dữ liệu phức tạp với khả năng xử lý nhiều giao dịch đồng thời, giúp tối ưu hóa băng thông và hiệu suất khi kết nối giữa các IP trên FPGA. Giao thức này sử dụng năm kênh độc lập để thực hiện truyền và nhận dữ liệu: kênh địa chỉ ghi, kênh dữ liệu ghi, kênh phản hồi ghi, kênh địa chỉ đọc, và kênh dữ liệu đọc.

Hình 4.19 mô tả chi tiết quá trình ghi dữ liệu trên giao thức AMBA AXI4 sử dụng ba kênh chính: kênh địa chỉ ghi, kênh dữ liệu ghi, và kênh phản hồi ghi, nhằm đảm bảo tính đồng bộ và hiệu suất cao của hệ thống. Trong quá trình ghi, chủ gửi tín hiệu điều khiển và địa chỉ cần ghi qua kênh địa chỉ ghi để chỉ định vị trí bộ nhớ, đảm bảo dữ liệu sẽ được ghi vào đúng địa chỉ. Tiếp theo, chủ truyền dữ liệu qua kênh dữ liệu ghi, nơi dữ liệu thực tế được chuyển từ chủ đến tớ. Cuối cùng, sau khi tớ ghi dữ liệu xong, kênh phản hồi ghi được sử dụng để tớ gửi phản hồi về trạng thái của giao dịch ghi, giúp chủ xác nhận việc ghi đã hoàn tất và tiếp tục các bước tiếp theo trong quá trình xử lý dữ liệu.



Hình 4.20: Quá trình đọc dữ liệu trên AMBA AXI4 Full sử dụng hai kênh chính: **kênh địa chỉ đọc** và **kênh dữ liệu đọc**[18].

Hình 4.20 mô tả quá trình đọc dữ liệu trên giao thức AXI4 sử dụng hai kênh chính: kênh địa chỉ đọc và kênh dữ liệu đọc. Trong quá trình đọc, chủ gửi tín hiệu điều khiển và địa chỉ qua kênh địa chỉ đọc để chỉ định địa chỉ nơi dữ liệu cần đọc. Sau đó, dữ liệu được

truyền từ từ tới chủ qua kênh dữ liệu đọc. Các kênh này cho phép việc truyền dữ liệu diễn ra đồng thời với các giao dịch khác, tối ưu hóa hiệu suất và giảm độ trễ trong hệ thống.

Sự phân chia rõ ràng của các kênh trong AXI4 Full giúp tăng cường khả năng kiểm soát và tối ưu hóa việc truyền tải dữ liệu, đặc biệt khi tích hợp nhiều IP với độ phức tạp cao trên FPGA.

Bảng 4.5 mô tả các tín hiệu được sử dụng trong 5 kênh của giao thức AXI Full, bao gồm các tín hiệu cho kênh địa chỉ ghi, kênh dữ liệu ghi, kênh phản hồi ghi, kênh địa chỉ đọc và kênh dữ liệu đọc. Theo đó, có đến 45 tín hiệu khác nhau được sử dụng trên 5 kênh. Mỗi kênh đóng vai trò quan trọng trong việc truyền tải dữ liệu, địa chỉ, và phản hồi giữa các chủ và tớ trong hệ thống.

**Bảng 4.5: Các tín hiệu ở 5 kênh dùng trong AXI Full**

Tín hiệu	Chiều tín hiệu	Mô tả
ACLK	Nguồn xung nhịp → tất cả các khối AXI	Tín hiệu xung nhịp chung
ARESETn	Nguồn reset → tất cả các khối AXI	Tín hiệu reset mức thấp chung

#### *Các tín hiệu kênh địa chỉ ghi*

AWID	Chủ → Tớ	ID địa chỉ ghi. Tín hiệu này là thẻ nhận dạng cho nhóm tín hiệu địa chỉ ghi.
AWADDR	Chủ → Tớ	Địa chỉ ghi. Địa chỉ ghi cung cấp địa chỉ của lần truyền đầu tiên trong một giao dịch Burst.
AWLEN	Chủ → Tớ	Độ dài Burst. Độ dài Burst cung cấp số lượng truyền chính xác trong một Burst. Thông tin này xác định số lần truyền dữ liệu liên quan đến địa chỉ.
AWSIZE	Chủ → Tớ	Kích thước Burst. Tín hiệu này chỉ ra kích thước của mỗi lần truyền trong Burst.

AWBURST	Chủ → Tớ	Loại Burst. Loại Burst và thông tin kích thước xác định cách tính địa chỉ cho mỗi lần truyền trong Burst.
AWLOCK	Chủ → Tớ	Loại khóa. Cung cấp thông tin bổ sung về đặc tính của lần truyền.
AWCACHE	Chủ → Tớ	Loại bộ nhớ. Tín hiệu này chỉ ra cách các giao dịch cần tiến hành qua hệ thống.
AWPROT	Chủ → Tớ	Loại bảo vệ. Tín hiệu này chỉ ra mức độ quyền và an ninh của giao dịch, và liệu giao dịch đó là truy cập dữ liệu hay lệnh.
AWQOS	Chủ → Tớ	Chất lượng dịch vụ (QoS). Tập định danh QoS cho mỗi lần truyền ghi. Chỉ được triển khai trong AXI4.
AWREGIO N	Chủ → Tớ	Định danh vùng. Cho phép một giao diện vật lý đơn trên tớ được sử dụng cho nhiều giao diện logic.
AWUSER	Chủ → Tớ	Tín hiệu người dùng. Tín hiệu do người dùng tùy chọn trong kênh địa chỉ ghi. Chỉ hỗ trợ trong AXI4.
AWVALID	Chủ → Tớ	Địa chỉ ghi hợp lệ. Tín hiệu này chỉ ra rằng kênh đang báo hiệu địa chỉ ghi hợp lệ và thông tin điều khiển.
AWREADY	Tớ → Chủ	Địa chỉ ghi sẵn sàng. Tín hiệu này chỉ ra rằng tớ đã sẵn sàng nhận địa chỉ ghi và thông tin điều khiển liên quan.
<b>Các tín hiệu kênh dữ liệu ghi</b>		

WID	Chủ → Tớ	ID ghi. Tín hiệu này là ID của việc truyền dữ liệu ghi. Chỉ hỗ trợ AXI3.
WDATA	Chủ → Tớ	Dữ liệu ghi. Dữ liệu có thể điều chỉnh tùy thuộc vào yêu cầu dữ liệu của IP với độ lớn thường là 32, 64, 128, và 256 bit.
WSTRB	Chủ → Tớ	Strobes ghi. Tín hiệu này chỉ ra các byte nào chứa dữ liệu hợp lệ. Mỗi 8 bit của bus dữ liệu ghi có một bit strobe ghi.
WLAST	Chủ → Tớ	Kết thúc ghi. Tín hiệu này chỉ ra lần truyền cuối cùng trong một chuỗi ghi.
WUSER	Chủ → Tớ	Tín hiệu người dùng. Tín hiệu tùy chọn do người dùng định nghĩa trong kênh dữ liệu ghi.
WVALID	Chủ → Tớ	Ghi hợp lệ. Tín hiệu này chỉ ra rằng dữ liệu ghi hợp lệ và strobes có sẵn.
WREADY	Tớ → Chủ	Sẵn sàng ghi. Tín hiệu này chỉ ra rằng tờ có thể chấp nhận dữ liệu ghi.

#### Các tín hiệu kênh phản hồi

BID	Tớ → Chủ	ID phản hồi. Tín hiệu này là thẻ ID của phản hồi ghi.
BRESP	Tớ → Chủ	Phản hồi ghi. Tín hiệu này chỉ ra trạng thái của giao dịch ghi.
BUSER	Tớ → Chủ	Tín hiệu người dùng. Tín hiệu tùy chọn do người dùng định nghĩa trong kênh phản hồi ghi. Chỉ hỗ trợ trong AXI4.

BVALID	Tớ → Chủ	Phản hồi ghi hợp lệ. Tín hiệu này chỉ ra rằng kênh đang báo hiệu một phản hồi ghi hợp lệ.
BREADY	Chủ → Tớ	Phản hồi sẵn sàng. Tín hiệu chỉ ra rằng chủ có thể chấp nhận phản hồi ghi.

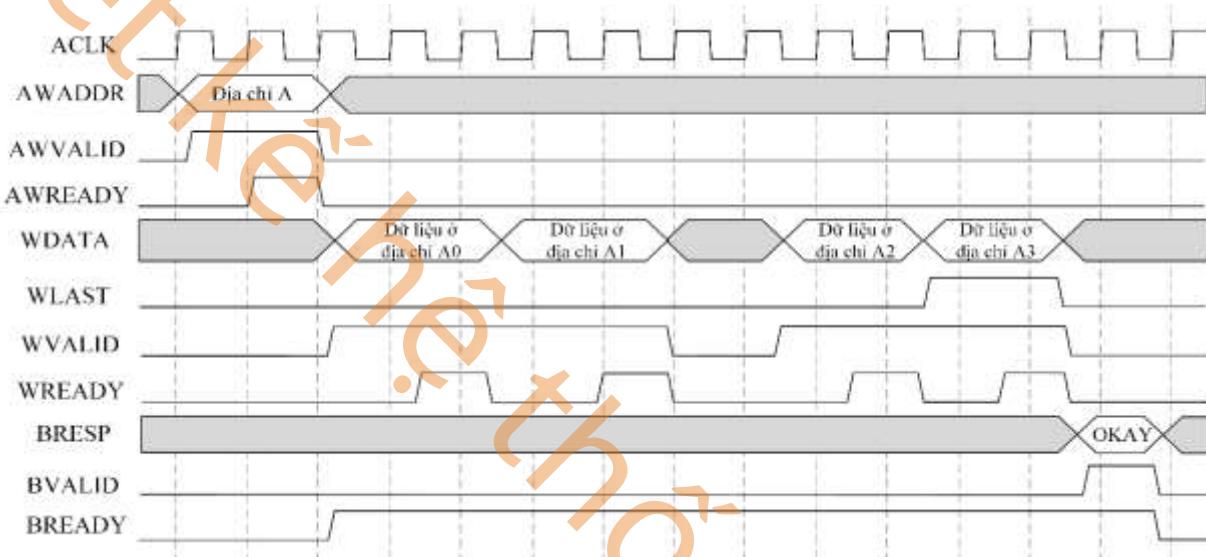
#### Các tín hiệu kênh địa chỉ đọc

ARID	Chủ → Tớ	ID địa chỉ đọc. Tín hiệu này là thẻ nhận dạng cho nhóm tín hiệu địa chỉ đọc.
ARADDR	Chủ → Tớ	Địa chỉ đọc. Tín hiệu địa chỉ đọc cung cấp địa chỉ của lần chuyển đầu tiên trong một giao dịch đọc dạng Burst.
ARLEN	Chủ → Tớ	Độ dài Burst. Tín hiệu này chỉ ra số lượng chuyển chính xác trong một Burst.
ARSIZE	Chủ → Tớ	Kích thước Burst. Tín hiệu chỉ ra kích thước của mỗi lần chuyển trong Burst.
ARBURST	Chủ → Tớ	Loại Burst. Loại Burst và thông tin kích thước xác định cách thức địa chỉ cho mỗi lần chuyển trong Burst được tính toán.
ARLOCK	Chủ → Tớ	Loại khóa. Tín hiệu này cung cấp thông tin bổ sung về đặc điểm của lần chuyển.
ARCACHE	Chủ → Tớ	Loại bộ nhớ. Tín hiệu này chỉ ra cách thức các giao dịch cần phải tiến hành trong hệ thống.
ARPROT	Chủ → Tớ	Loại bảo vệ. Tín hiệu này chỉ ra mức độ đặc quyền và an ninh của giao dịch, và liệu giao dịch là truy cập dữ liệu hay truy cập lệnh.

ARQOS	Chủ → Tớ	Chất lượng dịch vụ (QoS). Mã định danh QoS được gửi cho mỗi giao dịch đọc. Chỉ được triển khai trong AXI4.
ARREGION	Chủ → Tớ	Mã định danh vùng. Cho phép một giao diện vật lý đơn lẻ trên tớ được sử dụng cho nhiều giao diện logic. Chỉ được triển khai trong AXI4.
ARUSER	Chủ → Tớ	Tín hiệu người dùng. Tín hiệu tùy chọn do người dùng định nghĩa trong kênh địa chỉ đọc. Chỉ hỗ trợ trong AXI4.
ARVALID	Chủ → Tớ	Địa chỉ đọc hợp lệ. Tín hiệu này chỉ ra rằng kênh đang báo hiệu địa chỉ đọc hợp lệ và thông tin điều khiển.
ARREADY	Tớ → Chủ	Địa chỉ đọc sẵn sàng. Tín hiệu này chỉ ra rằng tớ đã sẵn sàng chấp nhận một địa chỉ và các tín hiệu điều khiển liên quan.
<b>Các tín hiệu kênh dữ liệu đọc</b>		
RID	Tớ → Chủ	ID đọc. Tín hiệu này là thẻ nhận dạng cho nhóm tín hiệu dữ liệu đọc được tạo ra bởi tớ
RDATA	Tớ → Chủ	Dữ liệu đọc.
RRESP	Tớ → Chủ	Phản hồi đọc. Tín hiệu này chỉ ra trạng thái của lần chuyển đọc.
RLAST	Tớ → Chủ	Dữ liệu đọc cuối cùng. Tín hiệu này chỉ ra lần chuyển cuối cùng trong một Burst đọc.
RUSER	Tớ → Chủ	Tín hiệu người dùng. Tín hiệu tùy chọn do người dùng định nghĩa trong kênh dữ liệu đọc.

RVALID	Tớ → Chủ	Dữ liệu đọc hợp lệ. Tín hiệu này chỉ ra rằng kênh đang báo hiệu dữ liệu đọc yêu cầu.
RREADY	Chủ → Tớ	Dữ liệu đọc sẵn sàng. Tín hiệu này chỉ ra rằng chủ có thể chấp nhận dữ liệu đọc và thông tin phản hồi.

#### 4.5.3. Chi tiết tín hiệu của AXI4 Full trong việc đọc và ghi dữ liệu

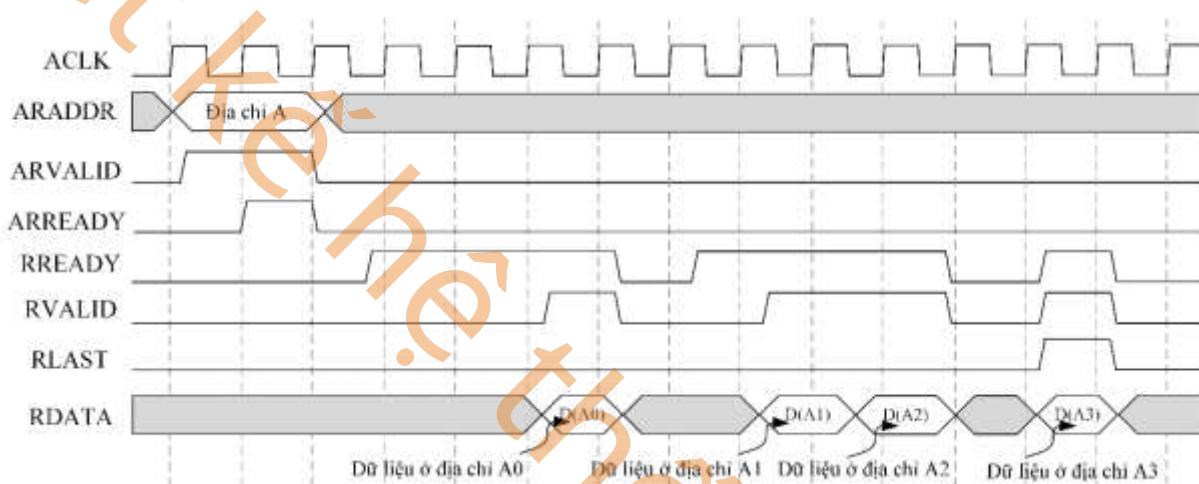


Hình 4.21: Sơ đồ thời gian chi tiết của tín hiệu chính cho ghi dữ liệu với Burst trên kênh địa chỉ ghi, kênh dữ liệu ghi, và kênh phản hồi ghi.

Trong phần này, chúng ta sẽ tập trung vào việc phân tích chi tiết các tín hiệu của giao thức AXI4 Full liên quan đến các hoạt động đọc và ghi dữ liệu với chế độ Burst. Chế độ Burst trong AXI4 Full cho phép truyền một loạt dữ liệu liên tiếp chỉ với một lệnh địa chỉ duy nhất, giúp cải thiện hiệu suất truyền thông và băng thông dữ liệu.

Hình 4.21 minh họa sơ đồ thời gian chi tiết cho giao dịch ghi dữ liệu trên bus AXI với chế độ truyền dữ liệu Burst, bao gồm ba kênh chính: kênh địa chỉ ghi, kênh dữ liệu ghi, và kênh phản hồi ghi. Giao dịch bắt đầu khi chủ gửi địa chỉ và thông tin điều khiển trên kênh địa chỉ ghi (AWADDR, AWVALID, AWREADY). Khi tín hiệu AWVALID và AWREADY đều lên mức cao, địa chỉ được gửi đi và được tờ chấp nhận. Sau đó, chủ gửi các mục dữ liệu trên kênh dữ liệu ghi (WDATA) kèm theo tín hiệu WVALID để báo rằng dữ liệu trên bus là hợp lệ. Tín hiệu WREADY từ tờ cho biết tờ đã sẵn sàng nhận dữ liệu.

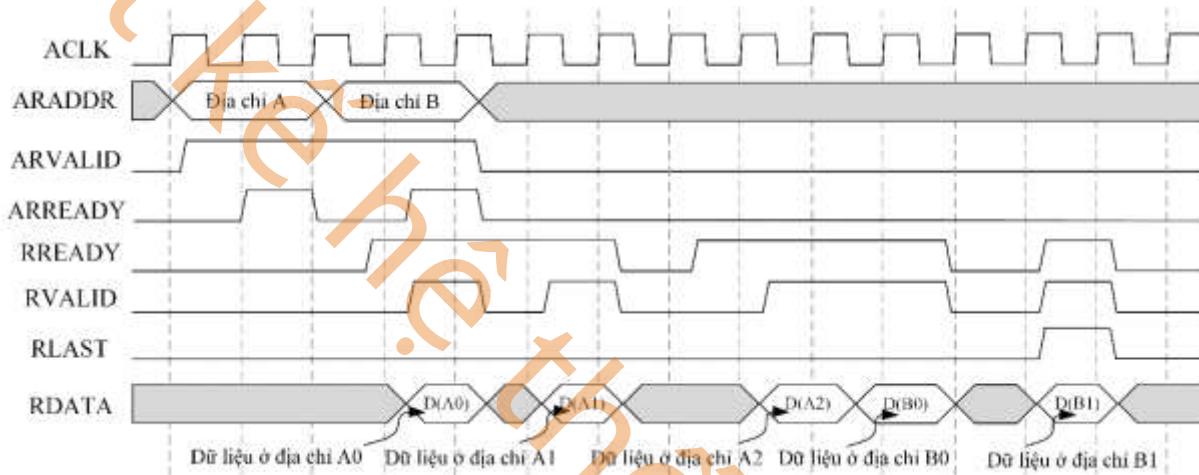
Trong chế độ Burst, dữ liệu được gửi liên tiếp đến khi hết dữ liệu; khi dữ liệu cuối cùng được gửi, tín hiệu WLAST sẽ lên mức cao. Khi tất cả dữ liệu đã được tờ chấp nhận, tờ sẽ gửi phản hồi trên kênh phản hồi ghi (BVALID, BREADY) để thông báo giao dịch ghi đã hoàn tất. Khi tín hiệu BVALID và BREADY đều lên mức cao, phản hồi ghi "OKAY" được gửi đến chủ, kết thúc quá trình giao dịch ghi. Hình này minh họa sự phối hợp chặt chẽ giữa các tín hiệu điều khiển và dữ liệu trong quá trình truyền thông tin giữa chủ và tờ trên bus AXI.



Hình 4.22: Sơ đồ thời gian chi tiết của tín hiệu chính cho đọc dữ liệu với Burst trên kênh địa chỉ đọc và kênh dữ liệu đọc.

Hình 4.22 minh họa sơ đồ thời gian chi tiết cho quá trình đọc dữ liệu với chế độ Burst trên bus AXI, tương tự như quá trình ghi dữ liệu đã trình bày trước đó. Trong ví dụ này, chủ sẽ gửi địa chỉ đọc trên kênh địa chỉ đọc (ARADDR) cùng với tín hiệu điều khiển báo hiệu chiều dài và loại của Burst (các tín hiệu này không hiển thị trong hình để đơn giản hóa). Tín hiệu ARVALID được chủ kích hoạt để báo rằng địa chỉ và thông tin điều khiển trên bus là hợp lệ, và khi tín hiệu ARREADY từ tờ lên mức cao sau một chu kỳ, địa chỉ đã được tờ chấp nhận. Sau khi địa chỉ xuất hiện trên bus địa chỉ, quá trình truyền dữ liệu xảy ra trên kênh dữ liệu đọc (RDATA). Trong quá trình này, tờ giữ tín hiệu RVALID ở mức thấp cho đến khi dữ liệu đọc sẵn sàng, và dữ liệu được truyền từ tờ đến chủ với các tín hiệu đồng bộ theo từng chu kỳ. Đối với dữ liệu cuối cùng trong chuỗi Burst, tín hiệu RLAST được tờ kích hoạt để báo hiệu rằng đây là dữ liệu cuối cùng đang được truyền.

Để tăng cường hiệu quả và tối ưu hóa băng thông trong các hệ thống sử dụng giao thức AXI4 Full, việc thực hiện các giao dịch đọc chồng chéo (overlapping read transactions) là một kỹ thuật quan trọng. Kỹ thuật này cho phép chủ gửi một địa chỉ Burst mới ngay sau khi địa chỉ Burst trước đó đã được tờ chấp nhận. Nhờ đó, tờ có thể chuẩn bị xử lý dữ liệu cho Burst tiếp theo trong khi Burst hiện tại vẫn đang diễn ra. Điều này giúp tăng cường hiệu suất tổng thể bằng cách tận dụng khả năng xử lý song song của hệ thống và giảm thiểu thời gian chờ giữa các giao dịch.



Hình 4.23: Sơ đồ thời gian chi tiết của tín hiệu chính cho đọc dữ liệu chồng chéo với Burst trên kênh địa chỉ đọc và kênh dữ liệu đọc.

Trong Hình 4.23, quá trình đọc chồng chéo được minh họa rõ ràng khi chủ điều khiển địa chỉ Burst thứ hai (địa chỉ B) ngay sau khi địa chỉ Burst đầu tiên (địa chỉ A) được tờ chấp nhận. Khi địa chỉ A được gửi và tín hiệu ARREADY từ tờ lên mức cao, địa chỉ A đã được chấp nhận, và ngay sau đó, chủ tiếp tục gửi địa chỉ B. Trong khi dữ liệu từ địa chỉ A được truyền trên kênh dữ liệu đọc (RDATA), tờ đồng thời chuẩn bị dữ liệu cho địa chỉ B. Khi dữ liệu tương ứng từ cả hai địa chỉ sẵn sàng, tờ sẽ sử dụng tín hiệu RVALID và RLAST để đồng bộ hóa việc truyền dữ liệu cho Burst.

#### 4.5.4. Tín hiệu và kết nối AXI4 Lite

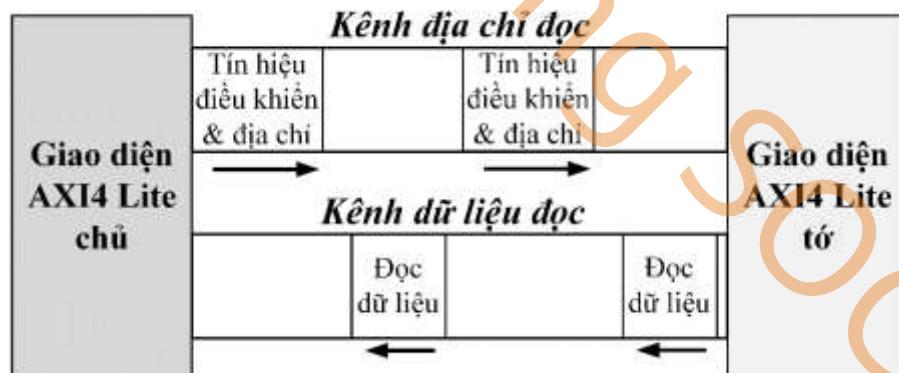
Trong phần này, chúng ta sẽ tìm hiểu về tín hiệu và kết nối của giao thức AXI4-Lite trong hệ thống AMBA. AXI4-Lite là một phiên bản đơn giản hóa của AXI4 Full, được thiết kế cho các ứng dụng không yêu cầu truyền dữ liệu phức tạp hoặc hiệu suất cao. Giao thức này vẫn giữ cấu trúc giao dịch tương tự như AXI4 Full nhưng đơn giản hơn, không

hỗ trợ burst, giúp giảm chi phí tài nguyên và độ phức tạp khi triển khai trong các thiết bị nhúng hoặc các IP đơn giản trên FPGA.

Mặc dù giảm bớt độ phức tạp, AXI4-Lite vẫn sử dụng ba kênh chính cho quá trình ghi dữ liệu: kênh địa chỉ ghi, kênh dữ liệu ghi, và kênh phản hồi ghi. Điều này đảm bảo các giao dịch ghi dữ liệu diễn ra mạch lạc và có phản hồi từ tự.



Hình 4.24: Quá trình ghi dữ liệu trên AMBA AXI4-Lite sử dụng ba kênh chính: kênh địa chỉ ghi, kênh dữ liệu ghi, và kênh phản hồi ghi [18].



Hình 4.25: Quá trình đọc dữ liệu trên AMBA AXI4-Lite sử dụng hai kênh chính: kênh địa chỉ đọc và kênh dữ liệu đọc [18].

Hình 4.24: Quá trình ghi dữ liệu trên AXI4-Lite sử dụng ba kênh chính: kênh địa chỉ ghi, kênh dữ liệu ghi, và kênh phản hồi ghi. Trong quá trình này, chủ bắt đầu bằng cách gửi tín hiệu điều khiển và địa chỉ qua kênh địa chỉ ghi để chỉ định địa chỉ nơi dữ liệu sẽ được ghi vào. Sau đó, dữ liệu được truyền từ chủ đến tớ qua kênh dữ liệu ghi. Khi dữ liệu

đã được ghi xong, tớ sẽ gửi tín hiệu phản hồi về trạng thái của giao dịch qua kênh phản hồi ghi.

Hình 4.25: Quá trình đọc dữ liệu trên AXI4-Lite sử dụng hai kênh chính: kênh địa chỉ đọc và kênh dữ liệu đọc. Chủ gửi tín hiệu điều khiển và địa chỉ qua kênh địa chỉ đọc để chỉ định địa chỉ nơi dữ liệu cần đọc. Dữ liệu được đọc từ tớ và truyền về chủ qua kênh dữ liệu đọc. Việc sử dụng hai kênh trong AXI4-Lite cho phép quá trình đọc dữ liệu diễn ra đồng thời với các giao dịch khác, giúp tối ưu hóa hiệu suất.

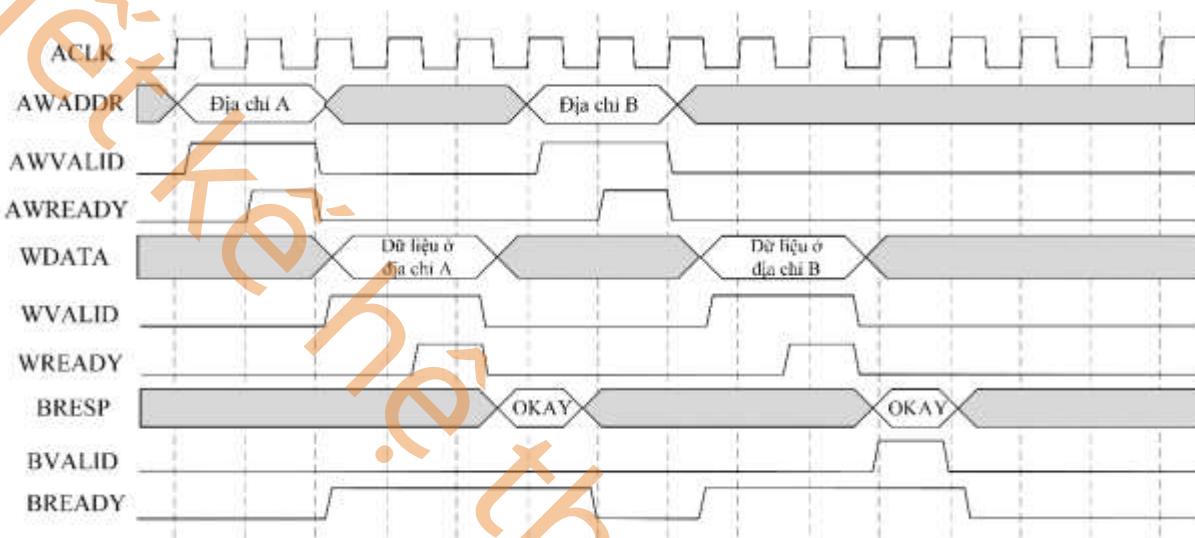
Bảng 4.6 mô tả các tín hiệu được sử dụng trong 5 kênh của giao thức AXI Lite, bao gồm các tín hiệu cho kênh địa chỉ ghi, kênh dữ liệu ghi, kênh phản hồi ghi, kênh địa chỉ đọc, và kênh dữ liệu đọc. Nội dung về chiều tín hiệu và mô tả của các tín hiệu trong bảng này tương tự như đã trình bày trong Bảng 4.5. Do đó, Bảng 4.6 chỉ liệt kê tên các tín hiệu sử dụng trong giao thức AXI Lite, không đi sâu vào chi tiết mô tả từng tín hiệu. AXI Lite, với thiết kế đơn giản hơn AXI Full, chỉ sử dụng 19 tín hiệu khác nhau trên 5 kênh này. Mỗi kênh trong AXI Lite đảm nhận vai trò cụ thể trong việc truyền tải thông tin điều khiển, địa chỉ, và dữ liệu giữa các chủ và tớ trong hệ thống. So với AXI Full, số lượng tín hiệu ít hơn giúp cho giao thức AXI Lite dễ dàng triển khai và tối ưu hóa hơn trong các ứng dụng yêu cầu đơn giản hóa kết nối và giao tiếp giữa các thành phần trong hệ thống.

**Bảng 4.6: Các tín hiệu ở 5 kênh dùng trong AXI Lite**

Kênh	Tên tín hiệu
Kênh địa chỉ ghi	AWVALID, AWREADY, AWADDR, AWPROT
Kênh dữ liệu ghi	WVALID, WREADY, WDATA, WSTRB
Kênh phản hồi ghi	BVALID, BREADY, BRESP
Kênh địa chỉ đọc	ARVALID, ARREADY, ARADDR, ARPROT
Kênh dữ liệu đọc	RVALID, RREADY, RDATA, RRESP

#### **4.5.5. Chi tiết tín hiệu của AXI4-Lite trong việc đọc và ghi dữ liệu**

Trong phần này, chúng ta sẽ tập trung vào việc phân tích chi tiết các tín hiệu của giao thức AXI4-Lite liên quan đến các hoạt động đọc và ghi dữ liệu. Không giống như AXI4 Full, AXI4-Lite được thiết kế cho các ứng dụng yêu cầu truyền thông đơn giản và không cần độ phức tạp của chế độ Burst. Trong AXI4-Lite, mỗi giao dịch đọc hoặc ghi dữ liệu chỉ liên quan đến một địa chỉ duy nhất và không có khả năng thực hiện các giao dịch Burst.

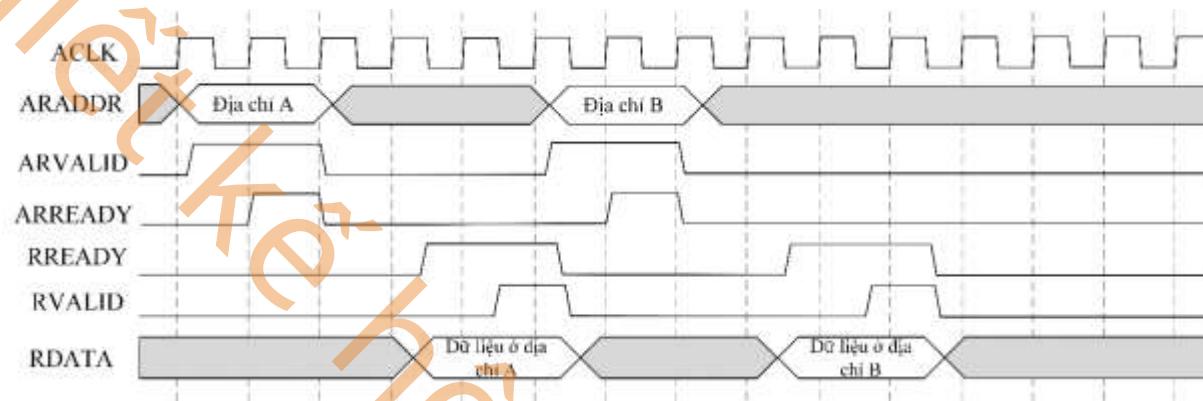


Hình 4.26: Sơ đồ thời gian chi tiết của tín hiệu chính cho ghi dữ liệu của AXI4-Lite trên kênh địa chỉ ghi, kênh dữ liệu ghi, và kênh phản hồi ghi.

Hình 4.26 minh họa quá trình ghi dữ liệu trên giao thức AXI4-Lite, sử dụng ba kênh: kênh địa chỉ ghi, kênh dữ liệu ghi, và kênh phản hồi ghi. Quy trình ghi dữ liệu bắt đầu khi chủ gửi địa chỉ ghi (AWADDR) và tín hiệu hợp lệ địa chỉ ghi (AWVALID) trên kênh địa chỉ ghi. Tờ xác nhận nhận địa chỉ khi tín hiệu sẵn sàng địa chỉ ghi (AWREADY) được kích hoạt. Sau đó, dữ liệu ghi (WDATA) được truyền từ chủ sang tớ thông qua kênh dữ liệu ghi khi tín hiệu hợp lệ dữ liệu ghi (WVALID) được kích hoạt và tờ sẵn sàng nhận dữ liệu (WREADY). Khi quá trình ghi hoàn tất, tờ gửi tín hiệu phản hồi ghi (BRESP) qua kênh phản hồi ghi để báo hiệu trạng thái của giao dịch, thường là "OKAY", cho biết quá trình ghi đã thành công. Tín hiệu hợp lệ phản hồi ghi (BVALID) và tín hiệu sẵn sàng phản hồi ghi (BREADY) cũng được sử dụng để đồng bộ hóa việc truyền tải phản hồi.

Hình 4.27 minh họa quá trình đọc dữ liệu trên giao thức AXI4-Lite, sử dụng hai kênh: kênh địa chỉ đọc và kênh dữ liệu đọc. Quá trình đọc bắt đầu khi chủ gửi địa chỉ đọc

(ARADDR) và tín hiệu hợp lệ địa chỉ đọc (ARVALID) trên kênh địa chỉ đọc. Tới xác nhận đã nhận địa chỉ khi tín hiệu sẵn sàng địa chỉ đọc (ARREADY) được kích hoạt. Sau khi nhận địa chỉ, dữ liệu đọc (RDATA) được truyền từ từ tới chủ qua kênh dữ liệu đọc khi tín hiệu hợp lệ dữ liệu đọc (RVALID) được kích hoạt và chủ sẵn sàng nhận dữ liệu (RREADY).



Hình 4.27: Sơ đồ thời gian chi tiết của tín hiệu chính cho đọc dữ liệu của AXI4-Lite trên kênh địa chỉ đọc và kênh dữ liệu đọc.

#### 4.5.6. Tín hiệu và kết nối AXI4 Stream

Trong phần này, chúng ta sẽ tìm hiểu về tín hiệu và kết nối của giao thức AXI4 Stream trong hệ thống AMBA. AXI4 Stream là một biến thể của giao thức AXI trong hệ thống AMBA, được thiết kế để truyền dữ liệu liên tục mà không cần địa chỉ hoặc sự điều khiển phức tạp như AXI4 Full hoặc AXI4 Lite. Giao thức này chủ yếu được sử dụng cho các ứng dụng đòi hỏi truyền dữ liệu lớn như video, âm thanh, hoặc dữ liệu sensor trong các thiết bị xử lý tín hiệu số (DSP) và FPGA. Khác với AXI4 Full và AXI4 Lite, AXI4 Stream không phân chia thành 5 kênh riêng biệt (địa chỉ đọc, địa chỉ ghi, dữ liệu đọc, dữ liệu ghi, và phản hồi ghi). Thay vào đó, nó chỉ có một kênh duy nhất cho truyền dữ liệu.

**Bảng 4.7: Các tín hiệu trong AXI Stream**

Tín hiệu	Chiều tín hiệu	Mô tả
ACLK	Nguồn xung nhịp → tất cả các khối AXI Stream	Tín hiệu xung nhịp chung

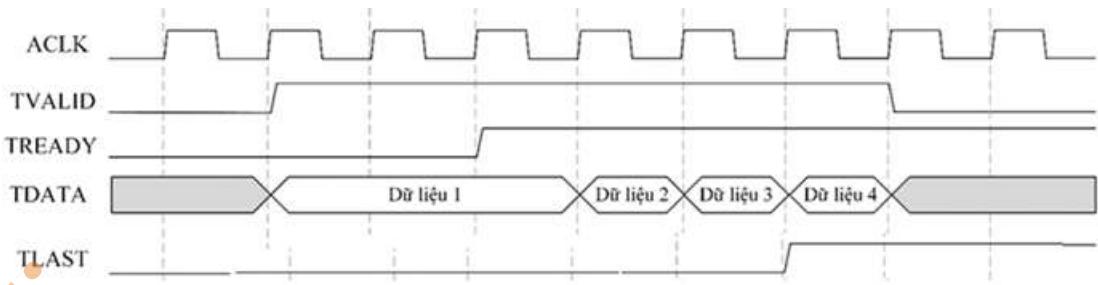
ARESETn	Nguồn reset → tất cả các khối AXI Stream	Tín hiệu reset mức thấp chung
TVALID	Chủ → Tớ hoặc (Tớ → Chủ)	TVALID chỉ ra rằng Chủ đang truyền dữ liệu hợp lệ. Một lần truyền xảy ra khi cả TVALID và TREADY đều bằng 1.
TREADY	Tớ → Chủ hoặc (Chủ → Tớ)	TREADY chỉ ra rằng Tớ có thể chấp nhận dữ liệu.
TDATA	Chủ → Tớ hoặc (Tớ → Chủ)	TDATA là tải trọng chính được sử dụng để cung cấp dữ liệu đi qua giao diện. Chiều rộng bit phải là số nguyên của byte và được khuyến nghị là 8, 16, 32, 64, 128, 256, 512 hoặc 1024 bit.
TSTRB	Chủ → Tớ hoặc (Tớ → Chủ)	TSTRB là tín hiệu byte qualifier chỉ ra nội dung của byte liên quan của TDATA được xử lý như byte dữ liệu hay byte vị trí.
TKEEP	Chủ → Tớ hoặc (Tớ → Chủ)	TKEEP là tín hiệu byte qualifier chỉ ra nội dung của byte liên quan của TDATA được xử lý như một phần của luồng dữ liệu.
TLAST	Chủ → Tớ hoặc (Tớ → Chủ)	TLAST chỉ ra ranh giới của một gói dữ liệu.
TID	Chủ → Tớ hoặc (Tớ → Chủ)	TID là tín hiệu định danh luồng dữ liệu. Chiều rộng bit được khuyến nghị không quá 8.
TDEST	Chủ → Tớ hoặc (Tớ → Chủ)	TDEST cung cấp thông tin định tuyến cho luồng dữ liệu. Chiều rộng bit được khuyến nghị không quá 8.
TUSER	Chủ → Tớ	TUSER là thông tin sideband do người dùng định nghĩa có thể được truyền kèm theo

	hoặc (Tớ → Chủ)	luồng dữ liệu. Chiều rộng bit được khuyến nghị là bội số nguyên của chiều rộng bit của TSTRB.
TWAKEUP	Chủ → Tớ hoặc (Tớ → Chủ)	TWAKEUP xác định bất kỳ hoạt động nào liên quan đến giao diện AXI-Stream.

Bảng 4.7 liệt kê các tín hiệu được sử dụng trong giao thức AXI Stream, giúp điều phối quá trình truyền dữ liệu giữa các thành phần trong hệ thống. Các tín hiệu như ACLK và ARESETn đóng vai trò quan trọng trong việc cung cấp xung nhịp đồng bộ và khởi tạo lại trạng thái của hệ thống. TVALID và TREADY có thể sử dụng từ chủ sang tớ hoặc tớ sang chủ, điều phối quá trình truyền dữ liệu và đảm bảo chỉ những dữ liệu hợp lệ được truyền. TDATA chứa dữ liệu thực tế được truyền đi, trong khi các tín hiệu như TSTRB và TKEEP xác định nội dung byte của dữ liệu và đảm bảo việc truyền tải dữ liệu đúng cách. Các tín hiệu bổ sung như TLAST, TID, TDEST, TUSER, và TWAKEUP có thể truyền từ chủ sang tớ hoặc ngược lại, cung cấp các thông tin bổ sung để xác định ranh giới dữ liệu, định danh luồng dữ liệu, và truyền thông tin phụ trợ, hỗ trợ tối ưu hóa quá trình giao tiếp và định tuyến dữ liệu qua hệ thống.

#### 4.5.7. Chi tiết tín hiệu của AXI4-Stream trong việc ghi dữ liệu

Trong phần này, chúng ta sẽ tập trung vào việc phân tích chi tiết các tín hiệu của giao thức AXI4-Stream liên quan đến các hoạt động truyền dữ liệu. Khác với AXI4 Full và AXI4 Lite, AXI4-Stream được thiết kế chuyên biệt cho các ứng dụng truyền dữ liệu liên tục, nơi dữ liệu được chuyển đi dưới dạng một luồng liên tục mà không có khái niệm về địa chỉ bộ nhớ. Điều quan trọng cần nhấn mạnh là giao thức AXI4-Stream chỉ hỗ trợ ghi dữ liệu và dữ liệu chỉ được truyền từ chủ sang tớ. Tuy nhiên, chủ và tớ có thể thay phiên nhau trong các giao dịch, ví dụ như thiết bị A là chủ để ghi dữ liệu cho tớ B, sau đó B sẽ trở thành chủ để ghi dữ liệu ngược lại cho A là tớ. Quá trình này cho phép luồng dữ liệu liên tục và linh hoạt, tối ưu hóa hiệu suất truyền tải dữ liệu trong hệ thống. Giao thức AXI4-Stream hỗ trợ truyền dữ liệu một chiều từ chủ sang tớ mà không cần phải quản lý địa chỉ bộ nhớ hoặc các giao dịch phản hồi phức tạp.



Hình 4.28: Sơ đồ thời gian chi tiết của tín hiệu chính cho ghi dữ liệu của AXI4-Stream.

Hình 4.28 minh họa quá trình ghi dữ liệu của giao thức AXI4-Stream, một giao thức truyền dữ liệu hiệu quả trong hệ thống AMBA. Trong quá trình này, tín hiệu ACLK đóng vai trò xung nhịp đồng bộ hóa toàn bộ hoạt động. Tín hiệu TVALID chỉ ra rằng dữ liệu trên đường TDATA hiện tại là hợp lệ và sẵn sàng để truyền. Khi cả hai tín hiệu TVALID và TREADY đều được kích hoạt (ở mức cao), dữ liệu sẽ được truyền từ chủ sang tớ. Trong quá trình truyền, dữ liệu được đóng gói trên đường TDATA và có thể bao gồm nhiều từ dữ liệu liên tiếp, như minh họa trong hình với các từ "Dữ liệu 1", "Dữ liệu 2", "Dữ liệu 3", và "Dữ liệu 4". Tín hiệu TLAST được kích hoạt để chỉ ra rằng từ dữ liệu hiện tại là từ cuối cùng trong một gói dữ liệu, đánh dấu kết thúc của quá trình truyền.

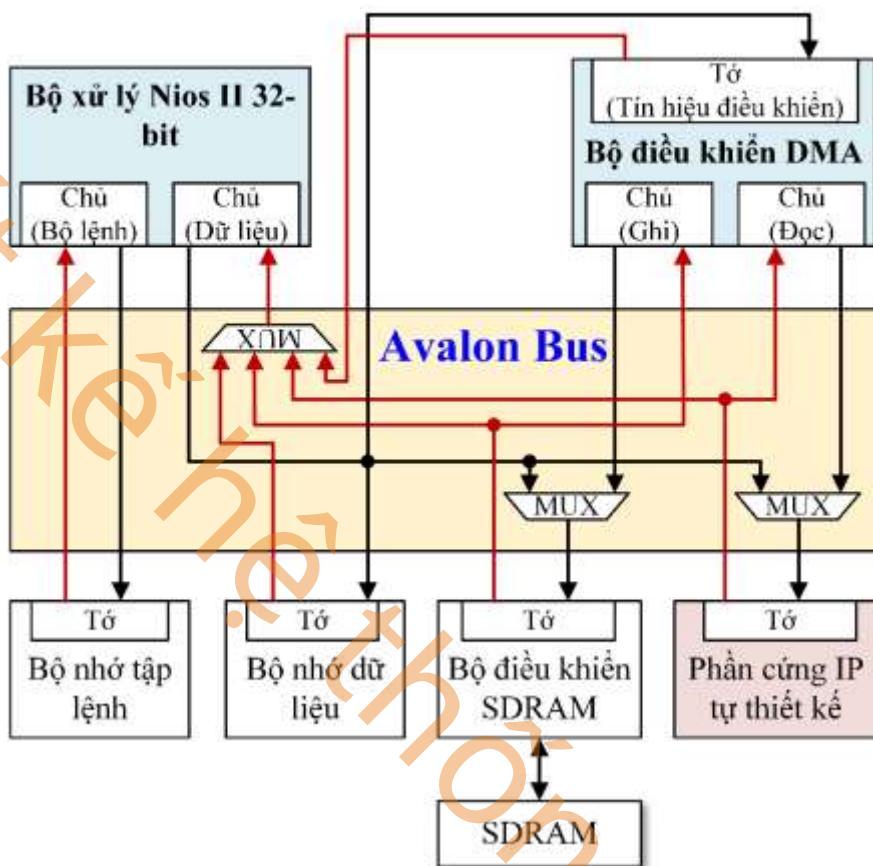
## 4.6. Hệ thống bus Avalon

### 4.6.1. Giới thiệu về hệ thống bus Avalon

Hệ thống bus Avalon là một kiến trúc bus đồng bộ đơn giản được thiết kế để kết nối các bộ xử lý và thiết bị ngoại vi trong các hệ thống SoC. Avalon bus giúp liên kết các thành phần như bộ xử lý, bộ nhớ và các thiết bị ngoại vi với giao diện đồng bộ, mang đến sự thuận tiện và hiệu quả trong thiết kế các hệ thống nhúng. Hệ thống bus Avalon được sử dụng rộng rãi trong các FPGA của Intel (trước đây là Altera), đặc biệt trong các dòng FPGA như Cyclone và Stratix. Avalon bus là lựa chọn chính trong các thiết kế SoC của Intel, giúp kết nối các bộ xử lý, bộ nhớ và thiết bị ngoại vi với nhau.

### Ghi chú:

↔ Ghi dữ liệu và tín hiệu điều khiển      ↔ Đọc dữ liệu



Hình 4.29: Sơ đồ khái niệm về mô-đun bus Avalon trong hệ thống ví dụ đơn giản [19].

Hệ thống bus Avalon hỗ trợ các giao dịch giữa các thiết bị chủ và tớ thông qua các chu kỳ bus với kích thước dữ liệu có thể là byte (8-bit), half-word (16-bit) hoặc word (32-bit). Sau mỗi giao dịch, bus sẽ sẵn sàng cho một giao dịch mới, có thể là giữa các cặp chủ-tớ tương tự hoặc giữa các cặp chủ và tớ khác. Điều này giúp tối ưu hóa hiệu suất và băng thông trong các hệ thống có nhiều thiết bị hoạt động đồng thời. Avalon bus cũng hỗ trợ các giao thức như truyền tải dữ liệu theo luồng và các giao dịch với độ trễ thấp, giúp tăng hiệu quả truyền tải trong các ứng dụng đòi hỏi băng thông cao. Hình 4.29 mô tả sơ đồ khái niệm của mô-đun bus Avalon trong một hệ thống ví dụ, bao gồm bộ xử lý Nios II 32-bit, bộ điều khiển DMA, bộ nhớ SDRAM và các thiết bị ngoại vi kết nối với bus Avalon. Các thiết bị này bao gồm các thiết bị chủ và tớ, với các giao dịch giữa các thiết bị được quản lý và điều

phối qua bus Avalon, tối ưu hóa việc chia sẻ dữ liệu và tín hiệu điều khiển trong hệ thống. Cấu trúc này giúp hệ thống có thể thực hiện các giao dịch hiệu quả và liên tục mà không gặp phải sự cố về hiệu suất, đặc biệt là trong các ứng dụng yêu cầu băng thông cao và tốc độ truyền tải dữ liệu lớn.

Một điểm mạnh của bus Avalon là giao thức dễ hiểu và tối ưu hóa việc sử dụng tài nguyên bus. Điều này giúp cho việc chia sẻ dữ liệu giữa các thiết bị trở nên hiệu quả, dễ dàng triển khai và tích hợp trong các hệ thống nhúng.

So với hệ thống bus AXI, Avalon bus có một số đặc điểm khác biệt:

- ✓ **Đơn giản:** Avalon bus có giao thức đơn giản hơn AXI, điều này giúp dễ dàng triển khai và quản lý trong các hệ thống nhỏ hoặc khi yêu cầu hệ thống bus không quá phức tạp.
- ✓ **Tính đồng bộ:** Avalon bus hoàn toàn đồng bộ, điều này giúp dễ dàng tích hợp với các hệ thống khác trong một FPGA mà không gặp phải vấn đề đồng bộ hóa phức tạp. Trong khi đó, AXI hỗ trợ nhiều kênh giao dịch bất đồng bộ, giúp các giao dịch có thể diễn ra song song mà không cần phải chờ đợi..
- ✓ **Độ phức tạp trong cấu trúc:** AXI có khả năng hỗ trợ các giao dịch phức tạp và các hệ thống đa chủ với độ phức tạp cao hơn, giúp đáp ứng tốt hơn với các ứng dụng yêu cầu băng thông và hiệu suất cao. Avalon, mặc dù có sự hỗ trợ cho nhiều giao dịch đồng thời, nhưng vẫn đơn giản hơn trong việc cấu hình và triển khai.

Mặc dù có nhiều lợi thế, Avalon cũng có một vài điểm yếu so với AXI Bus như sau:

- ✓ **Thiếu tính linh hoạt cao:** Mặc dù Avalon hỗ trợ nhiều giao dịch, nhưng tính linh hoạt và khả năng mở rộng của nó không bằng AXI, đặc biệt là trong các hệ thống có nhiều chủ và tớ hoặc yêu cầu băng thông rất cao.
- ✓ **Không hỗ trợ burst dài như AXI:** AXI có khả năng hỗ trợ burst dài hơn và có khả năng truyền dữ liệu với băng thông cao hơn, rất phù hợp với các ứng dụng yêu cầu truyền tải dữ liệu lớn và nhanh.

Tóm lại, Avalon bus là một giải pháp hiệu quả cho các hệ thống đơn giản và linh hoạt, trong khi AXI bus phù hợp với các hệ thống phức tạp và yêu cầu băng thông cao.

#### 4.6.2. Tín hiệu và kết nối Avalon Bus

Trong phần này, chúng ta sẽ tìm hiểu về tín hiệu và kết nối của giao thức Avalon Bus. Avalon Bus là một kiến trúc bus đồng bộ đơn giản, được thiết kế để kết nối các bộ xử lý và thiết bị ngoại vi trong các hệ thống SoC. Các tín hiệu trong Avalon Bus có vai trò quan trọng trong việc điều phối và quản lý giao tiếp giữa các thiết bị chủ và tớ, giúp tối đam bảo sự đồng bộ giữa các thiết bị trong hệ thống.

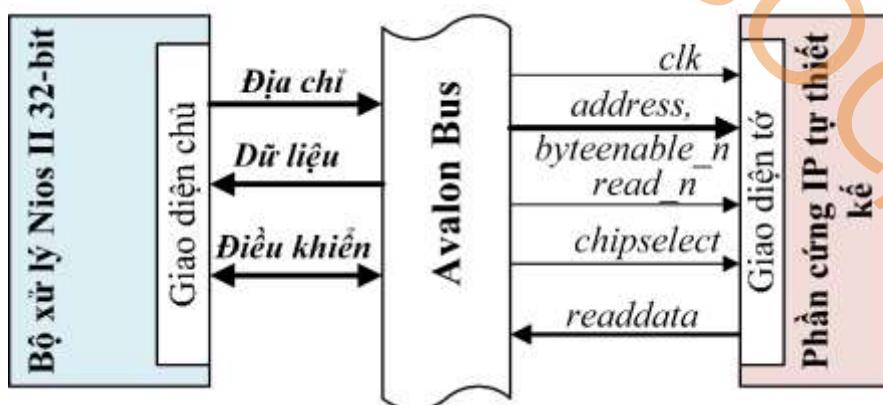
**Bảng 4.8: Các tín hiệu trong Avalon Bus**

Tín hiệu	Chiều tín hiệu	Mô tả
<i>clk</i>	Nguồn xung nhịp → tất cả các khối Avalon	Tín hiệu xung clock toàn cục cho môđun hệ thống và môđun bus Avalon. Tất cả các giao dịch bus đều đồng bộ với <i>clk</i> . Chỉ có các công tơ không đồng bộ mới có thể bỏ qua <i>clk</i> .
<i>reset</i>	Nguồn reset → tất cả các thiết bị ngoại vi	Tín hiệu thiết lập lại toàn cục. Triển khai cụ thể cho thiết bị ngoại vi
<i>chipselect</i>	Chủ → Tớ	Tín hiệu 1 bit chọn chip đến tớ. Công tơ phải bỏ qua tất cả các đầu vào tín hiệu Avalon khác trừ khi <i>chipselect</i> được khảng định.
<i>address</i>	Chủ → Tớ	Dòng địa chỉ của môđun bus Avalon. Tín hiệu này có độ rộng từ 1 đến 32 bit.
<i>begintransfer</i>	Chủ → Tớ	Tín hiệu được kích hoạt trong chu kỳ bus đầu tiên của mỗi giao dịch mới trên bus Avalon. Tín hiệu này có độ rộng 1 bit.
<i>byteenable</i>	Chủ → Tớ	Tín hiệu byte-enable để kích hoạt các byte lane cụ thể trong các giao dịch với bộ nhớ có độ rộng lớn hơn 8 bit. Tín hiệu này có độ rộng 0, 2 hoặc 4 bit.

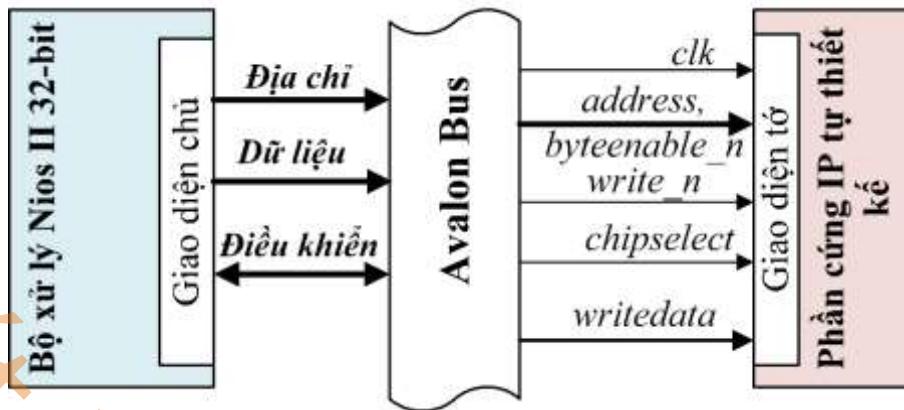
<i>read</i>	Chủ → Tớ	Tín hiệu yêu cầu đọc gửi đến tớ. Không cần thiết nếu tớ không bao giờ xuất dữ liệu ra chủ. Nếu sử dụng, tín hiệu <i>readdata</i> bắt buộc phải được dùng. Tín hiệu này có độ rộng 1 bit.
<i>readdata</i>	Tớ → Chủ	Đường dữ liệu đến mô-đun bus Avalon để chuyển dữ liệu đọc. Không bắt buộc nếu tớ không bao giờ xuất dữ liệu đến chủ. Nếu sử dụng, cũng phải sử dụng tín hiệu <i>read</i> . Tín hiệu này có độ rộng từ 1 đến 32 bit.
<i>write</i>	Chủ → Tớ	Ghi tín hiệu yêu cầu vào tớ. Không bắt buộc nếu tớ không bao giờ nhận dữ liệu từ chủ. Nếu sử dụng, tín hiệu <i>writedata</i> cũng phải được sử dụng. Tín hiệu này có độ rộng 1 bit.
<i>writedata</i>	Chủ → Tớ	Đường dữ liệu từ mô-đun bus Avalon để chuyển dữ liệu ghi. Không bắt buộc nếu tớ không bao giờ nhận dữ liệu từ chủ. Nếu sử dụng, cũng phải sử dụng tín hiệu ghi. Tín hiệu này có độ rộng 1 đến 32 bit.
<i>readdatavalid</i>	Tớ → Chủ	Chỉ được sử dụng bởi các tớ có độ trễ thay đổi. Đánh dấu cạnh lên xung clock khi tớ khẳng định <i>readdata</i> hợp lệ. Tín hiệu này có độ rộng 1 bit.
<i>waitrequest</i>	Tớ → Chủ	Được sử dụng để dừng mô-đun bus Avalon khi cổng phụ không thể phản hồi

		ngay lập tức. Tín hiệu này có độ rộng 1 bit.
<i>readyfordata</i>	Tớ → Chủ	Tín hiệu cho việc truyền dữ liệu trực tuyến. Chỉ ra rằng tờ trực tuyến có thể nhận dữ liệu. Tín hiệu này có độ rộng 1 bit.
<i>dataavailable</i>	Tớ → Chủ	Tín hiệu cho việc truyền dữ liệu trực tuyến. Chỉ ra rằng tờ truyền dữ liệu có sẵn. Tín hiệu này có độ rộng 1 bit.
<i>endofpacket</i>	Tớ → Chủ	Tín hiệu cho truyền phát trực tuyến. Có thể được sử dụng để chỉ ra tình trạng “kết thúc gói tin” cho cổng chính. Triển khai cụ thể cho từng thiết bị ngoại vi. Tín hiệu này có độ rộng 1 bit.
<i>irq</i>	Tớ → Chủ	Yêu cầu ngắt. Tờ khẳng định <i>irq</i> khi nó cần được dùng bởi chủ. Tín hiệu này có độ rộng 1 bit.
<i>irqnumber</i>	Tớ → Chủ	Mức độ ưu tiên ngắt của cổng tờ ngắt. Giá trị thấp hơn có mức độ ưu tiên cao hơn. Tín hiệu này có độ rộng 6 bit.
<i>resetrequest</i>	Tớ → Chủ	Tín hiệu đặt lại cho phép thiết bị ngoại vi đặt lại toàn bộ mô-đun hệ thống. Tín hiệu này có độ rộng 1 bit.
<i>flush</i>	Chủ → Tớ	Tín hiệu cho các chuyển giao đọc có độ trễ. Chủ có thể xóa bất kỳ chuyển giao đọc tiềm ẩn nào đang chờ xử lý bằng cách tín hiệu <i>flush</i> . Tín hiệu này có độ rộng 1 bit.

Bảng 4.8 liệt kê các tín hiệu trong Avalon Bus với các mô tả chi tiết về chức năng và vai trò của mỗi tín hiệu. Các tín hiệu này được phân thành các nhóm chính như tín hiệu xung clock (*clk*), tín hiệu reset (*reset*), tín hiệu chọn chip (*chipselect*), tín hiệu địa chỉ (*address*), và các tín hiệu dữ liệu (*read*, *write*, *writedata*, *readdata*), cùng với một số tín hiệu hỗ trợ khác. Tín hiệu *clk* là nguồn xung nhịp toàn cục, giúp đồng bộ hóa toàn bộ hệ thống và đảm bảo các giao dịch trên bus Avalon diễn ra một cách chính xác. Tín hiệu *reset* được sử dụng để thiết lập lại toàn bộ hệ thống, đảm bảo các thiết bị ngoại vi bắt đầu trong trạng thái mặc định khi cần thiết. Tín hiệu *chipselect* giúp chọn chip từ trong một giao dịch Avalon, giúp xác định thiết bị nhận dữ liệu và điều khiển các giao dịch. Tín hiệu *address* xác định địa chỉ bộ nhớ hoặc thiết bị ngoại vi trong hệ thống, là cơ sở cho việc định tuyến dữ liệu đến đúng nơi cần thiết. Tín hiệu *begintransfer* đánh dấu sự bắt đầu của một giao dịch mới trên bus Avalon. Các tín hiệu *read* và *write* lần lượt yêu cầu đọc và ghi dữ liệu giữa chủ và tớ, trong khi *writedata* và *readdata* chuyển dữ liệu vào và ra từ các thiết bị. Bên cạnh đó, Avalon Bus còn sử dụng các tín hiệu như *byteenable* để điều chỉnh byte lane trong các giao dịch với bộ nhớ có độ rộng lớn hơn 8 bit, *waitrequest* để tạm dừng giao dịch khi cần thiết, và *readyfordata* cùng *dataavailable* để thông báo liệu dữ liệu đã sẵn sàng để truyền tải. Các tín hiệu *endofpacket*, *irq*, và *irqnumber* hỗ trợ việc truyền dữ liệu, xử lý ngắn và quản lý các sự kiện ngoại lệ trong hệ thống. Các tín hiệu này không chỉ giúp điều phối quá trình truyền tải dữ liệu mà còn đảm bảo tính đồng bộ và hiệu suất trong giao tiếp giữa các thiết bị.



Hình 4.30: Sơ đồ sử dụng Avalon Bus cơ bản cho quá trình đọc để giao tiếp giữa bộ xử lý Nios II và phần cứng IP tự thiết kế.



Hình 4.31: Sơ đồ sử dụng Avalon Bus cơ bản cho quá trình ghi để giao tiếp giữa bộ xử lý Nios II và phần cứng IP tự thiết kế.

Giả sử trong một hệ thống cơ bản giao tiếp giữa bộ xử lý Nios II và phần cứng IP tự thiết kế, như thể hiện trong Hình 4.30 và Hình 4.31, các tín hiệu trong Avalon Bus sẽ hoạt động như sau:

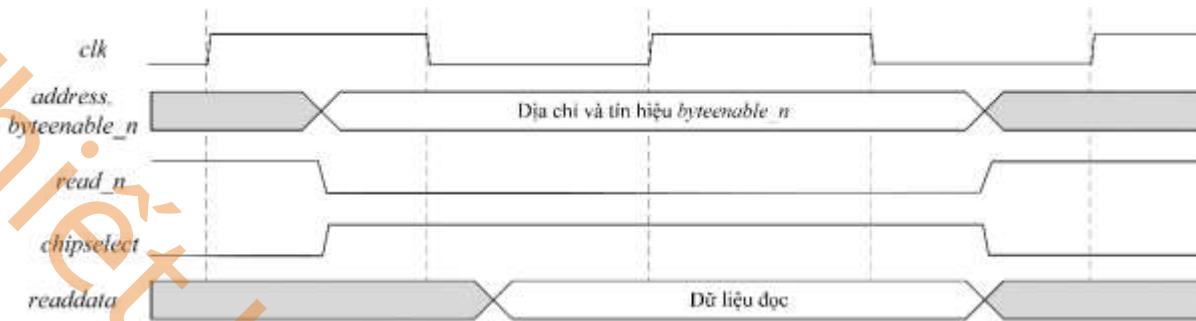
Hình 4.30 mô tả quá trình đọc dữ liệu từ IP tự thiết kế (tớ) về bộ xử lý Nios II (chủ) thông qua bus Avalon. Cụ thể, chủ sẽ yêu cầu dữ liệu từ tớ thông qua tín hiệu *address* để xác định địa chỉ của bộ nhớ hoặc thiết bị ngoại vi cần đọc dữ liệu. Khi tín hiệu *read* được kích hoạt, thiết bị tớ nhận biết yêu cầu và truyền dữ liệu qua tín hiệu *readdata* về cho chủ. Tín hiệu *byteenable* được sử dụng khi có yêu cầu kích hoạt các byte lane cụ thể trong các giao dịch với bộ nhớ có độ rộng lớn hơn 8 bit. Hình 4.31 mô tả quá trình ghi dữ liệu liệu từ bộ xử lý Nios II (chủ) sang IP tự thiết kế (tớ) thông qua Avalon. Tín hiệu *writedata* chứa dữ liệu mà chủ muốn ghi vào tớ. Đồng thời, tín hiệu *chipselect* được sử dụng để chọn chip tớ, đảm bảo rằng tín hiệu ghi và dữ liệu chỉ có hiệu lực khi thiết bị tớ đã được chọn. Tín hiệu *write* chỉ ra yêu cầu ghi dữ liệu và *byteenable* cho phép chủ chọn byte lane để ghi dữ liệu vào thiết bị tớ.

Nhìn chung, Avalon Bus có số lượng tín hiệu ít và đơn giản, nhưng vẫn đóng vai trò quan trọng trong việc đảm bảo sự đồng bộ và hiệu quả trong quá trình truyền dữ liệu giữa bộ xử lý Nios II và phần cứng IP tự thiết kế.

#### 4.6.3. Chi tiết tín hiệu của Avalon bus trong việc ghi đọc dữ liệu

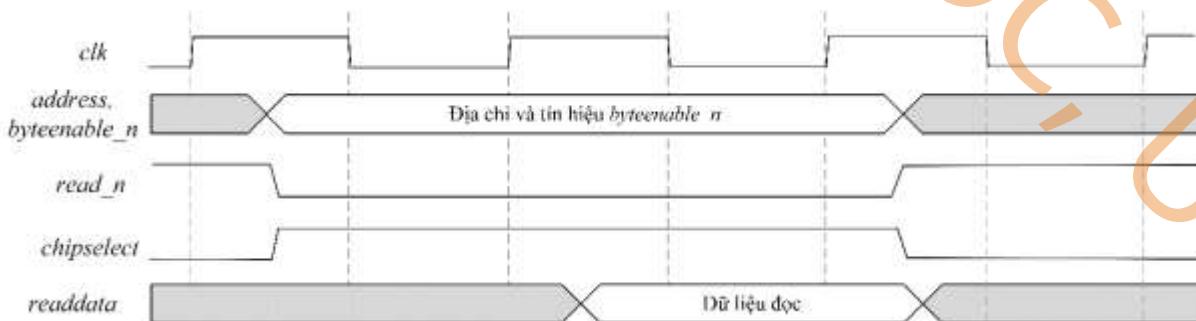
Trong phần này, chúng ta sẽ phân tích chi tiết các tín hiệu của Avalon Bus liên quan đến quá trình đọc dữ liệu. Avalon Bus hỗ trợ giao tiếp giữa bộ xử lý Nios II và phần cứng

IP tự thiết kế, với các tín hiệu được điều phối qua các chu kỳ đồng bộ để đảm bảo dữ liệu được truyền tải chính xác.



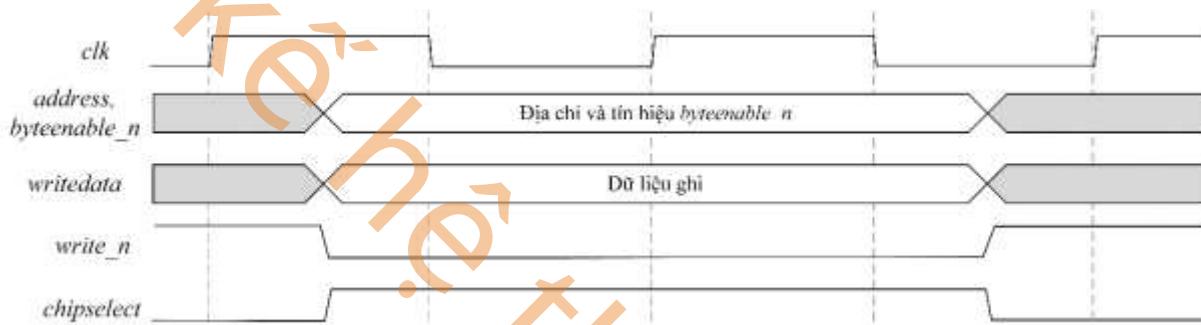
Hình 4.32: Sơ đồ thời gian chi tiết của tín hiệu chính cho đọc dữ liệu cơ bản của Avalon Bus.

Trong quá trình đọc dữ liệu cơ bản với không có trạng thái chờ, như được mô tả trong Hình 4.32, bộ xử lý Nios II (chủ) yêu cầu dữ liệu từ phần cứng IP tự thiết kế (tớ) qua bus Avalon. Tín hiệu *address* xác định địa chỉ bộ nhớ hoặc thiết bị ngoại vi cần đọc dữ liệu, và khi tín hiệu *read* được kích hoạt, nó nhận biết yêu cầu và chuẩn bị dữ liệu để trả về. Quá trình này không yêu cầu thời gian chờ, vì dữ liệu phải được trình bày ngay lập tức khi địa chỉ được chọn hoặc thay đổi. Tín hiệu *readdata* của tớ phải có giá trị hợp lệ và ổn định ngay trong chu kỳ xung clock tiếp theo để dữ liệu có thể được truyền tải chính xác từ tớ về chủ. Điều này đảm bảo quá trình đọc dữ liệu diễn ra ngay lập tức mà không có sự chậm trễ. Nếu dữ liệu không được truyền kịp thời, giao dịch có thể không thành công và bus Avalon sẽ không thu thập được dữ liệu chính xác từ tớ. Tuy nhiên, đối với các tớ đồng bộ, tức là các thiết bị có cổng nhập hoặc xuất đã được đăng ký, không thể sử dụng kiểu chuyển giao này vì chúng yêu cầu ít nhất một chu kỳ xung clock để lưu trữ và truyền tải dữ liệu hay đọc dữ liệu với trạng thái chờ cố định.



Hình 4.33: Sơ đồ thời gian chi tiết của tín hiệu chính cho đọc dữ liệu với một trạng thái chờ cố định của Avalon Bus.

Trong quá trình đọc dữ liệu với trạng thái chờ cố định, như được mô tả trong Hình 4.33, bus Avalon sẽ cung cấp địa chỉ và tín hiệu điều khiển trong chu kỳ xung clock đầu tiên, nhưng sẽ phải chờ thêm một chu kỳ xung clock trước khi nhận *readdata* từ tó. Điều này cho phép tó có đủ thời gian để chuẩn bị dữ liệu. Các tín hiệu như *address*, *byteenable\_n*, và *chipselect* vẫn được truyền tải từ chủ đến tó, nhưng tín hiệu *readdata* chỉ được truyền vào chu kỳ xung clock thứ hai và thu nhận bởi bus Avalon tại cạnh lên xung clock thứ ba, hoàn thành quá trình chuyển giao dữ liệu. Sự khác biệt với giao dịch đọc cơ bản là ở thời gian chờ, cho phép tó đồng bộ chuẩn bị dữ liệu một cách hợp lý.



Hình 4.34: Sơ đồ thời gian chi tiết của tín hiệu chính cho ghi dữ liệu cơ bản của Avalon Bus.

Tương tự với quá trình đọc, trong quá trình ghi dữ liệu cơ bản với không có trạng thái chờ, như được mô tả trong Hình 4.34, chủ sẽ gửi dữ liệu đến tó qua bus Avalon. Tín hiệu *address* xác định địa chỉ bộ nhớ hoặc thiết bị ngoại vi cần ghi dữ liệu, và khi tín hiệu *write* được kích hoạt, tó sẽ nhận yêu cầu ghi dữ liệu. Tín hiệu *writedata* chứa dữ liệu mà chủ muốn ghi vào tó. Quá trình này không yêu cầu thời gian chờ, vì dữ liệu phải được ghi ngay lập tức khi địa chỉ được chọn hoặc thay đổi. Tín hiệu *chipselect* giúp chọn tó để thực hiện giao dịch ghi. Nếu tín hiệu *writedata* không được truyền kịp thời, giao dịch có thể không thành công và bus Avalon sẽ không ghi dữ liệu chính xác vào tó.



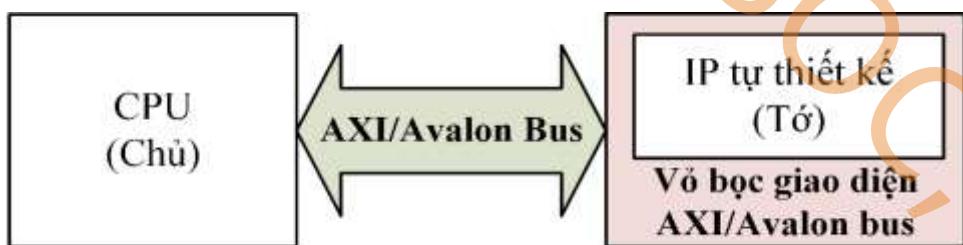
Hình 4.35: Sơ đồ thời gian chi tiết của tín hiệu chính cho ghi dữ liệu với một trạng thái chờ cố định của Avalon Bus.

Trong quá trình ghi dữ liệu với trạng thái chờ cố định, như được mô tả trong Hình 4.35, bus Avalon sẽ cung cấp địa chỉ và tín hiệu điều khiển trong chu kỳ xung clock đầu tiên, nhưng sẽ phải chờ thêm một chu kỳ xung clock trước khi nhận *writedata* từ chủ. Điều này cho phép có đủ thời gian để chuẩn bị ghi dữ liệu. Các tín hiệu như *address*, *byteenable\_n*, và *chipselect* vẫn được truyền tải từ chủ đến tớ, nhưng tín hiệu *writedata* chỉ được truyền vào chu kỳ xung clock thứ hai và thu nhận bởi bus Avalon tại cạnh lẻ xung clock thứ ba, hoàn thành quá trình ghi dữ liệu.

Nhìn chung, Avalon Bus cung cấp một cơ chế đồng bộ đơn giản và hiệu quả để truyền tải dữ liệu giữa bộ xử lý Nios II và phần cứng IP tự thiết kế, hỗ trợ cả giao dịch đọc và ghi với các tính năng như không có trạng thái chờ và có trạng thái chờ cố định. Các tín hiệu trong Avalon Bus, như *address*, *read*, *write*, và *writedata*, điều phối việc truyền dữ liệu qua các chu kỳ xung clock để đảm bảo hiệu quả và đồng bộ trong giao tiếp giữa chủ và tớ.

#### 4.7. Kết nối IP tự thiết kế vào hệ thống bus

Sau khi chúng ta cùng tìm hiểu chi tiết về các tín hiệu của AXI/Avalon bus trong việc ghi và đọc dữ liệu, phần này sẽ giải thích việc kết nối các IP tự thiết kế vào hệ thống bus và sử dụng các tín hiệu AXI/Avalon để giao tiếp với các thành phần khác. Đây là một phần quan trọng trong việc phát triển các hệ thống nhúng, giúp các IP tự thiết kế giao tiếp hiệu quả với các thành phần khác trong hệ thống như CPU và bộ nhớ.



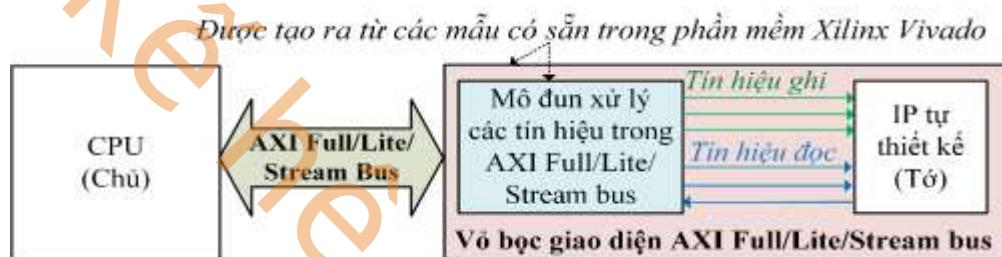
Hình 4.36: Minh họa hệ thống đơn giản kết nối giữa CPU và IP tự thiết kế thông qua AXI/Avalon Bus, với IP tự thiết kế đóng vai trò Tớ kết nối vào hệ thống AXI/Avalon bus.

Hình 4.36 minh họa một hệ thống đơn giản, trong đó CPU (Chủ) và IP tự thiết kế (Tớ) kết nối với nhau thông qua AXI/Avalon Bus. Trong đó, IP tự thiết kế đóng vai trò là Tớ và sẽ giao tiếp với CPU qua hệ thống bus AXI hoặc Avalon. Phần tiếp theo sẽ đi vào chi tiết

cách thức thiết kế và cấu hình giao diện AXI và Avalon để kết nối IP tự thiết kế vào hệ thống, bao gồm các bước trong việc xác định địa chỉ, xử lý tín hiệu điều khiển, và dữ liệu.

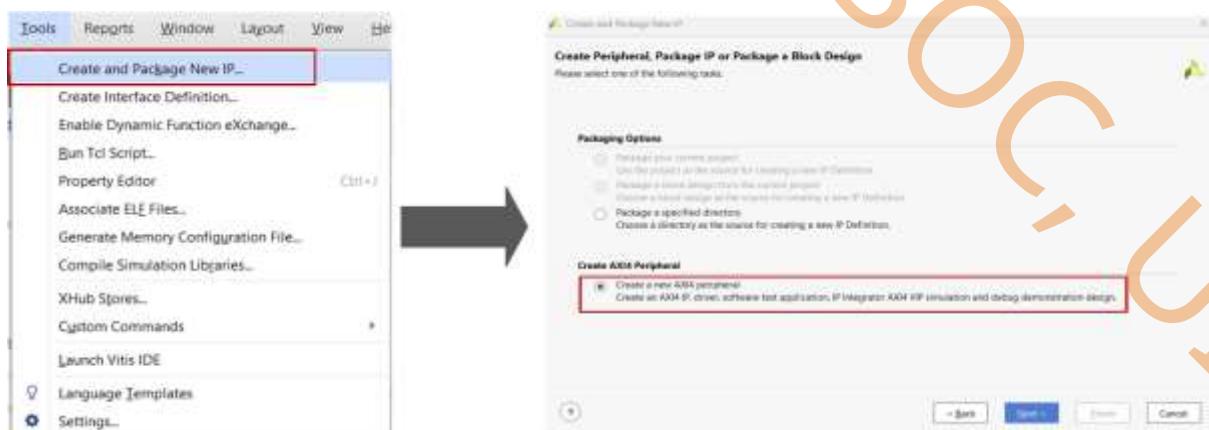
#### 4.7.1. Kết nối IP tự thiết kế vào hệ thống AXI bus

Hệ thống AXI bus hỗ trợ ba chuẩn khác nhau: AXI Full, AXI Lite và AXI Stream, mỗi chuẩn có cách thức hoạt động và số lượng tín hiệu khác nhau. Tuy nhiên, tất cả chúng đều có điểm chung là sử dụng vỏ bọc và mô-đun xử lý các tín hiệu trong AXI bus. Hình 4.37 minh họa sự kết nối giữa CPU và IP tự thiết kế qua AXI Full/Lite/Stream Bus, trong đó mô-đun xử lý tín hiệu sẽ đảm nhận việc truyền tải các tín hiệu ghi và đọc.



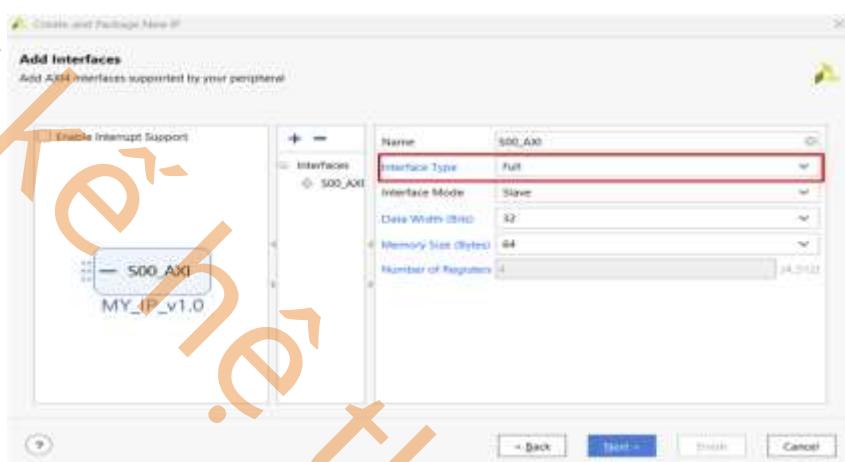
Hình 4.37: Minh họa hệ thống kết nối giữa CPU và IP tự thiết kế qua AXI Full/Lite/Stream Bus.

Người dùng có thể tự thiết kế mô-đun xử lý tín hiệu trong AXI bus để đáp ứng các yêu cầu cụ thể của hệ thống. Tuy nhiên, việc tự thiết kế mô-đun này là khá phức tạp. Do đó, phần này sẽ hướng dẫn người dùng sử dụng các mẫu có sẵn trong phần mềm Xilinx Vivado, giúp việc kết nối trở nên dễ dàng hơn. Các IP tự thiết kế sẽ chỉ sử dụng một vài tín hiệu cơ bản trong mô-đun xử lý tín hiệu được tạo ra từ các mẫu có sẵn, giúp CPU có thể ghi và đọc dữ liệu vào IP tự thiết kế thông qua hệ thống bus AXI.



Hình 4.38: Bước tạo một vỏ bọc giao diện AXI bus từ mẫu có sẵn cho IP tự thiết kế trong phần mềm Xilinx Vivado.

Bước tiếp theo là chọn Tools > Create and Package New IP từ thanh menu chính của phần mềm Vivado. Sau khi chọn tùy chọn này, cửa sổ Create Peripheral, Package IP or Package a Block Design sẽ xuất hiện, cho phép bạn chọn loại tác vụ muốn thực hiện. Trong trường hợp này, bạn sẽ chọn Create AXI Peripheral để tạo một vỏ bọc giao diện AXI cho IP tự thiết kế, như mô tả ở Hình 4.38. Đây là bước quan trọng để dễ dàng tích hợp IP vào hệ thống sử dụng AXI bus mà không cần thiết kế lại mô-đun xử lý tín hiệu từ đầu.



Hình 4.39: Chọn loại giao diện (Interface Type) khi tạo vỏ bọc AXI, với các tùy chọn AXI Full hoặc AXI Lite để xác định ba chuẩn khác nhau của giao diện AXI.

Trong phần Interface Type, bạn có thể chọn một trong ba loại giao diện AXI: AXI Full, AXI Lite, hoặc AXI Stream, như mô tả ở Hình 4.39. Sau khi chọn loại giao diện phù hợp, bạn sẽ tiếp tục cấu hình các tham số khác như Interface Mode (Tớ hoặc Chủ), Data Width, Memory Size, và Number of Registers để phù hợp với yêu cầu thiết kế của bạn.

#### Vỏ bọc giao diện AXI Full/Lite/Stream bus

#### Mô đun xử lý các tín hiệu trong AXI/Full/Lite/Stream bus

```

MY_IP_v1_0_500_AXI.v x Project Summary x Package IP + MY_IP x
/home/hoailuan/Research/2025/UIT_Book/AXI_Full_Project/MY_IP_v1_0/hdl/MY_IP_v1_0

1 *timescale 1 ns / 1 ps
2
3 module MY_IP_v1_0_500_AXI #(
4   // Users to add parameters here
5
6   // User parameters ends
7   // Do not modify the parameters beyond this line
8
9

```

Hình 4.40: Hai file code Verilog tương ứng với vỏ bọc giao diện AXI Full/Lite/Stream và mô đun xử lý các tín hiệu trong AXI Full/Lite/Stream bus được tạo ra sau bước chọn giao diện trong phần mềm Xilinx Vivado.

Sau bước chọn giao diện trong phần mềm Xilinx Vivado, hai file code Verilog tương ứng với vỏ bọc giao diện AXI Full/Lite/Stream bus và mô đun xử lý các tín hiệu trong AXI Full/Lite/Stream bus được tạo ra như Hình 4.40. Để làm rõ cách IP tự thiết kế lấy các tín hiệu điều khiển đọc ghi từ mô đun xử lý các tín hiệu AXI Full/Lite/Stream bus, chúng tôi phân tích code Verilog cho từng loại giao diện. Trong đó, IP tự thiết kế thực hiện phép tính  $y = a * x + b$ , sử dụng các tín hiệu từ mô đun xử lý để ghi và đọc dữ liệu từ các thanh ghi  $a$ ,  $b$ ,  $x$ , và  $y$ , qua đó thực hiện phép toán này.

```

reg [31:0] y, a, b, x;
always @(* posedge S_AXI_ACLK) begin
    if (S_AXI_ARESETN == 1'b0) begin
        a <= 0;
        b <= 0;
        x <= 0;
    end
    else begin
        if(axi_awv_awr_flag && (axi_awaddr == 32'hA0000000))
            a <= S_AXI_WDATA;
        else if(axi_awv_awr_flag && (axi_awaddr == 32'hA0000004))
            b <= S_AXI_WDATA;
        else if(axi_awv_awr_flag && (axi_awaddr == 32'hA0000008))
            x <= S_AXI_WDATA;
    end
end

```

Xử lý ghi dữ liệu vào biến  $a$   $b$   $x$

```

always @(* posedge S_AXI_ACLK) begin
    if (S_AXI_ARESETN == 1'b0)
        y <= 0;
    else
        y <= a*x + b;
end

```

Tính toán chính của IP tự thiết kế ( $y = a*x + b$ )

```

always @(* posedge S_AXI_ACLK) begin
    if (S_AXI_ARESETN == 1'b0) begin
        axi_rdata <= 0;
    end
    else begin
        if (axi_arv_awr_flag && (axi_araddr == 32'hA0000000))
            axi_rdata <= y;
        else
            axi_rdata <= axi_rdata;
    end
end

```

Xử lý đọc dữ liệu của biến  $y$

Hình 4.41: Code Verilog mô tả IP tự thiết kế sử dụng 6 tín hiệu từ mô đun xử lý các tín hiệu AXI Full bus để ghi, đọc dữ liệu vào các biến và thực hiện tính toán  $y = a*x + b$ .

Trong Hình 4.41, mã Verilog mô tả cách IP tự thiết kế sử dụng mô đun xử lý các tín hiệu AXI Full bus để thực hiện các thao tác ghi và đọc dữ liệu vào các biến, đồng thời thực hiện tính toán  $y = a * x + b$ . Cụ thể, mô đun này sử dụng 6 tín hiệu từ AXI Full bus để xử lý các yêu cầu ghi và đọc dữ liệu. Tín hiệu *axi\_awv\_awr\_flag* cho biết có yêu cầu ghi từ chủ (CPU) đến tớ (IP tự thiết kế). Tín hiệu *axi\_awaddr* chỉ ra địa chỉ của thanh ghi mà dữ

liệu sẽ được ghi vào. Trong ví dụ trên, các địa chỉ như  $32'hA0000000$ ,  $32'hA0000004$ , và  $32'hA0000008$  được sử dụng để lưu trữ các giá trị vào các thanh ghi a, b, và x. Tín hiệu  $S_AXI_WDATA$  mang theo dữ liệu ghi từ chủ đến tớ. Khi cần đọc dữ liệu, tín hiệu  $axi_arv_arr_flag$  được kích hoạt để xác nhận yêu cầu đọc, và  $axi_araddr$  chỉ ra địa chỉ của thanh ghi mà chủ muốn đọc dữ liệu, ở đây là địa chỉ cho thanh ghi y là  $32'hA0000000$ . Cuối cùng, tín hiệu  $axi_rdata$  mang dữ liệu trả về từ tớ về chủ. Các tín hiệu này giúp thực hiện các thao tác ghi và đọc dữ liệu vào các thanh ghi a, b, x, y, và thực hiện tính toán ở IP tự thiết kế.

```

reg [31:0] y, a, b, x;

always @(posedge S_AXI_ACLK) begin
    if (S_AXI_ARESETN == 1'b0) begin
        a <= 0;
        b <= 0;
        x <= 0;
    end
    else begin
        if(slv_reg_wren && (axi_awaddr == 32'hA0000000))
            a <= S_AXI_WDATA;
        else if(slv_reg_wren && (axi_awaddr == 32'hA0000004))
            b <= S_AXI_WDATA;
        else if(slv_reg_wren && (axi_awaddr == 32'hA0000008))
            x <= S_AXI_WDATA;
    end
end

```

*Xử lý ghi dữ liệu  
vào biến a b x*

```

always @(posedge S_AXI_ACLK) begin
    if (S_AXI_ARESETN == 1'b0)
        y <= 0;
    else
        y <= a*x + b;
end

```

*Tính toán chính  
của IP tự thiết kế  
 $(y = a*x + b)$*

```

always @(posedge S_AXI_ACLK) begin
    if (S_AXI_ARESETN == 1'b0) begin
        axi_rdata <= 0;
    end
    else begin
        if(slv_reg_rden && (axi_araddr == 32'hA0000000))
            axi_rdata <= y;
        else
            axi_rdata <= axi_rdata;
    end
end

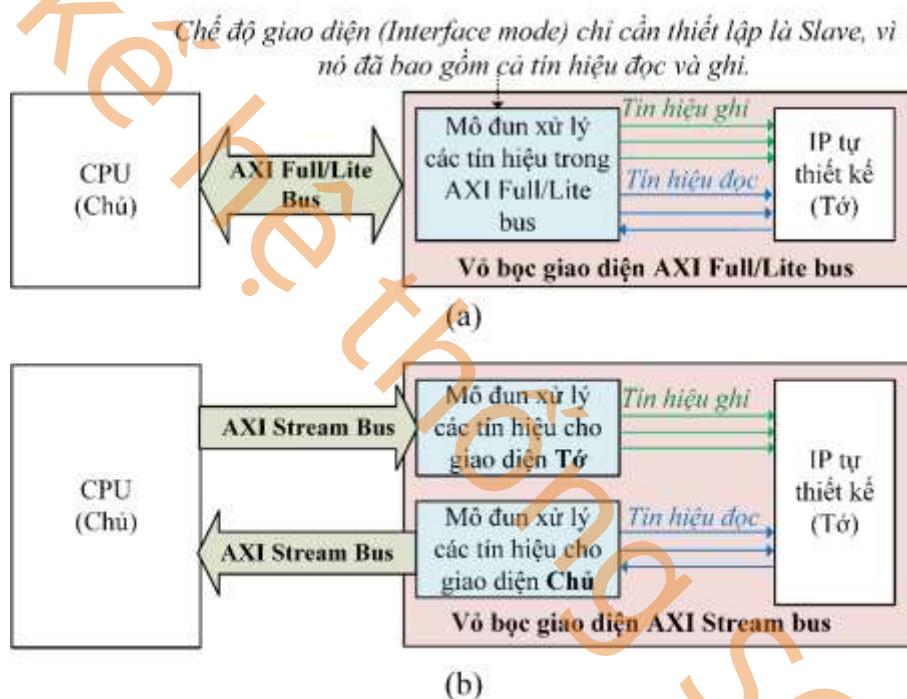
```

*Xử lý đọc dữ  
liệu của biến y*

Hình 4.42: Code Verilog mô tả IP tự thiết kế sử dụng 6 tín hiệu từ mô đun xử lý các tín hiệu AXI Lite bus để ghi, đọc dữ liệu vào các biến và thực hiện tính toán  $y = a * x + b$ .

Hình 4.42 mô tả mã Verilog cho IP tự thiết kế sử dụng mô đun xử lý các tín hiệu từ AXI Lite bus để ghi, đọc dữ liệu vào các biến và thực hiện tính toán  $y = a * x + b$ . Mô đun này sử dụng các tín hiệu như  $slv\_reg\_wren$  để xác nhận yêu cầu ghi dữ liệu từ chủ (CPU)

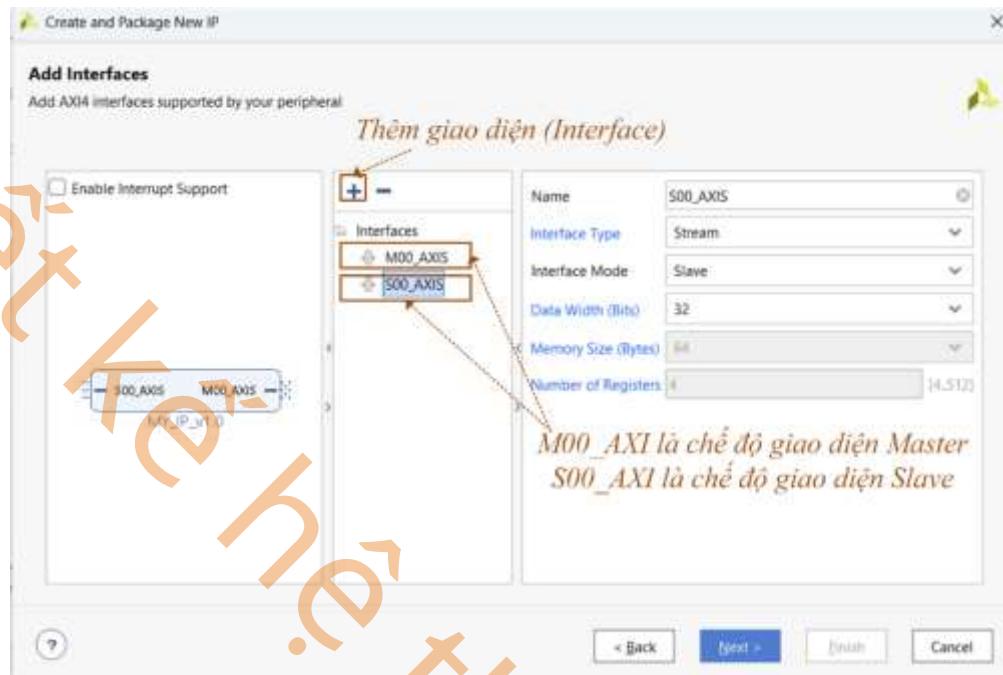
đến tớ (IP tự thiết kế), tín hiệu *axi\_awaddr* chỉ ra địa chỉ của thanh ghi để ghi dữ liệu vào (các địa chỉ như 32'hA0000000, 32'hA0000004 và 32'hA0000008 cho các thanh ghi a, b, và x), và tín hiệu *S\_AXI\_WDATA* mang dữ liệu ghi từ chủ đến tớ. Để thực hiện đọc dữ liệu, tín hiệu *slv\_reg\_rden* xác nhận yêu cầu đọc từ chủ, *axi\_araddr* chỉ ra địa chỉ thanh ghi mà chủ muốn đọc dữ liệu từ (ví dụ, thanh ghi y ở địa chỉ 32'hA0000000), và tín hiệu *axi\_rdata* mang dữ liệu trả về từ tớ về chủ. Các tín hiệu này giúp thực hiện thao tác ghi và đọc dữ liệu vào các thanh ghi a, b, x, y, đồng thời thực hiện phép toán  $y = a * x + b$  ở IP tự thiết kế.



Hình 4.43: Khác biệt cấu hình vỏ bọc giao diện AXI Full/Lite và AXI Stream: (a) Vỏ bọc giao diện AXI Full/Lite với chế độ Tớ hỗ trợ cả tín hiệu đọc và ghi; (b) Vỏ bọc giao diện AXI Stream với chế độ Tớ chỉ hỗ trợ tín hiệu ghi và chế độ Chủ chỉ hỗ trợ tín hiệu đọc.

Lưu ý là có sự khác biệt giữa giao diện AXI Full/Lite và AXI Stream ở cấu hình vỏ bọc giao diện và chế độ giao diện, như mô tả trong Hình 4.43. Đối với AXI Full/Lite, khi chế độ giao diện được thiết lập là Tớ, nó có thể hỗ trợ cả tín hiệu đọc và ghi, như mô tả trong Hình 4.43 (a). Trong khi đó, với AXI Stream, chế độ giao diện Tớ chỉ hỗ trợ tín hiệu ghi, và chế độ giao diện Chủ chỉ hỗ trợ tín hiệu đọc, như mô tả trong Hình 4.43 (b). Vì vậy, cấu hình vỏ bọc và mô-đun xử lý tín hiệu trong AXI Stream sẽ có sự khác biệt so với AXI

Full/Lite, do yêu cầu về tín hiệu đọc và ghi được phân tách giữa Tớ và Chủ trong hệ thống AXI Stream.



Hình 4.44: Thêm giao diện AXI Stream cho IP tự thiết kế, với M00\_AXIS là chế độ giao diện Chủ và S00\_AXIS là chế độ giao diện Tớ.

Để thêm giao diện IP thiết kế có cả giao diện Chủ và Tớ, cần thêm hai giao diện AXI Stream cho IP tự thiết kế như mô tả trong Hình 4.44. Trong đó, M00\_AXIS được cấu hình là chế độ giao diện Chủ, và S00\_AXIS được cấu hình là chế độ giao diện Tớ. Đây là sự khác biệt quan trọng khi làm việc với AXI Stream, vì chế độ giao diện Tớ chỉ hỗ trợ tín hiệu ghi, trong khi chế độ giao diện Chủ chỉ hỗ trợ tín hiệu đọc. Chế độ giao diện (Interface mode) cần phải chọn Stream cho cả hai giao diện Chủ và Tớ để đảm bảo hoạt động chính xác trong hệ thống AXI Stream.

Trong Hình 4.45, mã Verilog mô tả IP tự thiết kế sử dụng 6 tín hiệu từ mô đun xử lý các tín hiệu AXI Stream của giao diện Chủ và Tớ để ghi, đọc dữ liệu vào các biến và thực hiện tính toán  $y = a * x + b$ . Vì không có địa chỉ trong AXI Stream, tín hiệu write\_pointer (từ giao diện Tớ) được sử dụng để đếm số lượng input vào và xác định việc ghi dữ liệu vào các thanh ghi a, b, và x. Tín hiệu fifo\_wren (từ giao diện Tớ) kiểm tra và điều khiển quá trình ghi dữ liệu vào các thanh ghi này. Tín hiệu M\_AXIS\_ARESETN (từ giao diện chủ) và

S\_AXIS\_ARESETN (từ giao diện tớ) được sử dụng để thiết lập lại các giá trị khi có reset. Phép toán chính của IP tự thiết kế được thực hiện trong phần code thứ hai, nơi giá trị của y được tính toán từ a và x, rồi cộng với b. Cuối cùng, tín hiệu stream\_data\_out (từ giao diện chủ) đảm bảo việc truyền dữ liệu từ IP ra ngoài khi có yêu cầu đọc, với tín hiệu tx\_en (từ giao diện chủ) điều khiển việc xuất dữ liệu ra ngoài hệ thống. Các tín hiệu này giúp thực hiện các thao tác ghi, đọc và tính toán tại IP tự thiết kế trong môi trường AXI Stream.

```

reg [31:0] y, a, b, x;

always @(posedge S_AXI_ACLK) begin
    if (S_AXIS_ARESETN == 1'b0) begin
        a <= 0;
        b <= 0;
        x <= 0;
    end
    else begin
        if(fifo_wren && (write_pointer == 0))
            a <= S_AXIS_TDATA;
        else if(fifo_wren && (write_pointer == 1))
            b <= S_AXIS_TDATA;
        else if(fifo_wren && (write_pointer == 2))
            x <= S_AXIS_TDATA;
    end
end

```

*Xử lý ghi dữ liệu vào biến a b x*

```

always @(posedge S_AXI_ACLK) begin
    if (M_AXIS_ARESETN == 1'b0)
        y <= 0;
    else
        y <= a*x + b;
end

```

*Tính toán chính của IP tự thiết kế ( $y = a*x + b$ )*

```

always @(posedge S_AXI_ACLK) begin
    if (M_AXIS_ARESETN == 1'b0) begin
        stream_data_out <= 0;
    end
    else begin
        if (tx_en)
            stream_data_out <= y;
    end
end

```

*Xử lý đọc dữ liệu của biến y*

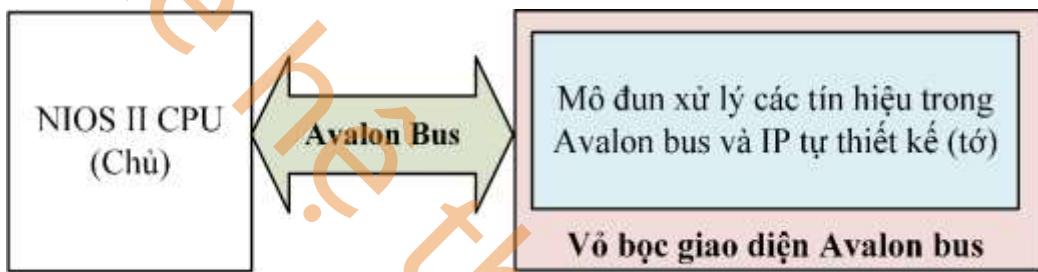
Hình 4.45: Code Verilog mô tả IP tự thiết kế sử dụng 5 tín hiệu từ mô đun xử lý các tín hiệu AXI Stream bus của giao diện Chủ và Tớ để ghi, đọc dữ liệu vào các biến và thực hiện tính toán  $y = a*x + b$ .

Nhìn chung, khi kết nối IP tự thiết kế vào hệ thống AXI bus, do tính phức tạp của AXI, người dùng thường sử dụng các vỏ bọc giao diện theo mẫu có sẵn do Xilinx cung cấp, kết hợp với việc sử dụng 5 hoặc 6 tín hiệu cần thiết để quản lý các thao tác ghi và đọc

dữ liệu một cách đơn giản. Đây là giải pháp phù hợp cho các hệ thống đơn giản hoặc yêu cầu tích hợp nhanh chóng. Tuy nhiên, đối với các hệ thống phức tạp, người dùng có thể cần phải tự thiết kế toàn bộ quá trình xử lý các tín hiệu trong AXI bus để đáp ứng yêu cầu cụ thể của ứng dụng.

#### 4.7.2. Kết nối IP tự thiết kế vào hệ thống Avalon bus

Trong phần này, chúng ta sẽ tìm hiểu về cách kết nối IP tự thiết kế vào hệ thống Avalon bus. Avalon bus là một giao thức bus đơn giản, yêu cầu chỉ một vài tín hiệu để kết nối các thành phần trong hệ thống, điều này làm cho việc triển khai trở nên dễ dàng hơn so với các giao thức phức tạp hơn như AXI bus.



Hình 4.46: Minh họa hệ thống kết nối giữa CPU và IP tự thiết kế qua Avalon Bus.

Như mô tả trong Hình 4.46, kết nối giữa CPU và IP tự thiết kế qua Avalon bus được thực hiện thông qua mô đun xử lý các tín hiệu giao tiếp trong Avalon bus. Vì sự đơn giản của Avalon bus, phép toán tính toán trong IP tự thiết kế (như phép tính  $y = a * x + b$ ) sẽ được gộp chung vào xử lý tín hiệu của Avalon bus, giúp giảm thiểu độ phức tạp trong thiết kế và triển khai hệ thống.

Trong Hình 4.47, mã Verilog mô tả cách IP tự thiết kế kết hợp với mô-đun xử lý các tín hiệu Avalon bus để ghi, đọc dữ liệu vào các biến và thực hiện phép tính  $y = a * x + b$ . Mô đun này sử dụng các tín hiệu như iChipSelect\_n, iWrite\_n, và iRead\_n để xác nhận các yêu cầu ghi và đọc dữ liệu giữa CPU (Chủ) và IP tự thiết kế (Tớ). Khi chế độ giao diện là ghi (tín hiệu iWrite\_n), các giá trị từ iData sẽ được ghi vào các thanh ghi a, b, và x dựa trên địa chỉ được chỉ ra bởi iAddress. Ngược lại, khi chế độ giao diện là đọc (tín hiệu iRead\_n), giá trị của các thanh ghi a, b, x, hoặc phép toán  $y = a * x + b$  sẽ được trả về thông qua oData. Tín hiệu iChipSelect\_n được sử dụng để chọn lựa việc thực hiện ghi hoặc đọc, và iAddress chỉ ra địa chỉ của thanh ghi mà dữ liệu sẽ được ghi vào hoặc từ đó sẽ được đọc.

Qua đó, các tín hiệu này giúp thực hiện các thao tác ghi, đọc và phép toán ở IP tự thiết kế trong môi trường Avalon bus, làm cho việc kết nối và xử lý trở nên mượt mà và dễ dàng hơn.

```

module Compute(
    input iClk,
    input iReset_n,
    input iChipSelect_n,
    input iWrite_n,
    input iRead_n,
    input [1:0] iAddress,
    input [31:0] iData,
    output reg [31:0] oData
);
    reg [31:0] a, b, x;

    always @(posedge iClk or negedge iReset_n) begin
        if (~iReset_n) begin
            oData <= 32'd0;
            a <= 32'd0;
            b <= 32'd0;
            x <= 32'd0;
        end else begin
            if (~iChipSelect_n & ~iWrite_n) begin
                case (iAddress)
                    2'd0: a <= iData[3:0];
                    2'd1: b <= iData[3:0];
                    2'd2: x <= iData[3:0];
                endcase
            end
            if (~iChipSelect_n & ~iRead_n) begin
                case (iAddress)
                    2'd0: oData <= a;
                    2'd1: oData <= b;
                    2'd2: oData <= x;
                    2'd3: oData <= a * x + b;
                endcase
            end
        end
    end
endmodule

```

*Xử lý ghi dữ liệu vào biến a b x*

*Kết hợp xử lý đọc dữ liệu và tính toán chính của IP tự thiết kế ( $y = a * x + b$ )*

Hình 4.47: Code Verilog mô tả IP tự thiết kế kết hợp với mô-đun xử lý các tín hiệu Avalon bus để ghi, đọc dữ liệu vào các biến và thực hiện phép tính  $y = a * x + b$ .

Trong Hình 4.48, chúng ta có thể thấy cấu hình các tín hiệu của IP tự thiết kế (mô-đun Compute) trong phần mềm Quartus Prime. Bảng này liệt kê các tín hiệu đầu vào và đầu ra của mô-đun, bao gồm các tín hiệu như iClk (đầu vào clock), iReset\_n (đầu vào reset), iChipSelect\_n (đầu vào chip select), iWrite\_n (đầu vào tín hiệu ghi), iRead\_n (đầu vào tín hiệu đọc), iAddress (đầu vào địa chỉ), iData (đầu vào dữ liệu), và oData (đầu ra dữ liệu).

Các tín hiệu này được cấu hình với các loại tín hiệu và chiều (input hoặc output) tương ứng, giúp IP tự thiết kế giao tiếp với các thành phần khác trong hệ thống, đặc biệt là với giao diện Avalon bus. Từ đây, việc cấu hình tín hiệu trong phần mềm Quartus Prime giúp đơn giản hóa quá trình kết nối và điều khiển dữ liệu cho IP tự thiết kế trong hệ thống.

Name	Interface	Signal Type	Width	Direction
iClk	clock_sink	clk	1	input
iReset_n	reset_sink	reset_n	1	input
iChipSelect_n	avalon_slave_0	chipselect_n	1	input
iWrite_n	avalon_slave_0	write_n	1	input
iRead_n	avalon_slave_0	read_n	1	input
iAddress	avalon_slave_0	address	2	input
iData	avalon_slave_0	writedata	32	input
oData	avalon_slave_0	readdata	32	output

Hình 4.48: Cấu hình các tín hiệu của IP tự thiết kế (mô đun Compute) trên phần mềm Quartus Prime.

Nhìn chung, việc kết nối IP tự thiết kế vào hệ thống Avalon bus giúp đơn giản hóa quá trình giao tiếp giữa các thành phần trong hệ thống nhờ vào giao thức Avalon bus với số tín hiệu tối thiểu. Như đã trình bày, Avalon bus yêu cầu ít tín hiệu hơn so với các giao thức phức tạp như AXI bus, giúp giảm bớt độ phức tạp trong thiết kế hệ thống. Các phép toán và thao tác đọc, ghi dữ liệu trong IP tự thiết kế có thể được tích hợp trực tiếp vào xử lý tín hiệu Avalon bus, giúp việc triển khai hệ thống trở nên dễ dàng hơn.

## 4.8. Tóm tắt

Chương này đã trình bày chi tiết về các giao thức kết nối trong SoC, bao gồm APB, AHB, AXI4 Full, AXI4-Lite và AXI4-Stream, cùng với các tín hiệu và kết nối đi kèm. Mỗi giao thức này có những đặc điểm và ứng dụng cụ thể nhằm đáp ứng yêu cầu kết nối giữa các IP cores khác nhau trên SoC. APB được sử dụng chủ yếu cho các thiết bị ngoại vi với băng thông thấp và độ phức tạp đơn giản, trong khi AHB cung cấp một giải pháp kết nối tốc độ cao với nhiều chủ và tớ. AXI4 là giao thức linh hoạt nhất, hỗ trợ các giao dịch Burst và truyền dữ liệu song song, phù hợp cho các ứng dụng đòi hỏi hiệu suất cao và băng thông lớn. Chương này cũng phân tích chi tiết cách thức hoạt động của các tín hiệu trong các giao dịch đọc và ghi dữ liệu, bao gồm các sơ đồ thời gian cho từng loại giao thức.

Vì môn học này tập trung thực hành vào thiết kế SoC trên Xilinx FPGA, việc hiểu rõ về AXI4 là rất quan trọng. AXI4 Full, với tính năng hỗ trợ các giao dịch phức tạp và khả năng tối ưu hóa băng thông, là giao thức thường được sử dụng nhất khi thiết kế các hệ thống SoC trên nền tảng FPGA. Tuy nhiên, bên cạnh AXI4, Avalon Bus của Intel (trước đây là Altera) cũng là một giao thức phổ biến trong các hệ thống SoC, đặc biệt là trong các thiết kế FPGA của Intel. Avalon Bus nổi bật với tính đơn giản, dễ triển khai và hỗ trợ giao tiếp hiệu quả giữa các bộ xử lý và thiết bị ngoại vi. Do đó, sinh viên cần lưu ý kỹ về cách thức hoạt động, tín hiệu, và các chế độ giao dịch của cả AXI4 và Avalon Bus để có thể thiết kế và tối ưu hóa các kết nối trong các ứng dụng thực tế.

#### 4.9. Câu hỏi và bài tập

1. Trình bày sự khác biệt giữa giao thức APB và AHB trong hệ thống SoC. Khi nào nên sử dụng APB thay vì AHB?
2. Giải thích cơ chế truyền dữ liệu trong giao thức AXI4 Full. Các kênh dữ liệu trong AXI4 Full hoạt động như thế nào để tối ưu hóa băng thông?
3. So sánh giữa AXI4 Full và AXI4-Lite. Đặc điểm nào khiến AXI4-Lite phù hợp hơn cho các ứng dụng yêu cầu kết nối đơn giản?
4. Mô tả chức năng và vai trò của các tín hiệu TVALID và TREADY trong giao thức AXI4-Stream. Tại sao chúng quan trọng đối với việc truyền dữ liệu?

5. Phân tích ưu và nhược điểm của việc sử dụng giao thức AXI4-Stream so với các giao thức khác trong AMBA khi truyền dữ liệu không cần địa chỉ.
6. Mô tả các tín hiệu chính của AXI4-Lite sử dụng trong giao dịch ghi và đọc dữ liệu.  
Tại sao AXI4-Lite không hỗ trợ giao dịch Burst?
7. Thiết kế một sơ đồ kết nối đơn giản sử dụng các giao thức APB, AHB, và AXI trong một hệ thống SoC. Giải thích cách các giao thức này tương tác với nhau thông qua các cầu nối (bridges).
8. Trong trường hợp sử dụng AXI4 Full, hãy mô tả quy trình giao dịch Burst từ chủ đến tớ. Những tín hiệu nào cần được chú ý để đảm bảo giao dịch diễn ra suôn sẻ?
9. Trình bày các tín hiệu chính trong giao thức Avalon Bus và giải thích vai trò của chúng trong quá trình giao tiếp giữa bộ xử lý Nios II và phần cứng IP tự thiết kế.
10. So sánh giữa Avalon Bus và AXI4 trong hệ thống SoC. Đặc điểm nào của Avalon Bus khiến nó trở thành một lựa chọn phù hợp cho các thiết kế FPGA của Intel?
11. Thiết kế IP thực hiện tính toán cho dãy số gồm N giá trị. Thực hiện bọc IP tính toán trên theo chuẩn giao tiếp avalon bus và AXI bus
12. Hiện thực hệ thống SoC có tích hợp IP tự thiết kế ở bài 11. Viết chương trình kiểm tra kết quả hoạt động của IP. So sánh kết quả khi chạy với chuẩn giao tiếp avalon bus và AXI bus.

## CHƯƠNG 5: TIÊU CHUẨN ĐÁNH GIÁ HỆ THỐNG SOC TRÊN FPGA

### 5.1. Giới thiệu

Thông qua nội dung từ chương 1 đến chương 4, chúng tôi hy vọng người đọc có thể hiểu được khái niệm về hệ thống SoC trên FPGA, nắm rõ quy trình thiết kế và cách tích hợp hiệu quả các thành phần quan trọng. Những thành phần này bao gồm các IP cơ bản như CPU, bộ nhớ, bus kết nối, và các IP tự thiết kế để đáp ứng nhu cầu cụ thể. Đặc biệt, chúng tôi nhấn mạnh tầm quan trọng của việc tích hợp các thành phần này nhằm tạo ra một hệ thống SoC hoàn chỉnh, vận hành ổn định trên nền FPGA. Sau khi thiết kế và hoàn thiện hệ thống SoC trên FPGA, việc đánh giá dựa trên các tiêu chí quan trọng sẽ giúp xác định xem hệ thống có đáp ứng được yêu cầu ứng dụng thực tế hay không, đồng thời nhận diện các ưu điểm và nhược điểm cần cải thiện.

Các tiêu chuẩn đánh giá hệ thống SoC trên FPGA là cần thiết để đảm bảo hiệu quả và độ tin cậy của hệ thống. Thứ nhất, các tiêu chuẩn giúp xác định các thông số kỹ thuật rõ ràng, bao gồm hiệu năng, tiêu thụ năng lượng, độ trễ và độ chính xác. Việc này hỗ trợ tối ưu hóa thiết kế để đáp ứng yêu cầu thực tế của ứng dụng. Thứ hai, các tiêu chuẩn cung cấp một khung tham chiếu để so sánh và đánh giá các thiết kế khác nhau, giúp các nhà phát triển lựa chọn giải pháp tối ưu nhất cho hệ thống. Thứ ba, việc tiêu chuẩn hóa đảm bảo sự tương thích với các thành phần phần cứng và phần mềm khác, tăng khả năng tái sử dụng và giảm chi phí phát triển. Cuối cùng, các tiêu chuẩn này giúp giảm rủi ro phát sinh lỗi và đảm bảo rằng hệ thống SoC trên FPGA hoạt động ổn định và đáp ứng được các yêu cầu ứng dụng thực tiễn. Những tiêu chuẩn này không chỉ là công cụ hỗ trợ kỹ thuật mà còn giúp hệ thống SoC đạt được chất lượng cao, độ tin cậy và phù hợp với mục tiêu thiết kế.

**Tài nguyên sử dụng** trên FPGA, bao gồm diện tích (như LUTs, FFs, DSPs, BRAMs) và năng lượng, là một tiêu chí quan trọng không thể bỏ qua trong thiết kế. Việc sử dụng hiệu quả các tài nguyên này không chỉ ảnh hưởng trực tiếp đến khả năng tích hợp và tối ưu của hệ thống, mà còn tác động đáng kể đến diện tích khi chuyển đổi thiết kế từ FPGA sang

IP hoặc chip thực tế. Một thiết kế tốt cần tận dụng tối đa các tài nguyên FPGA sẵn có mà không làm gia tăng độ phức tạp, gây xung đột giữa các chức năng, hoặc lãng phí diện tích. Bên cạnh đó, việc tối ưu hóa năng lượng tiêu thụ trên FPGA cũng đóng vai trò quan trọng, giúp giảm sự tiêu hao năng lượng không cần thiết, đồng thời cải thiện hiệu năng hoạt động của hệ thống.

**Hiệu năng hệ thống** luôn được xem là yếu tố hàng đầu trong việc đánh giá một hệ thống SoC trên FPGA. Hiệu năng phản ánh khả năng xử lý mạnh mẽ của SoC khi thực hiện các tác vụ với độ chính xác và tốc độ cao. Các chỉ số quan trọng để đo lường hiệu năng bao gồm độ trễ, thông lượng, tần số hoạt động, công suất và năng lượng tiêu thụ.

**Hiệu quả hệ thống** bao gồm hai yếu tố quan trọng: hiệu quả diện tích và hiệu quả năng lượng, đóng vai trò quyết định trong việc tối ưu hóa tài nguyên và đảm bảo hệ thống hoạt động hiệu quả. Hiệu quả diện tích liên quan đến việc sử dụng tài nguyên phần cứng như LUTs, FFs, DSPs và BRAMs một cách tối ưu để giảm thiểu không gian cần thiết mà vẫn đảm bảo hiệu năng cao. Trong khi đó, hiệu quả năng lượng tập trung vào việc giảm thiểu mức tiêu thụ năng lượng của hệ thống, giúp tiết kiệm chi phí vận hành và kéo dài tuổi thọ của thiết bị. Tuy nhiên, việc tối ưu hóa hai yếu tố này thường là một sự đánh đổi, vì tăng hiệu năng hoặc giảm tiêu thụ năng lượng có thể làm tăng diện tích sử dụng và ngược lại. Do đó, các nhà thiết kế cần phải tìm ra sự cân bằng hợp lý giữa hiệu năng, diện tích và năng lượng để đạt được một hệ thống tối ưu, phù hợp với yêu cầu ứng dụng và khả năng tài nguyên.

**Độ tin cậy** là một tiêu chí quan trọng trong thiết kế hệ thống SoC trên FPGA, đảm bảo hệ thống hoạt động ổn định và chính xác trong các điều kiện thực tế. Một hệ thống SoC đáng tin cậy phải có khả năng chịu đựng các lỗi phát sinh từ phần cứng, phần mềm, và môi trường như nhiệt độ cao, nhiễu điện từ, hoặc suy hao điện áp. Điều này đòi hỏi thiết kế phải tích hợp các cơ chế phát hiện và sửa lỗi (Error Detection and Correction), bảo vệ dữ liệu trong bộ nhớ, và quản lý nguồn hiệu quả. Ngoài ra, các hệ thống FPGA cũng cần kiểm tra kỹ lưỡng qua các giai đoạn kiểm thử để đảm bảo độ bền trong vận hành lâu dài, đặc biệt trong các ứng dụng quan trọng như y tế, quốc phòng, và hàng không vũ trụ. Độ tin cậy

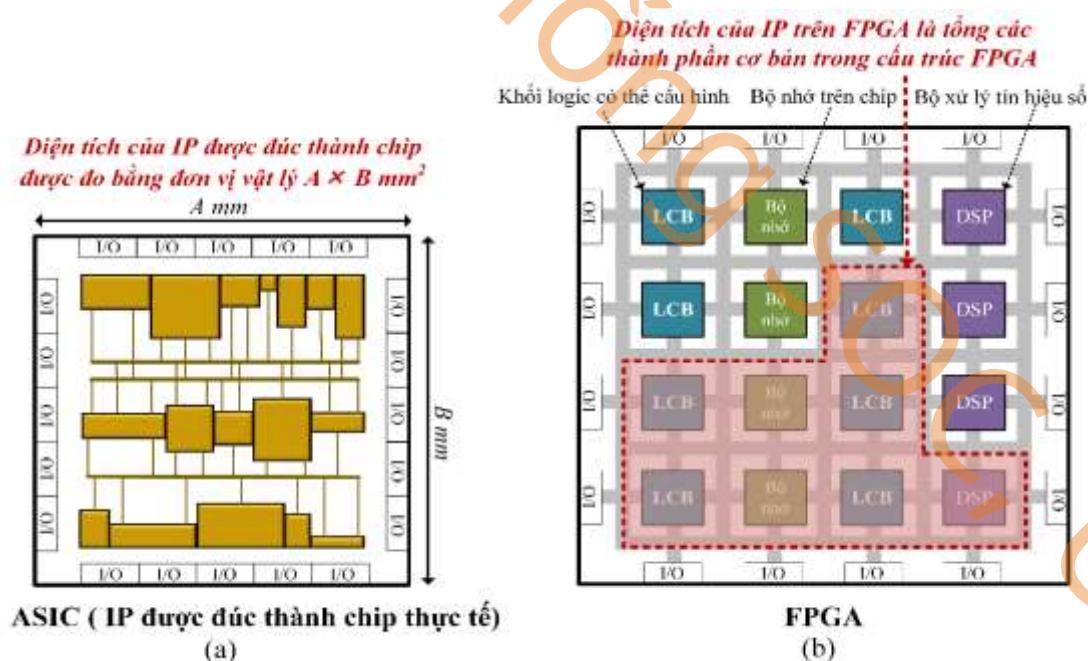
không chỉ giúp duy trì hiệu năng của hệ thống mà còn giảm chi phí bảo trì và tăng tuổi thọ sản phẩm.

**Tích hợp hệ thống** cũng đóng vai trò quan trọng, với việc kết hợp nhiều chức năng trên cùng một chip FPGA nhằm giảm kích thước, độ phức tạp của thiết kế và tăng hiệu quả vận hành. Tính linh hoạt và khả năng mở rộng là các yếu tố cần thiết, giúp hệ thống dễ dàng thích ứng với yêu cầu mới hoặc tích hợp công nghệ trong tương lai mà không cần thay đổi toàn bộ kiến trúc. Điều này đặc biệt quan trọng khi phải lựa chọn giữa IP cứng (hard IP) và IP mềm (soft IP), bởi sự lựa chọn này ảnh hưởng lớn đến tính linh hoạt, khả năng tùy chỉnh, và hiệu năng.

Chương này sẽ khám phá chi tiết những tiêu chí quan trọng trên khi đánh giá một hệ thống SoC trên FPGA. Việc hiểu rõ và áp dụng các tiêu chí này không chỉ giúp tối ưu hóa hệ thống mà còn mang lại giá trị cao, đáp ứng kỳ vọng của cả người dùng và nhà phát triển.

## 5.2. Đánh giá tài nguyên sử dụng

### 5.2.1. Đánh giá diện tích



Hình 5.1: Đánh giá diện tích của IP trên chip ASIC và FPGA: (a) Diện tích của IP trên chip ASIC được đo bằng các đơn vị vật lý; (b) Diện tích của IP trên FPGA là tổng diện tích của các thành phần cơ bản trong cấu trúc FPGA.

Đánh giá diện tích của một hệ thống SoC thực chất là đánh giá diện tích của IP tự thiết kế hoặc bao gồm cả hệ thống bus, còn các thiết bị/thành phần còn lại đều là các thành phần cố định trong hệ thống như CPU, DRAM, hay bộ nhớ Flash có diện tích cố định và không thể thay đổi được trong suốt quá trình thiết kế. Do đó, phần đánh giá diện tích này sẽ tập trung vào việc đánh giá diện tích của IP tự thiết kế trên FPGA, bao gồm cả Xilinx FPGA và Altera FPGA, với trọng tâm hướng đến Xilinx FPGA.

Hình 5.1 minh họa sự khác biệt giữa việc đánh giá diện tích của IP trên ASIC và FPGA. Trong trường hợp của ASIC, diện tích của IP được đo bằng các đơn vị vật lý như mm<sup>2</sup> hoặc cm<sup>2</sup>, phản ánh kích thước thực tế của chip. Theo đó, ASIC là các con chip thực tế ngày nay, và mục tiêu cuối cùng khi thiết kế trên FPGA là để có thể hiện thực hóa các thiết kế này và chuyển chúng thành các chip thực tế như ASIC trong sản xuất hàng loạt. Trong khi đó, đối với FPGA, diện tích không được đo bằng đơn vị vật lý mà thay vào đó là tổng diện tích của các thành phần cơ bản trong cấu trúc FPGA, chẳng hạn như LUT (bảng trả cứu), FF (Flip-Flops), và DSP (bộ xử lý tín hiệu số). Vì FPGA có thể tái cấu hình linh hoạt, diện tích được tính dựa trên số lượng các tài nguyên phần cứng được sử dụng. Các thành phần này có thể thay đổi tùy vào cách mà người thiết kế cấu hình FPGA để thực hiện các tác vụ khác nhau.

Nội dung kế tiếp sẽ đi sâu vào chi tiết về đánh giá diện tích trên Altera FPGA và Xilinx FPGA. Cần lưu ý rằng mỗi loại FPGA có những đặc điểm riêng biệt và công cụ phần mềm hỗ trợ đánh giá diện tích cũng sẽ có sự khác nhau. Altera FPGA sử dụng phần mềm Quartus Prime, trong khi Xilinx FPGA sử dụng Vivado. Cả hai phần mềm này đều cung cấp các công cụ mạnh mẽ giúp người thiết kế đánh giá diện tích của các thành phần trong SoC trên FPGA, bao gồm IP tự thiết kế.

### **Đánh giá diện tích trên Altera FPGA dùng phần mềm Quartus Prime**

Để ước tính tài nguyên phần cứng sử dụng trên Altera FPGA, phần mềm Quartus Prime của Intel được sử dụng rộng rãi. Quartus Prime cung cấp báo cáo chi tiết về việc sử dụng các tài nguyên phần cứng như các phần tử logic, các thanh ghi, chân I/O, bit bộ nhớ,

bộ nhân nhúng và PLLs. Thông qua các báo cáo này, nhà thiết kế có thể dễ dàng đánh giá mức độ sử dụng tài nguyên và xác định tính khả thi của thiết kế trên FPGA.

Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	2,727 / 114,480 ( 2 % )
Total registers	1496
Total pins	50 / 529 ( 9 % )
Total virtual pins	0
Total memory bits	1,231,492 / 3,981,312 ( 31 % )
Embedded Multiplier 9-bit elements	26 / 532 ( 5 % )
Total PLLs	0 / 4 ( 0 % )

Hình 5.2: Diện tích trên Altera FPGA trích xuất tài nguyên từ phần mềm Quartus Prime.

Hình 5.2 cung cấp thông tin chi tiết về việc sử dụng tài nguyên phần cứng cho một ứng dụng SoC được hiện thực trên FPGA Cyclone IV E EP4CE115F29C7. Thiết kế sử dụng chỉ 2% tổng số logic elements (2,727/114,480), cho thấy mức độ khai thác tài nguyên logic còn rất thấp, cho phép người thiết kế tiếp tục mở rộng hoặc tối ưu hóa thêm. Tổng số thanh ghi (registers) được sử dụng là 1,496, trong khi tổng số chân I/O (pins) sử dụng là 50, tương đương 9% tổng số chân có sẵn, cho thấy khả năng mở rộng giao tiếp với các thiết bị ngoại vi vẫn còn rất lớn. Bộ nhớ tích hợp sử dụng 31% dung lượng, tương ứng 1,231,492 bits trên tổng số 3,981,312 bits, cho thấy thiết kế đã tận dụng đáng kể tài nguyên bộ nhớ để lưu trữ dữ liệu. Ngoài ra, chỉ có 5% số nhân nhúng 9-bit được sử dụng (26/532), và không có PLL nào được khai thác, cho thấy các khối xử lý số học và điều khiển đồng hồ vẫn còn rất nhiều tiềm năng để triển khai thêm các tính năng hoặc tối ưu hóa hiệu năng. Nhìn chung, thiết kế hiện tại chiếm dụng rất ít tài nguyên, cho phép nâng cấp hoặc thêm chức năng trong tương lai mà không gặp giới hạn về phần cứng.

### Đánh giá diện tích trên Xilinx FPGA dùng phần mềm Vivado

Để ước tính tài nguyên phần cứng sử dụng trên Xilinx FPGA, phần mềm Vivado được sử dụng rộng rãi. Vivado cung cấp báo cáo chi tiết về việc sử dụng các tài nguyên phần cứng như LUTs, Registers Registers/Flip-Flop/FF, Block RAM, DSPs, F7 Muxes, LUT as Memory và các tài nguyên khác. Thông qua các báo cáo này, nhà thiết kế có thể dễ dàng

đánh giá mức độ sử dụng tài nguyên và xác định tính khả thi của thiết kế trên FPGA. Trong đó, các tài nguyên LUT, Registers/Flip-Flop (gọi tắt là FF), BRAM và DSP là những thông số quan trọng nhất và thường được đem đi so sánh với các thiết kế SoC khác để đánh giá độ lớn về diện tích của hệ thống. LUTs chủ yếu được sử dụng để thực hiện các phép toán logic và điều kiện, giúp thực thi các phép toán số học cơ bản, biểu thức điều kiện và các phép toán logic phức tạp. FF đóng vai trò quan trọng trong việc lưu trữ và giữ các giá trị tạm thời trong quá trình xử lý, thường được khai báo trong thiết kế Verilog dưới dạng reg. BRAM thường được tiêu thụ bởi các mô-đun liên quan đến bộ nhớ và các cấu trúc FIFO, nơi cần lưu trữ dữ liệu lớn hoặc tạm thời trong quá trình xử lý. DSPs được sử dụng chủ yếu cho các phép toán số học phức tạp như nhân, cộng, chia và các phép toán ma trận. Tùy thuộc vào loại tính toán và ứng dụng, các tài nguyên này sẽ được sử dụng nhiều hay ít, giúp nhà thiết kế tối ưu hóa tài nguyên và đạt được hiệu năng tốt nhất trong các ứng dụng cụ thể.

Name	CLB LUTs (274080)	CLB Registers (548160)	CARRYB (34260)	FF Mixers (137040)	CLB (34260)	LUT as Logic (274080)	LUT as Memory (144000)	Block RAM Tile (912)	DSPs (2520)	GLOBAL CLOCK BUFFERS (404)	PSB (1)
SoC_wrapper	31926	20637	300	16	5922	26990	4936	4.5	40	1	1
SoC_I (SoC)	31926	20637	300	16	5922	26990	4936	4.5	40	1	1
axi_smc [SoC_axi_t]	7059	9463	16	0	1459	5404	1655	0	0	0	0
CGRA_0 [SoC_CGRA]	24855	11111	284	16	4471	21575	3280	4.5	40	0	0
rst_ps8_0_99M [5c]	13	63	0	0	36	12	1	0	0	0	0
zynq_ultra_ps_e_0	0	0	0	0	0	0	0	0	0	1	1

Hình 5.3: Diện tích SoC trên ZCU102 FPGA trích xuất tài nguyên từ phần mềm Vivado.

Hình 5.3 minh họa các tài nguyên phần cứng được sử dụng trong một thiết kế SoC trên Xilinx ZCU102 FPGA, với ứng dụng IP tự thiết kế có tên CGRA\_0. Cụ thể, bảng cho thấy số lượng LUT, FF, BRAM và DSPs được sử dụng trong IP CGRA. Cụ thể, thiết kế CGRA\_0 sử dụng 24,855 LUTs, chiếm 9% tổng số LUTs (274,080) trong thiết kế. Số lượng FF được sử dụng là 11,111, chiếm 2% tổng số FF (548,160). BRAM được sử dụng là 4.5, chiếm 0.5% tổng số BRAM (912). Cuối cùng, thiết kế sử dụng 40 DSPs, chiếm 1.6% tổng số DSPs (2,520). Nhìn chung, thiết kế IP CGRA\_0 này vẫn còn nhỏ, chủ yếu tiêu tốn LUTs, và hoàn toàn có thể mở rộng khả năng tính toán của DSPs cũng như lưu trữ của BRAM để đáp ứng các yêu cầu phức tạp hơn trong tương lai.

Do đặc điểm cấu tạo của các khối DSP và BRAM khác nhau giữa các dòng FPGA, để so sánh một cách công bằng giữa các thiết kế, nhà thiết kế có thể quy đổi xấp xỉ tất cả tài nguyên về dạng FF và LUT. Do đặc điểm cấu tạo của các khối DSP và block RAM khác nhau giữa các dòng FPGA, để so sánh một cách công bằng giữa các thiết kế, nhà thiết kế có thể quy đổi xấp xỉ tất cả tài nguyên về dạng FF và LUT. Cách tiếp cận này giúp chuẩn hóa và tạo cơ sở đánh giá hiệu quả sử dụng tài nguyên một cách nhất quán, bất kể sự khác biệt về kiến trúc phần cứng.

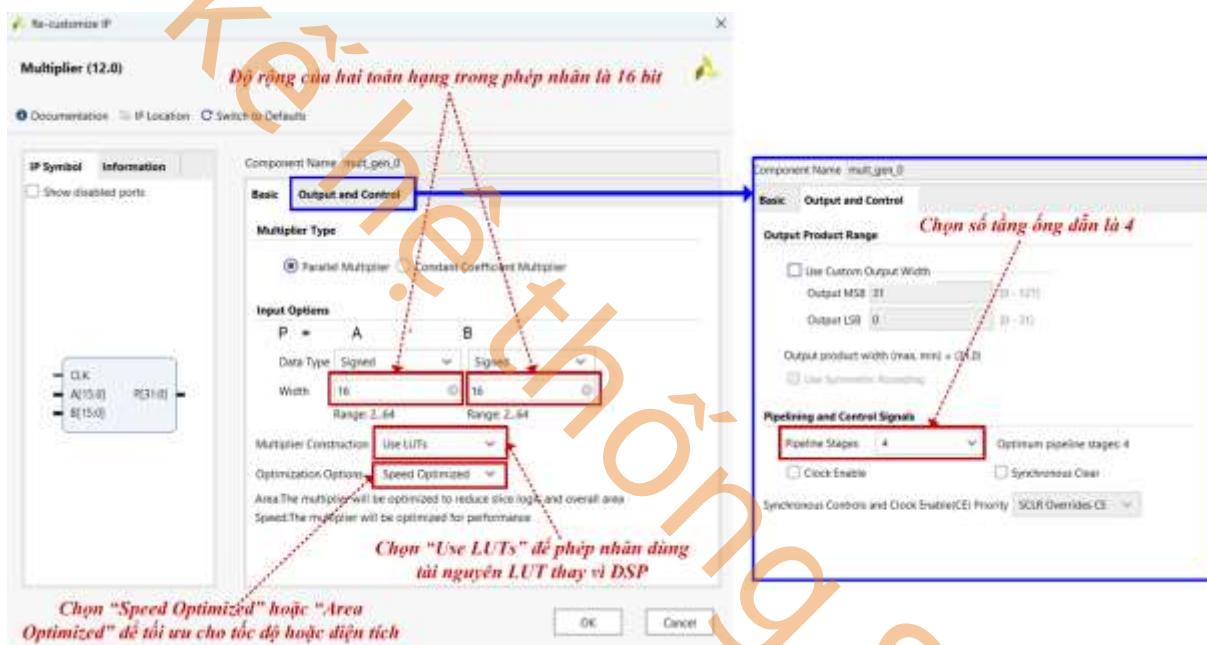
### **Quy đổi tài nguyên DSP sang LUT và FF**

Các DSP blocks trong FPGA có thể thực hiện các phép toán số học phức tạp như MAC (Multiply-Accumulate), floating-point (số phẩy động), các phép tính khác như cộng, trừ, nhân, chia, và các phép toán fixed-point (số phẩy tĩnh) như phép tính nhân và chia. Tuy nhiên, chỉ số quy đổi từ DSP sang LUT và FF có thể thay đổi tùy vào độ lớn của phép tính và loại phép tính. Cụ thể, đối với các phép toán có độ chính xác cao hoặc yêu cầu tính toán phức tạp (như floating-point), mức độ tiêu thụ tài nguyên LUT và FF sẽ cao hơn, trong khi các phép toán fixed-point đơn giản hơn sẽ yêu cầu ít tài nguyên hơn. Điều này có nghĩa là, tùy thuộc vào yêu cầu của ứng dụng và độ chính xác của phép toán, giá trị quy đổi từ DSP sang LUT và FF sẽ khác nhau.

Để biết chính xác mức độ tiêu thụ tài nguyên, trên phần mềm Vivado, khi sử dụng IP Catalog, người thiết kế có thể chọn loại tiêu thụ tài nguyên khi sử dụng DSP và chuyển đổi trong quá trình tổng hợp. Phần mềm Vivado sẽ cung cấp báo cáo chi tiết về việc sử dụng LUT và FF cho các phép toán cụ thể, giúp nhà thiết kế xác định chính xác tài nguyên cần thiết cho các ứng dụng của mình.

Hình 5.4 minh họa cách cấu hình phép nhân với độ rộng 16 bit cho hai toán hạng trong phần mềm Vivado. Trong ví dụ này, các toán hạng A và B có độ rộng là 16 bit, và người thiết kế đã chọn sử dụng LUTs thay vì DSP để thực hiện phép nhân. Việc chọn "Use LUTs" giúp sử dụng tài nguyên LUT thay vì DSP. Tương tự với các phép toán số học khác, khi chuyển từ DSP sang LUT, nhà thiết kế chỉ cần chọn "Use LUTs" trong các IP sử dụng DSP trong IP Catalog của Vivado để thay đổi cách thức sử dụng tài nguyên trong quá trình tổng

hợp. Ngoài ra, khi chọn Tùy chọn Tối ưu hóa (Optimization Option), người thiết kế có thể chọn "Tối ưu hóa tốc độ" (Speed Optimized) để tối ưu hóa tốc độ của phép nhân, điều này sẽ giúp giảm độ trễ và tăng tốc độ hoạt động, nhưng cũng có thể làm tăng số lượng LUT và FF sử dụng. Bên cạnh đó, giá trị chuyển đổi từ DSP sang LUT và FF còn bị ảnh hưởng bởi tầng ống dẫn (pipeline stages). Số tầng ống dẫn càng lớn thì đường có độ trễ lớn nhất (critical path) trong phép nhân sẽ nhỏ hơn, giúp tăng tốc độ xử lý, nhưng đổi lại số lượng LUT và FF sẽ tăng lên. Do đó, việc chọn số tầng ống dẫn và tùy chọn tối ưu hóa phù hợp sẽ là yếu tố quyết định trong việc tối ưu hóa tài nguyên và hiệu năng hệ thống.



Hình 5.4: Cấu hình phép nhân với độ rộng 16 bit cho hai toán hạng, chọn tối ưu tốc độ, và chọn tầng ống dẫn là 4, sử dụng LUT thay vì DSP trên phần mềm Vivado.

Name	1	CLB LUTs (274080)	CLB Registers (548160)	CARRY8 (34260)	Bonded IOB (328)	GLOBAL CLOCK BUFFERs (404)
> N mult_gen_0		282	301	45	65	1

Hình 5.5: Diện tích phép nhân với độ rộng 16 bit cho hai toán hạng, tối ưu hóa tốc độ, và 4 tầng đường ống trên ZCU102 FPGA.

Hình 5.5 minh họa diện tích phép nhân với độ rộng 16 bit cho hai toán hạng, tối ưu hóa tốc độ, và sử dụng 4 tầng ống dẫn trên ZCU102 FPGA. Trong ví dụ này, thay vì sử dụng tài nguyên 1 DSP48, phép nhân sử dụng số lượng LUT là 282, chiếm một phần nhỏ

so với tổng số LUT có sẵn (274,080). FF sử dụng là 301, cũng chiếm một tỷ lệ nhỏ so với tổng số FF (548,160).

Rất khó để ước lượng tài nguyên sử dụng theo kiểu tỷ lệ theo số bit của toán hạng hoặc số tầng ống dẫn, vì kết quả của phép tính sẽ khác nhau tùy thuộc vào toán hạng và số tầng ống dẫn. Do đó, thường nên chọn kết quả từ quá trình tổng hợp phần mềm Vivado để đảm bảo độ chính xác.

Name	CLB LUTs (274080)	CLB Registers (548160)	CARRY8 (34260)	Bonded IOB (328)	GLOBAL CLOCK BUFFERS (404)
> N_mult_gen_0	1466	1587	92	129	1

Hình 5.6: Diện tích phép nhân với độ rộng 32 bit cho hai toán hạng, tối ưu hóa diện tích, và 8 tầng đường ống trên ZCU102 FPGA.

Ví dụ, Hình 5.6 thể hiện diện tích cho phép nhân với độ rộng 32 bit cho hai toán hạng, tối ưu hóa diện tích, và sử dụng 8 tầng ống dẫn trên ZCU102 FPGA, không theo tỷ lệ gấp đôi với phép nhân với độ rộng 16 bit cho hai toán hạng, tối ưu hóa tốc độ, và sử dụng 4 tầng ống dẫn trên ZCU102 FPGA. Điều này cho thấy sự khác biệt trong mức độ tiêu thụ tài nguyên giữa các thiết kế, và lý do tại sao việc sử dụng kết quả từ phần mềm Vivado tổng hợp là quan trọng để có ước lượng chính xác.

Ở nội dung trên, chủ yếu là chuyển đổi DSP sang LUT và FF bằng IP sử dụng DSP trong IP Catalog của Vivado. Tuy nhiên, cách thức dùng phép nhân (\*) và chia (/) trong số phẩy tĩnh (fixed-point) trong mã Verilog vẫn có thể tồn tài nguyên DSP khi tổng hợp trên phần mềm Vivado. Khi chuyển sang sử dụng LUT và FF, thường phải dùng cách gián tiếp là mở một IP sử dụng DSP có hành vi tính toán và độ lớn bit giống như phép nhân và chia, sau đó chuyển đổi IP đó để ước lượng phép nhân và chia này sẽ tốn bao nhiêu LUT và FF.

Ví dụ trong Hình 5.6, mã Verilog mô tả phép nhân với độ rộng 16 bit cho hai toán hạng mà không sử dụng IP phép nhân có sẵn trong IP Catalog của Vivado. Trong trường hợp này, phép nhân được thực hiện trực tiếp bằng cách sử dụng các toán tử chuẩn trong Verilog. Tuy nhiên, nếu muốn thay đổi từ DSP sang LUT và FF, nhà thiết kế có thể sử dụng cách tiếp cận gián tiếp như đã đề cập để ước lượng tài nguyên bằng cách điều chỉnh IP như trong Hình 5.4, thiết lập độ rộng 16 bit cho hai toán hạng và chọn số tầng ống dẫn là 1, để việc thực hiện phép nhân trở nên tương đương với mã Verilog đã được mô tả ở

Hình 5.6. Bằng cách này, IP sẽ sử dụng LUT thay vì DSP, và số tầng ống dẫn được điều chỉnh để tối ưu hóa độ trễ và hiệu năng của phép nhân, nhưng vẫn giữ được độ chính xác và khả năng hoạt động tương đương với cách thực hiện trực tiếp trong Verilog.

```

module multiplier (
    input wire clk,           // Xung clock
    input wire rst,           // Tín hiệu reset
    input wire [15:0] a,       // Đầu vào a, 16-bit
    input wire [15:0] b,       // Đầu vào b, 16-bit
    output reg [31:0] product // Đầu ra là kết quả phép nhân, 32-bit
);

// Mô tả hành vi của module nhân
always @ (posedge clk or posedge rst) begin
    if (rst) begin
        // Nếu có tín hiệu reset, đưa kết quả về 0
        product <= 32'b0;
    end else begin
        // Nếu không có reset, thực hiện phép nhân
        product <= a * b;
    end
end

endmodule

```

Hình 5.7: Code Verilog mô tả phép nhân với độ rộng 16 bit cho hai toán hạng mà không sử dụng IP phép nhân có sẵn trong IP Catalog của Vivado.

Name	CLB LUTs (274080)	CLB Registers (548160)	CARRY8 (34260)	Bonded IOB (328)	GLOBAL CLOCK BUFFERS (404)
> N mult_gen_0	280	32	45	65	1

Hình 5.8: Diện tích phép nhân với độ rộng 16 bit cho hai toán hạng và 1 tầng ống dẫn của IP tương đương với mã Verilog mô tả trong Hình 5.6.

Hình 5.7 minh họa diện tích của phép nhân với độ rộng 16 bit cho hai toán hạng và 1 tầng ống dẫn của IP, tương đương với mã Verilog mô tả trong Hình 5.6. Kết quả cho thấy rằng việc điều chỉnh IP theo cách này cho phép sử dụng LUT thay vì DSP, đồng thời giữ được hiệu năng tối ưu và khả năng hoạt động tương đương với cách thực hiện phép nhân trong Verilog.

### Quy đổi BRAM sang LUT và FF

BRAM được sử dụng để lưu trữ dữ liệu hoặc làm bộ nhớ cache, nhưng có thể thay thế bằng LUT và FF nếu cần. BRAM thường chia làm 2 loại đơn vị: BRAM 18 Kb và BRAM 36 Kb. Cụ thể, một BRAM (18 Kb) tương đương với 0.5 BRAM và BRAM (36 Kb) tương đương với 1 BRAM trên các Xilinx FPGA.

Một BRAM (18 Kb) thường tương đương:

- **FF:** xấp xỉ 32-512.
- **LUT:** xấp xỉ 400

Một BRAM (36 Kb) thường tương đương:

- **FF:** xấp xỉ 32-512.
- **LUT:** xấp xỉ 800

Khi thực hiện quy đổi, tổng tài nguyên của một thiết kế được tính bằng cách cộng tổng các tài nguyên đã quy đổi về LUT và FF:

$$\text{Tổng tài nguyên} = FF_{thực tế} + LUT_{thực tế} + (\text{DSP quy đổi}) + (\text{BRAM quy đổi})$$

*Ví dụ 5.1: Thiết kế thứ nhất sử dụng 10 DSP (phép nhân với độ rộng 16 bit cho hai toán hạng và 1 tầng ống dẫn), 4 BRAM (36Kb), 3000 LUT và 3500 FF. Và thiết kế thứ hai sử dụng 15 DSP (phép nhân với độ rộng 16 bit cho hai toán hạng và 1 tầng ống dẫn), 12 BRAM (36Kb), 1500 LUT và 2000 FF. Thực hiện so sánh hiệu quả của hai thiết kế trên. Biết rằng 1 DSP tương đương với 280 LUT và 32 FF, và 1 BRAM (36Kb) tương đương 800 LUT và 32 FF.*

## Thiết kế 1

DSP block (10): LUT:  $10 \times 280 = 2,800$ ; FF:  $10 \times 32 = 320$

Block RAM (4): LUT:  $4 \times 800 = 3,200$ ; FF:  $2 \times 32 = 64$

FF thực tế: 5,000.

LUT thực tế: 3,000.

⇒ Tổng tài nguyên:

$$\text{LUT: } 3,000 + 2,800 + 3,200 = 9,000; \text{ FF: } 3,500 + 320 + 64 = 3,884$$

## Thiết kế 2

DSP block (15): LUT:  $15 \times 280 = 4,200$ ; FF:  $15 \times 32 = 480$

Block RAM (12): LUT:  $12 \times 800 = 9,600$ ; FF:  $6 \times 32 = 192$

FF thực tế: 2,000.

LUT thực tế: 1,500.

⇒ Tổng tài nguyên:

LUT:  $1,500 + 4,200 + 9,600 = 15,300$ ; FF:  $2,000 + 480 + 192 = 2,672$

### So sánh tổng tài nguyên giữa hai thiết kế

Tài nguyên	Thiết kế 1	Thiết kế 2
LUT	9,000	15,300
FF	3,884	2,672

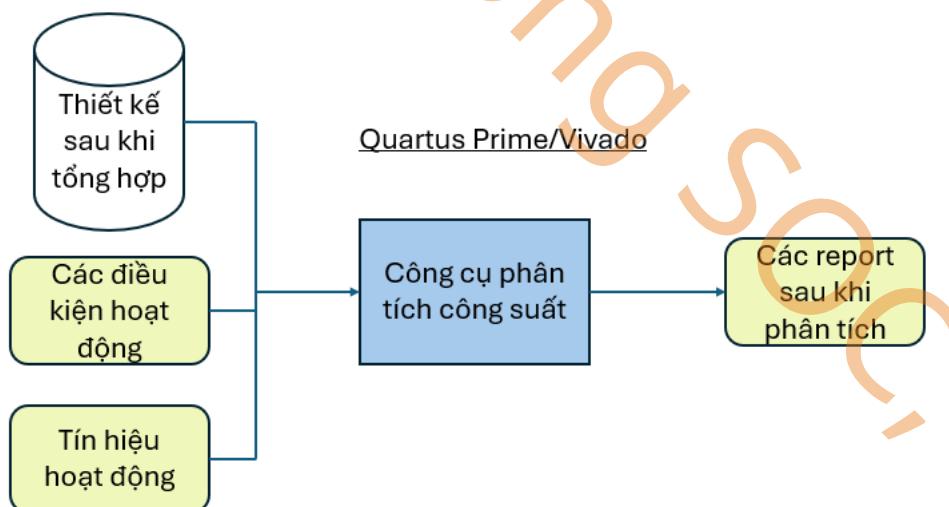
Mặc dù thiết kế 1 ban đầu có giá trị LUT và FF (3,000 LUT và 3,500 FF) lớn hơn thiết kế 2 (1,500 LUT và 2,000 FF), nhưng sau khi tính toán tổng LUT và FF, thiết kế 1 lại nhỏ hơn thiết kế 2, với tổng giá trị LUT là 9,000 và FF là 5,384, trong khi thiết kế 2 có tổng giá trị LUT là 15,300 và FF là 2,672. Thông thường, LUT là đơn vị chính để so sánh giữa các thiết kế, vì nó ảnh hưởng trực tiếp đến khả năng thực thi các phép toán và xử lý dữ liệu. Chỉ số FF ít được so sánh vì nó liên quan đến việc lưu trữ dữ liệu tạm thời trong các flip-flop, và không ảnh hưởng lớn đến hiệu năng chung của thiết kế. Kết luận, Thiết kế 1 có diện tích nhỏ hơn và hiệu quả hơn Thiết kế 2, làm cho nó phù hợp hơn cho các ứng dụng yêu cầu tiết kiệm tài nguyên và tối ưu hóa hiệu năng.

Khi có sự chênh lệch đáng kể giữa LUT và FF, việc chuyển đổi LUT và FF sang slice có thể giúp chúng ta đánh giá rõ hơn về mức độ sử dụng tài nguyên của thiết kế. Mỗi slice trong FPGA thường chứa một số lượng LUT và FF nhất định, ví dụ như một slice có thể bao gồm 4 LUTs và 8 FFs. Do đó, để chuyển đổi LUT và FF sang số slice, ta có thể sử dụng cách sau. Một slice có thể chứa 4 LUTs và 8 FFs, vì vậy ta sẽ tính số slice từ mỗi loại tài nguyên và chọn số slice lớn hơn. Ví dụ, đối với thiết kế 1, số LUT là 9,000, do đó số slice từ LUT là 2,250 slice ( $9,000 / 4$ ). Từ FF, số FF là 5,384, do đó số slice từ FF là 673 slice ( $5,384 / 8$ ). Vì vậy, số slice là 2,250. Tương tự, đối với thiết kế 2, số LUT là 15,300, do đó số slice từ LUT là 3,825 slice ( $15,300 / 4$ ). Từ FF, số FF là 2,672, do đó số slice từ FF là 334 slice ( $2,672 / 8$ ). Vì vậy, số slice là 3,825. Khi so sánh số slice giữa hai thiết kế,

ta thấy rằng thiết kế 1 yêu cầu 2,250 slice, trong khi thiết kế 2 yêu cầu 3,825 slice. Vì số slice trong thiết kế 1 ít hơn, điều này cho thấy thiết kế 1 sử dụng tài nguyên hiệu quả hơn và có thể coi là thiết kế nhỏ hơn và tốt hơn khi xét về mức độ tối ưu hóa tài nguyên.

### 5.2.2. Đánh giá năng lượng

Việc đánh giá năng lượng tiêu thụ của hệ thống FPGA là một yếu tố quan trọng trong quá trình tối ưu hóa thiết kế, đặc biệt khi so sánh với các yêu cầu tài nguyên phần cứng. Để ước tính mức năng lượng tiêu thụ, các phần mềm như Quartus Prime và Vivado được sử dụng phổ biến, cung cấp các báo cáo chi tiết về việc sử dụng tài nguyên phần cứng. Hình 5.8 minh họa quy trình phân tích công suất của một thiết kế SoC trên FPGA, sử dụng các công cụ như Quartus Prime hoặc Vivado. Quy trình bắt đầu từ thiết kế đã được tổng hợp (synthesis), nơi các điều kiện hoạt động cụ thể như điện áp, tần số, nhiệt độ và các tín hiệu hoạt động của hệ thống được nhập vào. Những thông tin này sau đó được chuyển đến công cụ phân tích công suất, nơi mức tiêu thụ năng lượng của từng thành phần trong thiết kế, bao gồm logic, DSP, RAM và I/O, sẽ được ước tính. Kết quả phân tích được xuất dưới dạng các báo cáo chi tiết, giúp nhà thiết kế đánh giá chính xác mức năng lượng tiêu thụ của hệ thống và tối ưu hóa thiết kế để đạt hiệu năng tốt nhất trong khi vẫn tiết kiệm năng lượng.



Hình 5.9: Quy trình phân tích công suất cho hệ thống SoC trên FPGA. [7]  
Công suất tiêu thụ của FPGA thường được chia thành hai loại chính:

- Công suất tĩnh (Static Power): Là công suất tiêu thụ ngay cả khi FPGA không thực hiện bất kỳ hoạt động nào, chủ yếu do dòng rò và điện áp cung cấp. Công suất tĩnh phụ thuộc vào công nghệ chế tạo chip, nhiệt độ hoạt động, và điện áp cung cấp.
- Công suất động (Dynamic Power): Là công suất tiêu thụ khi các khối logic và mạch giao tiếp hoạt động.

### **Đánh giá năng lượng trên Altera FPGA dùng phần mềm Quartus Prime**

Để ước tính năng lượng tiêu thụ trên FPGA Altera, phần mềm Quartus Prime của Intel được sử dụng rộng rãi. Quartus Prime cung cấp các báo cáo chi tiết về việc sử dụng tài nguyên phần cứng và tính toán mức tiêu thụ năng lượng của hệ thống, bao gồm các thành phần logic, bộ nhớ, DSP, và I/O.

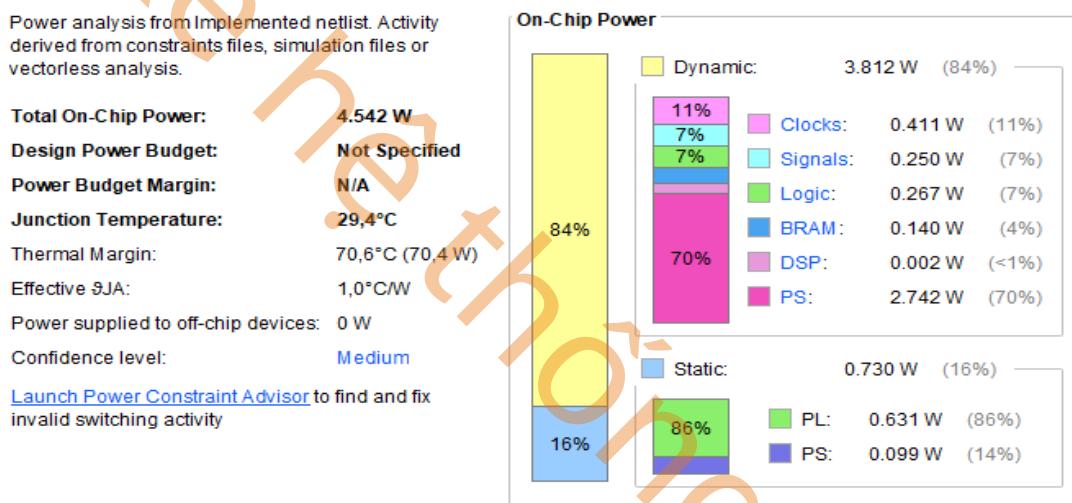
PowerPlay Power Analyzer Summary	
PowerPlay Power Analyzer Status	Successful - Wed Jan 22 20:18:52 2025
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	fifidemo
Top-level Entity Name	fifidemo
Family	Cyclone II
Device	EP2C5T144C6
Power Models	Final
Total Thermal Power Dissipation	31.48 mW
Core Dynamic Thermal Power Dissipation	0.00 mW
Core Static Thermal Power Dissipation	18.01 mW
I/O Thermal Power Dissipation	13.47 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

*Hình 5.10: Năng lượng trên Altera FPGA trích xuất tài nguyên từ phần mềm Quartus Prime.*

Hình 5.10 minh họa báo cáo về năng lượng tiêu thụ của một thiết kế SoC trên FPGA, được trích xuất từ phần mềm Quartus Prime. Báo cáo này cung cấp các thông tin về công suất tiêu thụ, bao gồm tổng công suất nhiệt (31.48 mW), công suất tĩnh (18.01 mW), và công suất động (0.00 mW), cùng với mức độ tự tin trong ước tính công suất. Thông qua các báo cáo này, nhà thiết kế có thể đánh giá chính xác mức năng lượng tiêu thụ của hệ thống và tối ưu hóa thiết kế để đạt hiệu năng tốt nhất trong khi vẫn tiết kiệm năng lượng, giúp đảm bảo sự phù hợp với các yêu cầu về hiệu năng và tiêu thụ năng lượng trong các ứng dụng SoC.

### **Đánh giá năng lượng trên Xilinx FPGA dùng phần mềm Vivado**

Để ước tính mức năng lượng tiêu thụ trên Xilinx FPGA, hiện tại phần mềm Vivado là phần mềm chính thức được sử dụng. Vivado không chỉ cung cấp các báo cáo chi tiết về việc sử dụng các tài nguyên phần cứng như LUTs, Registers/Flip-Flop (FF), Block RAM (BRAM), DSPs, F7 Muxes, và các tài nguyên khác mà còn có khả năng tính toán mức tiêu thụ năng lượng của hệ thống. Thông qua các báo cáo này, nhà thiết kế có thể đánh giá mức độ tiêu thụ năng lượng của các thành phần như logic, bộ nhớ, và các khối xử lý số học phức tạp. Điều này giúp tối ưu hóa việc sử dụng tài nguyên và đạt được hiệu năng tốt nhất trong khi vẫn đảm bảo tiết kiệm năng lượng, từ đó giúp nhà thiết kế đưa ra các quyết định chính xác về phân bổ tài nguyên của SoC.



Hình 5.11: Ví dụ trích xuất thông tin công suất tiêu thụ từ công cụ phân tích công xuất được tích hợp trong Vivado Xilinx [6]

Báo cáo công suất của thiết kế SoC trên FPGA, như được mô tả trong hình 5.5 và phân tích bằng công cụ Vivado, cho thấy tổng mức tiêu thụ điện năng là 4.542 W. Công suất này được chia thành hai phần chính: công suất động và công suất tĩnh. Công suất động chiếm 3.812 W (84% tổng mức tiêu thụ), bao gồm các thành phần: xung đồng hồ 0.411 W (11%), tín hiệu 0.250 W (7%), các khối logic 0.267 W (7%), các khối BRAM 0.140 W (4%), các khối DSP 0.002 W (dưới 1%), và CPU (PS) 2.742 W (70%). Công suất tĩnh chiếm 0.730 W (16%), với khối logic lập trình (PL) tiêu thụ 0.631 W (86%) và phần mềm PS tiêu thụ 0.099 W (14%). Nhiệt độ nội của thiết kế ở mức 29.4°C, với biên độ nhiệt độ 70.6°C, đảm bảo thiết kế hoạt động trong phạm vi nhiệt độ an toàn. Báo cáo cũng nêu bật

các điều kiện vận hành như nhiệt độ nút tại 29.4°C và ngân sách công suất dành cho các thành phần thiết kế, cung cấp cái nhìn tổng quan về các yếu tố ảnh hưởng đến hiệu năng và hiệu quả năng lượng của thiết kế.

Một số FPGA của Xilinx, như Zynq ZCU102, ZCU104, và RFSoC ZCU111, được trang bị cảm biến INA226, cho phép đo lường chính xác mức tiêu thụ năng lượng thực tế của hệ thống. Cảm biến INA226 có khả năng đo điện áp và dòng điện, từ đó tính toán công suất tiêu thụ của các thành phần như logic, bộ nhớ và các khối xử lý số học phức tạp. Thông qua giao tiếp I2C, dữ liệu đo lường có thể được thu thập và phân tích, giúp nhà thiết kế đánh giá chính xác mức tiêu thụ năng lượng và tối ưu hóa thiết kế để đạt hiệu năng tốt nhất trong khi vẫn tiết kiệm năng lượng. Để thu thập dữ liệu từ cảm biến INA226, có thể sử dụng một ứng dụng Linux đơn giản tương tác với các điện trở shunt trên bo mạch Zynq ZCU102, ZCU104, và RFSoC ZCU111. Ứng dụng này tận dụng giao diện `/sys/class/hwmon`, cho phép truy cập thông tin về điện áp và dòng điện của từng đường nguồn. Ứng dụng đọc các thuộc tính `curr1_input` và `in1_input` để trích xuất thông tin về dòng điện và điện áp cho mỗi đường nguồn. Dữ liệu thu thập được có thể được sử dụng để tính toán công suất tiêu thụ của các thành phần trong hệ thống, hỗ trợ nhà thiết kế trong việc đánh giá và tối ưu hóa mức tiêu thụ năng lượng của thiết kế. Để biết thêm chi tiết và truy cập mã nguồn C cho ứng dụng Linux, bạn có thể tham khảo bài viết "*Accurate Design Power Measurement Made Easier*" trên trang web hướng dẫn của Xilinx tại đường dẫn "<https://www.xilinx.com/developer/articles/accurate-design-power-measurement.html>". Bài viết này cung cấp hướng dẫn chi tiết về cách sử dụng cảm biến INA226 trên ZCU102 để đo lường và phân tích mức tiêu thụ năng lượng của hệ thống, cùng với mã nguồn C minh họa cách thu thập và xử lý dữ liệu từ cảm biến.

### Phương pháp giám sát năng lượng tiêu thụ

Thông qua quá trình phân tích và trích xuất công suất tiêu thụ, các nhà thiết kế có thể xác định các hạn chế và điểm yếu còn tồn tại trong hệ thống, từ đó thực hiện các cải tiến để tối ưu hóa thiết kế. Dưới đây là một số giải pháp giúp giảm công suất tiêu thụ cho hệ thống SoC trên FPGA thông qua việc tối ưu thiết kế:

- ✓ Tối ưu hóa thiết kế logic: Loại bỏ các logic không sử dụng (Unused Logic Removal).  
Ưu tiên thiết kế đồng bộ hóa hệ thống thay vì bất đồng bộ hóa.
- ✓ Tối ưu hóa đường dữ liệu (Data Path Optimization): Sử dụng khối DSP thay cho LUT/FF để thực hiện các phép toán phức tạp nhằm giảm số lần chuyển mạch.
- ✓ Tối ưu hóa bộ nhớ: Tận dụng bộ nhớ cục bộ như BRAM hoặc URAM trên FPGA để giảm truy cập bộ nhớ ngoài. Sử dụng các kỹ thuật quản lý bộ nhớ hiệu quả để giảm độ trễ và công suất.
- ✓ Tối ưu hóa giao tiếp I/O: Giảm hoạt động I/O không cần thiết hoặc sử dụng chế độ I/O tiết kiệm năng lượng.
- ✓ Thiết kế kiến trúc song song hiệu quả: Sử dụng các khối xử lý song song tối ưu để tăng throughput mà không tăng tần số xung nhịp. Chia nhỏ dữ liệu và sử dụng các đơn vị xử lý cục bộ để giảm truy cập tài nguyên chung.
- ✓ Tối ưu hóa hoạt động của FPGA: Sử dụng IP tối ưu năng lượng: Chọn các IP cứng thay cho IP mềm khi có thể, vì IP cứng thường tối ưu hóa năng lượng tốt hơn.

Trong thiết kế SoC, kỹ thuật đồng bộ hóa đóng vai trò quan trọng trong việc quản lý công suất tiêu thụ một cách hiệu quả. Đồng bộ hóa thấp năng (Low-Power Synchronization) là một phương pháp được phát triển để giảm tiêu thụ năng lượng khi dữ liệu được truyền giữa các khối chức năng trên chip. Phương pháp này đặc biệt cần thiết trong các thiết kế SoC, nơi việc tối ưu hóa năng lượng không chỉ kéo dài thời lượng pin mà còn giảm nhiệt độ hoạt động của chip, từ đó nâng cao độ tin cậy và tuổi thọ của thiết bị. Một kỹ thuật phổ biến trong đồng bộ hóa thấp năng là sử dụng các cơ chế đồng bộ hóa động, trong đó tín hiệu đồng bộ hóa chỉ được kích hoạt khi có dữ liệu thực sự cần truyền, giúp tránh lãng phí năng lượng do đồng bộ hóa không cần thiết. Ngoài ra, các kỹ thuật như tắt clock ở các khối không hoạt động (clock gating) và tắt nguồn của các phần không sử dụng (power gating) cũng được áp dụng để giảm tiêu thụ năng lượng đáng kể trong quá trình đồng bộ hóa. Những phương pháp này không chỉ giúp giảm thiểu năng lượng tiêu thụ mà còn góp phần nâng cao hiệu năng và hiệu quả tổng thể của hệ thống SoC.

### 5.3. Đánh giá hiệu năng của hệ thống

Phần này chúng ta đi sâu vào đánh giá hiệu năng của hệ thống, trong đó hai đại lượng quan trọng là độ trễ và thông lượng sẽ được mô tả chi tiết. Đây là hai yếu tố chủ chốt trong việc xác định hiệu quả hoạt động của hệ thống, đặc biệt trong các hệ thống xử lý dữ liệu hoặc hệ thống truyền thông. Độ trễ thể hiện thời gian cần thiết để hệ thống xử lý hoặc truyền tải dữ liệu, trong khi thông lượng phản ánh khả năng xử lý dữ liệu của hệ thống trong một khoảng thời gian nhất định. Phần này sẽ phân tích cách thức hai đại lượng này tác động đến hiệu năng của hệ thống, từ đó đề xuất các phương pháp tối ưu hóa hiệu quả.



Hình 5.12: Hình ảnh trực quan giải thích khái niệm về độ trễ và thông lượng.

Hình 5.12 có thể được hiểu như một mô phỏng ống nước để giải thích mối quan hệ giữa độ trễ và thông lượng. Giả sử ống nước tượng trưng cho hệ thống truyền tải dữ liệu, với độ dài của ống nước đại diện cho độ trễ của hệ thống - tức là thời gian cần thiết để dữ liệu đi qua từ đầu đến cuối hệ thống. Còn lượng nước đi qua ống trong một khoảng thời gian sẽ tượng trưng cho thông lượng, phản ánh khả năng hệ thống xử lý và truyền tải dữ liệu trong một đơn vị thời gian. Với ví dụ này, người đọc có thể dễ dàng hình dung tác động của độ trễ và thông lượng đối với hiệu năng tổng thể của hệ thống, khi độ trễ dài hơn có thể làm giảm thông lượng của hệ thống.

Để tính được độ trễ của hệ thống, một yếu tố quan trọng cần được xác định là chu kỳ dài nhất hoặc tần số hoạt động tối đa của hệ thống. Chu kỳ dài nhất phản ánh thời gian cần thiết để hoàn thành một chu kỳ xử lý trong hệ thống, trong khi tần số hoạt động tối đa xác định tốc độ mà các tác vụ trong hệ thống có thể được thực hiện. Việc xác định được chu kỳ dài nhất hoặc tần số hoạt động tối đa là điều kiện tiên quyết để tính toán chính xác độ trễ và thông lượng, bởi vì độ trễ và thông lượng trực tiếp phụ thuộc vào thời gian cần thiết để xử lý và truyền tải dữ liệu qua các thành phần trong hệ thống. Trước khi đi vào các phương pháp đánh giá độ trễ và thông lượng, chúng ta cần hiểu rõ tần số hoạt động của hệ

thống để đưa ra những ước lượng chính xác về hiệu năng của hệ thống trong quá trình xử lý dữ liệu.

### 5.3.1. *Ước tính chu kỳ dài nhất và tần số hoạt động tối đa*

Để tính toán tần số tối đa ( $F_{MAX}$ ) của một IP hoặc hệ thống SoC, các công cụ như Vivado và Quartus cung cấp các phương pháp hiệu quả dựa trên phân tích thời gian. Trên Vivado, quá trình bắt đầu bằng việc thực hiện tổng hợp thiết kế (Synthesis), nơi công cụ phân tích và tối ưu hóa các đường dữ liệu, sau đó cung cấp thông tin về thời gian trễ trong báo cáo phân tích thời gian (Timing Analysis). Trong báo cáo các thông số quan trọng như Slack âm nhất (WNS - Worst Negative Slack) và tổng số Slack âm (TNS - Total Negative Slack) cần được kiểm tra. Nếu cả hai giá trị đều bằng 0, thiết kế đáp ứng yêu cầu thời gian. Cụ thể chi tiết để tính toán  $F_{MAX}$  ta cần làm rõ các thông tin sau:

**WNS:** là giá trị Slack nhỏ nhất (âm nhất) trong tất cả các đường tín hiệu trong thiết kế. Slack là chênh lệch giữa thời gian yêu cầu (Required Time) và thời gian thực hiện thực tế (Arrival Time) của một tín hiệu. Nếu  $WNS \geq 0$ : Tất cả các đường tín hiệu đều đáp ứng yêu cầu thời gian  $\Rightarrow$  thiết kế hợp lệ. Nếu  $WNS < 0$ : Có ít nhất một đường tín hiệu không đáp ứng yêu cầu thời gian, gây vi phạm thời gian (timing violation).

- Công thức:

$$\text{Slack} = \text{Thời gian yêu cầu} - \text{Thời gian thực tế} \quad (5.1)$$

Với:

- Thời gian yêu cầu : Thời gian yêu cầu mà tín hiệu phải đến đích để đáp ứng chu kỳ xung nhịp.
- Thời gian thực tế: Thời gian thực tế mà tín hiệu đến đích.

**TNS:** là tổng tất cả các giá trị Slack âm trong thiết kế. Nó đại diện cho tổng mức độ vi phạm thời gian trên toàn bộ các đường tín hiệu. **TNS = 0:** Thiết kế không có vi phạm thời gian, tất cả các đường tín hiệu đều đáp ứng yêu cầu. **TNS > 0:** Không thể xảy ra vì chỉ tính Slack âm. **TNS < 0:** Giá trị âm biểu thị tổng mức độ vi phạm thời gian trong thiết kế. TNS càng nhỏ, thiết kế càng có nhiều lỗi thời gian nghiêm trọng và cần tối ưu hóa.

- Công thức:

$$TNS = \sum \text{slack} \text{ với các đường tín hiệu có Slack} < 0 \quad (5.2)$$

**Đường dài nhất:** là đường tín hiệu dài nhất (critical path) từ đầu vào đến đầu ra, với thời gian trễ lớn nhất. Critical Path Delay: thời gian trễ tổng cộng của tín hiệu khi di chuyển theo đường critical path, từ đầu vào đến đầu ra. Công thức tính độ trễ đường dài nhất (Critical Path Delay)  $T_{\text{dài nhất}} (T_{\text{critical}})$  khi ta biết tần số cài đặt giả định ban đầu  $F_{\text{Khởi tạo}}$  của một thiết kế:

$$T_{\text{dài nhất}} = \text{Chu kỳ xung đồng hồ Khởi tạo} - WNS \quad (5.3)$$

$$\text{Ta có : } F_{\text{MAX}} = \frac{1}{T_{\text{dài nhất}}} \quad (5.4)$$

Ví dụ: Một hệ thống SoC được khởi tạo giá trị tần số hoạt động ban đầu  $100MHz$ . Sau khi tổng hợp, từ thông tin thời gian ta trích xuất được thông số WNS bằng  $0.25 ns$ . Tìm tần số hoạt động tối đa của hệ thống.

$$T_{\text{dài nhất}} = \frac{1}{F_{\text{Khởi tạo}}} - WNS = 10 ns - 0.25 ns = 9.75 ns$$

Suy ra

$$F_{\text{MAX}} = \frac{1}{T_{\text{dài nhất}}} = \frac{1}{9.75 ns} = 102.56 MHz$$

### 5.3.2. Đánh giá độ trễ

Trong hệ thống SoC trên FPGA, chúng ta thường dùng chỉ số độ đẽ (latency) để đánh giá hệ thống. Độ trễ thường được định nghĩa là khoảng thời gian từ khi một tác vụ hoặc dữ liệu được bắt đầu xử lý ở đầu vào đến khi kết quả hoàn tất ở đầu ra. Công thức tính độ trễ cho một hệ thống SoC trên FPGA phụ thuộc vào cấu trúc hệ thống, các thành phần được tích hợp và cách dữ liệu di chuyển qua hệ thống. Một cách tổng quát, độ trễ của hệ thống có thể được tính bằng cách cộng độ trễ của từng thành phần trong đường dữ liệu chính.

Công thức chung:

$$\text{Độ trễ (tổng)} = \sum_{i=1}^N \text{độ trễ}_{\text{thành phần thứ } i} + \text{độ trễ}_{\text{giao tiếp}} \quad (5.5)$$

Trong đó:

- N: Số lượng thành phần trong đường dữ liệu (datapath)
- Độ trễ<sub>thành phần thứ i</sub>: Độ trễ của từng thành phần, bao gồm CPU, bộ nhớ, bus kết nối, IP

- Độ trễ giao tiếp: Độ trễ phát sinh từ việc truyền dữ liệu giữa các thành phần qua bus, giao tiếp nội, hoặc các cơ chế giao tiếp khác.

Cách tính độ trễ từng thành phần trong hệ thống SoC trên FPGA như sau:

### Đối với IP CPU

$$\text{Độ trễ}_{CPU} = \frac{\text{Số chu kỳ xử lý}}{\text{Tần số hoạt động của CPU}} \quad (5.6)$$

Trong đó số chu kỳ xử lý phụ thuộc vào đặc trưng cấu tạo của CPU, tập lệnh CPU, và cấu trúc toán học của CPU.

### Đối với IP bộ nhớ

$$\begin{aligned} \text{Độ trễ}_{bộ nhớ} &= \text{Thời gian truy cập cơ bản (Access Time)} + \\ &\quad \text{Độ trễ giao tiếp (Transfer Latency)} \end{aligned} \quad (5.7)$$

Trong đó:

- Thời gian truy cập cơ bản: Là thời gian từ lúc bắt đầu yêu cầu đọc/ghi (read/write) đến khi dữ liệu sẵn sàng để truyền.
- Độ trễ giao tiếp: Thời gian cần thiết để truyền dữ liệu từ bộ nhớ đến thiết bị yêu cầu qua bus hoặc giao tiếp.

### ✓ Công thức tính độ trễ đối với bộ nhớ chính (SRAM, DRAM):

$$\text{Độ trễ} = \frac{\text{Số chu kỳ xung nhịp cần thiết (Clock cycles)}}{\text{Tần số hoạt động của bộ nhớ (Clock frequency)}} \quad (5.8)$$

Ví dụ một DRAM cần 5 chu kỳ để hoàn thành truy cập dữ liệu, tần số hoạt động của DRAM là 200 MHz thì độ trễ sẽ bằng:

$$\text{Độ trễ} = \frac{5}{100 \times 10^6} = 25\text{ns}$$

### ✓ Công thức tính độ trễ đối với bộ nhớ Flash hoặc bộ nhớ ngoài (thẻ SD)

$$\text{Độ trễ} = \text{thời gian truy cập cơ bản} + \frac{\text{kích thước dữ liệu (data size)}}{\text{Băng thông truyền tải (bandwidth)}} \quad (5.9)$$

Ví dụ một flash memory có thời gian truy cập cơ bản là 50 ns, kích thước dữ liệu là 16 byte, băng thông đường truyền là 800 MB/s, khi đó độ trễ sẽ bằng:

$$\text{Độ trễ} = 50\text{ ns} + \frac{16\text{ bytes}}{800 \times 10^6 \text{ bytes/s}} = 50\text{ ns} + 20\text{ ns} = 70\text{ ns}$$

### ✓ Công thức tính độ trễ đối với bộ nhớ đệm (cache)

Trong hệ thống sử dụng bộ nhớ đệm, độ trễ phụ thuộc vào việc truy cập có trùng (cache hit) hay trượt (cache miss). Trong đó độ trễ cache hit thường rất thấp vì cache hit thường nằm gần CPU và được cho bởi công thức:

$$\text{Độ trễ} = \frac{\text{Chu kỳ bộ đệm}}{\text{Tần số CPU}} \quad (5.10)$$

Độ trễ cache miss bao gồm thời gian truy cập cache + thời gian truy cập bộ nhớ chính (SRAM, DRAM).

Nếu hệ thống SoC có cả bộ nhớ trong và bộ nhớ ngoài thì tổng độ trễ sẽ là:

$$\text{Tổng độ trễ bộ nhớ} = \sum_{i=1}^N \text{Độ trễ từng loại bộ nhớ} \quad (5.11)$$

Trong đó N là số loại bộ nhớ được sử dụng trong hệ thống SoC gồm cache, SRAM, DRAM hoặc bộ nhớ ngoài.

Độ trễ của bộ nhớ phản ánh thời gian cần thiết để truy cập dữ liệu từ bộ nhớ. Trong cấu trúc tổ chức bộ nhớ của một hệ thống SoC, các loại bộ nhớ trong (on-chip memory) thường có độ trễ thấp hơn so với các loại bộ nhớ ngoài (off-chip memory). Điều này là do bộ nhớ trong được tích hợp trực tiếp trên chip, cho phép dữ liệu được truy cập nhanh hơn nhờ loại bỏ độ trễ do giao tiếp bên ngoài. Ngược lại, bộ nhớ ngoài thường đòi hỏi quá trình truyền dữ liệu qua các bus hoặc giao thức kết nối, dẫn đến độ trễ cao hơn.

### **Đối với IP bus hoặc kết nối nội (interconnect)**

$$\text{Độ trễ bus} = \frac{\text{Số chu kỳ giao tiếp}}{\text{Tần số bus}} \quad (5.12)$$

### **Đối với IP hoặc khối logic tự thiết kế:**

Trường hợp không có pipeline:

$$\text{Độ trễ IP không pipeline} = \frac{\text{Số chu kỳ cần thiết cho IP hoạt động}}{\text{Tần số hoạt động của IP}} \quad (5.13)$$

Trường hợp có pipeline:

$$\text{Độ trễ IP có pipeline} = \frac{\text{Độ trễ IP không pipeline}}{\text{Số tần pipeline}} \quad (5.14)$$

Trường hợp IP hoạt động lặp lại:

$$\text{Độ trễ IP} = \frac{\text{Số chu kỳ mỗi tác vụ} \times \text{số lần lặp}}{\text{Tần số hoạt động của IP}} \quad (5.15)$$

Ví dụ: Giả sử một hệ thống SoC gồm CPU ARM Cortex có tần số hoạt động: 1 GHz, thời gian xử lý trung bình một tác vụ: 50 chu kỳ xung nhịp CPU. Một bộ nhớ trong (on chip memory) có độ trễ truy cập là 2 chu kỳ CPU. Bus AXI có độ trễ truyền nhận (mỗi chiều): 15 chu kỳ CPU. Một IP hoạt động ở tần số 200 MHz, biết tổng số chu kỳ tính toán của IP là 100 chu kỳ. Tính độ trễ của toàn bộ hệ thống SoC trong trường hợp IP không có pipeline và có pipeline. Giả sử khi có pipeline IP sẽ được thiết kế với 5 tầng pipeline.

Đầu tiên ta tính độ trễ của từng thành phần có trong hệ thống SoC như sau:

Độ trễ xử lý của CPU:

$$\text{Độ trễ (CPU)} = \frac{50 \text{ chu kỳ}}{\text{Tần số CPU}} = \frac{50}{1 \times 10^9} = 50 \text{ ns}$$

Độ trễ truy cập bộ nhớ:

$$\text{Độ trễ (bộ nhớ)} = \frac{2 \text{ chu kỳ}}{\text{Tần số CPU}} = \frac{2}{1 \times 10^9} = 2 \text{ ns}$$

Độ trễ đường truyền bus:

$$\text{Độ trễ (bus)} = 2 \times \frac{15 \text{ chu kỳ}}{\text{Tần số CPU}} = 2 \times \frac{15}{1 \times 10^9} = 30 \text{ ns}$$

Độ trễ của IP:

Trường hợp 1: không pipeline

$$\text{Độ trễ (IP)} = \frac{100 \text{ chu kỳ}}{\text{Tần số IP}} = \frac{100}{200 \times 10^6} = 500 \text{ ns}$$

Trường hợp 2: có pipeline

$$\text{Độ trễ (IP)} = \frac{500}{5} = 100 \text{ ns}$$

Độ trễ toàn hệ thống SoC được tính như sau:

Trường hợp 1: không pipeline

Độ trễ (tổng)

$$\begin{aligned} &= \text{độ trễ (CPU)} + \text{độ trễ (bộ nhớ)} + \text{độ trễ (bus)} \\ &+ \text{độ trễ (IP không pipeline)} + \text{độ trễ (bus)} \\ &= 50 + 2 + 30 + 500 + 30 = 612 \text{ ns} \end{aligned}$$

Trường hợp 2: có pipeline

$$\begin{aligned}
 \text{Độ trễ (tổng)} &= \text{độ trễ (CPU)} + \text{độ trễ (bộ nhớ)} + \text{độ trễ (bus)} + \\
 &\quad \text{độ trễ (IP có pipeline)} + \text{độ trễ (bus)} \\
 &= 50 + 2 + 30 + 100 + 30 = 212 \text{ ns}
 \end{aligned}$$

**Nhận xét:** Trong trường hợp không có pipeline, độ trễ của IP là rất lớn do toàn bộ 100 chu kỳ được thực hiện tuần tự. Trong trường hợp có pipeline, độ trễ của IP giảm đáng kể, giúp tổng độ trễ hệ thống giảm xuống chỉ còn 212 ns. Tuy nhiên, thiết kế pipeline đòi hỏi chi phí tài nguyên phần cứng cao hơn và phức tạp hơn trong việc đồng bộ dữ liệu.

### 5.3.3. Đánh giá thông lượng

Thông lượng (throughput) trong hệ thống SoC FPGA là lượng dữ liệu mà hệ thống có thể xử lý trong một đơn vị thời gian. Thông lượng là một thước đo hiệu năng quan trọng, đặc biệt trong các ứng dụng xử lý tín hiệu số, truyền dữ liệu, và các thuật toán tính toán song song. Nó thường được biểu diễn bằng các đơn vị như bit/giây (bps), mẫu/giây hoặc phép tính/giây, tùy thuộc vào ngữ cảnh.

Công thức chung:  $\text{Thông lượng} = \frac{\text{Số lượng dữ liệu xử lý}}{\text{Thời gian thực hiện}}$  (5.16)

Hoặc

$$\text{Thông lượng (bps)} = \text{tần số hoạt động} \times \text{số bit dữ liệu xử lý trong một chu kỳ}$$

Yếu tố ảnh hưởng đến thông lượng

- Tần số hoạt động:** Tăng tần số hoạt động cho phép hệ thống thực hiện nhiều chu kỳ xử lý hơn trong cùng một khoảng thời gian, từ đó cải thiện thông lượng.
- Độ rộng dữ liệu (Data width):** Việc tăng độ rộng dữ liệu, tức là số bit được xử lý mỗi chu kỳ, giúp hệ thống xử lý lượng dữ liệu lớn hơn trong một khoảng thời gian, nâng cao hiệu năng thông lượng.
- Độ trễ:** Giảm độ trễ trong quá trình xử lý, đặc biệt trong các hệ thống xử lý liên tục, có thể trực tiếp cải thiện thông lượng bằng cách giảm thời gian chờ giữa các lệnh.
- Kiến trúc song song:** Triển khai nhiều đường dữ liệu song song trên FPGA cho phép xử lý đồng thời nhiều luồng dữ liệu, mang lại sự cải thiện đáng kể về thông lượng của hệ thống.

Dưới đây, chúng ta sẽ xem xét một ví dụ tính toán thông lượng trong bài toán nhân hai ma trận, qua đó phân tích sự ảnh hưởng của các yếu tố như tần số hoạt động, độ rộng dữ liệu, và độ trễ kiến trúc đến chỉ số thông lượng. Thông qua ví dụ này, chúng ta có thể hiểu rõ cách các yếu tố kỹ thuật tác động đến hiệu quả xử lý và tối ưu hóa hệ thống.

*Ví dụ 5.2: Giả sử một hệ thống SoC có CPU, bộ nhớ và một IP tự thiết kế thực hiện bài toán nhân hai ma trận có kích thước  $M \times K$  và  $K \times N$ . Mỗi phần tử là một số thực (float 32 bit). Tần số hoạt động của FPGA là 200 MHz, tần số hoạt động của IP tự thiết kế là 500 MHz, độ trễ truy cập bộ nhớ 100 ns, độ trễ bus là 50 ns, các độ trễ khác xem như bỏ qua. Biết mỗi phép tính nhân cộng dồn (MAC – Multiply-Accumulate) tốn 5 ns và giả sử mỗi phép tính cần tải dữ liệu từ bộ nhớ một lần. Tính toán thông lượng của hệ thống SoC này trong trường hợp IP thiết kế không xử lý song song và có xử lý song song.*

Tổng số phép tính MAC ( $P_{MAC}$ ) cần thực hiện cho bài toán nhân hai ma trận  $M \times K$  và  $K \times N$  là:

$$P_{MAC} = M \times K \times N$$

Tổng thời gian thực hiện hết các phép toán MAC, với giả sử thời gian thực hiện tính cho 1 lần MAC là  $T_{MAC}$  được cho bởi công thức:

Trường hợp 1: kiến trúc không song song:

$$T_{tổng} = P_{MAC} \times T_{MAC}$$

Trường hợp 2: kiến trúc song song L lõi tính toán, mỗi lõi tính 1 phép tính MAC

$$T_{tổng} = \frac{P_{MAC}}{L} \times T_{MAC}$$

Độ trễ truy cập dữ liệu để tính toán được cho bởi công thức:

$$T_{độ trễ} = T_{memory} + T_{bus}$$

Vì mỗi phép tính MAC cần truy cập bộ nhớ 1 lần nên ta có tổng thời gian thực thi được tính như sau:

$$T = T_{tổng} + P_{MAC} \times T_{độ trễ}$$

Tổng dữ liệu:

$$D_{tổng} = (M \times K + K \times N + M \times N) \times 32 \text{ bit}$$

Khi đó thông lượng được tính bởi công thức:

$$\text{Thông lượng} = \frac{D_{tổng}}{T}$$

Áp dụng cho trường hợp  $M = K = N = 128$ ,  $L = 4$

Trường hợp không song song

$$P_{MAC} = 128 \times 128 \times 128 = 2,097,152$$

$$T_{tổng} = 2,097,152 \times 5ns = 10,485,760 ns$$

$$T = 10,485,760 + (100 ns + 50 ns) \times 2,097,152 = 352,058,560 ns$$

$$D_{tổng} = (128 \times 128 + 128 \times 128 + 128 \times 128) = 1,572,864 bit$$

$$\text{thông lượng} = \frac{1,572,864}{352,058,560} \approx 4.84 Mbps$$

Trường hợp song song với  $L = 4$

$$P_{MAC} = 128 \times 128 \times 128 = 2,097,152$$

$$T_{tổng} = \frac{2,097,152}{4} \times 5ns = 524,288 ns$$

$$T = 524,288 + (100 ns + 50 ns) \times 2,097,152 = 315,097,088 ns$$

$$D_{tổng} = (128 \times 128 + 128 \times 128 + 128 \times 128) = 1,572,864 bit$$

$$\text{thông lượng} = \frac{1,572,864}{315,097,088} \approx 4.99 Mbps$$

Tương tự nếu thử với  $L = 64$  thì thông lượng của hệ thống này sẽ bằng:

$$\text{thông lượng} = \frac{1,572,864}{314,736,640} \approx 5 Mbps$$

Qua bài ví dụ trên, có thể thấy rằng độ trễ và kiến trúc song song đóng vai trò quan trọng trong việc quyết định hiệu năng tổng thể của hệ thống SoC khi thực hiện bài toán nhân ma trận.

Đầu tiên, độ trễ (bao gồm độ trễ truy cập bộ nhớ và bus) có ảnh hưởng lớn đến tổng thời gian xử lý. Dù thời gian thực hiện MAC khá ngắn, nhưng khi độ trễ truy cập bộ nhớ được nhân lên theo số lượng phép tính cần thực hiện, nó chiếm phần lớn tổng thời gian xử lý. Điều này đặc biệt rõ ràng trong ví dụ khi độ trễ truy cập bộ nhớ và bus (150 ns) được nhân với tổng số phép MAC, dẫn đến thời gian thực hiện tăng đáng kể, làm giảm thông

lượng hệ thống. Thứ hai, kiến trúc song song giúp cải thiện hiệu năng đáng kể bằng cách chia nhỏ công việc cho nhiều nhân xử lý đồng thời. Khi số lượng lõi song song tăng lên, thời gian thực hiện các phép MAC giảm đi tương ứng. Tuy nhiên, ngay cả khi sử dụng kiến trúc song song, nếu độ trễ truy cập dữ liệu vẫn lớn, hiệu quả cải thiện thông lượng bị hạn chế. Ví dụ, với  $L=4$  thì thời gian tính toán có giảm, nhưng độ trễ truy cập dữ liệu vẫn là yếu tố chi phối thời gian tổng, khiến thông lượng chỉ đạt khoảng 4.99 Mbps.

Kết luận, để đạt được hiệu năng cao, việc giảm độ trễ truy cập dữ liệu và tăng số lượng nhân song song là những yếu tố cần được tối ưu đồng thời. Điều này nhấn mạnh tầm quan trọng của việc cân bằng giữa thiết kế phần cứng và tối ưu hóa truy cập dữ liệu trong các hệ thống SoC.

## 5.4. Đánh giá hiệu quả của hệ thống

Trong các phần trước, chúng ta đã thực hiện đánh giá về diện tích, năng lượng, và hiệu năng của hệ thống. Tuy nhiên, khi chỉ dựa vào một trong hai tiêu chí này, việc so sánh sự hiệu quả giữa các hệ thống trở nên khó khăn và không đầy đủ. Diện tích và năng lượng có thể ảnh hưởng trực tiếp đến chi phí và khả năng mở rộng của hệ thống, trong khi hiệu năng lại phản ánh khả năng xử lý của hệ thống. Vì vậy, để có cái nhìn tổng quan và công bằng hơn về sự hiệu quả của các thiết kế, cần phải có một tiêu chí đánh giá kết hợp, thể hiện sự tương quan giữa diện tích, năng lượng, và hiệu năng. Phần này sẽ trình bày về hai khía cạnh quan trọng trong việc đánh giá này: hiệu quả diện tích và hiệu quả năng lượng, qua đó giúp nhà thiết kế có cái nhìn sâu sắc hơn về sự tối ưu của hệ thống.

### 5.4.1. Đánh giá hiệu quả diện tích

Đối với hiệu năng của hệ thống, chúng ta có hai đại lượng quan trọng là độ trễ và thông lượng, và đánh giá hiệu quả diện tích cũng sẽ dựa trên hai đại lượng này. Cụ thể, khi đánh giá hiệu quả diện tích, chúng ta xem xét hiệu quả diện tích cho độ trễ và hiệu quả diện tích cho thông lượng. Đối với độ trễ, chỉ số **tích diện tích trễ (ADP)** được sử dụng, trong đó diện tích được tính bằng LUT tổng (bao gồm LUT thực tế và LUT chuyển đổi tương đương từ BRAM và DSP) và độ trễ tính bằng giây, mili giây, hoặc micro giây. Mục tiêu là giảm ADP để hệ thống hoạt động hiệu quả với diện tích tối thiểu và thời gian xử lý ngắn nhất.

Đối với thông lượng, chỉ số **thông lượng trên mỗi đơn vị diện tích** được tính bằng thông lượng / diện tích, và giá trị này càng lớn càng tốt, phản ánh khả năng hệ thống xử lý dữ liệu hiệu quả mà không chiếm dụng quá nhiều tài nguyên phần cứng.

### Tích diện tích trễ (ADP)

Tích diện tích trễ được tính bằng công thức dưới đây:

$$ADP = \text{Diện tích} \times \text{Độ trễ} \quad (5.17)$$

Trong đó:

- Diện tích: Là tổng số LUT sử dụng trong thiết kế.
- Độ trễ: Là thời gian cần thiết để hoàn thành một tác vụ hoặc truyền tải dữ liệu trong hệ thống.

ADP càng nhỏ thì hệ thống càng tốt, phản ánh được hệ thống vừa có diện tích nhỏ và độ trễ nhỏ, hoặc hệ thống có diện tích rất nhỏ và độ trễ chấp nhận được, hoặc hệ thống có diện tích chấp nhận được và độ trễ rất thấp. Điều này cho thấy thiết kế tối ưu, không chỉ tiết kiệm tài nguyên phần cứng mà còn đảm bảo thời gian xử lý nhanh chóng.

*Ví dụ 5.3: Giả sử chúng ta có hai thiết kế hệ thống SoC khác nhau. Thiết kế 1 có diện tích là 5000 LUT và độ trễ là 200 mili giây (ms), trong khi Thiết kế 2 có diện tích là 4000 LUT và độ trễ là 300 ms. Xác định thiết kế nào hiệu quả hơn.*

Đầu tiên, chúng ta tính Tích diện tích trễ (ADP) cho mỗi thiết kế.

Thiết kế 1:

$$ADP_{1_1} = 5000 \text{LUT} \times 200\text{ms} = 1,000,000 \text{(LUT.ms)}$$

Thiết kế 2:

$$ADP_{1_2} = 4000 \text{LUT} \times 300\text{ms} = 1,200,000 \text{(LUT.ms)}$$

Kết luận:

Thiết kế 1 có ADP thấp hơn (1,000,000 LUT.ms so với 1,200,000 LUT.ms của Thiết kế 2), chứng tỏ rằng Thiết kế 1 là tối ưu hơn về sự kết hợp giữa diện tích và độ trễ. Mặc dù Thiết kế 2 có diện tích nhỏ hơn, nhưng độ trễ lớn hơn khiến ADP của Thiết kế 2 cao hơn, cho thấy Thiết kế 1 là thiết kế hiệu quả hơn.

Thông qua ví dụ, nếu các hệ thống chỉ được đánh giá hiệu năng bằng độ trễ, việc sử dụng ADP sẽ giúp chúng ta so sánh và xác định hệ thống nào hiệu quả hơn về sự kết hợp giữa diện tích và độ trễ. ADP không chỉ phản ánh thời gian xử lý mà còn giúp đánh giá việc sử dụng tài nguyên phần cứng một cách hợp lý. Vì vậy, thông qua ADP, chúng ta có thể nhận diện được những thiết kế có sự cân bằng tốt giữa diện tích và độ trễ, từ đó xác định thiết kế nào cần cải tiến để đạt hiệu quả cao hơn. Trong trường hợp này, ADP thấp sẽ là mục tiêu tối ưu, giúp cải thiện không chỉ hiệu năng mà còn tiết kiệm tài nguyên và giảm chi phí hệ thống.

### Thông lượng trên mỗi đơn vị diện tích

Thông lượng trên mỗi đơn vị diện tích được tính bằng công thức sau đây:

$$\text{Thông lượng trên mỗi đơn vị diện tích} = \frac{\text{Thông lượng}}{\text{Diện tích}} \quad (5.18)$$

Trong đó:

- Diện tích: Là tổng số LUT sử dụng trong thiết kế.
- Thông lượng: Là lượng dữ liệu mà hệ thống có thể xử lý trong một khoảng thời gian nhất định (thường là bit/giây hoặc phép toán/giây).

Chỉ số thông lượng trên mỗi đơn vị diện tích càng lớn càng tốt, phản ánh khả năng hệ thống có thể xử lý dữ liệu hiệu quả mà không cần sử dụng quá nhiều tài nguyên phần cứng. Khi thiết kế hệ thống SoC, việc tối ưu hóa Thông lượng trên mỗi đơn vị diện tích giúp đảm bảo rằng hệ thống có thể xử lý nhiều dữ liệu trong khi vẫn duy trì mức độ sử dụng tài nguyên hợp lý.

*Ví dụ 5.4: Giả sử chúng ta có hai thiết kế hệ thống SoC. Thiết kế 1 có thông lượng là 5 Gbps và diện tích là 5000 LUT, trong khi Thiết kế 2 có thông lượng là 2 Gbps và diện tích là 1000 LUT. Xác định thiết kế nào hiệu quả hơn.*

Đầu tiên, chúng ta tính thông lượng trên mỗi đơn vị diện tích cho mỗi thiết kế.

Thiết kế 1:

$$\begin{aligned}\text{Thông lượng trên mỗi đơn vị diện tích 1} &= \frac{5 \text{ Gbps}}{5000 \text{ LUT}} = \frac{5000 \text{ Mbps}}{5000 \text{ LUT}} \\ &= 1 \text{ Mbps/LUT}\end{aligned}$$

Thiết kế 2:

$$\begin{aligned} \text{Thông lượng trên mỗi đơn vị diện tích 2} &= \frac{2 \text{ Gbps}}{1000 \text{ LUT}} = \frac{2000 \text{ Mbps}}{1000 \text{ LUT}} \\ &= 2 \text{ Mbps/LUT} \end{aligned}$$

Kết luận:

Khi tính thông lượng trên mỗi đơn vị diện tích, ta thấy rằng Thiết kế 2 có hiệu quả diện tích cao hơn, với 2 Mbps/LUT, trong khi Thiết kế 1 chỉ đạt 1 Mbps/LUT. Điều này cho thấy Thiết kế 2 sử dụng tài nguyên phần cứng hiệu quả hơn trong việc xử lý thông lượng, khi có thể xử lý nhiều dữ liệu hơn trong mỗi LUT. Tuy nhiên, khi xét đến thông lượng trên mỗi đơn vị diện tích, ta cần phải đảm bảo rằng không chỉ hiệu quả về mặt diện tích mà còn tối ưu hóa khả năng xử lý thông lượng, đảm bảo rằng thiết kế không chỉ tiết kiệm tài nguyên mà còn đạt được hiệu năng cao.

Qua ví dụ này, ta thấy rằng thông lượng trên mỗi đơn vị diện tích là một yếu tố quan trọng trong việc đánh giá hiệu quả sử dụng tài nguyên và khả năng xử lý của hệ thống. Thiết kế 2, mặc dù có thông lượng nhỏ hơn, lại đạt hiệu quả cao hơn trong việc sử dụng tài nguyên phần cứng để xử lý thông lượng. Điều này cho thấy Thiết kế 2 là tối ưu hơn về mặt tài nguyên phần cứng và hiệu năng, vì nó có thể xử lý nhiều dữ liệu với ít tài nguyên hơn.

#### 5.4.2. Đánh giá hiệu quả năng lượng

Tương tự như đánh giá hiệu quả diện tích, khi đánh giá hiệu quả năng lượng, chúng ta cũng sẽ xét đến hai đại lượng hiệu năng của hệ thống, bao gồm độ trễ và thông lượng. Cụ thể, khi đánh giá hiệu quả năng lượng, chúng ta sẽ xem xét hiệu quả năng lượng cho độ trễ và hiệu quả năng lượng cho thông lượng. Đối với độ trễ, chỉ số **tích năng lượng độ trễ (PDP)** có thể được sử dụng, trong đó năng lượng được tính theo công suất tiêu thụ trong quá trình xử lý và độ trễ tính bằng giây, mili giây hoặc micro giây. Mục tiêu là giảm PDP trong khi vẫn duy trì độ trễ thấp, từ đó giúp hệ thống hoạt động hiệu quả về mặt năng lượng. Đối với thông lượng, chỉ số **thông lượng trên mỗi đơn vị năng lượng** có thể được tính bằng thông lượng chia cho công suất tiêu thụ, và giá trị này càng cao càng tốt, phản ánh khả năng hệ thống xử lý dữ liệu hiệu quả trong khi tiêu thụ ít năng lượng. Điều này giúp

hệ thống duy trì hiệu năng cao mà không làm tăng quá mức mức tiêu thụ năng lượng, từ đó tối ưu hóa hiệu quả năng lượng của toàn bộ hệ thống.

### Tích năng lượng độ trễ (PDP)

Tích năng lượng độ trễ được tính bằng công thức dưới đây:

$$PDP = \text{Năng lượng} \times \text{Độ trễ} \quad (5.19)$$

Trong đó:

- Năng lượng: Là năng lượng tiêu thụ của thiết kế, tính bằng Watt hoặc mili Watt.
- Độ trễ: Là thời gian cần thiết để hoàn thành một tác vụ hoặc truyền tải dữ liệu trong hệ thống.

PDP càng nhỏ thì hệ thống càng tốt, phản ánh được hệ thống vừa có năng lượng tiêu thụ thấp và độ trễ nhỏ, hoặc hệ thống có năng lượng tiêu thụ rất thấp và độ trễ chấp nhận được, hoặc hệ thống có năng lượng tiêu thụ chấp nhận được và độ trễ rất thấp. Điều này cho thấy thiết kế tối ưu, không chỉ tiết kiệm tài nguyên phần cứng mà còn đảm bảo thời gian xử lý nhanh chóng.

*Ví dụ 5.5: Giả sử chúng ta có hai thiết kế hệ thống SoC khác nhau. Thiết kế 1 có năng lượng tiêu thụ là 1 W và độ trễ 2 giây, trong khi Thiết kế 2 có năng lượng tiêu thụ là 4 W và độ trễ là 1 giây. Xác định thiết kế nào hiệu quả hơn.*

Đầu tiên, chúng ta tính Tích năng lượng độ trễ (PDP) cho mỗi thiết kế.

Thiết kế 1:

$$PDP_{1_1} = 1 \times 2 = 2 \text{ (Joule)}$$

Thiết kế 2:

$$PDP_{1_2} = 4 \times 1 = 4 \text{ (Joule)}$$

Kết luận:

Thiết kế 1 có PDP thấp hơn (2 Joule so với 4 Joule của Thiết kế 2), chứng tỏ rằng Thiết kế 1 là tối ưu hơn về sự kết hợp giữa năng lượng tiêu thụ và độ trễ. Mặc dù Thiết kế 2 có năng lượng tiêu thụ lớn hơn, nhưng độ trễ ngắn hơn khiến PDP của Thiết kế 2 cao hơn, cho thấy Thiết kế 1 là thiết kế hiệu quả hơn.

Qua ví dụ này, ta thấy rằng khi các hệ thống chỉ được đánh giá hiệu năng bằng độ trễ hoặc năng lượng, việc sử dụng PDP sẽ giúp chúng ta so sánh và xác định hệ thống nào hiệu quả hơn về sự kết hợp giữa năng lượng và độ trễ. PDP không chỉ phản ánh thời gian xử lý mà còn giúp đánh giá việc sử dụng tài nguyên phần cứng một cách hợp lý. Vì vậy, thông qua PDP, chúng ta có thể nhận diện được những thiết kế có sự cân bằng tốt giữa năng lượng và độ trễ, từ đó xác định thiết kế nào cần cải tiến để đạt hiệu quả cao hơn. Trong trường hợp này, PDP thấp sẽ là mục tiêu tối ưu, giúp cải thiện không chỉ hiệu năng mà còn tiết kiệm tài nguyên và giảm chi phí hệ thống.

### Thông lượng trên mỗi đơn vị năng lượng

Thông lượng trên mỗi đơn vị năng lượng được tính bằng công thức sau đây:

$$\text{Thông lượng trên mỗi đơn vị năng lượng} = \frac{\text{Thông lượng}}{\text{Năng lượng}} \quad (5.20)$$

Trong đó:

- Năng lượng: Là năng lượng tiêu thụ của thiết kế, tính bằng Watt hoặc mili Watt.
- Thông lượng: Là lượng dữ liệu mà hệ thống có thể xử lý trong một khoảng thời gian nhất định (thường là bit/giây hoặc phép toán/giây).

Chỉ số thông lượng trên mỗi đơn vị năng lượng càng lớn càng tốt, phản ánh khả năng hệ thống có thể xử lý dữ liệu hiệu quả trong khi tiêu thụ ít năng lượng. Khi thiết kế hệ thống SoC, việc tối ưu hóa thông lượng trên mỗi đơn vị năng lượng giúp đảm bảo rằng hệ thống có thể xử lý nhiều dữ liệu mà không làm tăng quá mức mức tiêu thụ năng lượng. Điều này giúp duy trì hiệu năng cao mà không gây ra sự tiêu hao năng lượng lớn.

*Ví dụ 5.6: Giả sử chúng ta có hai thiết kế hệ thống SoC. Thiết kế 1 có thông lượng là 5 Gbps và năng lượng là 5 Watt, trong khi Thiết kế 2 có thông lượng là 3 Gbps và diện tích là 1.5 Watt. Xác định thiết kế nào hiệu quả hơn.*

Đầu tiên, chúng ta tính thông lượng trên mỗi đơn vị năng lượng mỗi thiết kế.

Thiết kế 1:

$$\text{Thông lượng trên mỗi đơn vị năng lượng 1} = \frac{5 \text{ Gbps}}{5 \text{ W}} = 1 \text{ Gbps/W}$$

Thiết kế 2:

$$\text{Thông lượng trên mỗi đơn vị năng lượng} 2 = \frac{3 \text{ Gbps}}{1.5 \text{ W}} = 2 \text{ Gbps/W}$$

Kết luận:

Khi tính thông lượng trên mỗi đơn vị năng lượng, ta thấy rằng Thiết kế 2 có hiệu quả năng lượng cao hơn, với 2 Gbps/W, trong khi Thiết kế 1 chỉ đạt 1 Gbps/W. Điều này cho thấy Thiết kế 2 sử dụng năng lượng hiệu quả hơn trong việc xử lý thông lượng, khi có thể xử lý nhiều dữ liệu hơn trong mỗi Watt năng lượng tiêu thụ. Tuy nhiên, khi xét đến thông lượng trên mỗi đơn vị năng lượng, ta cần đảm bảo rằng không chỉ hiệu quả về mặt năng lượng mà còn tối ưu hóa khả năng xử lý thông lượng, đảm bảo rằng thiết kế không chỉ tiết kiệm năng lượng mà còn đạt được hiệu năng cao.

Thông qua ví dụ này, ta thấy rằng thông lượng trên mỗi đơn vị năng lượng là một yếu tố quan trọng trong việc đánh giá hiệu quả sử dụng tài nguyên và khả năng xử lý của hệ thống. Thiết kế 2, mặc dù có thông lượng nhỏ hơn, lại đạt hiệu quả cao hơn trong việc sử dụng năng lượng để xử lý thông lượng. Điều này cho thấy Thiết kế 2 là tối ưu hơn về mặt hiệu quả năng lượng và hiệu năng, vì nó có thể xử lý nhiều dữ liệu với ít năng lượng hơn.

## 5.5. Độ tin cậy

Độ tin cậy trong thiết kế vi mạch SoC đề cập đến khả năng của một vi mạch để hoạt động đúng theo các thông số kỹ thuật đã được định trước, trong suốt một khoảng thời gian nhất định và dưới những điều kiện vận hành cụ thể. Độ tin cậy không chỉ bao gồm khả năng chống lại lỗi hệ thống thông qua việc phát hiện và sửa lỗi, mà còn liên quan đến việc duy trì hiệu năng, độ bền và tuổi thọ của sản phẩm trong điều kiện sử dụng thực tế.

Độ tin cậy của một vi mạch nói chung hay một thiết kế SoC nói riêng là một yếu tố phức tạp, liên quan trực tiếp đến diện tích die, tần số đồng hồ và công suất tiêu thụ. Ví dụ khi diện tích die tăng lên để tạo không gian cần thiết để áp dụng các kỹ thuật sửa lỗi và phát hiện lỗi, như ECC (Error Correction Code) thì lượng mạch điện trên chip cũng nhiều hơn, điều này không chỉ tăng khả năng phát sinh lỗi mà còn có thể dẫn đến việc tăng chi phí sản xuất và giảm hiệu năng năng lượng tổng thể. Hoặc việc tăng tần số clock có thể dẫn đến sự gia tăng của nhiễu điện và độ nhạy với nhiễu, vì tín hiệu điện tử di chuyển

nhanh hơn có thể bị ảnh hưởng bởi nhiều từ các nguồn khác nhau bên trong và bên ngoài chip. Các mạch hoạt động ở tốc độ cao thường được thiết kế nhỏ gọn hơn, điều này làm cho chúng dễ bị ảnh hưởng bởi bức xạ và các tác động môi trường khác, từ đó làm giảm độ tin cậy. Qua đó cho thấy mối quan hệ phức tạp giữa kích thước, tốc độ và công suất trong việc thiết kế vi mạch. Các nhà thiết kế cần cân nhắc kỹ lưỡng giữa việc mở rộng diện tích die để tăng cường độ tin cậy thông qua các kỹ thuật phát hiện và sửa lỗi, đồng thời giữ cho tần số clock ở mức độ thích hợp để không làm tăng nhiễu và giảm sự nhạy cảm với các yếu tố gây hại bên ngoài. Đây là một bài toán tối ưu mà các kỹ sư phải giải quyết để đạt được sự cân bằng giữa hiệu năng, công suất và độ tin cậy trong thiết kế vi mạch.

Các hệ thống SoC trên FPGA, nhờ tính năng tái cấu hình, đã nâng cao đáng kể độ tin cậy, đặc biệt trong các ứng dụng yêu cầu hoạt động ổn định và liên tục. Khả năng này cho phép FPGA linh hoạt thích nghi với các điều kiện hoạt động thay đổi và giảm thiểu tác động của các lỗi tiềm ẩn. Khi một phần mạch trên FPGA gặp sự cố, chẳng hạn do quá nhiệt hoặc hư hỏng vật lý, FPGA có thể tái cấu hình để "bỏ qua" phần mạch bị lỗi, đồng thời định tuyến lại các chức năng qua các khối logic còn hoạt động. Quá trình này diễn ra mà không làm gián đoạn toàn bộ hệ thống, giúp duy trì hiệu năng ổn định và kéo dài tuổi thọ thiết bị. Nhờ khả năng "tự sửa chữa" này, FPGA không chỉ giảm thiểu rủi ro hỏng hóc hệ thống mà còn giảm chi phí thay thế phần cứng. Tính năng vượt trội này khiến FPGA trở thành lựa chọn ưu tiên trong các lĩnh vực đòi hỏi độ tin cậy cao, như hàng không vũ trụ, y tế và tự động hóa công nghiệp, nơi mà bất kỳ sự cố nào cũng có thể dẫn đến hậu quả nghiêm trọng.

### 5.5.1. Lỗi và sửa lỗi

Trong quá trình phát triển và sử dụng FPGA (Field Programmable Gate Array), các loại lỗi có thể xảy ra và cần được phát hiện cũng như sửa chữa kịp thời để đảm bảo hệ thống hoạt động chính xác. Dưới đây là các loại lỗi phổ biến và cách xử lý:

- ✓ **Lỗi tín hiệu:** Đây là lỗi do giá trị tín hiệu không chính xác, có thể xảy ra do nhiễu hoặc các vấn đề liên quan đến thiết kế đường truyền tín hiệu. Ví dụ, nếu một tín hiệu

xung clock bị méo hoặc không đáp ứng yêu cầu thời gian, hệ thống có thể hoạt động không đúng. Để khắc phục, kỹ sư thường sử dụng các kỹ thuật như lọc tín hiệu, tối ưu hóa thiết kế mạch hoặc thêm các mạch kiểm tra lỗi (Error Detection Circuit).

- ✓ **Lỗi logic:** Lỗi logic thường biểu hiện dưới dạng kết quả đầu ra không đúng so với mong đợi. Ví dụ, trong một thiết kế FPGA thực hiện phép cộng, nếu kết quả phép toán luôn sai khi một số đầu vào cụ thể được đưa vào, thì đó là lỗi logic. Nguyên nhân có thể là do sai sót trong quá trình viết mã HDL (Hardware Description Language). Sửa lỗi này yêu cầu kiểm tra kỹ lưỡng logic thiết kế thông qua mô phỏng hoặc phân tích trạng thái.
- ✓ **Lỗi vật lý:** Lỗi này do các yếu tố môi trường, chẳng hạn như lão hóa linh kiện, bức xạ, hoặc biến đổi nhiệt độ. Ví dụ, khi FPGA hoạt động trong một môi trường có nhiệt độ cao liên tục, một số phần tử logic có thể bị hỏng hoặc hoạt động không ổn định. Cách xử lý bao gồm thiết kế mạch chịu nhiệt tốt hơn, sử dụng vật liệu bền bỉ, hoặc áp dụng các thuật toán phát hiện và sửa lỗi (Error Correction Code - ECC).
- ✓ **Lỗi thiết kế:** Lỗi thiết kế xuất phát từ việc thực hiện không đúng các yêu cầu hoặc quy định thiết kế. Ví dụ, nếu trong một thiết kế cần sử dụng chuẩn giao tiếp SPI (Serial Peripheral Interface) nhưng lại sử dụng nhầm cấu hình chân hoặc xung clock, hệ thống sẽ không hoạt động như mong muốn. Để khắc phục, cần kiểm tra toàn bộ quy trình thiết kế, áp dụng các công cụ kiểm tra tự động như linting tool, và thực hiện kiểm tra thiết kế kỹ lưỡng trước khi triển khai.

Tốc độ phát sinh lỗi  $\theta$  theo thời gian trung bình  $T$  với 1 xác suất lỗi  $P(t)$  được cho bởi công thức:

$$\theta = \frac{1}{T} \quad (5.21)$$

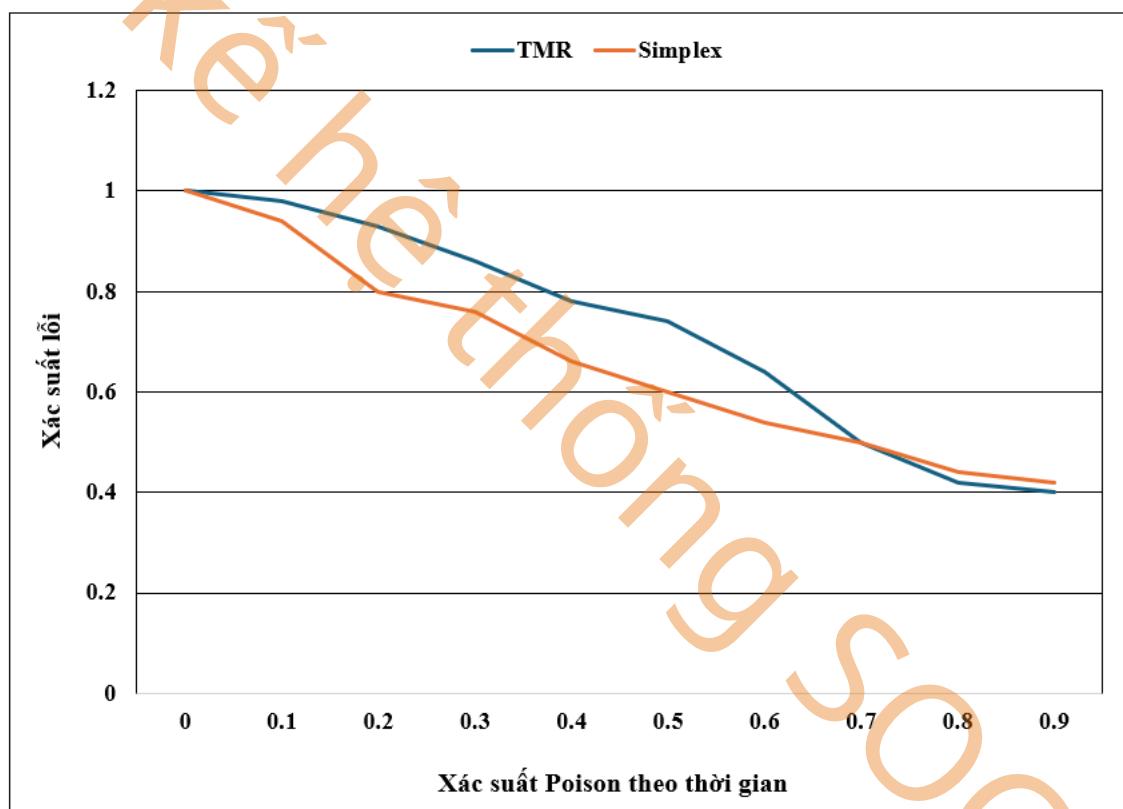
Và một xác suất lỗi theo phân bố Poisson được cho bởi công thức:

$$P(t) = e^{-\frac{t}{T}} = e^{-t\theta} \quad (5.22)$$

Khi đó để cải thiện độ tin cậy, chúng ta phải làm giảm giá trị  $P(t)$  xuống.

### 5.5.2. Kỹ thuật cải tiến độ tin cậy

Một kỹ thuật nổi tiếng được biết là triple modular redundancy (TMR). TMR là một kỹ thuật cải tiến độ tin cậy cao được sử dụng trong thiết kế hệ thống điện tử, đặc biệt là trong những ứng dụng đòi hỏi mức độ an toàn và độ tin cậy rất cao như hệ thống điều khiển hàng không và y tế. TMR hoạt động dựa trên nguyên tắc ba lần lặp: ba mô-đun chức năng giống hệt nhau thực hiện cùng một tác vụ và kết quả của chúng được đưa vào một bộ lọc hậu xử lý, thường là một bộ biểu quyết, để chọn ra kết quả cuối cùng. Nếu một trong ba mô-đun gặp sự cố, hai mô-đun còn lại đảm bảo rằng hệ thống vẫn hoạt động chính xác, qua đó giảm thiểu rủi ro hệ thống thất bại do lỗi đơn lẻ.



Hình 5.13: Độ tin cậy TMR so sánh với độ tin cậy Simplex

So với mô hình đơn giản (simplex), nơi chỉ có một mô-đun duy nhất thực hiện tác vụ mà không có dự phòng, TMR cung cấp một mức độ độ tin cậy cao hơn đáng kể. Trong khi một hệ thống simplex có thể dừng hoạt động ngay khi gặp phải lỗi, hệ thống TMR có khả năng chịu lỗi, có nghĩa là nó có thể tiếp tục hoạt động mà không bị gián đoạn, ngay cả khi một phần của nó không hoạt động. Điều này làm cho TMR trở thành lựa chọn ưu tiên cho các ứng dụng mà sự thất bại có thể dẫn đến hậu quả nghiêm trọng. Tuy cải tiến mức

độ độ tin cậy cao hơn nhưng hệ thống TMR yêu cầu nhiều phần cứng hơn, dẫn đến tăng chi phí, diện tích, và công suất tiêu thụ. Cần cân nhắc kỹ lưỡng giữa chi phí và lợi ích khi quyết định áp dụng TMR cho một ứng dụng cụ thể.

Nhìn vào hình 5.13, chúng ta thấy theo thời gian tổng số lỗi phát sinh ra khi áp dụng kỹ TMR cao hơn mô hình đơn giản, nhưng ngược lại hệ thống vẫn tiếp tục hoạt động vì có mô-đun dự phòng có thể sửa khắc phục lỗi khi xảy ra sự cố như đã đề cập ở trên. Lỗi xuất hiện theo TMR cao hơn mô hình Simplex có thể ước tính dựa vào công thức sau:

$$t = T \times 2 \quad (5.25)$$

### 5.5.3. Khắc phục lỗi từ nhà máy sản xuất chip

Việc xử lý các lỗi sản xuất trong quá trình chế tạo ASIC hoặc SoC được thực hiện chủ yếu qua kiểm tra và thử nghiệm. Khi mật độ transistor trên mỗi die tăng lên, tổng số lượng transistor trên die cũng tăng theo tỷ lệ, và vấn đề kiểm thử trở nên phức tạp hơn nhiều. Các tổ hợp có thể kiểm tra tăng lên theo hàm số mũ so với số lượng transistor, đưa ra một lượng lớn các trường hợp cần được xác minh để đảm bảo rằng vi mạch hoạt động đúng đắn. Nếu không có một đột phá nào trong lĩnh vực kiểm thử, dự kiến trong vài năm tới, chi phí kiểm thử die sẽ vượt qua phần còn lại của chi phí sản xuất. Điều này không chỉ nêu bật một thách thức về mặt kỹ thuật mà còn là một vấn đề kinh tế, buộc ngành công nghiệp phải tìm kiếm các phương pháp mới để tối ưu hóa quá trình kiểm thử hoặc phát triển các công nghệ sản xuất ít lỗi hơn.

Người thiết kế phần cứng có thể đóng góp đáng kể vào công tác kiểm thử và xác nhận thông qua một quá trình được gọi là thiết kế với khả năng kiểm thử (Design for Testability - DFT). Kỹ thuật scan cho phép cấu hình dữ liệu được xác định trước và lưu trữ ở đầu vào, đầu ra của các san được so sánh với các kết quả đã xác định trước. Chuỗi scan trong hình thức đơn giản nhất của nó bao gồm một điểm nhập và một điểm xuất riêng biệt từ mỗi ô nhớ. Mỗi điểm này được ghép nối (multiplexed) vào một bus nối tiếp, có thể được nạp vào/ra từ ô nhớ độc lập với phần còn lại của hệ thống. Kỹ thuật scan ban đầu được phát triển trong những năm 1960 như một phần của công nghệ máy tính lớn. Kỹ thuật này đã trở nên vô giá đối với ngành công nghiệp thiết kế vi mạch, vì nó cho phép kiểm tra mức độ

phức tạp cao của vi mạch hiện đại một cách có hệ thống và hiệu quả hơn. Sự tích hợp của các chuỗi scan vào thiết kế SoC là một phần quan trọng trong quy trình phát triển, giúp giảm thiểu rủi ro và đảm bảo rằng sản phẩm cuối cùng sẽ hoạt động đúng như mong đợi khi được triển khai trên thực tế. DFT bao gồm nhiều kỹ thuật khác nhau, mỗi kỹ thuật giải quyết các thách thức riêng biệt liên quan đến kiểm thử.

## 5.6. Sự đánh đổi trong hệ thống SoC

Trong việc triển khai hệ thống SoC trên FPGA, sự đánh đổi giữa hiệu năng, tài nguyên phần cứng (diện tích), và các yếu tố khác như độ tin cậy, tích hợp hệ thống là những yếu tố cốt lõi cần được cân nhắc kỹ lưỡng.

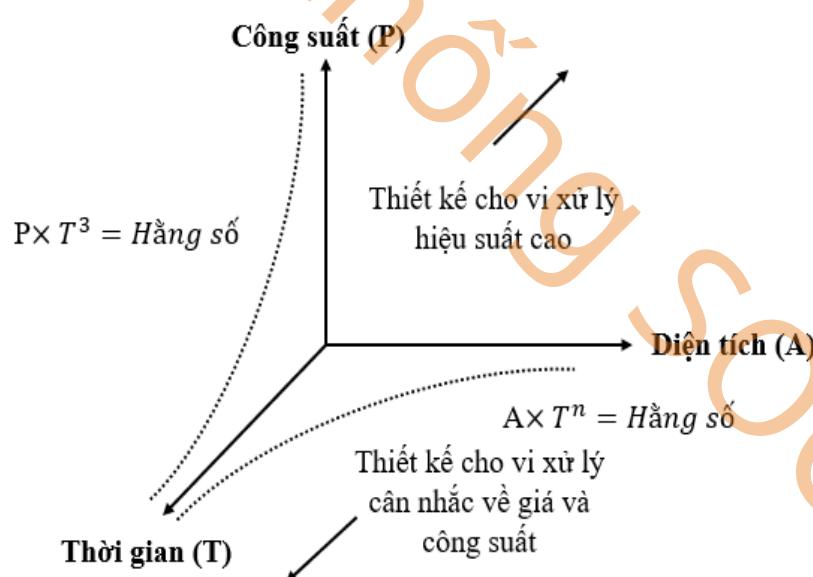
Hiệu năng là yếu tố trung tâm, thể hiện qua các chỉ số quan trọng như độ trễ, thông lượng, tần số hoạt động, công suất và năng lượng. Để giảm độ trễ hoặc tăng thông lượng, các nhà thiết kế thường phải tăng tần số hoạt động hoặc sử dụng nhiều tài nguyên phần cứng hơn, dẫn đến gia tăng diện tích chiếm dụng trên FPGA. Mặc dù điều này có thể cải thiện tốc độ xử lý dữ liệu và khả năng đáp ứng thời gian thực, nhưng nó cũng làm tăng chi phí và mức tiêu thụ năng lượng. Công suất tiêu thụ và năng lượng cũng là những yếu tố quan trọng khi đánh giá hiệu năng của hệ thống SoC, đặc biệt trong các ứng dụng tiết kiệm năng lượng như IoT, thiết bị di động hoặc hệ thống nhúng. Giảm công suất tiêu thụ không chỉ kéo dài thời lượng pin mà còn giảm nhiệt lượng tỏa ra, đảm bảo sự ổn định và an toàn của hệ thống. Tuy nhiên, việc này có thể dẫn đến giảm tần số hoạt động hoặc yêu cầu các thiết kế phức tạp hơn để duy trì hiệu năng mong muốn.

Diện tích phản ánh lượng tài nguyên phần cứng như LUTs, FFs, DSPs và BRAMs, ảnh hưởng trực tiếp đến chi phí triển khai và khả năng tích hợp của hệ thống. Tối ưu hóa diện tích có thể yêu cầu tái sử dụng hoặc giảm bớt các thành phần phần cứng, nhưng điều này thường dẫn đến sự gia tăng độ trễ hoặc giảm thông lượng, ảnh hưởng tiêu cực đến hiệu năng tổng thể.

Độ tin cậy là yếu tố ưu tiên hàng đầu trong các ứng dụng an toàn như hàng không hoặc y tế. Việc tăng cường độ tin cậy thường đòi hỏi bổ sung tài nguyên dự phòng hoặc cơ chế bảo vệ, điều này làm tăng diện tích và chi phí.

Cuối cùng, tích hợp hệ thống đòi hỏi sự cân bằng giữa việc tích hợp nhiều tính năng trên một con chip và duy trì hiệu năng tổng thể. Tích hợp quá mức có thể gây phức tạp trong thiết kế, làm tăng độ trễ và giảm tính linh hoạt khi kiểm tra và bảo trì hệ thống.

Hình 5.14 minh họa sự đánh đổi giữa tài nguyên phần cứng, thời gian, và công suất trong thiết kế mạch IP vi xử lý. Nghiên cứu lý thuyết chỉ ra rằng, trong các thiết kế bộ xử lý, tồn tại một giới hạn được biểu diễn bởi phương trình  $A \times T^n$ , trong đó  $n$  thường có giá trị từ 1 đến 2, phản ánh sự đánh đổi giữa diện tích  $A$  và thời gian thực thi  $T$ . Ngoài ra, cũng tồn tại một mối quan hệ đánh đổi giữa thời gian  $T$  và công suất  $P$ , biểu diễn qua giới hạn  $P \times T^3 = \text{Hằng số}$ . Các bộ xử lý nhúng và bộ xử lý cao cấp thường được thiết kế để hoạt động trong các vùng khác nhau của không gian ba chiều này. Cụ thể, đối với các bộ vi xử lý nhúng, các yếu tố như công suất và diện tích thường được tối ưu hóa để phù hợp với yêu cầu tiết kiệm năng lượng và kích thước nhỏ gọn. Ngược lại, đối với các bộ vi xử lý cao cấp, trực thời gian thường được ưu tiên để đạt hiệu năng cao, đáp ứng các tác vụ phức tạp và yêu cầu tính toán nhanh chóng.



Hình 5.14: Minh họa sự đánh đổi trong thiết kế vi xử lý

Việc đánh giá và lựa chọn giữa các tiêu chí đánh đổi trong thiết kế SoC không tuân theo một quy luật cố định. Trong nhiều trường hợp, một tiêu chí cụ thể sẽ được ưu tiên hơn các tiêu chí khác, tùy thuộc vào mục tiêu thiết kế và yêu cầu của ứng dụng. Ví dụ, trong

các thiết kế ưu tiên hiệu năng cao, việc giảm thiểu diện tích có thể không được chú trọng bằng việc tối ưu hóa tốc độ xử lý. Ngược lại, trong những ứng dụng mà mức tiêu thụ năng lượng là yếu tố quan trọng hàng đầu, hiệu quả năng lượng sẽ được ưu tiên hơn bất kỳ tiêu chí nào khác.

Quá trình lựa chọn và tối ưu hóa giữa các tiêu chí này đòi hỏi các nhà thiết kế phải có sự hiểu biết sâu sắc về kỹ thuật, nhận thức rõ các yêu cầu thị trường, cũng như khả năng dự báo và linh hoạt trước những thay đổi trong công nghệ và nhu cầu của người dùng. Chỉ bằng cách cân nhắc kỹ lưỡng và sáng tạo, các nhà thiết kế SoC mới có thể phát triển những giải pháp vừa đáp ứng các yêu cầu kỹ thuật, vừa thỏa mãn kỳ vọng thương mại. Dưới đây liệt kê một số ứng dụng thể hiện sự đánh đổi trong thiết kế SoC:

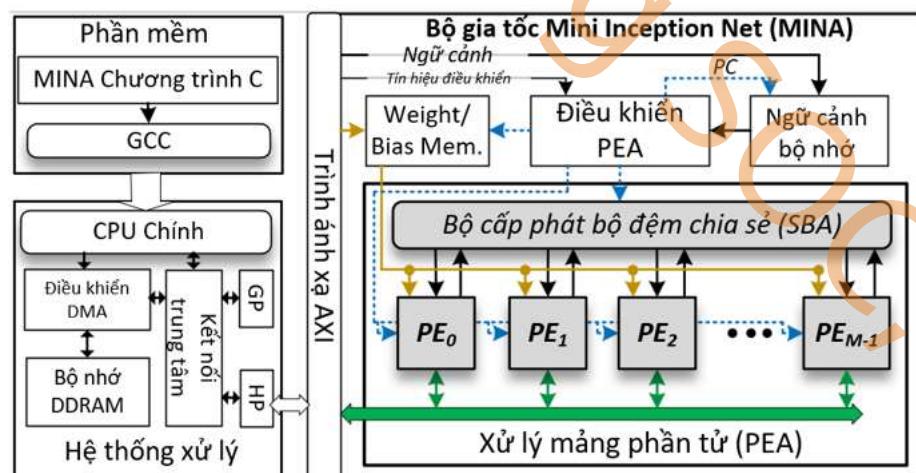
- ✓ **Hệ thống yêu cầu hiệu năng cao:** Tập trung tối ưu hóa thời gian xử lý, nhưng điều này thường đi kèm với sự gia tăng diện tích chip và mức tiêu thụ năng lượng.
- ✓ **Hệ thống giá thấp:** Uy tiên giảm chi phí sản xuất, đồng thời tăng khả năng tái sử dụng thiết kế và khả năng cấu hình lại, giúp đạt được hiệu quả kinh tế tối ưu. Tuy nhiên, trong trường hợp này, hệ thống thường sẽ chấp nhận hy sinh một phần hiệu năng, coi đó là yếu tố ít được ưu tiên để tập trung vào mục tiêu tiết kiệm chi phí và nâng cao tính linh hoạt trong sử dụng.
- ✓ **Hệ thống đeo đạc:** Nhấn mạnh mức tiêu thụ năng lượng thấp, vì nguồn cung cấp năng lượng ảnh hưởng trực tiếp đến trọng lượng và sự tiện lợi của thiết bị. Các hệ thống như điện thoại di động thường phải đáp ứng các yêu cầu thời gian thực và hiệu năng cao.
- ✓ **Hệ thống nhúng trong máy bay và các ứng dụng quan trọng về an toàn:** Tập trung vào độ tin cậy, với ưu tiên dành cho tuổi thọ thiết kế và khả năng cấu hình lại để đảm bảo hoạt động ổn định và an toàn trong môi trường khắc nghiệt.
- ✓ **Hệ thống chơi game:** Chú trọng đến chi phí, đặc biệt là chi phí sản xuất, trong khi hiệu năng được duy trì ở mức chấp nhận được. Độ tin cậy thường không phải là yếu tố ưu tiên hàng đầu trong loại hệ thống này.

Trong quá trình xem xét các yêu cầu, nhà thiết kế hệ thống trên chi cần phải cân nhắc kỹ lưỡng từng yếu tố đánh đổi để từ đó rút ra các thông số kỹ thuật tương ứng. Điều này đặt nền móng cho việc phát triển các giải pháp SoC không chỉ đáp ứng được yêu cầu về mặt kỹ thuật mà còn hợp lý về mặt chi phí và hiệu quả, qua đó mở ra hướng đi mới cho việc thiết kế và tối ưu hóa hệ thống.

## 5.7. Tích hợp hệ thống

Tính tích hợp hệ thống là một trong những ưu điểm nổi bật của SoC trên FPGA, cho phép gói gọn nhiều chức năng phức tạp trên một chip duy nhất. Với SoC trên FPGA, các thành phần như CPU, bộ nhớ, giao thức bus, và các khối IP được tích hợp chặt chẽ trên cùng một nền tảng. Điều này không chỉ giúp giảm kích thước hệ thống mà còn cải thiện hiệu năng, giảm độ trễ truyền dữ liệu, và tăng cường khả năng tương tác giữa các khối chức năng.

SoC trên FPGA cho phép tích hợp cả phần cứng tùy chỉnh và phần mềm, nhờ khả năng lập trình động của FPGA. Điều này mang lại tính linh hoạt cao, cho phép nhà thiết kế dễ dàng bổ sung, thay đổi hoặc tối ưu hóa các chức năng mà không cần thay đổi thiết kế vật lý. Ngoài ra, việc tích hợp hệ thống cũng cải thiện khả năng tiêu thụ năng lượng, vì các thành phần giao tiếp nội bộ tiêu tốn ít năng lượng hơn so với các kết nối ngoại vi.



Hình 5.12: Minh họa một hệ thống SoC thiết kế kiến trúc mạng Inception Net

Hệ thống SoC trong hình 5.12 thể hiện khả năng tích hợp vượt trội giữa phần cứng và phần mềm. **Phần cứng** bao gồm các **IP cứng** như CPU, AXI bus, bộ điều khiển DMA, và

bộ nhớ DDRAM. Đây là các IP được tích hợp sẵn từ nhà sản xuất FPGA, được tối ưu hóa để thực hiện các tác vụ quan trọng như quản lý dữ liệu, giao tiếp nội bộ, và điều khiển luồng dữ liệu, đảm bảo hiệu năng cao và độ ổn định cho SoC.

**IP mềm** Mini Inception Net Accelerator là khối tăng tốc mạng nơ-ron tích chập tùy chỉnh, với cấu trúc gồm mảng phần tử xử lý (PEA - Processing Element Array) được kiểm soát bởi bộ điều khiển PEA. Các trọng số được lưu trữ trong Weight/Bias Memory, và các tham số vận hành được lưu trong Context Memory, giúp hệ thống xử lý nhanh chóng và tái sử dụng tài nguyên hiệu quả. Bộ phân phối dữ liệu (SBA - Sharing Buffer Allocator) chịu trách nhiệm phân phối dữ liệu đầu vào tới các phần tử xử lý trong PEA, hỗ trợ thực hiện các phép nhân và cộng (MAC) song song, từ đó tối ưu hóa hiệu năng xử lý. IP này còn cho phép tùy chỉnh số lượng phần tử xử lý, đáp ứng linh hoạt các yêu cầu về diện tích và tốc độ cho nhiều thiết bị và ứng dụng khác nhau.

**Phần mềm** của hệ thống, được viết bằng ngôn ngữ C và biên dịch bằng GCC, tạo ra các cấu hình (CTX) để định nghĩa tham số mạng nơ-ron tích chập và điều khiển phần cứng. Hệ thống này minh họa sự tích hợp chặt chẽ giữa phần cứng và phần mềm, tối ưu hóa giao tiếp nội bộ và hỗ trợ quy trình pipeline. Điều này mang lại hiệu năng cao, tính linh hoạt và khả năng mở rộng, đáp ứng tốt các ứng dụng tính toán mạng nơ-ron tích chập.

Khả năng tích hợp hệ thống này đặc biệt hữu ích trong các ứng dụng như thiết bị nhúng, xử lý tín hiệu số, hệ thống truyền thông, và điều khiển công nghiệp. Nhờ tích hợp nhiều chức năng trên một chip, SoC trên FPGA không chỉ giúp giảm chi phí sản xuất mà còn tăng tính cạnh tranh và khả năng phát triển sản phẩm nhanh chóng trên thị trường.

## 5.8. Tóm tắt

Chương này tập trung vào các tiêu chí quan trọng để đánh giá hệ thống SoC trên FPGA, bao gồm hiệu năng, tiêu thụ năng lượng, tài nguyên phần cứng, độ tin cậy, và khả năng tích hợp. Hiệu năng hệ thống, được đo lường qua các chỉ số như độ trễ, thông lượng, và năng lượng, là yếu tố hàng đầu, đảm bảo SoC thực hiện các tác vụ với tốc độ và độ chính xác cao. Tiêu thụ năng lượng, đặc biệt quan trọng trong các ứng dụng nhúng và di động, cần được tối ưu hóa để kéo dài tuổi thọ pin và giảm nhiệt lượng. Tài nguyên phần

cứng, bao gồm LUT, FF, DSP, và RAM, cần được sử dụng hiệu quả để giảm diện tích, tối ưu chi phí sản xuất, và đảm bảo tích hợp mượt mà. Độ tin cậy của hệ thống, đặc biệt trong môi trường khắc nghiệt, yêu cầu các cơ chế phát hiện và sửa lỗi cùng quy trình kiểm thử nghiêm ngặt. Tích hợp hệ thống trên FPGA giúp giảm kích thước, tăng hiệu quả vận hành, và cho phép dễ dàng mở rộng hoặc điều chỉnh thiết kế nhờ sự linh hoạt của các IP cứng và mềm. Những tiêu chí này không chỉ đảm bảo chất lượng và ổn định của hệ thống mà còn tăng tính tương thích, tái sử dụng, và khả năng đáp ứng các yêu cầu ứng dụng thực tiễn.

## 5.9. Câu hỏi và bài tập

1. Những tài nguyên phần cứng chính trong Xilinx FPGA là gì, và chúng ảnh hưởng như thế nào đến thiết kế và hiệu năng của hệ thống SoC?
2. Một SoC điều khiển động cơ bao gồm CPU hoạt động ở tần số 500 MHz. Một bộ nhớ ngoài có độ trễ truy cập là 10 chu kỳ CPU. Bus I2C truyền dữ liệu với độ trễ mỗi chiều là 50 chu kỳ. Một khối IP điều khiển động cơ hoạt động ở 100 MHz, cần 1,000 chu kỳ để xử lý một lệnh điều khiển. Hệ thống hỗ trợ 8 tầng pipeline. Tính tổng độ trễ của hệ thống khi xử lý một lệnh điều khiển động cơ trong hai trường hợp:
  - a) Không sử dụng pipeline.
  - b) Có sử dụng pipeline.
3. Một hệ thống SoC có CPU, bộ nhớ và một IP tự thiết kế thực hiện bài toán tính tổng của dãy số có NNN phần tử. Mỗi phần tử là một số thực (float 32-bit). Tần số hoạt động của FPGA là 250 MHz, tần số hoạt động của IP tự thiết kế là 500 MHz. Độ trễ truy cập bộ nhớ là 80 ns, độ trễ bus là 40 ns, các độ trễ khác xem như bỏ qua. Biết mỗi phép tính cộng tốn 2 ns, và mỗi phép tính cần tải dữ liệu từ bộ nhớ một lần. Hãy tính toán:
  - a) Tổng thời gian thực hiện và thông lượng của hệ thống trong trường hợp IP thiết kế không xử lý song song.

- b) Tổng thời gian thực hiện và thông lượng của hệ thống trong trường hợp IP thiết kế có xử lý song song với 16 nhân xử lý.

4. Cho bảng thông tin sau:

	<b>BRAM</b>	<b>DSP</b>	<b>FF</b>	<b>LUT</b>
<b>VGG16</b>	1182	227	36343	49213
<b>Resnet34</b>	1118	275	89307	112823
<b>InceptionV1</b>	31466	765	156170	249723

- a) Dựa vào bảng thông tin này thực hiện chuyển đổi các khối DSP, BRAM sang FF và LUT. Biết rằng 1 DSP sẽ tương đương 32 FF và 280 LUT. 1 BRAM tương đương với 800 LUT và 32 FF.
- b) Tính số lượng FF và LUT cho ba thiết kế trên
- c) Nhận xét mức độ hiệu quả của ba thiết kế trên.
5. Nêu các cách để tối ưu hóa công suất tiêu thụ trên FPGA. Cho ví dụ minh họa 1 trường hợp cụ thể
6. Nêu một số giải pháp thực hiện kiểm tra mạch khi sản xuất SoC để hạn chế lỗi có thể xảy ra thấp nhất.
7. Giả sử bạn đang thiết kế một hệ thống SoC trên FPGA với hai thiết kế khác nhau cho cùng ứng dụng. Thiết kế 1 sử dụng 5000 LUTs, 2000 FFs, 20 DSP (1 DSP bằng 280 LUT và 32 FF), 10 BRAM (1 BRAM tương đương 800 LUT và 32 FF) và có công suất tiêu thụ là 3 W với độ trễ 500 mili giây. Thiết kế 2 sử dụng 6000 LUTs, 2500 FFs, 10 DSP (1 DSP bằng 280 LUT và 32 FF), 7 BRAM (1 BRAM tương đương 800 LUT và 32 FF) và có công suất tiêu thụ là 2.5 W với độ trễ 200 mili giây. Xác định thiết kế nào hiệu quả hơn.
8. Giả sử bạn đang thiết kế một hệ thống SoC trên FPGA với hai thiết kế khác nhau cho cùng ứng dụng. Thiết kế 1 sử dụng 2,500 LUTs, 3000 FFs, 28 DSP (2 DSP bằng 1,466 LUT và 1,587 FF), 25 BRAM (1 BRAM tương đương 800 LUT và 32 FF) và có công suất tiêu thụ là 5 W với thông lượng 5 Gbps. Thiết kế 2 sử dụng 12,000 LUTs, 2500 FFs, 22 DSP (2 DSP bằng 1,466 LUT và 1,587 FF), 42 BRAM (1 BRAM tương đương 800 LUT và 32 FF) và có công suất tiêu thụ là 6 W với thông lượng 7 Gbps. Xác định thiết kế nào hiệu quả hơn.

9. Giả sử bạn cần chọn một thiết kế SoC cho ứng dụng nhúng yêu cầu tiêu thụ điện thấp.

- a) Thiết kế 1: Thông lượng 1 Gbps, năng lượng 0.5 W, xử lý tác vụ trong 3 s.
  - b) Thiết kế 2: Thông lượng 1.5 Gbps, năng lượng 1.5 W, xử lý tác vụ trong 1.5 s
- Theo bạn thiết kế nào tốt hơn.

10. Nhìn vào bảng số liệu sau:

	Thiết kế 1		Thiết kế 2			
<b>Phương pháp</b>	1-D CNN			1-D CNN		
<b>Dữ liệu</b>	MIT-BIH			MIT-BIH		
<b>FPGA</b>	ZC706			ZC706		
<b>Tần số (MHz)</b>	200			200		
<b>GOP</b>	$1.028 \times 10^{-3}$			$0.32768 \times 10^{-3}$		
<b>Thông số</b>	11,065			6,261		
<b>Độ chính xác</b>	16-bit Fixed			16-bit Fixed		
<b>Độ thura</b>	0%	70%	0%	30%	70%*	
<b>Nguồn</b>	[-0.046875 : 0.046875]					
<b>LUT</b>	16,033	19,677	26,315			
	10.5	10.5	4.5			
	0	0	40			
	24,265	27,909	39,413			
<b>IT (<math>\mu</math>s)</b>	84	45	43.06	30.77	14.39*	
<b>ADP (s <math>\times</math> eLUT)</b>	2.04	1.26	1.77	1.26	0.59*	

a) So sánh tài nguyên phần cứng giữa thiết kế 1 và thiết kế 2.

b) Theo bạn thiết kế nào tốt hơn. Vì sao.

## CHƯƠNG 6: TỔNG HỢP CẤP CAO TRONG SoC FPGA

### 6.1. Giới thiệu

Tổng hợp cấp cao (HLS - High-Level Synthesis) trong thiết kế SoC trên FPGA mang lại nhiều lợi ích quan trọng, đặc biệt trong bối cảnh các hệ thống ngày càng phức tạp và yêu cầu thời gian phát triển nhanh. Tổng hợp cấp cao cho phép các nhà phát triển sử dụng ngôn ngữ cấp cao như Python, C/C++ thay vì các ngôn ngữ mô tả phần cứng truyền thống như Verilog hoặc VHDL, giúp giảm đáng kể độ phức tạp của việc phát triển. Điều này đặc biệt cần thiết khi thiết kế SoC trên FPGA, nơi các ứng dụng thường yêu cầu tích hợp nhiều thành phần như xử lý tín hiệu, quản lý bộ nhớ, giao tiếp ngoại vi, và các thuật toán tính toán hiệu năng cao.

Ngày nay, tổng hợp cấp cao không chỉ tự động chuyển đổi mã cấp cao thành mô tả phần cứng, mà còn hỗ trợ chuyển đổi các mô hình AI được phát triển bằng Python, ví dụ như TensorFlow hoặc PyTorch, sang RTL. Điều này cho phép các mô hình trí tuệ nhân tạo AI (Artificial Intelligent) được triển khai trực tiếp trên FPGA, mang lại hiệu suất cao, độ trễ thấp, và khả năng tối ưu hóa tài nguyên phần cứng. Nhờ đó, quá trình triển khai phần kiểm tra (inference) của các ứng dụng AI như nhận diện hình ảnh, xử lý ngôn ngữ tự nhiên có thể được tích hợp một cách dễ dàng vào thiết kế SoC.

Ngoài ra, tổng hợp cấp cao hỗ trợ các chiến lược tối ưu hóa như song song hóa và pipelining, giúp tăng tốc độ thực thi và cải thiện hiệu năng hệ thống. Với tổng hợp cấp cao, các kỹ sư có thể tập trung vào việc giải quyết các vấn đề thiết kế ở cấp hệ thống, thay vì tốn nhiều thời gian vào chi tiết kỹ thuật của phần cứng. Điều này không chỉ tăng năng suất mà còn rút ngắn thời gian đưa sản phẩm ra thị trường, đáp ứng nhanh chóng nhu cầu ngày càng cao về các hệ thống AI và SoC trên FPGA.

### 6.2. Giới thiệu Vivado HLS và Intel HLS

Tổng hợp ngôn ngữ cấp cao Vivado (Vivado High-Level Synthesis - Vivado HLS), hiện nay được đổi tên thành Vitis HLS, là một công cụ mạnh mẽ do Xilinx phát triển, cho phép chuyển đổi thiết kế phần mềm thành phần cứng một cách hiệu quả. Công cụ này hỗ

trợ người dùng tạo ra các IP dưới dạng ngôn ngữ mô tả phần cứng (HDL - Hardware Description Language), từ đó tích hợp trực tiếp vào các hệ thống. Vivado HLS hỗ trợ chuyển đổi các thuật toán từ các ngôn ngữ lập trình cấp cao như C/C++ hoặc SystemC sang mức RTL. Công cụ này cho phép phần cứng được xây dựng trực tiếp từ thuật toán ban đầu được viết dưới dạng phần mềm, giúp đẩy nhanh quá trình phát triển và thử nghiệm, đồng thời cho phép kỹ sư nhanh chóng kiểm tra hiệu quả và tính khả thi của thiết kế phần mềm trên phần cứng thực tế. Quy trình này có thể được lặp lại nhiều lần để tối ưu hóa hệ thống và đảm bảo đáp ứng đầy đủ các yêu cầu thiết kế ban đầu. Bên cạnh đó, Vivado HLS cung cấp nhiều tính năng hữu ích như biên dịch và kiểm lỗi (debug) mã C/C++, mô phỏng dạng sóng, và hỗ trợ các thư viện phong phú phục vụ thiết kế phần cứng. Các thư viện bao gồm xử lý số dấu phẩy động (floating-point), số dấu chấm cố định (fixed-point), thư viện tính toán, và các công cụ tối ưu hóa như phân giải vòng lặp, thiết kế pipeline, tối ưu hóa luồng dữ liệu và bộ nhớ. Tất cả những tối ưu hóa này được thực hiện thông qua các chỉ dẫn `#pragma` đơn giản, giúp người dùng dễ dàng kiểm soát thiết kế phần cứng và đạt được hiệu suất tối ưu.

Tương tự, trình biên dịch Intel HLS (Intel HLS Complier) là một công cụ tổng hợp cấp cao do Intel phát triển, cho phép chuyển đổi mã nguồn cấp cao như C/C++ hoặc SystemC thành thiết kế RTL. Đặc biệt, Intel HLS nổi bật với khả năng hỗ trợ chuyển đổi trực tiếp các mô hình AI từ các nền tảng như TensorFlow hoặc PyTorch thành mã RTL, giúp triển khai hiệu quả các ứng dụng AI và xử lý thời gian thực lên FPGA. Công cụ này tích hợp chặt chẽ với Intel Quartus Prime, cung cấp các tính năng tương tự Vivado HLS như kiểm lỗi, mô phỏng dạng sóng và tối ưu hóa thiết kế phần cứng. Ngoài ra, trình biên dịch Intel HLS còn cung cấp các thư viện và công cụ tối ưu hóa đa dạng để khai thác tối đa hiệu suất phần cứng của các dòng FPGA Intel như Stratix, Arria và Cyclone.

**Bảng 6.1: So sánh các đặc trưng chính giữa hai bộ công cụ Vivado HLS và Intel HLS**

Tiêu chí	Vivado HLS (Vitis HLS)	Intel HLS
Nhà phát triển	Xilinx (nay là một phần của AMD)	Intel
Tên hiện tại	Vitis HLS	Intel® HLS Compiler
Mục tiêu thiết bị	FPGA dòng Xilinx (Zynq, Virtex, Artix, Kintex)	FPGA dòng Intel (Stratix, Arria, Cyclone)
Ngôn ngữ đầu vào	C/C++, SystemC	C/C++, SystemC
Khả năng tích hợp với mô hình AI	Có thể tích hợp mô hình AI qua các công cụ như Vitis AI	Hỗ trợ chuyển đổi trực tiếp mô hình AI từ TensorFlow, PyTorch thành RTL
Môi trường tích hợp	Tích hợp chặt chẽ với Vivado Design Suite và Vitis Unified Platform	Tích hợp chặt chẽ với Intel® Quartus® Prime
Thư viện hỗ trợ	Thư viện đa dạng như DSP, xử lý video, fixed-point, floating-point	Hỗ trợ các thư viện tính toán khoa học, fixed-point, floating-point
Tối ưu hóa thiết kế	Hỗ trợ phân giải vòng lặp (loop unrolling), pipeline, và tối ưu hóa bộ nhớ	Tương tự, với các khả năng tối ưu hóa như pipeline, loop unrolling
Công cụ mô phỏng và debug	Hỗ trợ mô phỏng dạng sóng và kiểm lỗi với Vivado Simulator	Hỗ trợ mô phỏng dạng sóng và kiểm lỗi tích hợp
Khả năng nền tảng	Hỗ trợ trên Windows và Linux	Hỗ trợ trên Windows và Linux
Ứng dụng chính	Tăng tốc các ứng dụng nhúng như DSP, xử lý hình ảnh, video, AI	Tăng tốc các ứng dụng DSP, HPC, AI, và nhúng

Mức độ sử dụng trong công nghiệp	Phổ biến trong các hệ thống nhúng, đặc biệt với SoC dòng Zynq	Phổ biến trong các ứng dụng HPC và AI yêu cầu hiệu năng cao
Hỗ trợ hệ sinh thái	Là một phần của hệ sinh thái Xilinx, kết hợp chặt chẽ với các công cụ Vitis AI và Vivado	Là một phần của hệ sinh thái Intel, tích hợp tốt với các công cụ OneAPI và OpenVINO

Cả hai công cụ, Vivado HLS và Intel HLS Compiler, đều giúp rút ngắn thời gian phát triển, hỗ trợ các kỹ sư phần mềm và phần cứng làm việc đồng bộ, và tối ưu hóa quá trình thiết kế hệ thống SoC. Nhìn vào bảng 6.1 ta có thể thấy được cả hai bộ công cụ gần như tương đồng các tính năng được tích hợp, tuy nhiên vẫn có một số đặc trưng nổi trội giữa hai bộ công cụ. Vivado HLS tập trung vào việc hỗ trợ các thiết bị FPGA của Xilinx, đặc biệt hữu ích trong các thiết kế SoC nhúng như dòng Zynq SoC, và có lợi thế với các công cụ tối ưu hóa chuyên biệt như Vitis AI. Intel HLS là một công cụ mạnh mẽ dành riêng cho các thiết bị FPGA của Intel, đặc biệt phù hợp cho các ứng dụng HPC (High-Performance Computing) và AI, nhờ tích hợp tốt với hệ sinh thái như OpenVINO và khả năng triển khai trực tiếp mô hình AI từ các nền tảng phổ biến. Tuy nhiên, lựa chọn công cụ nào phụ thuộc vào hệ sinh thái FPGA mà người dùng đang làm việc và yêu cầu cụ thể của ứng dụng.

### 6.2.1. Quy trình tạo IP từ công cụ HLS

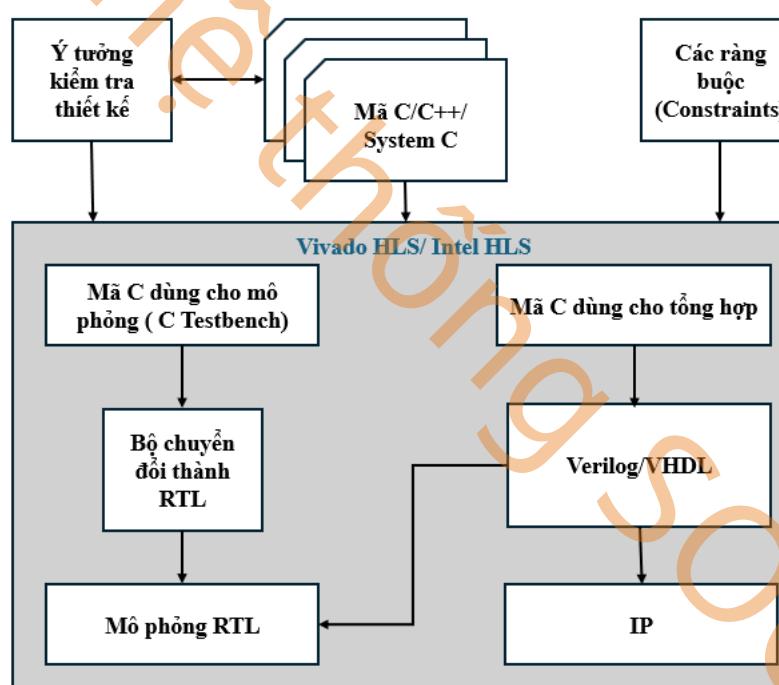
Quy trình để chuyển ngôn ngữ cấp cao sang RTL bằng Vivado HLS hoặc Intel HLS được nêu rõ ở hình 6.9. Đầu vào của một project trong quy trình tổng hợp cấp cao gồm:

- ✓ Hàm chính (top function) được viết bằng ngôn ngữ C, C++, hoặc SystemC: đây là đầu vào chính cho Vivado HLS hoặc Intel HLS. Hàm này có thể chứa một hệ thống phân cấp của các hàm con.
- ✓ Các ràng buộc (constraints): các ràng buộc là bắt buộc, bao gồm xung đồng hồ (clock), độ không đảm bảo của xung đồng hồ (uncertainly clock) và loại FPGA được sử dụng. Độ không đảm bảo của xung đồng hồ mặc định là 12,5% nếu không được chỉ định.

- ✓ Chỉ thị (directives): chỉ thị là tùy chọn và chỉ đạo quá trình tổng hợp để thực hiện một hành vi cụ thể hoặc một kỹ thuật tối ưu hóa cụ thể.
- ✓ Mã C dùng cho mô phỏng (C testbench) và những tệp liên quan: quy trình tổng hợp cấp cao sử dụng C testbench để mô phỏng hàm trước khi tổng hợp và kiểm thử đầu ra RTL nhờ vào quá trình mô phỏng C/RTL (C/RTL simulation).

Đầu ra của một project trong HLS gồm:

- ✓ Tệp thiết kế RTL dưới dạng ngôn ngữ mô tả phần cứng: đây là đầu ra chủ yếu của một project dưới dạng các ngôn ngữ VHDL (IEEE 1076-2000) hoặc Verilog (IEEE 1364-2001). Vivado HLS đóng gói các tệp triển khai dưới dạng khối IP để sử dụng với các IP khác trong quy trình thiết kế của Xilinx.
- ✓ Tệp báo cáo: báo cáo kết quả của quá trình tổng hợp, giả lập, đóng gói thành IP.



Hình 6.9: Quy trình tạo IP tự thiết kế với công cụ Vivado HLS/Intel HLS

### 6.2.2. Các lợi ích của phương pháp tổng hợp cấp cao

Tổng hợp cấp cao là cầu nối giữa phần cứng và phần mềm, mang lại các lợi ích chính sau:

- ✓ Cải thiện năng suất của các kỹ sư thiết kế phần cứng: Có thể làm việc ở mức trừu tượng cao hơn trong khi vẫn tạo ra được phần cứng có hiệu năng cao.

- ✓ Cải thiện năng suất hệ thống cho các kỹ sư thiết kế phần mềm: Các kỹ sư thiết kế phần mềm có thể tăng tốc độ tính toán chuyên sâu của thuật toán trên mục tiêu biên dịch mới là FPGA.
- ✓ Phát triển các thuật toán ở mức ngôn ngữ cấp trung như C: Làm việc ở mức độ trừu tượng thay vì quá trình thiết kế chi tiết tiêu tốn nhiều thời gian.
- ✓ Phát triển các thuật toán ở mức ngôn ngữ cấp trung như C: Làm việc ở mức độ trừu tượng thay vì quá trình thiết kế chi tiết tiêu tốn nhiều thời gian.
- ✓ Kiểm soát quá trình tổng hợp ngôn ngữ C thông qua các chỉ thị tối ưu (optimization directives): tạo ra các bản thiết kế phần cứng cụ thể có hiệu suất cao.
- ✓ Tạo ra nhiều thiết kế từ ngôn ngữ C nhờ vào các chỉ thị tối ưu: mở rộng không gian thiết kế, làm tăng khả năng tìm được các thiết kế tối ưu nhất.
- ✓ Tạo ra mã nguồn dễ đọc và tiện lợi: nhằm mục tiêu tái sử dụng vào các thiết bị hoặc dự án khác.

### 6..3. Các kỹ thuật tối ưu hóa thiết kế với HLS

#### 6.3.1. Phân giải vòng lặp

Phân giải vòng lặp ("Loop Unrolling" hay "Unrolling Loops") là một kỹ thuật tối ưu hóa mạch trong lĩnh vực tổng hợp cấp cao và lập trình phần mềm. Kỹ thuật này liên quan đến việc tối ưu hóa vòng lặp bằng cách sao chép phần thân của vòng lặp nhiều lần, giảm số lần lặp cần thiết và tăng tốc độ thực thi chương trình hoặc mạch. Ví dụ hãy xem xét một vòng lặp *for* trong ngôn ngữ lập trình cấp cao. Nó lặp một số lần lặp và có một số lệnh được thực thi bên trong thân vòng lặp mỗi lần. Bằng cách phân giải vòng lặp, số lần lặp sẽ giảm đi và chỉ thực hiện các lệnh trong thân vòng lặp. Mục đích chính của phân giải vòng lặp là tăng tốc độ thực thi bằng cách giảm điều kiện kiểm tra của vòng lặp và số lần nhảy vòng lặp, qua đó tối ưu hóa hiệu suất. Điều này đặc biệt hữu ích trong các thiết kế mạch nơi mà việc giảm số lần lặp có thể dẫn đến giảm độ trễ và tăng throughput.

*Chúng ta sẽ xem xét một tình huống đơn giản là cộng một mảng các số. Khi không sử dụng phân giải vòng lặp ta có đoạn code sau:*

```

int sum = 0;
int array[8] = {1, 2, 3, 4, 5, 6, 7, 8};
for (int i = 0; i < 8; i++) {
    sum += array[i];
}

```

Hình 6.10: Minh họa đoạn mã xử lý không sử dụng kỹ thuật phân giải vòng lặp

Đoạn mã C trong hình 6.10 sẽ thực hiện 8 lần lặp, mỗi lần cộng một phần tử vào biến sum. Áp dụng kỹ thuật phân giải vòng lặp, ta có thể giảm số lần lặp của vòng lặp bằng cách thực hiện nhiều phép cộng trong mỗi lần lặp:

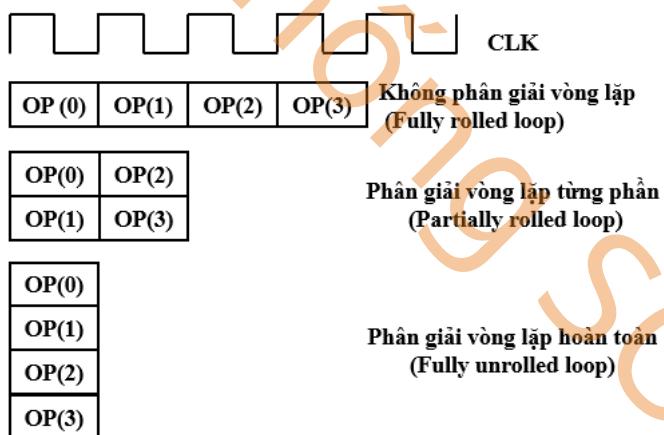
```

int sum = 0;
int array[8] = {1, 2, 3, 4, 5, 6, 7, 8};
// Mỗi lần lặp cộng 2 phần tử, giảm số lần lặp xuống còn một nửa
for (int i = 0; i < 8; i += 2) {
    sum += array[i] + array[i+1];
}

```

Hình 6.11: Minh họa đoạn mã xử lý với kỹ thuật phân giải vòng lặp từng phần

Đoạn mã C trong hình 6.11 chỉ cần chạy 4 lần thay vì 8 lần, mỗi lần cộng 2 phần tử vào tổng sum điều này giảm độ phức tạp của vòng lặp và có thể cải thiện hiệu suất, đặc biệt là trong các ứng dụng xử lý dữ liệu lớn hoặc yêu cầu thời gian thực.



Hình 6.12: Minh họa một số kỹ thuật phân giải vòng lặp.

Hình 6.12 cho thấy một ví dụ về kỹ thuật không phân giải vòng lặp (fully rolled loop), phân giải vòng lặp từng phần (partially rolled loop), và phân giải vòng lặp hoàn toàn (fully unrolled loop) qua thao tác thực hiện một nhiệm vụ lặp lại 4 lần. Trong vòng lặp không áp dụng kỹ thuật phân giải vòng lặp, công việc phải thực hiện bốn lần lặp lại, tương đương với bốn chu kỳ đồng hồ. Vì mỗi lần chỉ thực hiện một thao tác nên phần cứng cho hoạt động này chỉ cần thiết một lần. Trong kỹ thuật phân giải vòng lặp từng phần, lần lặp đầu

tiên OP(0) và OP(1) được thực thi cùng một thời điểm, tiếp theo lần lặp thứ hai OP(2) và OP(3) được thực hiện. Về mặc phần cứng sẽ yêu cầu gấp đôi phần cứng cần thiết cho hoạt động, nhưng nó chỉ mất hai chu kỳ đồng hồ. Trong trường hợp phân giải vòng lặp hoàn toàn, cả 4 thao tác đều được thực hiện cùng một lúc. Vòng lặp bây giờ chỉ mất một chu kỳ để hoàn thành, tuy nhiên nó cần bốn lần phần cứng.

### Lợi ích

- Tăng thông lượng: Cải thiện hiệu suất bằng cách thực hiện nhiều hơn một lần lặp trong mỗi chu kỳ.
- Giảm độ trễ: Giảm số lần kiểm tra điều kiện vòng lặp và số lần nhảy, giúp giảm độ trễ tổng thể.
- Hiệu suất tối ưu: Trong một số trường hợp, phân giải vòng lặp giúp tận dụng tốt hơn các nguồn lực phần cứng, như bộ nhớ cache hoặc đơn vị xử lý song song.

### Hạn Chế

- Tăng diện tích sử dụng: Trong thiết kế mạch, việc sử dụng nhiều sao chép của phần thân vòng lặp có thể dẫn đến tăng diện tích sử dụng.
- Quản lý phức tạp: Cần phải cân nhắc kỹ lưỡng mức độ unrolling phù hợp để tránh làm giảm hiệu quả do quá trình quản lý phức tạp hóa.
- Phụ thuộc vào kích thước vòng lặp: Hiệu quả của kỹ thuật này có thể phụ thuộc vào kích thước của vòng lặp và số lần lặp cố định.

#### 6.3.2. Phân chia mảng (Array Partitioning)

Kỹ thuật phân chia mảng (Array Partitioning) trong tổng hợp cấp cao là một phương pháp quan trọng giúp tối ưu hóa hiệu suất và sử dụng hiệu quả tài nguyên phần cứng bằng cách phân chia mảng dữ liệu lớn thành các phần nhỏ hơn. Mục tiêu chính của kỹ thuật này là tăng cường khả năng truy cập đồng thời dữ liệu, qua đó tăng thông lượng và giảm độ trễ trong xử lý. Khi một mảng được phân chia, việc xử lý dữ liệu có thể được thực hiện song song trên nhiều phần tử, thay vì tuần tự, từ đó tối ưu hóa thời gian thực thi tổng thể của chương trình. Phân chia mảng có thể được thực hiện theo nhiều cách khác nhau, bao gồm phân chia theo chiều dọc (phân chia mỗi mảng thành các cột), phân chia theo chiều ngang

(phân chia mảng thành các hàng), hoặc phân chia thành các khối nhỏ hơn. Hình 6.13 minh họa các loại phân chia mảng dữ liệu theo dạng khối, dạng đa chiều, và theo chu kỳ.

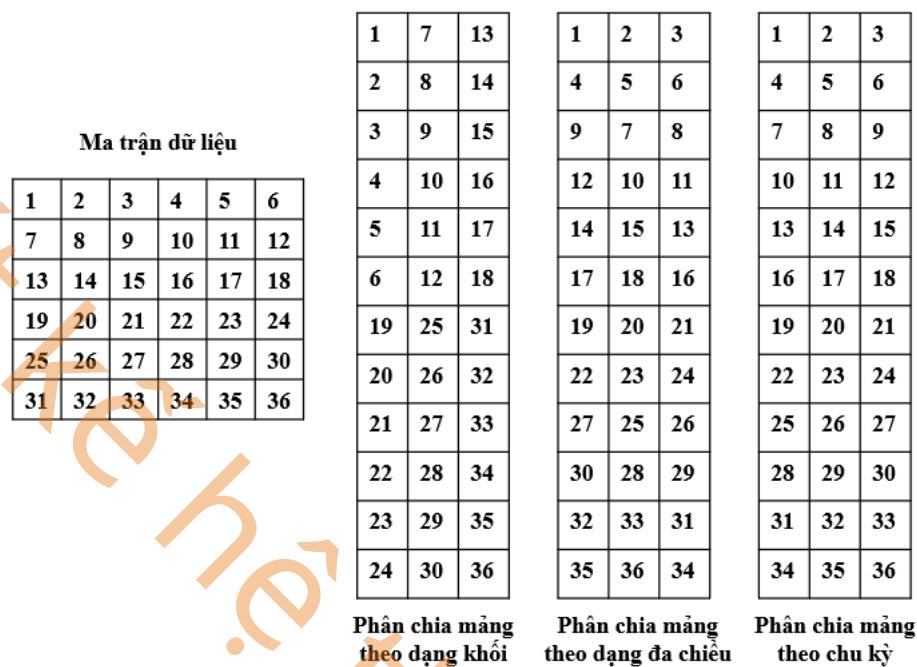
Phương pháp đầu tiên, **phân chia mảng theo dạng khối**, tổ chức dữ liệu bằng cách chia ma trận gốc thành các khối liền kề, cụ thể là từng cột của ma trận. Dữ liệu từ mỗi cột được trích xuất tuần tự từ trên xuống dưới rồi ghép nối lại thành một cột dọc lớn. Ví dụ, cột đầu tiên gồm các phần tử 1, 7, 13, 19, 25, 31, sau đó đến cột tiếp theo gồm 2, 8, 14, 20,... Cách phân chia này thường phù hợp khi ánh xạ từng khối vào một phần tử xử lý độc lập (processing element – PE) trong hệ thống phần cứng.

Phương pháp thứ hai, **phân chia theo dạng đa chiều**, chia ma trận thành các khối con 2D (gọi là tiles), giữ nguyên cấu trúc dữ liệu trong từng khối. Mỗi tile có kích thước cố định (ví dụ 3x3), sau đó các tile này được quét lần lượt theo hàng và được "dàn phẳng" để ghép lại thành chuỗi dữ liệu. Ví dụ, khối 3x3 đầu tiên gồm các phần tử từ 1 đến 15 được ghép lại theo hàng, sau đó đến khối tiếp theo chứa các phần tử từ 4 đến 18, v.v. Cách làm này giữ được tính liên kết dữ liệu trong không gian hai chiều và tối ưu hiệu năng truy xuất bộ nhớ đệm, rất hữu ích trong xử lý ảnh, video hoặc các phép toán ma trận cục bộ.

Phương pháp cuối cùng là **phân chia theo chu kỳ**, trong đó các phần tử được phân phối theo thứ tự tuần hoàn (round-robin) vào các nhóm hoặc PE. Cụ thể, phần tử đầu tiên sẽ thuộc về nhóm 1, phần tử thứ hai thuộc nhóm 2, phần tử thứ ba thuộc nhóm 3, sau đó quay lại nhóm 1 cho phần tử thứ tư, và cứ như vậy. Cách này tạo ra các cột dữ liệu mà mỗi cột bao gồm các phần tử cách đều nhau trong ma trận gốc. Ví dụ, nhóm 1 nhận các phần tử 1, 4, 7, 10,... nhóm 2 nhận 2, 5, 8, 11,... và nhóm 3 nhận 3, 6, 9, 12,... Phân chia theo chu kỳ rất phù hợp trong các kiến trúc xử lý song song như SIMD hoặc trong những hệ thống yêu cầu cân bằng tải giữa nhiều bộ xử lý.

Lựa chọn phương pháp phân chia phụ thuộc vào cấu trúc của dữ liệu và yêu cầu cụ thể của thuật toán đang được thực hiện. Ngoài ra, kỹ thuật này không chỉ cải thiện hiệu suất xử lý mà còn giúp tối ưu hóa sử dụng bộ nhớ, bởi lẽ nó cho phép việc sử dụng hiệu quả các khối bộ nhớ nhỏ hơn, giảm thiểu việc sử dụng bộ nhớ không cần thiết. Tuy nhiên, việc áp dụng phân chia mảng cũng đòi hỏi cân nhắc kỹ lưỡng về sự cân bằng giữa tăng thông lượng

và tăng sử dụng tài nguyên phần cứng. Một thiết kế tổng hợp cấp cao tốt cần phải tìm được điểm cân bằng này để đạt được hiệu suất cao nhất với chi phí tài nguyên phù hợp.



Hình 6.13: Minh họa các kỹ thuật phân chia mảng trong tổng hợp cấp cao

### 6.3.3. Pragma trong tổng hợp cấp cao

Trong tổng hợp cấp cao, *pragma* là một công cụ mạnh mẽ giúp các nhà thiết kế giao tiếp trực tiếp với bộ biên dịch tổng hợp cấp cao để hướng dẫn cách thức tổng hợp một đoạn mã cụ thể. *Pragma* không thay đổi logic của chương trình nhưng có thể ảnh hưởng đáng kể đến hiệu suất, diện tích, và công suất của thiết kế phần cứng kết quả. Chúng cho phép nhà thiết kế kiểm soát tối ưu hóa mà không cần thay đổi mã nguồn của thuật toán. Các *pragma* thường được sử dụng trong tổng hợp cấp cao bao gồm:

**#pragma HLS pipeline**: Tối ưu hóa vòng lặp bằng cách pipelining, giúp giảm độ trễ và tăng thông lượng bằng cách cho phép nhiều lần lặp của vòng lặp thực hiện đồng thời ở các giai đoạn khác nhau của việc xử lý.

**#pragma HLS unroll**: Giải nén vòng lặp để thực hiện song song, cải thiện tốc độ thực thi nhưng chi phí sử dụng tài nguyên phần cứng nhiều hơn.

**#pragma HLS array\_partition:** Chỉ định cách một mảng được phân chia thành các phần nhỏ hơn, cho phép truy cập đồng thời nhiều phần tử, điều này rất hữu ích trong các thiết kế yêu cầu xử lý song song cao.

**#pragma HLS allocate:** Kiểm soát việc phân bổ tài nguyên phần cứng cho các biến hoặc mảng, giúp tối ưu hóa việc sử dụng tài nguyên.

**#pragma HLS inline:** Chỉ định rằng một hàm nên được inline, tức là mã của hàm được chèn trực tiếp vào nơi gọi hàm, có thể giúp giảm độ trễ và tối ưu hóa hiệu suất.

```
#include <ap_int.h>
#define DATA_SIZE 256
#define KERNEL_SIZE 3
// Giả sử sử dụng fixed-point integer 8 bit cho dữ liệu và kernel
typedef ap_int<> data_t;
typedef ap_int<8> kernel_t;
// Hàm tính convolution
void convolution(data_t input[DATA_SIZE], kernel_t kernel[KERNEL_SIZE],
                 data_t output[DATA_SIZE]) {
    int i, j;
    // Áp dụng pragma HLS pipeline để tối ưu hóa việc thực hiện vòng lặp chính
    #pragma HLS PIPELINE
    for(i = 0; i < DATA_SIZE; i++) {
        data_t sum = 0;
        // Áp dụng pragma HLS unroll để tối ưu hóa việc tính toán convolution
        #pragma HLS UNROLL factor=2
        for(j = 0; j < KERNEL_SIZE; j++) {
            // Xử lý biên của mảng
            int idx = i + j - KERNEL_SIZE / 2;
            if (idx >= 0 && idx < DATA_SIZE) {
                sum += input[idx] * kernel[j];
            }
        }
        output[i] = sum;
    }
}
```

Hình 6.14: Minh họa đoạn mã xử lý có tích hợp các chỉ định pragma

Sử dụng *pragma* trong tổng hợp cấp cao đòi hỏi sự hiểu biết sâu sắc về cả thuật toán và kiến trúc phần cứng mục tiêu. Việc áp dụng *pragma* một cách cẩn thận và có chọn lọc có thể dẫn đến những cải thiện đáng kể về hiệu suất, diện tích, và tiêu thụ năng lượng của thiết kế cuối cùng. Dưới đây là một ví dụ về đoạn code C đơn giản để tính toán tích chập (convolution) sử dụng *#pragma HLS pipeline* và *# pragma HLS unroll* để tối ưu hóa. Tích chập là một phép toán quan trọng trong nhiều lĩnh vực như xử lý ảnh, xử lý tín hiệu số, và học máy, đặc biệt là trong các mạng nơ-ron tích chập.

Ở đoạn mã trong hình 6.14 chúng tôi sử dụng **#pragma HLS PIPELINE** để tối ưu hóa việc thực hiện vòng lặp ngoài, giúp giảm độ trễ bằng cách cho phép các lần lặp khác nhau của vòng lặp thực hiện đồng thời ở các giai đoạn khác nhau của việc xử lý. Pragma này nâng cao thông lượng của vòng lặp. **#pragma HLS UNROLL factor=2** được áp dụng cho vòng lặp nội bộ để tăng tốc độ thực thi bằng cách "giải cuốn" vòng lặp, cho phép thực hiện đồng thời nhiều lần lặp. Trong trường hợp này, chúng tôi giải cuốn vòng lặp với thừa số là 2, điều này có nghĩa là hai lần lặp của vòng lặp sẽ được thực hiện đồng thời.

#### 6.3.4 Kỹ thuật lượng tử hóa

Kỹ thuật lượng tử hóa (quantization) trong tổng hợp cấp cao là một phương pháp tối ưu hóa quan trọng, giúp giảm độ phức tạp tính toán và tài nguyên phần cứng trong các thiết kế số. Lượng tử hóa chuyển đổi các giá trị dấu phẩy động (floating-point) sang các giá trị số nguyên hoặc số thực sử dụng dấu chấm cố định (fixed-point) với độ chính xác thấp hơn, nhưng vẫn đảm bảo hiệu suất và tính chính xác cần thiết cho ứng dụng. Trong tổng hợp cấp cao, kỹ thuật này được áp dụng rộng rãi để giảm kích thước bộ nhớ, giảm tiêu thụ năng lượng và tăng tốc độ xử lý, đặc biệt trong các ứng dụng như AI, xử lý tín hiệu số, và các hệ thống nhúng trên FPGA.

Công cụ tổng hợp cấp cao thường cung cấp các thư viện hỗ trợ số dấu chấm cố định, cho phép kỹ sư kiểm soát độ dài bit, định dạng dữ liệu, và độ chính xác của các phép toán. Kỹ thuật này đặc biệt hiệu quả trong các mô hình AI, nơi các trọng số (weights) thường được lượng tử hóa để giảm khối lượng tính toán và yêu cầu lưu trữ mà không làm giảm đáng kể hiệu suất mô hình. Nhờ đó, lượng tử hóa trong tổng hợp cấp cao giúp thiết kế phần cứng vừa đạt được hiệu suất cao, vừa tiết kiệm tài nguyên và phù hợp với các ứng dụng có yêu cầu tối ưu hóa nghiêm ngặt.

Thư viện **ap\_fixed** trong công cụ tổng hợp cấp cao là một thư viện mạnh mẽ được sử dụng để hỗ trợ kỹ thuật lượng tử hóa, cho phép thiết kế phần cứng sử dụng dữ liệu dạng số dấu chấm cố định. Thư viện này cho phép tùy chỉnh độ chính xác từ đó người dùng có thể kiểm soát chính xác số lượng bit dành cho phần nguyên, phần thập phân và cả bit dấu.

Ví dụ, viết mã C hiện thực bài toán nhân hai ma trận với kiểu dữ liệu số dấu phẩy động (float) và kiểu dữ liệu số dấu chấm cố định (ap\_fixed<16,6>), với 6 bit dành cho phần thập phân và 10 bit dành cho phần nguyên.

```
#include <hls_stream.h>

void matrix_multiply(float A[4][4], float B[4][4], float C[4][4]) {
    #pragma HLS PIPELINE
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            float sum = 0.0;
            for (int k = 0; k < 4; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}
```

Hình 6.15: Mã C sử dụng dấu phẩy động để tính toán

```
#include <hls_stream.h>
#include <ap_fixed.h>

void matrix_multiply(ap_fixed<16,6> A[4][4], ap_fixed<16,6> B[4][4], ap_fixed<16,6> C[4][4]
# pragma HLS PIPELINE
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        ap_fixed<16,6> sum = 0;
        for (int k = 0; k < 4; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

Hình 6.16: Mã C sử dụng dấu chấm cố định để tính toán

Ví dụ minh họa cụ thể về kỹ thuật lượng tử hóa trong tổng hợp cấp cao thông qua bài toán nhân hai ma trận. Giả sử chúng ta cần thực hiện phép nhân hai ma trận 4x4 trong phần

cứng FPGA theo hướng tổng hợp cấp cao. Đầu tiên, thuật toán được viết bằng số dấu phẩy động như mô tả trong hình 6.15 cho các biến đầu vào và đầu ra khai báo với kiểu dữ liệu ‘float’, sau đó chúng ta áp dụng kỹ thuật lượng tử hóa để chuyển đổi sang số dấu chấm cố định với độ chính xác  $<16,6>$  như trong hình 6.16. Ở đây các biến đầu vào đầu ra sẽ được khai báo với kiểu dữ liệu ‘ap\_fixed  $<16,6>$ ’ với 6 bit dành cho phần thập phân và 10 bit dành cho phần nguyên nhằm tối ưu tài nguyên phần cứng.

**Bảng 6.2: Báo cáo tài nguyên phần cứng với dữ liệu đầu vào ở hai kiểu dữ liệu: dấu chấm cố định và dấu phẩy động.**

Tài nguyên	Thiết kế 1 (dấu chấm cố định)	Thiết kế 2 (dấu phẩy động)
BRAM	0	0
DSP	64	28
FF	1829	5326
LUT	425	3281
URAM	0	0
Độ trễ (s)	32	40

Kết quả báo cáo trong bảng 6.2 giữa hai module, sử dụng số dấu chấm cố định (fixed-point) và số thực dấu phẩy động (floating-point), cho thấy thiết kế 2 giảm mức sử dụng DSP đến **56.25%**, nhưng đồng thời làm tăng mạnh tài nguyên FF lên **191.20%** và LUT lên **672%**. Độ trễ (latency) của thiết kế 2 cũng tăng thêm **25%** so với thiết kế 1. Khi quy đổi tài nguyên DSP sang LUT và FF, có thể nhận thấy rõ ràng việc triển khai bài toán theo kiểu số dấu chấm cố định tiêu tốn ít tài nguyên phần cứng hơn,

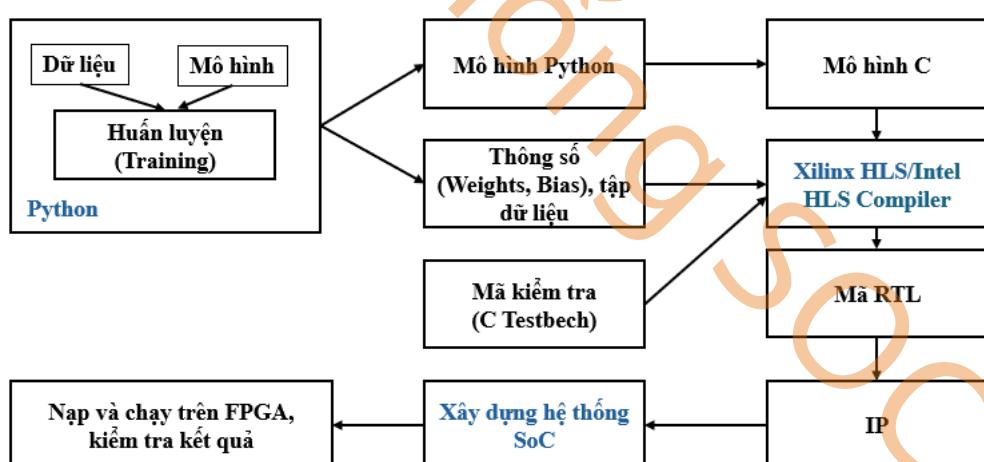
#### 6.4. Quy trình thiết kế SoC theo phương pháp tổng hợp cấp cao

Quy trình thiết kế một hệ thống SoC bằng phương pháp tổng hợp cấp cao được thực hiện theo các bước cụ thể và tuần tự để chuyển đổi thuật toán phần mềm thành thiết kế phần cứng hiệu quả. Đầu tiên, thuật toán được mô tả bằng các ngôn ngữ lập trình cấp cao như C/C++, SystemC hoặc Python. Sau đó, mã nguồn này được nhập vào các công cụ tổng hợp cấp cao như (Vivado HLS hoặc Intel HLS) để chuyển đổi thành ngôn ngữ mô tả phần cứng ở mức RTL nhằm tạo ra khôi IP tích hợp vào hệ thống SoC. Trong quá trình chuyển đổi các kỹ thuật như phân giải vòng lặp, thiết kế pipeline, và tối ưu hóa luồng dữ liệu cũng như bộ nhớ được áp dụng.

Khi mã RTL đã được tạo, bước tiếp theo là mô phỏng và xác minh để đảm bảo thiết kế phần cứng hoạt động đúng như thuật toán phần mềm ban đầu. Công cụ tổng hợp cấp cao cung cấp các tính năng hỗ trợ như kiểm lỗi (debug) và mô phỏng dạng sóng, giúp kỹ sư so sánh và đảm bảo tính chính xác của thiết kế. Tiếp theo, IP hoàn thiện được tích hợp vào hệ thống SoC thông qua các công cụ như Vivado Design Suite hoặc Intel Quartus Prime để hoàn thiện hệ thống và triển khai trên FPGA.

Quy trình này thường được lặp lại nhiều lần (iterative design flow) để tinh chỉnh hiệu suất và đảm bảo hệ thống đáp ứng đầy đủ các yêu cầu về hiệu năng, năng lượng và chi phí. Cách tiếp cận này không chỉ giúp rút ngắn thời gian phát triển mà còn tăng cường sự phối hợp giữa các kỹ sư phần mềm và phần cứng, từ đó tối ưu hóa toàn bộ quy trình thiết kế SoC.

Hình 6.17 minh họa quy trình thiết kế hoàn chỉnh của hệ thống SoC theo hướng tổng hợp cấp cao, áp dụng cho các ứng dụng sử dụng mô hình mạng trí tuệ nhân tạo viết bằng Python, đến bước triển khai thực tế trên FPGA. Quy trình thiết kế này được thực hiện thông qua các bước sau:



Hình 6.17: Quy trình thiết kế hệ thống từ HLS xuông SoC

- ✓ Bước 1: chuẩn bị dữ liệu gồm thu thập dữ liệu, xử lý dữ liệu ví dụ khử nhiễu, chuẩn hóa, loại bỏ các đột biến..., và xây dựng mô hình mạng tương ứng từ các nền tảng như Keras hoặc Pytorch sử dụng ngôn ngữ lập trình Python.
- ✓ Bước 2: huấn luyện mô hình đến khi mô hình hội tụ được các chuẩn đánh giá như mong muốn như độ chính xác (accuracy), chỉ số F1..., thì ta có được giá trị của các

thông số (parameters) gồm weights, bias. Đây là những trọng số quan trọng trong một mô hình trí tuệ nhân tạo. Các trọng số này cũng sẽ lưu ở dạng file .dat hoặc file .txt.

- ✓ Bước 3: chuyển mô hình được viết bằng mã python sang mã C ở bước mạng lan truyền tới (forwarding). Đưa tất cả các tệp mã C và tệp trọng số vào công cụ tổng hợp cấp cao để chuyển qua mã RTL. Bước chuyển từ mã C sang mã RTL mang đậm tính chất phần cứng nên cần chú trọng các yêu cầu về cấu trúc phần cứng IP như bộ nhớ, số bit của dữ liệu,... Do đó, ở giai đoạn này ta có thể tối ưu mã C sử dụng các phương pháp tối ưu như lượng tử hóa, phân giải vòng lặp, kỹ thuật pipeline, phân chia mảng...
- ✓ Bước 4: Mã RTL được tạo ra ban đầu chỉ có thể được sử dụng để kiểm tra chức năng của thiết kế trong phạm vi của một công cụ tổng hợp cấp cao cụ thể. Để mở rộng khả năng tương tác với các công cụ khác và tiến tới thiết kế chip mà không bị ràng buộc bởi bất kỳ công cụ nào, ta cần thiết lập một lớp giao tiếp tuân theo chuẩn bus phổ biến, chẳng hạn như AXI của Xilinx hoặc Avalon của Altera. Điều này cho phép dễ dàng tích hợp IP vừa thiết kế vào hệ thống SoC tương ứng.
- ✓ Bước 5: Sau khi hoàn tất việc kiểm tra chức năng bằng mã kiểm tra (testbench) và đóng gói IP với giao diện tuân theo chuẩn bus mong muốn, bước tiếp theo là tiến hành thiết kế hệ thống SoC và tích hợp IP vừa thiết kế vào hệ thống.
- ✓ Bước 6: kiểm tra kết quả sau khi thực hiện hệ thống trên FPGA. Ta có thể lấy kết quả được lưu ở dạng tệp .dat hoặc tệp .txt. sau đó so sánh trực tiếp với kết quả từ Python hoặc C, từ đó ta có thể đánh giá được mô hình đã hoàn thành yêu cầu đặt ra hay chưa.

## 6.5. Ví dụ thiết kế SoC bằng phương pháp tổng hợp cấp cao

Sau khi trình bày quy trình thiết kế SoC bằng phương pháp tổng hợp cấp cao và các phương pháp tối ưu hóa tích hợp trong tổng hợp cấp cao, phần này nhóm tác giả sẽ đưa ra các ví dụ minh họa cụ thể. Những ví dụ này nhằm giúp người đọc hiểu rõ hơn và tiếp cận thực tế với quy trình thiết kế SoC bằng phương pháp tổng hợp cấp cao.

### 6.5.1. Ví dụ tạo IP từ phương pháp tổng hợp cấp cao

Để giúp người đọc dễ dàng tiếp cận, nhóm tác giả bắt đầu với ví dụ về việc xây dựng một IP cho bài toán nhân hai ma trận. Trong ví dụ này, dữ liệu sẽ được xử lý với độ chính xác 16-bit (16,6) và áp dụng phương pháp tối ưu pipeline để cải thiện hiệu suất xử lý.

```
1. #include <hls_stream.h>
2. #include <ap_fixed.h>
3.
4. void matrix_multiply(ap_fixed<16,6> A[4][4], ap_fixed<16,6> B[4][4], ap_fixed<16,6> C[4][4]) {
5. #pragma HLS PIPELINE
6. #pragma HLS INTERFACE mode=ap_memory port=A storage_type=ram_1p
7. #pragma HLS INTERFACE mode=ap_memory port=B storage_type=ram_1p
8. #pragma HLS INTERFACE mode=ap_memory port=C storage_type=ram_1p
9.     for (int i = 0; i < 4; i++) {
10.         for (int j = 0; j < 4; j++) {
11.             ap_fixed<16,6> sum = 0;
12.             for (int k = 0; k < 4; k++) {
13.                 sum += A[i][k] * B[k][j];
14.             }
15.             C[i][j] = sum;
16.         }
17.     }
18. }
```

Hình 6.18: Minh họa đoạn mã nguồn để tạo IP nhân hai ma trận từ mã C++.

Đoạn mã nguồn C++ trong hình 6.18 minh họa cách mô tả bài toán nhân hai ma trận bằng ngôn ngữ cấp cao để sang ngôn ngữ mô tả phần cứng bằng công cụ tổng hợp cấp cao. Trong đoạn mã, các ngoặc vào được khai báo dưới dạng ma trận kích thước  $4 \times 4$ , sử dụng kiểu dữ liệu `ap_fixed<16, 6>` thông qua thư viện `<ap_fixed.h>`. Tối ưu hóa pipeline được thực hiện bằng cách áp dụng chỉ thị `#pragma HLS PIPELINE`, cho phép chồng lấp các giai đoạn thực thi trong vòng lặp, từ đó cải thiện hiệu suất xử lý và hỗ trợ tính toán song song. Đồng thời, các chỉ thị `#pragma HLS INTERFACE` được sử dụng để cấu hình các cổng giao tiếp cho các ma trận A, B, và C ở chế độ `ap_memory`, với kiểu lưu trữ `ram_1p` (RAM đơn cổng), nhằm tối ưu hóa khả năng truy xuất dữ liệu một cách hiệu quả.

Như mô tả trong hình 6.9, sau khi hoàn thành mã nguồn thiết kế, người dùng tiếp tục chuẩn bị một đoạn mã nguồn kiểm tra chức năng (testbench), cũng được viết bằng C/C++. Mã nguồn kiểm tra này có nhiệm vụ kiểm tra tính đúng đắn của mã nguồn thiết kế ban đầu,

đồng thời được sử dụng để kiểm tra lại chức năng của mã RTL sau khi quá trình biên dịch hoàn tất.

```
1 #include <iostream>
2 #include <hls_stream.h>
3 #include <ap_fixed.h>
4
5 void matrix_multiply(ap_fixed<16,6> A[4][4], ap_fixed<16,6> B[4][4], ap_fixed<16,6> C[4][4]);
6
7 int main() {
8     const int SIZE = 4;
9     ap_fixed<16,6> A[SIZE][SIZE] = {
10         {1.0, 2.0, 3.0, 4.0},
11         {5.0, 6.0, 7.0, 8.0},
12         {9.0, 10.0, 11.0, 12.0},
13         {13.0, 14.0, 15.0, 16.0}
14     };
15
16     ap_fixed<16,6> B[SIZE][SIZE] = {
17         {1.0, 0.0, 0.0, 0.0},
18         {0.0, 1.0, 0.0, 0.0},
19         {0.0, 0.0, 1.0, 0.0},
20         {0.0, 0.0, 0.0, 1.0}
21     };
22
23     ap_fixed<16,6> C[SIZE][SIZE] = {0};
24
25     matrix_multiply(A, B, C);
26
27     std::cout << "Result of matrix multiplication:" << std::endl;
28     for (int i = 0; i < SIZE; i++) {
29         for (int j = 0; j < SIZE; j++) {
30             std::cout << C[i][j].to_float() << " ";
31         }
32         std::cout << std::endl;
33     }
34
35     return 0;
36 }
```

Hình 6.19: Minh họa đoạn mã nguồn kiểm tra C++ dùng để kiểm tra chức năng bài toán nhân hai ma trận.

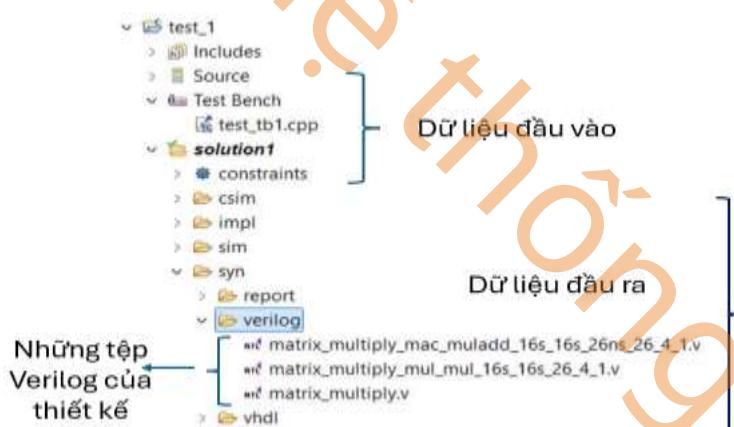
Để kiểm tra chức năng của bài toán nhân hai ma trận, đoạn mã trong hình 6.19 thực hiện việc khai báo dữ liệu ban đầu cho hai ma trận A và B. Sau đó, hàm *matrix\_multiply* (đã được định nghĩa trước đó trong phần mã nguồn) được gọi vào, tương tự như cách gọi một chương trình con trong ngôn ngữ lập trình C thông thường. Cuối cùng, kết quả được kiểm tra bằng cách in các phần tử của ma trận C sau khi tính toán để xác minh tính đúng đắn của phép nhân. Tiếp theo, tần số hoạt động ban đầu của mạch được thiết lập thông qua việc định nghĩa các ràng buộc, như được mô tả trong hình 6.20.

```
# This constraints file contains default clock frequencies to be used during out-of-context flows such as
# OOC Synthesis and Hierarchical Designs. For best results the frequencies should be modified
# to match the target frequencies.
# This constraints file is not used in normal top-down synthesis (the default flow of Vivado)
create_clock -name ap_clk -period 10.000 [get_ports ap_clk]
```

Hình 6.20: Tạo tần số hoạt động ban đầu cho thiết kế.

Sau khi các điều kiện đầu vào đã được chuẩn bị đầy đủ, người dùng sẽ sử dụng công cụ tổng hợp cấp cao để thực thi quá trình tổng hợp và tạo ra mã RTL cho bài toán. Các tệp mã RTL này sẽ được đóng gói thành IP tự thiết kế để tích hợp vào hệ thống SoC tương ứng.

Hình 6.21 mô tả cấu trúc thư mục của một dự án thiết kế sử dụng phương pháp tổng hợp cấp cao, bao gồm cả dữ liệu đầu vào và đầu ra trong quá trình tổng hợp. Các tệp mã nguồn thiết kế được lưu trữ trong thư mục "Source", trong khi các tập kiểm tra chức năng được đặt trong thư mục "Test Bench". Kết quả đầu ra từ quá trình tổng hợp được tổ chức trong các thư mục riêng biệt, bao gồm "csim" để lưu báo cáo mô phỏng từ mã nguồn C++, và "impl" cùng "sim" để lưu các báo cáo liên quan đến vấn đề mô phỏng mã nguồn RTL. Các tệp RTL được tạo ra, bao gồm mã Verilog hoặc VHDL, được đặt trong thư mục "verilog" hoặc "vhdl" tương ứng nằm trong thư mục "syn".



Hình 6.21: Tổ chức lưu trữ trong bộ công cụ Vitis HLS cho một dự án

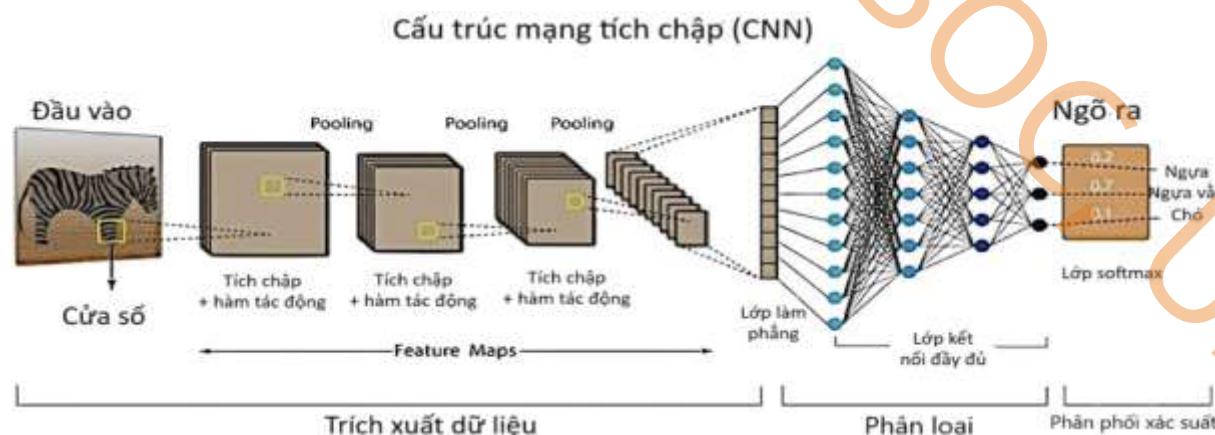
## 6.5.2. Ví dụ thiết kế một SoC sử dụng phương pháp tổng hợp cấp cao cho ứng dụng mạng trí tuệ nhân tạo

Như chúng ta biết học máy (machine learning) và học sâu (deep learning) đang có những bước phát triển vượt bậc và xuất hiện ngày càng phổ biến hơn, trở thành xu thế trong ngành công nghệ trên thế giới. Các mô hình học sâu, đặc biệt là mạng tích chập (CNN - Convolutional Neural Network), một loại mô hình học sâu rất mạnh trong các bài toán về thị giác máy tính, cần một lượng rất lớn tài nguyên tính toán cho việc vận hành, do đó các bộ xử lý đồ họa (GPU - Graphics Processing Unit) được phát triển và áp dụng cho các nền tảng học sâu nhằm tăng tốc độ xử lý nhờ kỹ thuật xử lý song song và đa luồng. FPGA cũng đang được hướng đến để sử dụng như một nền

tăng phần cứng cho việc tính toán của các mô hình học sâu. FPGA đang cho thấy một sự cải thiện lớn về năng lượng tiêu thụ và hiệu năng trong các ứng dụng của các mạng học sâu yêu cầu độ chính xác cao như các bài toán phân lớp. Hơn thế nữa, FPGA cho phép triển khai các mô hình học máy và học sâu lên những hệ thống nhúng chiếm diện tích nhỏ mà vẫn đảm bảo về tốc độ xử lý và độ chính xác nhờ vào việc xử lý song song trên phần cứng, cùng với đó là việc dễ dàng thay đổi và nâng cấp nhờ vào đặc tính khả trinh của FPGA. Do đó FPGA đang được áp dụng rộng rãi hơn trong lĩnh vực học máy nói chung và trong các mạng học sâu nói riêng. Phần này sẽ giúp người đọc hiểu rõ quy trình thực hiện ứng dụng nhận dạng ký tự số từ ngôn ngữ cấp cao xuống hệ thống SoC trên FPGA.

## Mạng tích chập

Mạng nơ-ron tích chập, là một trong những kiến trúc mạng học sâu cực kỳ tiên tiến trong việc giúp người dùng cho tạo được các ứng dụng có độ chính xác cao trên nguồn dữ liệu đa dạng và phức tạp. Từ đặc trưng vượt trội đó, mạng tích chập được ứng dụng trong rất nhiều lĩnh vực khác nhau, chẳng hạn như bài toán nhận dạng và phân loại đối tượng. Sự ra đời của mạng tích chập bắt nguồn từ ý tưởng cải tiến cách thức hoạt động của các mạng nơ-ron nhân tạo truyền thống nhằm giảm bớt sự nặng nề nhưng nâng cao hiệu quả. Vì việc sử dụng các liên kết đầy đủ giữa các dữ liệu đầu vào, các mạng nơ-ron nhân tạo truyền thống (ANN - Artificial Neural Network) bị có nhược điểm rất lớn bởi khi kích thước của dữ liệu đầu vào càng lớn thì số lượng liên kết càng nhiều kết quả là cần một khối lượng tính toán lớn. Mạng tích chập với kiến trúc thay đổi, có khả năng trích xuất đặc trưng mà chỉ sử dụng một phần cục bộ trong dữ liệu để tạo kết nối đến nút trong lớp tiếp theo thay vì toàn bộ dữ liệu như trong mạng nơ-ron nhân tạo.



Hình 6.22: Cấu trúc một mô hình mạng tích chập cơ bản

Cấu trúc cơ bản của một mạng nơ-ron tích chập được mô tả trong hình 6.22. Mạng mạng tích chập được hình thành bởi 2 thành phần là **chiết xuất đặc trưng** và **phân loại**. Thành phần **chiết xuất đặc trưng** gồm có lớp tích chập (Convolutional layer) và lớp gộp (Pooling layer) và thành phần **phân loại** gồm lớp kết nối đầy đủ (fully Connected) được thực hiện như mạng nơ-ron thông thường.

### Lớp tích chập:

Lớp tích chập (Convolution Layer) được dùng để phát hiện và trích xuất đặc trưng hay chi tiết của ảnh hoặc tín hiệu. Lớp tích chập nhận dữ liệu đầu vào, thực hiện các phép chuyển đổi để tạo ra dữ liệu đầu ra cho lớp tiếp theo. Phép biến đổi được sử dụng là phép tích chập hai chiều hay một chiều. Mỗi lớp tích chập chứa một hoặc nhiều bộ lọc - bộ phát hiện đặc trưng (filter - feature detector hay kernel) cho phép phát hiện và trích xuất những đặc trưng khác nhau của ảnh hay dữ liệu. Mức đa dạng của đặc trưng phụ thuộc vào số lượng và đặc tính của bộ lọc. Trong mạng tích chập, những lớp tích chập đầu tiên thường được huyền huận với các bộ lọc hình học (geometric filters) nhằm lấy được những đặc trưng đơn giản như cạnh ngang, dọc, chéo của bức ảnh hoặc tín hiệu. Những lớp tích chập tiếp theo được thiết lập dùng để phát hiện các đặc trưng chi tiết của đối tượng trong ảnh như các bộ phận trên cơ thể người hoặc động vật, các chi tiết của máy móc.

Dữ liệu đầu vào của mạng tích chập, hay còn gọi là các đặc trưng (kênh), được biểu diễn dưới dạng ma trận nhiều chiều với các chiều lần lượt là chiều dài, chiều rộng và số kênh. Ví dụ, ảnh xám thường có một kênh, trong khi ảnh màu RGB có 3 kênh. Phép tích chập hoạt động bằng cách thực hiện phép nhân từng phần tử giữa bộ lọc và vùng dữ liệu tương ứng trên đặc trưng đầu vào, sau đó tính tổng tất cả các giá trị này. Đầu ra của lớp tích chập là một ma trận, trong đó mỗi giá trị được tạo ra bằng cách trượt bộ lọc trên toàn bộ đặc trưng đầu vào theo thứ tự từ trái sang phải, từ trên xuống dưới. Khi tích chập trên các ma trận có nhiều kênh, giá trị của mỗi điểm ảnh đầu ra được tính bằng tổng của các phép tích chập giữa từng kênh của ma trận đầu vào và kênh tương ứng của bộ lọc. Số kênh của bộ lọc phải bằng số kênh của ma trận đầu vào, và số lượng bộ lọc sẽ quyết định số kênh của ma trận đầu ra. Bộ lọc thường có dạng ma trận vuông với kích thước là một số lẻ (ví

dụ:  $3 \times 3$  hoặc  $5 \times 5$ ). Các giá trị trong bộ lọc ban đầu được khởi tạo ngẫu nhiên và sẽ được điều chỉnh trong quá trình học của mạng. Sau khi thực hiện phép tích chập, mỗi điểm ảnh đầu ra còn được cộng thêm một giá trị gọi là hệ số điều chỉnh (bias). Hệ số điều chỉnh này giúp tăng độ chính xác của mô hình và giảm nguy cơ quá khớp với dữ liệu. Tương tự như bộ lọc, các hệ số điều chỉnh cũng được khởi tạo và học trong quá trình huấn luyện. Các giá trị của bộ lọc và hệ số điều chỉnh được gọi chung là tham số (parameters) của mạng tích chập.

Kích thước của ma trận đầu ra của lớp tích chập được xác định theo công thức:

$$w_0 * h_0 = \left( \frac{w_i + 2p - k}{s} + 1 \right) * \left( \frac{h_i + 2p - k}{s} + 1 \right) \quad (6.1)$$

Trong đó

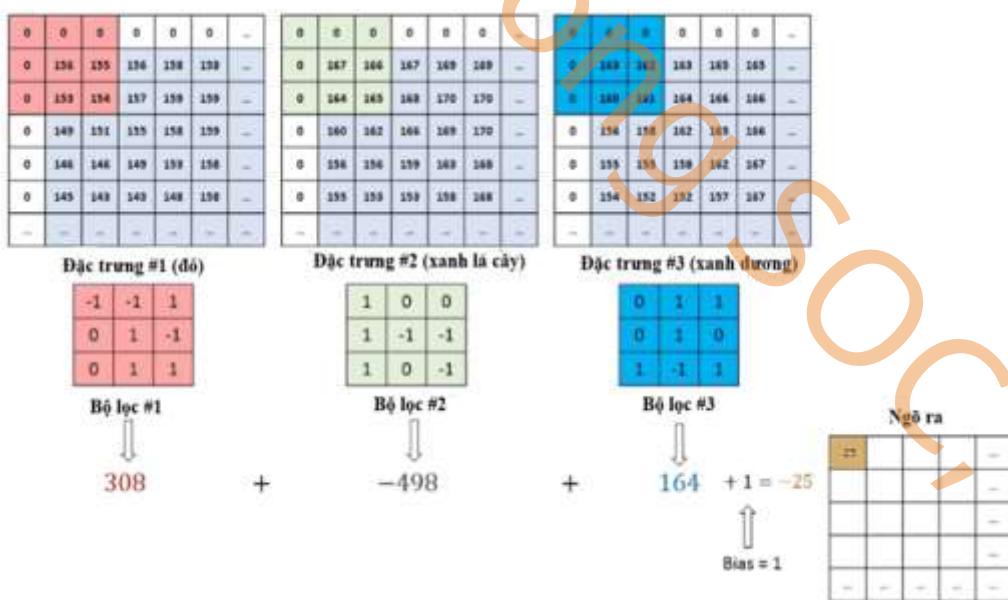
$w_0$  và  $h_0$  là chiều rộng và chiều cao của ma trận đầu ra.

$w_i$  và  $h_i$  là chiều rộng và chiều cao của ma trận đầu vào.

$p$  là kích thước đệm.

$s$  là khoảng cách giữa mỗi lần trượt của bộ lọc.

$k$  là kích thước của bộ lọc.



Hình 6.23: Ví dụ về cách tính nhân chập của lớp tích chập

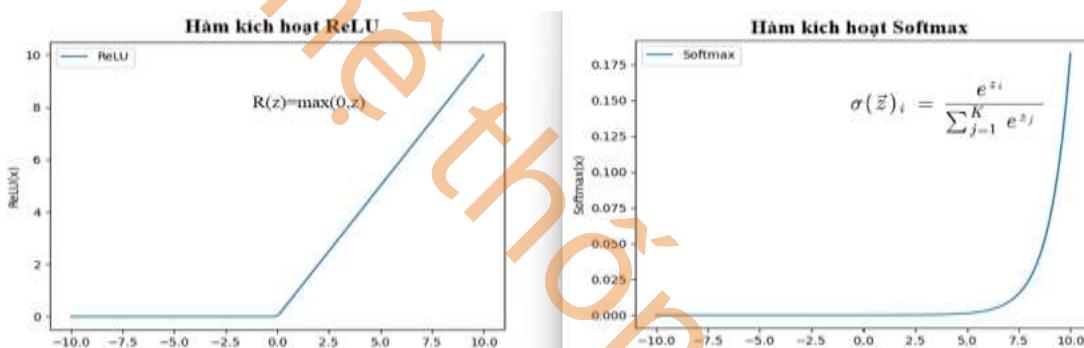
Hình trên 6.23 minh họa quá trình tính tích chập trên một đặc trưng đầu vào có ba kênh (màu đỏ, xanh lá cây, xanh dương) với ba bộ lọc tương ứng. Mỗi kênh đầu vào được

nhân từng phần tử với bộ lọc tương ứng, sau đó tính tổng giá trị, tạo ra các giá trị đầu ra như 308, -498, và 164. Sau khi cộng các giá trị này lại, kết quả là -25 sau khi thêm hệ số điều chỉnh (bias) là 1. Quá trình này được lặp lại cho toàn bộ ảnh để tạo thành đặc trưng đầu ra.

### Đệm:

Đệm (Zero-padding) là kỹ thuật thêm các giá trị 0 vào dữ liệu đầu vào của lớp tích chập. Nếu dữ liệu đầu vào là một vector, các giá trị 0 được thêm vào cuối vector. Nếu dữ liệu là một ma trận (ví dụ: hình ảnh), các giá trị 0 sẽ được thêm xung quanh viền của ma trận. Kỹ thuật này giúp mô hình dễ dàng trích xuất đặc trưng từ các vùng biên hoặc góc của dữ liệu, đồng thời đảm bảo kích thước đầu ra giữ nguyên so với kích thước đầu vào.

### Hàm kích hoạt:



Hình 6.24: Công thức và đồ thị một số hàm kích hoạt phổ biến

Đầu ra của các lớp tích chập và các lớp kết nối đầy đủ là các hàm tuyến tính. Vì hợp của các hàm tuyến tính vẫn chỉ là một hàm tuyến tính, các lớp này có thể bị rút gọn thành một lớp duy nhất, khiến mô hình không thể học được các quan hệ phức tạp trong dữ liệu. Để khắc phục vấn đề này, các hàm kích hoạt phi tuyến (non-linear activation functions) được sử dụng. Những hàm này được áp dụng lên từng điểm giá trị đầu ra của lớp tích chập và từng nút trong lớp kết nối đầy đủ, tạo ra tính phi tuyến cho mô hình. Hình 6.24 minh họa vùng hoạt động cụ thể của một số hàm kích hoạt phổ biến trong mạng học sâu, tiêu biểu là hàm ReLU và Softmax. Hàm Relu hoạt động bằng cách đặt tất cả các giá trị âm về 0 và giữ nguyên các giá trị dương. Hàm Softmax chuẩn hóa đầu ra thành xác suất, với tổng các xác suất bằng một.

Trong bài toán phân lớp, bao gồm cả bài toán phân lớp sử dụng mạng tích chập, mục tiêu là dự đoán xác suất mà một dữ liệu đầu vào  $x$  thuộc về một lớp cụ thể. Để đảm bảo tính chính xác, các giá trị xác suất dự đoán  $a_i$  cho từng lớp cần thỏa mãn các điều kiện: phải là số dương ( $a_i > 0$ ), và tổng tất cả các xác suất phải bằng 1. Giá trị đầu ra  $z_i$  của mô hình càng lớn thì xác suất  $a_i$  tương ứng với lớp  $i$  càng cao. Để đáp ứng những yêu cầu này, hàm **softmax** được sử dụng, đây là một hàm đồng biến có khả năng chuyển đổi các giá trị  $z_i$  thành các xác suất  $a_i$  vừa dương, vừa có tổng bằng 1, đồng thời duy trì thứ tự giữa các  $z_i$ . Hàm softmax được áp dụng trong các bài toán phân lớp với nhiều hơn hai lớp, trong khi với bài toán phân lớp hai lớp, hàm sigmoid thường được sử dụng ở lớp cuối cùng để dự đoán xác suất. Công thức của hàm softmax như sau:

$$a_i = \frac{\exp(z_i)}{\sum_{j=1}^C \exp(z_j)}, \forall i = 1, 2, 3, \dots, C, \quad (6.2)$$

với  $C$  là số lớp cần phân loại

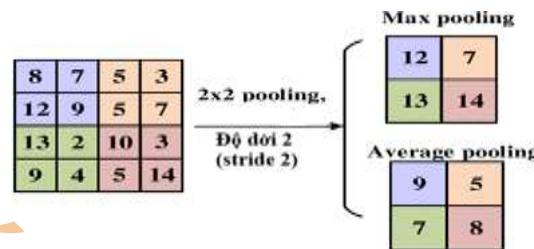
### Lớp loại bỏ:

Lớp loại bỏ (dropout layer) là một kỹ thuật đơn giản nhằm giảm bớt các nút mạng dư thừa trong quá trình đào tạo bằng cách loại bỏ ngẫu nhiên một số nút. Khi một nút bị loại bỏ, nó sẽ không tham gia vào quá trình tính toán trong giai đoạn lan truyền tiến (forwarding) và lan truyền ngược (backwarding) của mạng. Xác suất giữ lại một nút trong mỗi bước huấn luyện được gọi là  $p$ , do đó xác suất để nút đó bị loại bỏ là  $1-p$ . Lớp loại bỏ thường được thêm vào khi một lớp tích chập hoặc lớp kết nối đầy đủ có quá nhiều kết nối dư thừa, gây ảnh hưởng đến quá trình trích xuất đặc trưng và phân loại. Sử dụng lớp này giúp giảm số lượng tham số và ngăn các nút mạng trong lớp quá phụ thuộc lẫn nhau, từ đó giảm thiểu hiện tượng mô hình học quá mức vào tập dữ liệu huấn luyện (overfitting).

### Lớp gộp:

Lớp gộp (Pooling layer) thường được tích hợp giữa các lớp tích chập để giảm kích thước dữ liệu trong khi vẫn giữ được các thuộc tính quan trọng. Việc giảm kích thước này giúp giảm số lượng tham số và phép tính trong các lớp tiếp theo, đồng thời tăng hiệu quả tính toán của mô hình. Lớp gộp sử dụng một cửa sổ trượt, tương tự như lớp tích chập, để quét qua toàn bộ các vùng trong ảnh. Tuy nhiên, thay vì thực hiện phép tích chập, nó thực

hiện phép lấy mẫu, chọn ra một giá trị đại diện duy nhất cho toàn bộ các giá trị trong cửa sổ trượt. Cửa sổ thường được sử dụng phổ biến nhất có kích thước  $2 \times 2$  với khoảng cách trượt là 2. Hình 6.25 minh họa kết quả khi áp dụng hai phương pháp gộp phổ biến: max pooling, lấy giá trị lớn nhất trong cửa sổ, và average pooling, lấy giá trị trung bình của toàn bộ các giá trị trong cửa sổ, trên một đặc trưng có kích thước  $4 \times 4$ .



Hình 6.25: Ví dụ về max pooling và average pooling

### Làm phẳng:

Sau quá trình trích xuất đặc trưng của dữ liệu từ lớp tích chập và lớp gộp, ta bắt đầu phân loại dữ liệu thành các đầu ra mong muốn bằng cách sử dụng nhiều lớp kết nối đầy đủ. Trước tiên cần làm phẳng (flatten) dữ liệu với công thức:

$$y_{hs} = y_{j*a*c+k*c+i} = X_{i,j,k} \quad (6.3)$$

Trong đó:

$$c > i \geq 0 \forall i \in N, a > k \geq 0 \forall k \in N, b > j \geq 0 \forall j \in N$$

### Lớp kết nối đầy đủ:

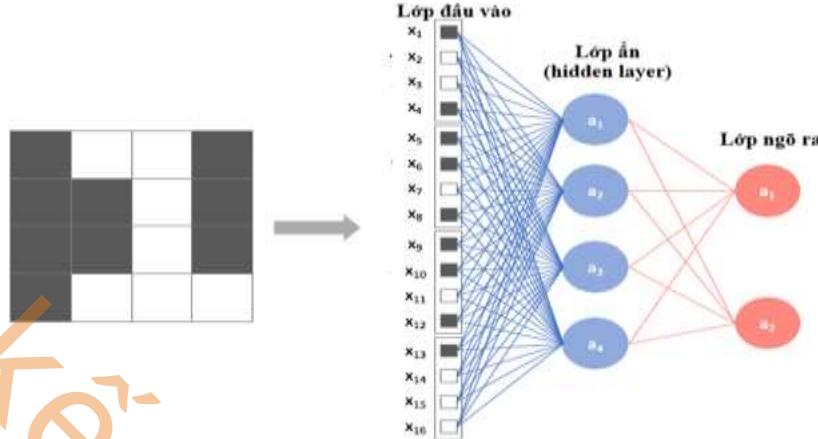
Hình 6.26 minh họa cách hoạt động của lớp kết nối đầy đủ (fully connected layer). Trong lớp này, các giá trị đặc trưng đầu ra từ các lớp trước, có thể ở dạng 1 chiều hoặc 2 chiều, sẽ được chuyển đổi thành đầu vào cho lớp kết nối đầy đủ. Lớp này kết nối tất cả các nút từ lớp trước với tất cả các nút trong lớp hiện tại, nhằm tạo ra mối quan hệ phi tuyến và học các đặc trưng toàn cục. Quá trình tính toán trong lớp kết nối đầy đủ được thực hiện thông qua công thức toán học:

$$y_k = Activation(\sum x_i w_{k,i} + b_k) \quad (6.4)$$

Trong đó:

- $x$  là dữ liệu đầu vào
- $y$  là dữ liệu đầu ra

- $w$  là trọng số weight
- $b$  là tập bias



Hình 6.26: Cấu trúc lớp làm phẳng và kết nối dày đặc

### 6.5.3. Áp dụng quy trình thiết kế SoC vào ứng dụng nhận dạng chữ số bằng AI



Hình 6.27: Minh họa hình ảnh cụ thể trong tập dữ liệu MNIST

Phần này minh họa chi tiết từng bước tích hợp IP nhận dạng chữ số bằng mạng neural, được tạo ra thông qua phương pháp tổng hợp cấp cao, vào hệ thống SoC. Dữ liệu sử dụng là cơ sở dữ liệu **MNIST** (Modified National Institute of Standards and Technology Database), một tập dữ liệu lớn chứa các chữ số viết tay, thường được dùng để huấn luyện các hệ thống xử lý hình ảnh. Tập dữ liệu bao gồm 60.000 mẫu, mỗi mẫu là một hình ảnh đơn sắc với kích thước  $28 \times 28$  điểm ảnh. Hình 6.27 minh họa một số hình ảnh tiêu biểu

trong tập dữ liệu MNIST. Mạng Lenet5 sẽ được sử dụng để huấn luyện trên tập dữ liệu này bằng ngôn ngữ Python. Cấu trúc chi tiết của mạng Lenet5 được trình bày trong bảng 6.3.

**Bảng 6.3: Mô tả chi tiết cấu trúc mạng Lenet 5**

Lớp	Đầu vào	Đầu ra	Bộ lọc	Tham số	Đệm/trượt	Tổng tham số
Lớp tích chập Kích hoạt (ReLU)	1x28x2 8	6x28x28	-	-	1-1	60
Lớp gộp (Max Pooling)	6x28x2 8	6x14x14	3x3	-	0-2	0
Lớp tích chập Kích hoạt (ReLU)	6x14x1 4	16x10x10	-	-	0-1	2416
Lớp gộp (Max Pooling)	16x10x 10	16x5x5	3x3	-	0-2	0
Làm phẳng	16x5x5	400	-	-	-	0
Lớp kết nối đầy đủ Kích hoạt (ReLU)	400	128	-	400x128	-	47120
Lớp kết nối đầy đủ Kích hoạt (ReLU)	128	84	-	128x84	-	10164
Lớp kết nối đầy đủ Kích hoạt (Softmax)	84	10	-	84x10	-	850

Tổng tham số	61,610
--------------	--------

Bảng 6.3 mô tả chi tiết cấu trúc mạng Lenet5, bao gồm các lớp tích chập, lớp gộp, lớp làm phẳng, và các lớp kết nối đầy đủ. Mạng bắt đầu với đầu vào có kích thước  $28 \times 28 \times 1$  và trải qua các lớp tích chập với số lượng bộ lọc, kích thước bộ lọc, và hàm kích hoạt được chỉ định, như ReLU ở các lớp tích chập và kết nối đầy đủ, hoặc Softmax ở lớp cuối cùng. Sau các lớp gộp và làm phẳng, đầu ra của mạng giảm dần từ 400 xuống 84 và cuối cùng là 10 lớp (ứng với 10 chữ số). Tổng số tham số trong mạng là 61.610, thể hiện mức độ phức tạp phù hợp để xử lý dữ liệu từ tập MNIST.

Tập dữ liệu và mô hình sau khi được xác định rõ ràng sẽ được sử dụng để huấn luyện mạng, nhằm đạt được hiệu suất và tiêu chuẩn như mong muốn của người dùng. Trong quá trình này, các tham số của mạng (như trọng số và bias) sẽ được tối ưu hóa thông qua quá trình học. Sau khi hoàn tất huấn luyện, các tham số đã học được kết hợp với kiến trúc mạng, tạo thành mô hình hoàn chỉnh để kiểm tra trên các mẫu thử mới không thuộc tập dữ liệu huấn luyện ban đầu. Đoạn code trong hình 6.28 minh họa việc lập trình kiến trúc mạng Lenet5 bằng ngôn ngữ Python, sử dụng nền tảng Pytorch, thể hiện cách triển khai và thiết lập các lớp của mạng.

```

class LeNet5(nn.Module):
    def __init__(self, *args, **kwargs):
        super(LeNet5, self).__init__()
        self.cnn1 = nn.Conv2d(1, 6, (5, 5), 1) # 1 input channel, 6 output channels, 5x5 kernel
        self.pool = nn.AvgPool2d(kernel_size=(2, 2), stride=2) # Average pooling
        self.cnn2 = nn.Conv2d(6, 16, (5, 5), 1) # 6 input channels, 16 output channels, 5x5 kernel
        self.cnn3 = nn.Conv2d(16, 120, (5, 5), 1) # 16 input channels, 120 output channels, 5x5 kernel
        self.fc1 = nn.Linear(120, 84) # Fully connected layer 120 -> 84
        self.fc2 = nn.Linear(84, 10) # Fully connected layer 84 -> 10

    def forward(self, x):
        # Pad the image to maintain same dimensions
        x = F.pad(x, (2, 2, 2, 2)) # Pad 2 pixels on all sides
        x = F.relu(self.cnn1(x)) # Apply ReLU activation after first convolution
        x = self.pool(x) # Apply pooling
        x = F.relu(self.cnn2(x)) # Apply ReLU activation after second convolution
        x = self.pool(x) # Apply pooling
        x = F.relu(self.cnn3(x)) # Apply ReLU activation after third convolution
        x = x.view(-1, 120) # Flatten the layer to a vector
        x = F.relu(self.fc1(x)) # Apply ReLU activation after first fully connected layer
        x = F.softmax(self.fc2(x), dim=1) # Apply softmax at the output layer
        return x

```

Hình 6.28: Minh họa đoạn mã Python của mô hình mạng Lenet5 được tạo bằng nền tảng Keras

Đoạn code trong hình 6.28 minh họa cách xây dựng mạng LeNet5 bằng thư viện PyTorch. Mạng bao gồm các lớp tích chập (nn.Conv2d), lớp gộp trung bình (nn.AvgPool2d), các lớp kết nối đầy đủ (nn.Linear), và hàm kích hoạt (Relu).

```

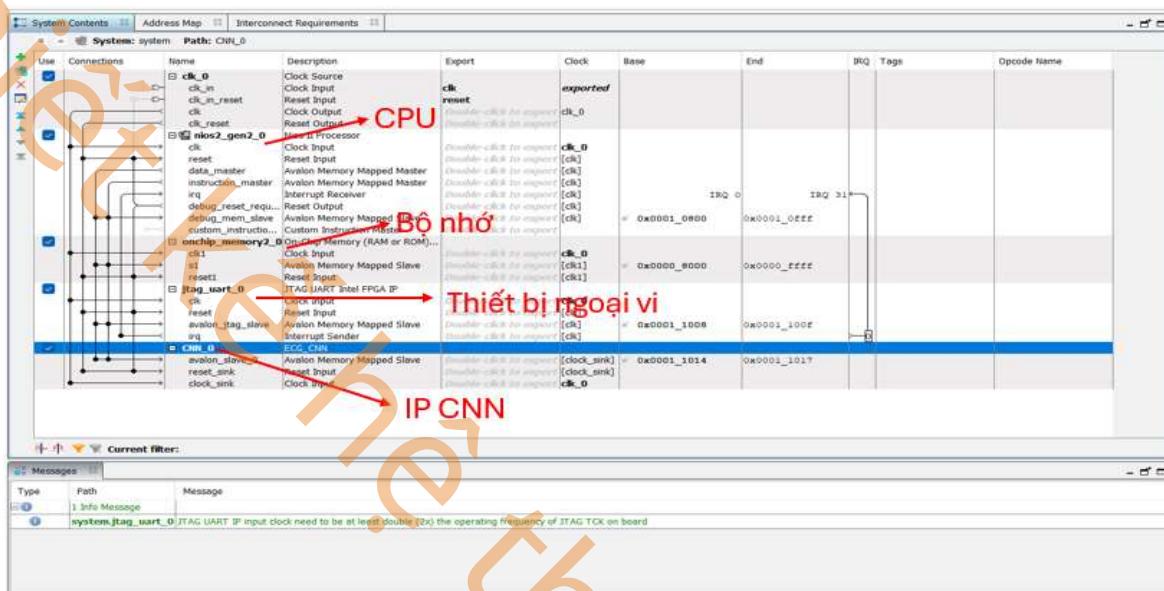
void convolution(float in[28][28], float kernel[6][5][5], float out[6][24][24])
{
    int i,j,k,l,c;
    for(c = 0; c < 6; c++) {
        for(i = 0; i < 24; i++) {
            for(j = 0; j < 24; j++) {
                for(k = 0; k < 5; k++) {
                    for(l = 0; l < 5; l++) {
                        if (k==0 && l==0) out[c][i][j] = in[i+k][j+l] * kernel[c][k][l];
                        else out[c][i][j] += in[i+k][j+l] * kernel[c][k][l];
                    } } } } }
void relu_bias(float in[6][24][24], float bias[6])
{
    int i,j,c;
    for(c = 0; c < 6; c++) {
        for(i = 0; i < 24; i++) {
            for(j = 0; j < 24; j++) {
                in[c][i][j] += bias[c];
                in[c][i][j] = (in[c][i][j] < 0.0f) ? 0.0f : in[c][i][j];
            } } } }
void maxpooling(float in[6][24][24], float out[6][12][12])
{
    int i,j,c;
    for(c = 0; c < 6; c++) {
        for(i = 0; i < 12; i++) {
            for(j = 0; j < 12; j++) {
                out[c][i][j] = in[c][i*2][j*2];
                if (in[c][i*2][j*2+1] > out[c][i][j]) out[c][i][j] = in[c][i*2][j*2+1];
                if (in[c][i*2+1][j*2] > out[c][i][j]) out[c][i][j] = in[c][i*2+1][j*2];
                if (in[c][i*2+1][j*2+1] > out[c][i][j]) out[c][i][j] = in[c][i*2+1][j*2+1];
            } } } }
```

Hình 6.29: Minh họa đoạn mã C dùng để tính toán lớp tích chập, lớp kích hoạt (relu) và lớp maxpooling.

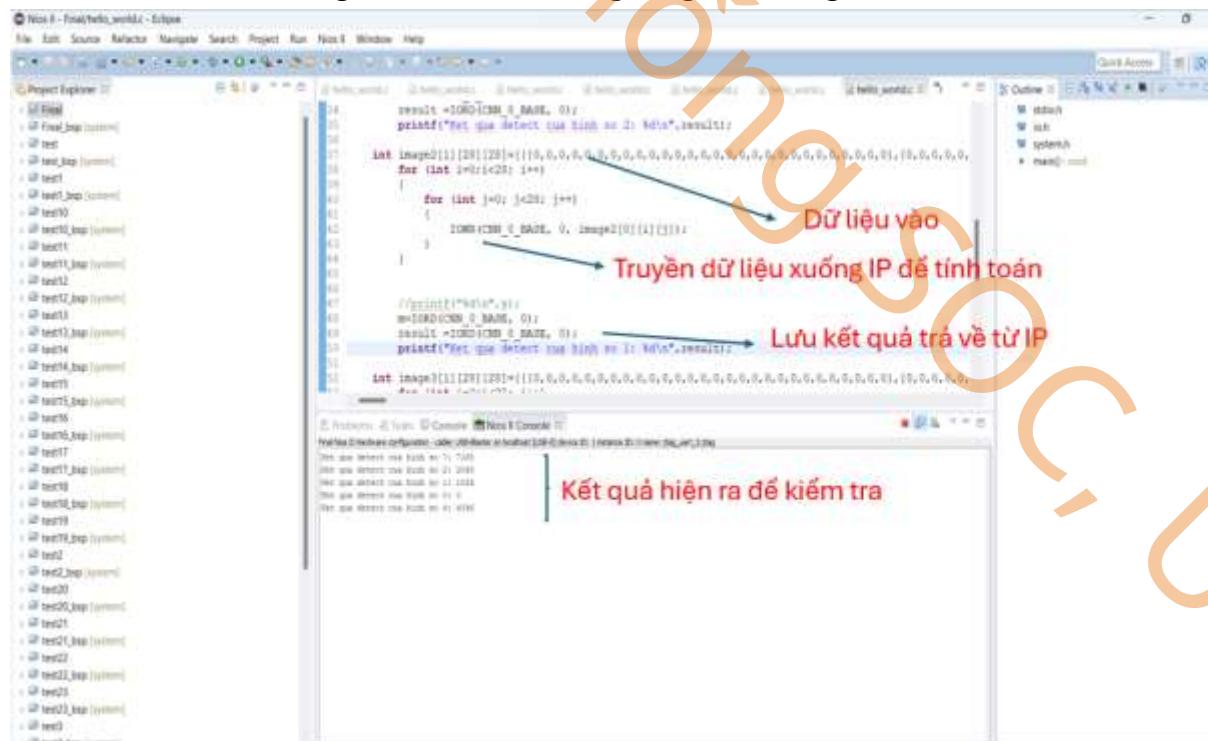
Từ mô hình được xây dựng bằng ngôn ngữ Python, bước tiếp theo trong quy trình hiện thực IP để tích hợp vào hệ thống SoC là chuyển mô hình này sang mã C, sau đó sử dụng công cụ hỗ trợ tự động, như Vivado HLS hoặc Intel HLS, để chuyển mã C thành mã RTL. Đoạn code minh họa trong hình 6.29 trình bày cách viết mã C cho các lớp cơ bản trong mạng LeNet5, bao gồm lớp tích chập, lớp kích hoạt, và lớp pooling, dựa trên mã Python tương ứng.

Code RTL sau khi thực hiện mô phỏng và kiểm thử với các mẫu thử khác nhau cho kết quả như mong muốn của người thiết kế, thì sẽ được đóng gói lại thành IP và chuyển tới bước hiện thực hệ thống SoC để chạy trên các bảng mạch FPGA mà người dùng mong muốn. Sau khi hiện thực xong phần cứng SoC như minh họa ở hình 6.30 và nạp xuống

bảng mạch FPGA tương ứng, người thiết kế chuyển sang bước lập trình điều khiển hệ thống bằng phần mềm Elipse của Altera hay SDK của Xilinx để hoàn thành bước cuối cùng trong quy trình thiết kế SoC. Hình 6.31 minh họa đoạn code điều khiển hệ thống SoC cho bài toán nhận dạng chữ số.



Hình 6.30: Hệ thống SoC thiết kế cho ứng dụng nhận dạng chữ số trên FPGA Altera



Hình 6.31: Minh họa lập trình code trên phần mềm để điều khiển và lấy kết quả từ phần cứng

## 6.6. Tóm tắt

Trong chương này, chúng tôi giới thiệu về ngôn ngữ tổng hợp cấp cao (HLS) và các lợi ích mà HLS mang lại trong quá trình thiết kế hệ thống SoC trên FPGA. Đầu tiên, chúng tôi phân tích chi tiết các đặc điểm và so sánh hai bộ công cụ HLS phổ biến từ Xilinx và Altera. Tiếp theo, chúng tôi trình bày quy trình thiết kế một IP bằng HLS, bao gồm việc chuyển đổi từ ngôn ngữ mô tả cấp cao sang ngôn ngữ mô tả phần cứng. Chúng tôi cũng thảo luận một số kỹ thuật tối ưu hóa trong HLS nhằm nâng cao hiệu suất và tối ưu hóa việc sử dụng tài nguyên phần cứng. Cuối cùng là phần đề cập đến vài ứng dụng cụ thể để minh họa chi tiết cách tổng hợp bằng HLS thông qua ví dụ tạo một IP từ bài toán nhân hai ma trận và xây dựng một hệ thống SoC hoàn chỉnh, từ Python đến SoC, cho bài toán nhận dạng chữ số sử dụng mạng nơ-ron tích chập (CNN). Chúng tôi hy vọng rằng chương này sẽ mang đến cho người đọc một cách tiếp cận hiệu quả trong thiết kế SoC, giúp rút ngắn thời gian phát triển và tối ưu hóa hệ thống để đáp ứng tốt các yêu cầu thực tế.

## 6.7. Câu hỏi và bài tập

1. HLS là gì. Nêu điểm giống và khác nhau Vivado HLS và Intel HLS.
2. Vì sao phải sử dụng HLS trong thiết kế SoC? Nếu quy trình thiết kế SoC dùng HLS.
3. Kể tên một số phương pháp tối ưu hóa thiết kế bằng HLS. Cho ví dụ minh họa từng phương pháp
4. Viết chương trình C thực hiện bài toán tính tổng tất cả các phần tử của một dãy số gồm N số thực. Tối ưu hóa mạch nhân sang số fixpoint 16 bit, với 5 bit độ chính xác cho phần thập phân. Dùng HLS chuyển code C sang dạng RTL. Thực hiện mô phỏng RTL code vừa tạo ra từ HLS.
5. Dựa vào thông tin tài nguyên phần cứng trích xuất được từ bài 4, cho biết thiết kế nào tốn nhiều tài nguyên phần cứng hơn.
6. Thực hiện tính toán tần số FMAX, công suất tiêu thụ cho thiết kế ở bài tập 4. Với giả thuyết tần số cài đặt ban đầu là 200 Mhz.
7. Viết chương trình C hiện thực hóa kiến trúc mạng sau. Dùng HLS chuyển code C này sang RTL.

Lớp	Đầu vào	Đầu ra	Kernel	Pooling	weights	Padding /Stride	tham số
Conv1D ReLU	1x101	32x97	32x1x5	-	-	0-1	192
Dropout	32x97	32x97	-	-	-	-	0
Pooling	32x97	32x49	-	1x3	-	1-2	0
Conv1D ReLU	32x49	32x25	32x32x 25	-	-	0-1	25632
Dropout	32x25	32x25	-	-	-	-	0
Flatten	32x25	800	-	-	-	-	0
Dense ReLU	800	512	-	-	800x51 2	-	410112
Dropout	512	512	-	-	-	-	0
Dense ReLU	512	256	-	-	512x25 6	-	131328
Dropout	256	256	-	-	-	-	0
Dense 3 Softmax	256	2	-	-	256x2	-	514
Tổng tham số							567,778

8. Tạo kết nối AXI với IP vừa được thiết kế ở trên
9. Thiết kế đầy đủ hệ thống SoC ở trên gồm CPU, RAM, bus kết nối và IP mạng tự thiết kế.
10. Viết code phần mềm điều khiển hệ thống trên, với dữ liệu mô phỏng đầu vào là 1 chuỗi số ngẫu nhiên kiểu float.

## TÀI LIỆU THAM KHẢO

- [1] Flynn, Michael J., and Wayne Luk. Computer system design: system-on-chip. John Wiley & Sons, 2011.
- [2] Schaumont, Patrick R. A practical introduction to hardware/software codesign. Springer Science & Business Media, 2012.
- [3] Yiu J. "System-on-Chip Design: With Arm® Cortex®-M Processors: Reference Book". Arm Education Media; 2019.
- [4] Neuendorffer, Stephen, and Fernando Martinez-Vallina. "Building zynq® accelerators with Vivado® high level synthesis." *FPGA*. 2013.
- [5] Altera DE2 Board – User Manual, Altera Corporation, 2012.
- [6][https://www.xilinx.com/support/documents/sw\\_manuals/xilinx2021\\_2/ug907-vivado-power-analysis-optimization.pdf](https://www.xilinx.com/support/documents/sw_manuals/xilinx2021_2/ug907-vivado-power-analysis-optimization.pdf), pp. 1- 118, 2021.
- [7]<https://www.intel.com/content/dam/support/us/en/programmable/support-resources/bulk-container/pdfs/literature/ug/archives/ug-qpp-power-18-1.pdf>, pp. 1-53, 2018.
- [8] Xilinx, "Vivado Design Suite Users Guide: High-Level Synthesis", UG902 (v2019.2), Jan 2020, pp. 5-6, 12-13.
- [9]<https://people.ece.cornell.edu/land/courses/ece5760/DE1-SOC/aib-01020-soc-fpga-cortex-a9-processor.pdf>, pp. 1-20, 2012.
- [10] <https://docs.amd.com/v/u/en-US/ds190-Zynq-7000-Overview>, pp. 1-25, 2018.
- [11]<https://www.intel.com/content/www/us/en/docs/programmable/683126/21-2/hard-processor-system-technical-reference.html>, 2024.
- [12] <https://www.scs.stanford.edu/~zyedidia/docs/riscv/riscv-privileged.pdf>, pp.1-155, 2021.
- [13] Maxfield C. The design warrior's guide to FPGAs: devices, tools and flows. Elsevier; 2004 Jun 16.

- [14] Xilinx Inc, “7 Series FPGAs Memory Resources”,  
[https://www.xilinx.com/support/documentation/user\\_guides/ug473\\_7Series\\_Memory\\_Resources.pdf](https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf), 2019
- [15] <https://semiwiki.com/wikis/semiconductor-ip-wikis/amba-advanced-microcontroller-bus-architecture/>, 2020.
- [16] Arm Ltd, “AMBA Specification version 2”,  
<https://developer.arm.com/documentation/ihi0011/latest/>, 2004.
- [17] Synopsys, “AMBA SoC Infrastructure IP”, <https://www.synopsys.com/designware-ip/soc-infrastructure-ip/amba.html> , 2020.
- [18] Arm Ltd, “AMBA AXI and ACE Protocol Specification”,  
<https://developer.arm.com/documentation/ihi0022/latest>, 2023.
- [19] Intel Corporation, “Avalon® Interface Specifications”,  
<https://www.intel.com/content/www/us/en/docs/programmable/683385/current/avalon-interface-specifications.html>, 2022.
- [20] [https://www.xilinx.com/support/documents/sw\\_manuals/xilinx2019\\_1/ug940-vivado-tutorial-embedded-design.pdf](https://www.xilinx.com/support/documents/sw_manuals/xilinx2019_1/ug940-vivado-tutorial-embedded-design.pdf), 2019
- [21] [https://docs.amd.com/r/3.2English/pg338dpu/Introduction?tocId=iBBrQ7pinvaWB\\_KbQH6hQ](https://docs.amd.com/r/3.2English/pg338dpu/Introduction?tocId=iBBrQ7pinvaWB_KbQH6hQ)
- [22] <https://www.amd.com/en/products/software/adaptive-socs-and-pgas/microblaze.html>