

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH  
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN  
KHOA KỸ THUẬT MÁY TÍNH

ĐÀO PHƯỚC TÀI  
NGUYỄN ANH KHÔI

KHÓA LUẬN TỐT NGHIỆP  
THIẾT KẾ HỆ THỐNG SOC DỰA TRÊN BỘ XỬ LÝ  
RISC-V 32-BIT VỚI MÃ HOÁ H.264 NỘI KHUNG TRÊN  
FPGA

DESIGN OF A RISC-V 32-BIT PROCESSOR-BASED SOC WITH  
H.264 INTRA-FRAME ENCODING ON FPGA

CỬ NHÂN NGÀNH KỸ THUẬT MÁY TÍNH

TP. HỒ CHÍ MINH, 2025

**ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH**  
**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**  
**KHOA KỸ THUẬT MÁY TÍNH**

**ĐÀO PHƯỚC TÀI- 21521391**  
**NGUYỄN ANH KHÔI - 21520301**

**KHÓA LUẬN TỐT NGHIỆP**  
**THIẾT KẾ HỆ THỐNG SOC DỰA TRÊN BỘ XỬ LÝ**  
**RISC-V 32-BIT VỚI MÃ HOÁ H.264 NỘI KHUNG TRÊN**  
**FPGA**

**DESIGN OF A RISC-V 32-BIT PROCESSOR-BASED SOC WITH**  
**H.264 INTRA-FRAME ENCODING ON FPGA**

**CỬ NHÂN NGÀNH KỸ THUẬT MÁY TÍNH**

**GIẢNG VIÊN HƯỚNG DẪN**  
**TH.S NGÔ HIẾU TRƯỜNG**

**TP. HỒ CHÍ MINH, 2025**

## **THÔNG TIN HỘI ĐỒNG CHẤM KHÓA LUẬN TỐT NGHIỆP**

Hội đồng chấm khóa luận tốt nghiệp, thành lập theo Quyết định số .....  
ngày ..... của Hiệu trưởng Trường Đại học Công nghệ Thông tin.

## LỜI CẢM ƠN

Sau bốn năm học tập và rèn luyện tại Trường Đại học Công nghệ Thông tin – Đại học Quốc gia TP. Hồ Chí Minh, chặng đường cuối cùng của hành trình đại học được đánh dấu bằng khoá luận tốt nghiệp – một cột mốc quan trọng mở ra những bước đầu tiên trên con đường nghề nghiệp phía trước.

Chúng em xin được bày tỏ lòng biết ơn sâu sắc đến quý thầy cô nhà trường. Đặc biệt, chúng em xin tri ân các thầy cô khoa Kỹ thuật máy tính, những người đã đồng hành, hỗ trợ chúng em trong suốt quá trình học tập và thực hiện khoá luận.

Chúng em xin gửi lời cảm ơn chân thành đến Thạc sĩ Ngô Hiếu Trường – giảng viên hướng dẫn, người đã luôn tận tình chỉ bảo, đưa ra những nhận xét quý báu và giúp định hướng rõ ràng để chúng em có thể hoàn thiện đề tài tốt nhất.

Chúng em cũng muốn dành những lời cảm ơn chân thành nhất đến gia đình – điểm tựa vững chắc và là nguồn động viên thầm lặng giúp chúng em vững bước trên mọi hành trình.

Mặc dù đã nỗ lực, khoá luận của chúng em vẫn không tránh khỏi những thiếu sót nhất định. Chúng em rất mong nhận được sự thông cảm và những góp ý quý báu từ quý thầy cô trong Hội đồng để hoàn thiện hơn trong tương lai.

Chúng em xin chân thành cảm ơn!

*TP. Hồ Chí Minh, ngày 13 tháng 6 năm 2025*

Sinh viên thực hiện

Đào Phước Tài

Nguyễn Anh Khôi

## MỤC LỤC

Chương 1. GIỚI THIỆU ĐỀ TÀI .....	15
1.1. Tổng quan đề tài .....	15
1.2. Mục tiêu đề tài .....	16
1.3. Giới hạn đề tài .....	17
Chương 2. CƠ SỞ LÝ THUYẾT.....	18
2.1. Sơ lược về hệ thống SoC .....	18
2.1.1. Giới thiệu .....	18
2.1.2. Các thành phần cơ bản của một SoC .....	18
2.2. Kiến trúc tập lệnh RISC-V .....	20
2.2.1. Giới thiệu .....	20
2.2.2. Kiến trúc tập lệnh RV32I.....	20
2.2.3. Kỹ thuật pipeline 5 tầng.....	22
2.2.4. Các loại xung đột trong RISC-V.....	23
2.3. Sơ lược về chuẩn mã hoá video H.264/AVC .....	27
2.3.1. Giới thiệu .....	27
2.3.2. Nguyên lý hoạt động của bộ mã hoá video H.264/AVC .....	28
2.4. Sơ lược Directed Memory Access (DMA) .....	33
2.5. Sơ lược về giao thức AMBA AXI.....	33
2.6. Sơ lược về board Xilinx FPGA Virtex-7 VC707 .....	35
Chương 3. HIỆN THỰC THIẾT KẾ .....	36
3.1. Thiết kế bộ xử lý RISC-V RV32I .....	36
3.1.1. Kiến trúc tổng quát.....	36
3.1.2. Thiết kế khối Datapath pipeline .....	37

3.1.2.1.	Tầng Fetch (IF).....	37
3.1.2.2.	Tầng Decode (ID).....	37
3.1.2.3.	Tầng Execute (EX).....	38
3.1.2.4.	Tầng Memory (MA).....	39
3.1.2.5.	Tầng Write-Back (WB).....	39
3.1.3.	Thiết kế khối Controller.....	39
3.1.4.	Bộ xử lý xung đột.....	41
3.1.5.	Thiết kế giao diện và đóng gói IP .....	45
3.2.	Thiết kế bộ mã hoá H.264/AVC.....	48
3.2.1.	Kiến trúc lõi H.264 IE.....	48
3.2.2.	Bộ LOAD_MB_16x16.....	48
3.2.3.	Bộ CONTROLLER .....	51
3.2.4.	Giải pháp xử lý vấn đề Cross Domain Crossing (CDC).....	54
3.2.2.1.	FIFO bất đồng bộ.....	54
3.2.2.2.	Phương pháp đồng bộ tín hiệu nhiều bit.....	56
3.2.5.	Đóng gói và tích hợp bộ mã hoá.....	57
3.3.	Hiện thực hệ thống SoC cho mô phỏng trên Vivado .....	58
3.3.1.	Hệ thống SoC tổng quát.....	58
3.3.2.	Giải thuật điều khiển bộ mã hoá .....	60
3.3.3.	Một số waveform mô tả hệ thống .....	65
3.3.2.1.	Đọc thanh ghi trạng thái MM2S .....	65
3.3.2.2.	Đọc thanh ghi trạng thái S2MM.....	65
3.4.	Hiện thực hệ thống SoC cho việc triển khai xuống FPGA .....	67
3.5.	Các IP của Xilinx được sử dụng trong Block Design .....	68

3.5.1.	MicroBlaze.....	68
3.5.2.	Memory Intergrater Generator (MIG 7 series).....	68
3.5.3.	Khối AXI 1G/2.5G Ethernet Subsystem.....	69
3.5.4.	Khối AXI DMA .....	70
3.5.5.	Khối AXI SmartConnect.....	72
3.5.6.	Khối AXI UartLite .....	72
Chương 4.	KẾT QUẢ VÀ ĐÁNH GIÁ.....	73
4.1.	Bộ xử lý RISC-V RV32I.....	73
4.1.1.	Kiểm tra 37 lệnh cơ bản qua mô phỏng và triển khai trên FPGA .....	73
4.1.2.	Kiểm tra truy xuất ngoại vi qua MMIO Interface.....	78
4.1.3.	Đánh giá tài nguyên sử dụng .....	80
4.2.	Bộ mã hoá video H264/AVC .....	81
4.2.1.	Kịch bản kiểm tra.....	81
4.2.2.	Đánh giá hiệu suất.....	82
4.2.3.	Đánh giá tài nguyên sử dụng .....	83
4.3.	Kết quả thực thi của toàn bộ hệ thống SoC.....	84
4.4.	Đánh giá thiết kế và so sánh với các đề tài khác.....	87
4.4.1.	Đánh giá bộ xử lý RV32I.....	87
4.4.2.	Đánh giá bộ mã hoá video H.264.....	88
Chương 5.	KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN.....	89
5.1.	Kết luận .....	89
5.2.	Hướng phát triển.....	89

## DANH MỤC HÌNH

Hình 2.1 Minh hoạ các thành phần cơ bản của một SoC [9].....	18
Hình 2.2 Tập lệnh cơ bản của bộ xử lý RISC-V RV32I [21].....	21
Hình 2.3 Mô hình kỹ thuật pipeline 5 tầng của bộ xử lý RV32I.....	22
Hình 2.4 Xung đột dữ liệu RAW .....	24
Hình 2.5 Xung đột dữ liệu ở lệnh Load .....	25
Hình 2.6 Phương pháp xử lý xung đột ở lệnh load.....	25
Hình 2.7 Xung đột điều khiển .....	25
Hình 2.8 Phương pháp xử lý xung đột điều khiển .....	26
Hình 2.9 Quá trình mã hoá video trong chuẩn H.264.....	29
Hình 2.10 Quá trình giải mã video trong chuẩn H.264 .....	29
Hình 2.11 Các chế độ dự đoán nội khung cho thành phần Luma (Y).....	30
Hình 2.12 Các chế độ dự đoán nội khung cho thành phần Chroma (Cr, Cb).....	30
Hình 2.13 Bảng tham chiếu cho ma trận lượng tử hoá.....	32
Hình 2.14 Minh hoạ 5 kênh giao tiếp của giao thức AXI .....	34
Hình 2.15 Minh hoạ board Virtex 7 – VC707.....	35
Hình 3.1 Kiến trúc RV32I pipeline .....	36
Hình 3.2 Khối IMEM. ....	37
Hình 3.3 Khối Controller của bộ xử lý RV32I.....	39
Hình 3.4 Khối Forwarding Unit.....	41
Hình 3.5 Khối Hazard Handler .....	42
Hình 3.6 Waveform minh hoạ xung đột load-hazard 1.....	43
Hình 3.7 Waveform minh hoạ xung đột load-hazard 2.....	43
Hình 3.8 Waveform minh hoạ xung đột điều khiển 1.....	44
Hình 3.9 Waveform minh hoạ xung đột điều khiển 2.....	44
Hình 3.10 Kiến trúc IP bộ xử lý RV32I sau khi đóng gói .....	45
Hình 3.11 Waveform minh ho giao dịch AXI ghi từ lệnh store-byte .....	46
Hình 3.12 Waveform minh hoạ giao dịch AXI đọc từ lệnh load-half .....	46



Hình 3.13 Mô hình kiểm tra bộ xử lý RV32I bằng AXI VIP .....	47
Hình 3.14 Thông báo vi phạm từ protocol check của AXI VIP .....	47
Hình 3.15 Kiến trúc bộ mã hoá video H.264 IE pipeline.....	48
Hình 3.16 Khối Load MB 16x16 .....	49
Hình 3.17 Khối H.264 Controller .....	51
Hình 3.18 Sơ đồ biến đổi trạng thái trong bộ H.264 Controller .....	53
Hình 3.19 Biểu đồ trạng thái hoạt động pipeline theo thời gian.....	54
Hình 3.20 Kiến trúc khối FIFO bất đồng bộ.....	55
Hình 3.21 Mạch nguyên lý phương pháp kiểm tra sự ổn định của giá trị nhận [10]	56
Hình 3.22 Kiến trúc bộ mã hoá video H.264 sau khi đóng gói.....	57
Hình 3.23 Thanh ghi cấu hình của bộ mã hoá video H.264.....	57
Hình 3.24 Hệ thống SoC mô phỏng .....	58
Hình 3.25 Lưu đồ giải thuật mô tả chương trình SoC cho mã hoá video .....	60
Hình 3.26 Lệnh đọc từ bộ xử lý RV32I đến thanh ghi MM2S_DMA_SR.....	65
Hình 3.27 Lệnh đọc từ bộ xử lý RV32I đến thanh ghi S2MM_DMA_SR.....	65
Hình 3.28 Lệnh đọc từ bộ xử lý RV32I đến thanh ghi S2MM_LENGTH.....	66
Hình 3.29 Kết quả so sánh số byte truyền từ DMA so với byte thực tế .....	66
Hình 3.30 Kiến trúc SoC triển khai xuống FPGA .....	67
Hình 3.31 Soft-core MicroBlaze trong Vivado Block Design.....	68
Hình 3.32 Memory Intergrater Generator trên Vivado Block Design .....	68
Hình 3.33 AXI Ethernet Subsystem trong Vivado Block Design .....	69
Hình 3.34 Khối AXI DMA trong Vivado Block Design .....	70
Hình 3.35 Kiến trúc khối AXI DMA [12] .....	70
Hình 3.36 Khối AXI SmartConnect Trong Vivado Block Design .....	72
Hình 3.37 Khối AXI Uartlite trên Vivado Block Design .....	72
Hình 4.1 Minh hoạ chương trình nạp lệnh qua Vitis Serial Console .....	76
Hình 4.2 Kết quả mô phỏng 37 lệnh qua phần mềm RARS.....	77
Hình 4.3 Kết quả thanh ghi đọc từ Vitis Serial Console.....	77
Hình 4.4 Kết quả so sánh thanh ghi giữa Vivado và phần mềm Rars.....	79

Hình 4.5 Kết quả timing sau khi gắn constraint của bộ xử lý RV32I .....	80
Hình 4.6 Mô tả constraint của bộ xử lý RV32I (100Mhz).....	80
Hình 4.7 Kịch bản kiểm tra bộ mã hoá video H.264/AVC .....	81
Hình 4.8 Kết quả timing sau khi gắn constraint của bộ mã hoá video H.264 .....	83
Hình 4.9 Mô tả 2 clock domain của bộ mã hoá video H.264.....	83
Hình 4.10 Kết quả mô phỏng hệ thống SoC .....	84
Hình 4.11 Kiểm tra video đã mã hoá .264 bằng phần mềm VLC .....	85
Hình 4.12 File tổng hợp kết quả mô phỏng dựa trên số xung clock.....	85
Hình 4.13 Kết quả PSNR tính từ phần mềm ffmpeg .....	86

## **DANH MỤC BẢNG**

Bảng 2.1 Bảng so sánh các chuẩn nén video phổ biến .....	28
Bảng 3.1 Bảng mô tả chức năng bộ ALU .....	38
Bảng 3.2 Bảng mô tả chức năng khối Controller .....	40
Bảng 3.3 Bảng mô tả tín hiệu khối Load MB 16x16 .....	50
Bảng 3.4 Bảng mô tả các tín hiệu của bộ H.264 Controller .....	52
Bảng 3.5 Bảng mô tả chức năng các trạng thái của bộ H.264 Controller .....	53
Bảng 3.6 Bảng mô tả không gian địa chỉ truy cập của bộ xử lý RV32I .....	59
Bảng 3.7 Bảng mô tả thanh ghi DMA trong chế độ Directed Register .....	71
Bảng 4.1 Chương trình kiểm tra 37 lệnh cơ bản của tập lệnh RV32I .....	73
Bảng 4.2 Tập lệnh kiểm tra truy xuất ngoại vi qua giao diện MMIO .....	78
Bảng 4.3 Bảng mô tả tài nguyên sử dụng của bộ xử lý RV32I .....	80
Bảng 4.4 Bảng kết quả hiệu suất của bộ mã hoá video H.264/AVC .....	82
Bảng 4.5 Bảng mô tả tài nguyên của bộ mã hoá video H.264 .....	83
Bảng 4.6 Bảng so sánh thiết kế RV32I của nhóm với các đề tài khác .....	87
Bảng 4.7 So sánh thiết kế bộ mã hoá video H.264 so với các đề tài khác .....	88

## DANH MỤC TỪ VIẾT TẮT

Viết tắt	Viết đầy đủ	Ý nghĩa
ALU	Arithmetic Logic Unit	Bộ xử lý số học và logic
AMBA	Advanced Microcontroller Bus Architecture	Kiến trúc bus tiên tiến cho vi điều khiển do ARM phát triển
AVC	Advanced Video Codec	Chuẩn mã hoá video nâng cao
AXI	Advanced eXtensible Interface	Giao thức bus mở rộng hiệu năng cao
CPU	Central Processing Unit	Bộ xử lý trung tâm
DMA	Direct Memory Access	Khối truy cập bộ nhớ trực tiếp
FPGA	Field Programmable Gate Array	Vi mạch lập trình được có thể cấu hình lại sau sản xuất
IP	Intellectual Property	Khối thiết kế phần cứng (IP nhân - sở hữu trí tuệ)
ISA	Instruction Set Architecture	Kiến trúc tập lệnh
RAM	Random Access Memory	Bộ nhớ truy cập ngẫu nhiên
RISC	Reduce Instruction Set Computer	Kiến trúc tập lệnh rút gọn
SOC	System On Chip	Hệ thống tích hợp trên một vi mạch
VIP	Verification Intellectual Property	Khối IP xác minh chức năng

## TÓM TẮT KHÓA LUẬN

Khóa luận này trình bày quá trình thiết kế và hiện thực một hệ thống SoC (System-on-Chip) tích hợp bộ xử lý RISC-V 32-bit RV32I cùng bộ mã hoá video H.264/AVC Intra-frame trên nền tảng FPGA Xilinx Virtex-7 VC707. Mục tiêu chính của đề tài là xây dựng một hệ thống có khả năng nén video theo chuẩn H.264/AVC, dưới sự điều khiển hoàn toàn của bộ xử lý RISC-V do nhóm tự thiết kế, qua đó chứng minh tính khả thi của việc triển khai kiến trúc xử lý mở RISC-V trong các ứng dụng nhúng có yêu cầu xử lý dữ liệu phức tạp.

Trong quá trình thực hiện, nhóm đã thiết kế bộ xử lý RV32I với kiến trúc pipeline 5 tầng, có khả năng xử lý đầy đủ 37 lệnh cơ bản, tích hợp cơ chế giải quyết xung đột dữ liệu và điều khiển. Bộ xử lý được kết nối đến các ngoại vi thông qua giao thức AXI-Lite, cho phép điều khiển và cấu hình các khối chức năng như bộ mã hoá H.264 hoặc bộ điều khiển DMA một cách linh hoạt. Về phía bộ mã hoá video, nhóm sử dụng một lõi mã hoá H.264 Intra từ mã nguồn mở và thiết kế lại toàn bộ cơ chế nạp dữ liệu, điều khiển tiến trình mã hoá, xử lý bitstream đầu ra, đồng thời giải quyết các vấn đề bất đồng bộ clock domain (CDC) giữa các thành phần hoạt động ở các tần số khác nhau.

Hệ thống SoC được xây dựng bao gồm các IP lõi của Xilinx như AXI DMA, SmartConnect, UARTLite, MIG DDR3 và tích hợp trong môi trường Vivado Block Design. Quá trình mô phỏng được thực hiện để kiểm chứng chức năng của từng thành phần, sau đó hệ thống được triển khai thực tế trên FPGA. Bộ xử lý RISC-V được nạp chương trình qua AXI VIP hoặc MicroBlaze hỗ trợ debug, còn dữ liệu video đầu vào được xử lý bằng công cụ Python, nạp vào bộ nhớ và truyền qua DMA đến bộ mã hoá. Bitstream đầu ra sau khi mã hoá được lưu lại để so sánh với dữ liệu nén từ phần mềm, và đánh giá chất lượng thông qua chỉ số PSNR.

Kết quả thực nghiệm cho thấy hệ thống hoạt động ổn định, bộ xử lý RISC-V thực hiện chính xác tập lệnh RV32I, bộ mã hoá H.264 cho ra bitstream đúng định dạng và có thể giải mã lại thành video đạt chất lượng tốt với chỉ số đánh giá chất lượng PSNR trên 35dB ở video FHD@30fps với tham số lượng tử hoá QP bằng 28. Hệ thống hỗ trợ độ phân giải lên đến Full HD và hoạt động ở hai miền xung nhịp riêng biệt (100 MHz cho CPU, 62.5 MHz cho H.264 encoder). Qua đó, đề tài không chỉ khẳng định khả năng hiện thực của SoC sử dụng RISC-V và H.264, mà còn mở ra hướng phát triển cho các ứng dụng nhúng xử lý video theo thời gian thực trên nền tảng FPGA.

## **Chương 1. GIỚI THIỆU ĐỀ TÀI**

### **1.1. Tổng quan đề tài**

Trong bối cảnh nhu cầu phát trực tuyến video, hội nghị truyền hình và giám sát an ninh ngày càng tăng, yêu cầu về khả năng mã hóa hiệu quả và tiết kiệm tài nguyên tính toán trở nên quan trọng. Các thuật toán nén video liên tục được cải tiến nhằm tối ưu hóa dung lượng lưu trữ và băng thông truyền tải mà vẫn đảm bảo chất lượng hình ảnh. H.264/AVC là chuẩn nén video phổ biến nhờ khả năng nén hiệu quả mà vẫn giữ chất lượng hình ảnh cao [1][2]. So với H.263 [3], H.264 giúp giảm băng thông mà không làm giảm đáng kể chất lượng video. Dù H.265/HEVC [4] có hiệu suất nén tốt hơn và hỗ trợ thiết bị hiện đại, nhưng đòi hỏi nhiều tài nguyên tính toán hơn. Do đó, H.264 vẫn là lựa chọn tối ưu trong nhiều hệ thống hiện tại nhờ khả năng tương thích, độ trễ thấp và yêu cầu phần cứng vừa phải. Việc triển khai H.264 bằng phần cứng giúp tăng tốc xử lý, hỗ trợ mã hóa thời gian thực và tối ưu điện năng. Các giải pháp dựa trên DSP hoặc ARM đều có hạn chế riêng: DSP bị giới hạn về mở rộng và tính linh hoạt [5], trong khi ARM khó đạt tối ưu khi tích hợp phần cứng chuyên biệt [6]. Bên cạnh đó, bộ xử lý thương mại thường đi kèm chi phí và yêu cầu bản quyền. Trong bối cảnh đó, RISC-V [7] nổi lên như một giải pháp tiềm năng nhờ kiến trúc mở, cho phép tùy chỉnh linh hoạt mà không chịu ràng buộc bản quyền, phù hợp với các ứng dụng xử lý video tích hợp sâu giữa phần cứng và phần mềm.

RISC-V là một kiến trúc tập lệnh (ISA) được giới thiệu vào năm 2010 tại Đại học California, Berkeley. Kiến trúc này có nhiều biến thể như RV32I, RV64I và các phiên bản mở rộng hỗ trợ số học dấu chấm động, vector, ... phù hợp với nhiều loại ứng dụng từ nhúng đến hiệu năng cao giúp mang lại sự linh hoạt vượt trội, giúp phát triển hệ thống tối ưu hơn. Việc kết hợp bộ mã hóa H.264 với SoC dựa trên RISC-V giúp tăng hiệu suất xử lý, giảm độ trễ và cải thiện khả năng tích hợp trong các hệ thống nhúng. Trong đề tài này, RV32I được chọn làm bộ xử lý vì đây là biến thể 32-bit đơn giản, chỉ gồm tập lệnh số nguyên, giúp giảm tài nguyên phần cứng, phù hợp với các hệ thống nhúng và đảm bảo tính khả thi khi triển khai trên FPGA.

## 1.2. Mục tiêu đề tài

Đề tài “Thiết kế hệ thống SoC dựa trên bộ xử lý RISC-V 32-bit với Mã Hoá H.264 Nội Khung trên FPGA” hướng đến việc xây dựng một hệ thống SoC tùy biến, có khả năng thực hiện chức năng mã hoá video theo chuẩn H.264/AVC dưới sự điều khiển và phân luồng dữ liệu được đảm nhiệm bởi bộ xử lý RISC-V RV32I do nhóm tự thiết kế.

Các mục tiêu cụ thể của đề tài bao gồm:

- Thiết kế và hiện thực phần cứng bộ xử lý RISC-V RV32I: Bộ xử lý được xây dựng theo kiến trúc 32-bit với pipeline 5 tầng, hỗ trợ xử lý đầy đủ các lệnh cơ bản và tích hợp các cơ chế xử lý xung đột dữ liệu và điều khiển. Ngoài ra, CPU còn được tích hợp giao diện AXI-Lite để giao tiếp và điều khiển các ngoại vi bên ngoài như bộ mã hoá và DMA.
- Nghiên cứu và tích hợp bộ mã hoá video H.264/AVC: Nhóm tiến hành tìm hiểu nguyên lý hoạt động của bộ mã hoá H.264 Intra-frame thông qua các tài liệu và mô phỏng chức năng bằng phần mềm MATLAB. Do giới hạn về thời gian và năng lực, nhóm lựa chọn sử dụng một lõi mã hoá mã nguồn mở [20], từ đó thiết kế RTL các khối giao tiếp AXI cần thiết và giải quyết các vấn đề về bất đồng bộ xung nhịp (CDC) để tích hợp hiệu quả vào hệ thống SoC. Sau đó, nhóm sẽ kết hợp các thành phần đã thiết kế cùng với các IP có sẵn từ Xilinx để tạo thành một hệ thống SoC đáp ứng đúng các chức năng đã đề ra.
- Xây dựng hệ thống SoC hoàn chỉnh: Kết hợp các thành phần tự thiết kế (CPU và interface logic) với các IP lõi có sẵn từ Xilinx như DMA, bộ nhớ DDR3, UARTLite... để hình thành một hệ thống SoC đồng bộ, có khả năng xử lý và điều phối toàn bộ chuỗi nén video từ đầu vào đến đầu ra.
- Triển khai và kiểm thử thực tế trên FPGA: Toàn bộ hệ thống được triển khai trên board FPGA Xilinx Virtex-7 VC707. Bộ xử lý RISC-V sẽ hoạt động ở tần số 100 MHz, trong khi bộ mã hoá H.264 được tối ưu hoạt động ở tần số 62,5 MHz. Các chương trình điều khiển và kiểm thử được xây dựng để đánh giá hiệu năng hoạt động và chất lượng video đầu ra của hệ thống.



### 1.3. Giới hạn đề tài

Đối với bộ xử lý RISC-V RV32I:

- Bộ xử lý sẽ không tách riêng I-Cache và D-Cache, mà sẽ tích hợp vào bên trong bộ xử lý. Ngoài ra, nhóm cần phải kết hợp thêm một soft-core MicroBlaze để tiến hành nạp chương trình cũng như debug thông qua phần mềm Vitis IDE.
- Bộ xử lý sẽ không hỗ trợ Control and Status Register Instructions (CSR) cũng như các lệnh liên quan đến Environment Call Environment Breakpoints.
- Bộ xử lý sẽ không có cơ chế dự đoán nhảy (branch prediction), thay vào đó bộ xử lý sẽ sử dụng các cơ chế stall và flush nếu gặp các xung đột trong quá trình thực thi.

Đối với bộ mã hoá H.264/AVC, vì bị giới hạn về thời gian và kiến thức, nhóm chỉ dừng lại ở việc tìm hiểu hoạt động của bộ mã hoá này và sử dụng một bộ mã hoá open-source để tích hợp vào hệ thống và có những hạn chế sau:

- Bộ mã hóa H.264/AVC chỉ triển khai dự đoán nội khung (Intra-Prediction), không hỗ trợ các phương pháp dự đoán liên khung (Inter-Prediction), do đó chưa tối ưu hóa hiệu suất nén so với các cấu hình có khung P/B.
- Hệ thống chỉ hỗ trợ độ phân giải tối đa Full HD (1920x1080) với định dạng màu YUV 4:2:0, phù hợp với các ứng dụng có yêu cầu băng thông trung bình.
- Bộ mã hóa chỉ hỗ trợ Baseline Profile, không bao gồm các tính năng nâng cao như CABAC (Context-Adaptive Binary Arithmetic Coding) hay B-frames, làm hạn chế mức độ nén so với các profile cao hơn.
- Các module biến đổi (Transform) và lượng tử hóa (Quantization) chỉ thực hiện trên ma trận số nguyên, có thể gây sai lệch nhỏ giữa video gốc và video sau khi giải mã, nhưng vẫn nằm trong mức chất lượng chấp nhận được.
- Chưa thực hiện tích hợp Deblocking Filter, có thể dẫn đến hiện tượng "khối vuông" (blocking artifacts) trong một số điều kiện mã hóa nhất định.

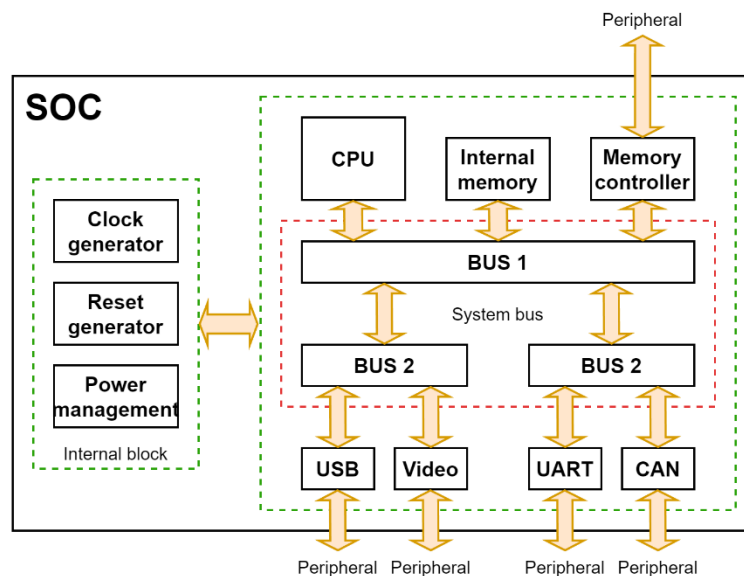
## Chương 2. CƠ SỞ LÝ THUYẾT

### 2.1. Sơ lược về hệ thống SoC

#### 2.1.1. Giới thiệu

System-on-Chip (SoC) là một hệ thống vi mạch tích hợp toàn bộ các thành phần cần thiết của một hệ thống máy tính hoặc hệ thống nhúng vào một chip đơn. Một SoC điển hình có thể bao gồm bộ xử lý (CPU), bộ nhớ, các giao diện truyền thông (UART, SPI, I2C), bộ điều khiển DMA, khối xử lý tín hiệu số (DSP), khối nén/giải nén video, và các IP ngoại vi khác.

#### 2.1.2. Các thành phần cơ bản của một SoC



Hình 2.1 Minh họa các thành phần cơ bản của một SoC [9]

Hình 2.1 minh họa các thành phần cơ bản của một SoC, hầu hết các SoC đều có các thành phần cơ bản sau đây:

- CPU: lõi vi xử lý là thành phần không thể thiếu có nhiệm vụ quản lý toàn bộ hoạt động chính của SoC như đảm nhiệm luồng xử lý, điều phối hoạt động giữa các thành phần hay thực thi tính toán, ... Một SoC có thể có nhiều lõi CPU, trong đó nổi bật như:
  - o Lõi CPU ARM dùng tập lệnh ARM và là sản phẩm của hãng ARM.
  - o Lõi CPU dựa trên tập lệnh mã nguồn mở và miễn phí RISC-V.

- SH (SuperH) dùng tập lệnh RISC được phát triển bởi Hitachi và là sản phẩm của Renesas.
- Bus hệ thống: có nhiệm vụ kết nối thông suốt các thành phần chức năng khác nhau trong vi xử lý, một số cấu trúc bus hệ thống điển hình như:
  - AMBA (AXI, AHB, APB) là chuẩn bus phát triển bởi ARM.
  - Avalon là chuẩn bus phát triển bởi Altera, nay thuộc Intel.
  - STBus của STMicroelectronic.
  - Wishbone của Silicore Corporation.
- Bộ nhớ:
  - ROM lưu firmware/bootloader chương trình ban đầu.
  - RAM để lưu thông tin hoặc giá trị tính toán của SoC.
- Thành phần điều khiển nội: là thành phần chỉ điều khiển hoạt động bên trong SoC mà không điều khiển trực tiếp port nào của SoC như:
  - Khối tạo clock (clock generator): Cung cấp clock cho toàn bộ các khối chức năng trong SoC, kể cả CPU.
  - Khối tạo Reset (reset generator): Cung cấp reset cho toàn bộ các khối chức năng trong SoC, kể cả CPU.
  - Khối quản lý năng lượng (power management): Điều khiển cấp nguồn (bật/tắt) cho các khối chức năng trong SoC.
  - Các khối giám sát (Monitor) là các khối có chức năng giám sát hoạt động của SoC, kịp thời phát hiện ra các lỗi trong quá trình hoạt động để khởi động lại một phần hoặc toàn bộ hệ thống.
- Ngoại vi (Peripheral): là các khối có thể lái trực tiếp các chân (pin hoặc port) của SoC để thực thi một chức năng điều khiển bên ngoài SoC, ví dụ như:
  - UART: truyền nhận dữ liệu nối tiếp bất đồng bộ.
  - SegLCD: điều khiển hiển thị trên segment LCD.
  - Video: điều khiển camera.
  - Audio: thu phát âm thanh.
  - ADC: bộ chuyển đổi tín hiệu tương tự thành tín hiệu số.

## **2.2. Kiến trúc tập lệnh RISC-V**

### **2.2.1. Giới thiệu**

RISC-V là một kiến trúc tập lệnh máy tính (ISA) dạng RISC (Reduced Instruction Set Computing) được phát triển bởi nhóm nghiên cứu tại Đại học California, Berkeley với mục tiêu là đơn giản hoá hoạt động của bộ xử lý, qua đó giảm độ phức tạp trong phần cứng và tối ưu hiệu năng. Khác với các kiến trúc thương mại truyền thống, RISC-V được thiết kế theo mô hình mở, cho phép sử dụng và tùy chỉnh mà không bị ràng buộc bởi chi phí bản quyền. RISC-V hiện hỗ trợ ba phiên bản chính theo độ dài thanh ghi: RV32 (32-bit), RV64 (64-bit) và RV128 (128-bit), đáp ứng đa dạng nhu cầu từ các thiết bị nhúng nhỏ gọn đến các hệ thống tính toán hiệu năng cao. Nhờ kiến trúc đơn giản, dễ hiện thực phần cứng, khả năng mở rộng linh hoạt thông qua các phần mở rộng chính thức và tùy biến, RISC-V đã và đang được ứng dụng rộng rãi trong nhiều lĩnh vực như vi điều khiển, thiết bị IoT, hệ thống nhúng, máy tính học thuật, thậm chí cả trong lĩnh vực điện toán đám mây và AI, trở thành lựa chọn tiềm năng thay thế cho các kiến trúc độc quyền hiện nay.

Trong số các tập lệnh của RISC-V, RV32I được xem là nền tảng cơ bản và phổ biến nhất. Đây là tập lệnh chuẩn cho các kiến trúc 32-bit, bao gồm những lệnh thao tác với số nguyên, đủ để xây dựng một bộ xử lý chức năng hoàn chỉnh với các nhóm lệnh như số học, logic, nhảy có điều kiện, truy xuất bộ nhớ và hệ thống. Với những ưu điểm đó, RV32I là một lựa chọn lý tưởng cho việc hiện thực một CPU tùy chỉnh, là điểm khởi đầu trong học thuật và nghiên cứu.

### **2.2.2. Kiến trúc tập lệnh RV32I**

Tập lệnh này bao gồm 40 lệnh nhưng trong đề tài này nhóm sẽ bỏ qua một số lệnh như ECALL/EBREAK và FENCE để cho việc xử lý đơn giản hơn, rút ngắn tổng số lệnh được triển khai xuống còn 37 lệnh được mô tả qua Hình 2.2 bên dưới với chức năng được mô tả qua cột <Description>. Mỗi lệnh sẽ có các trường <opcode> có nhiệm vụ nhóm các lệnh có cùng chức năng tổng quát vào cùng một danh mục xử lý, trường <funct3> xác định loại lệnh cụ thể khi nhiều lệnh cùng chia

sẽ một opcode, trường <funct7> ở một số lệnh dùng để phân biệt các lệnh có cùng trường <opcode> và <funct3> đặc biệt trong loại lệnh R-type. Ngoài ra, một số lệnh có phần mở rộng dựa vào bit MSB hoặc zero-extend mở rộng bằng các bit 0 ở một số lệnh sltu, lbu, ...

#### RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1   rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1   imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srl	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 ≥ rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 ≥ rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1101111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	

Hình 2.2 Tập lệnh cơ bản của bộ xử lý RISC-V RV32I [21]

Trong đó được chia thành 4 nhóm lệnh cơ bản:

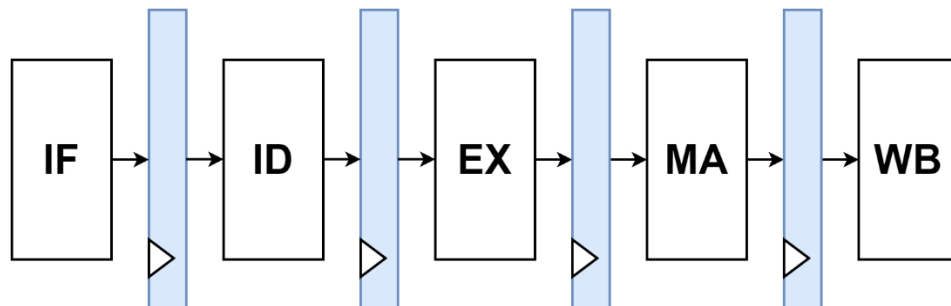
- Nhóm lệnh số học – logic: là các lệnh thao tác trực tiếp trên các thanh ghi (R-type) hoặc giữa thanh ghi và hằng số (I-type) để thực hiện các phép toán số học, so sánh và dịch bit và là nền tảng cho mọi xử lý logic trong bộ xử lý.
- Nhóm lệnh truy cập bộ nhớ: là các lệnh dùng để đọc và ghi dữ liệu giữa thanh ghi và bộ nhớ để truy xuất dữ liệu ở nhiều kích thước như byte,

half-word, word kèm theo các phiên bản mở rộng có dấu, không dấu (lbu, lhu, ...).

- Nhóm lệnh điều khiển, rẽ nhánh: là các lệnh dùng để thay đổi luồng thực thi chương trình theo điều kiện logic giữa 2 thanh ghi (B-type) để so sánh hai thanh ghi và thay đổi giá trị PC (Program Counter) nếu điều kiện đúng, hỗ trợ thực hiện vòng lặp và câu lệnh điều kiện trong chương trình.
- Nhóm lệnh điều khiển nhảy: là các lệnh dùng để điều khiển dòng chảy chương trình một cách trực tiếp hoặc thông qua thanh ghi (jal, jalr) giúp nhảy đến vị trí khác trong chương trình và lưu địa chỉ quay lại, hỗ trợ cho việc gọi hàm và điều hướng chương trình.

### 2.2.3. Kỹ thuật pipeline 5 tầng

Trong thiết kế bộ xử lý theo kiến trúc RISC-V, kỹ thuật pipeline 5 tầng là một phương pháp phổ biến nhằm tăng hiệu suất xử lý lệnh bằng cách chia quá trình thực thi một lệnh thành nhiều giai đoạn độc lập và xử lý song song. Một kiến trúc RV32I pipeline được mô tả thông qua Hình 2.3:



Hình 2.3 Mô hình kỹ thuật pipeline 5 tầng của bộ xử lý RV32I

Mỗi lệnh trong RV32I được chia thành năm giai đoạn chính:

- IF – Instruction Fetch (Nạp lệnh): Trong giai đoạn này, địa chỉ lệnh hiện tại được lấy từ thanh ghi Program Counter (PC) và lệnh tương ứng được đọc từ bộ nhớ chỉ lệnh (Instruction Memory). Sau đó, PC được cập nhật để trỏ đến lệnh tiếp theo.
- ID – Instruction Decode (Giải mã lệnh và đọc thanh ghi): Bộ giải mã sẽ phân tích lệnh vừa nạp để xác định loại lệnh và thông tin điều khiển

cần thiết. Đồng thời, các thanh ghi nguồn (rs1 và rs2) được đọc từ tập thanh ghi. Nếu lệnh sử dụng immediate, giá trị immediate cũng được trích xuất từ mã lệnh.

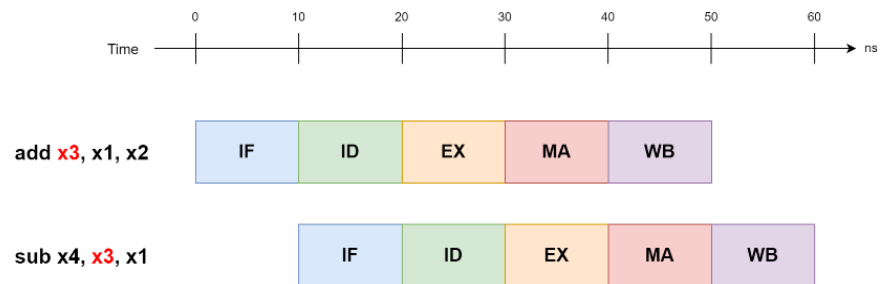
- EX – Execute / (Thực thi): ALU (Arithmetic Logic Unit) thực hiện phép toán tương ứng: phép cộng, trừ, so sánh, hoặc dịch bit. Với lệnh load/store, ALU sẽ tính địa chỉ bộ nhớ cần truy cập. Với lệnh nhảy, địa chỉ đích cũng được tính ở giai đoạn này.
- MEM – Memory Access (Truy cập bộ nhớ): Nếu lệnh yêu cầu đọc (lw, lh, lb,...) hoặc ghi (sw, sh, sb) bộ nhớ, việc truy cập sẽ diễn ra tại đây. Với các lệnh không yêu cầu thao tác bộ nhớ (như add, sub), giai đoạn này có thể bị bỏ qua.
- WB – Write Back (Ghi kết quả về thanh ghi): là bước cuối cùng hoàn tất chu kỳ xử lý của một lệnh, có nhiệm vụ ghi kết quả vào thanh ghi đích (rd).

#### 2.2.4. Các loại xung đột trong RISC-V

Trong kỹ thuật pipeline, các xung đột có thể xảy ra khi các lệnh được thực thi đồng thời tại các giai đoạn khác nhau và có liên quan đến nhau dẫn đến các tình huống làm gián đoạn dòng chảy liên tục của pipeline và nếu không được xử lý thích hợp, sẽ gây lỗi logic hoặc làm giảm hiệu suất của hệ thống. Trong kiến trúc RISC-V, đặc biệt với pipeline 5 tầng, các xung đột thường được chia thành ba loại chính: Structural Hazard, Data Hazard và Control Hazard.

- Structural Hazard (xung đột cấu trúc): xảy ra khi hai lệnh cùng lúc cần truy cập một tài nguyên phần cứng duy nhất nhưng hệ thống không đủ tài nguyên để xử lý đồng thời, nhưng trong thiết kế này tách riêng bộ nhớ dữ liệu (DMEM) và bộ nhớ lệnh (IMEM) do đó vấn đề này sẽ không xảy ra.
- Data Hazard (xung đột dữ liệu): xảy ra khi một lệnh phụ thuộc vào kết quả của lệnh trước đó, nhưng kết quả đó chưa sẵn sàng tại thời điểm lệnh sau cần sử dụng. Trong đó có 2 nhóm lệnh có thể dẫn đến xung đột là nhóm lệnh R-type và nhóm lệnh LOAD:

- RAW (Read After Write): khi một lệnh cần đọc một thanh ghi mà lệnh trước đó vẫn đang trong quá trình ghi vào. Hình 2.4 mô tả xung đột dữ liệu khi lệnh sub cần giá trị được lưu ở thanh ghi x3, nhưng giá trị thanh ghi này đang được thực thi ở lệnh add x3, x1, x2 dẫn đến việc truy xuất sai giá trị hoặc giá trị cũ tại thời điểm lệnh sub cần dữ liệu (tầng EX).

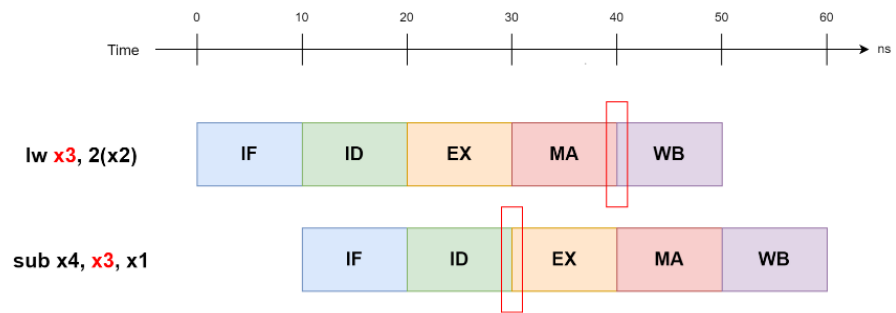


Hình 2.4 Xung đột dữ liệu RAW

Do đó chúng ta phải sử dụng giải quyết bằng cách stall lệnh tiếp theo này và chờ cho đến khi dữ liệu được ghi vào thanh ghi, hoặc chúng ta phải dùng phương pháp ‘forwarding’ dữ liệu để lấy dữ liệu trực tiếp từ lệnh trước. Hai biện pháp này đều giải quyết được vấn đề nhưng sử dụng ‘forwarding’ sẽ giúp chúng ta không tốn thêm chu kỳ như phương pháp stall lệnh, nhưng bù lại sẽ cần thêm logic xử lý để lựa chọn giá trị forwarding cho lệnh.

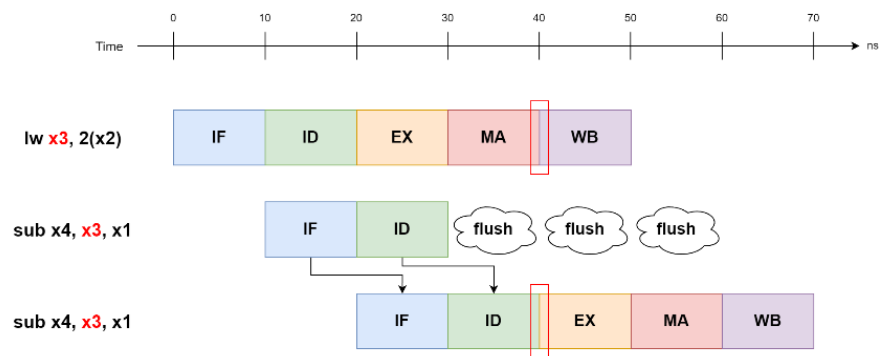
- Load: là một tình huống đặt biệt của data hazard khi thanh ghi cần đọc chưa thể cập nhật đúng giá trị khi lệnh trước đó là lệnh load bộ nhớ (Hình 2.5). Khi lệnh sub cần đọc giá trị thanh ghi x3, lúc này thanh ghi x3 đang được load từ bộ nhớ DMEM ở lệnh lw phía trên, dẫn đến x3 sẽ lấy giá trị chưa được cập nhật mới nhất.





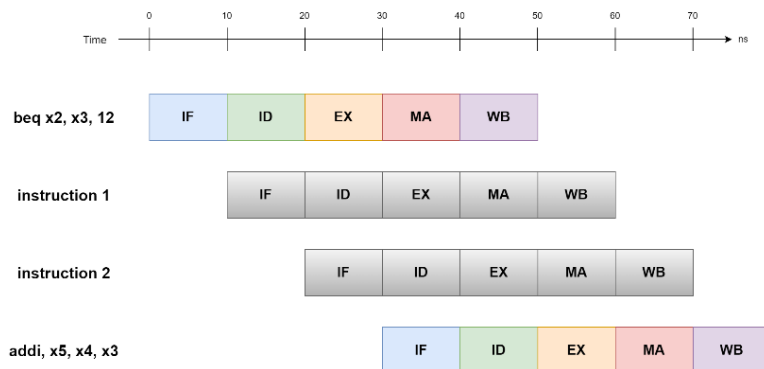
Hình 2.5 Xung đột dữ liệu ở lệnh Load

Để khắc phục tình trạng này, lệnh sub sẽ stall tầng IF, ID một chu kì và flush tầng EX trở đi cho đến khi có kết quả được tải từ DMEM. Lúc này chúng ta sẽ dùng phương pháp forwarding dữ liệu để truyền kết quả lệnh lw ở tầng MA đến trực tiếp tầng EX của lệnh sub (Hình 2.6).



Hình 2.6 Phương pháp xử lý xung đột ở lệnh load

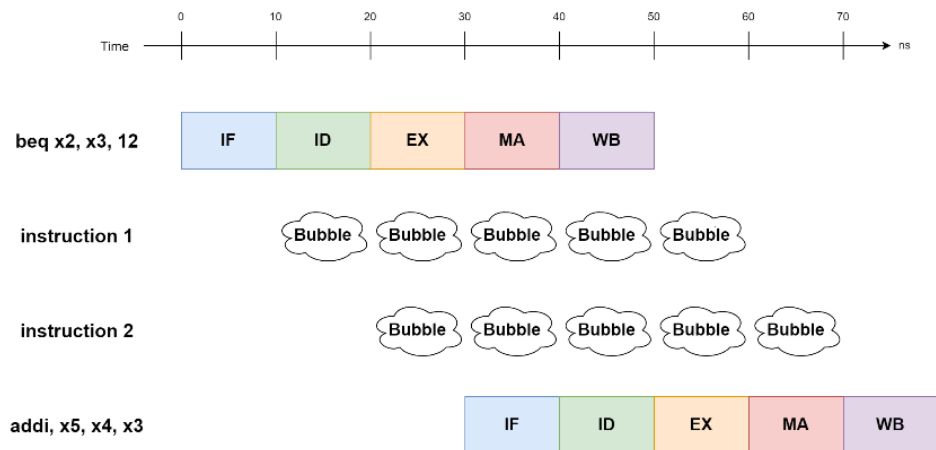
- Control Hazard (xung đột điều khiển): xảy ra khi pipeline không biết lệnh tiếp theo là gì do phải đợi kết quả của một lệnh rẽ nhánh (beq, bne, jal, ...) dẫn đến việc nạp sai lệnh vào pipeline (Hình 2.7).



Hình 2.7 Xung đột điều khiển

Có 2 phương pháp chính để giải quyết vấn đề này đó chính là stall lệnh phía sau và dự đoán lệnh thực thi:

- Stall: lệnh phía sau chờ cho đến khi có kết quả so sánh và tính toán xong giá trị nhảy đến (tầng execute). Đây là phương pháp đơn giản nhất nhưng sẽ tốn 2 chu kì thực thi để có kết quả có rẽ nhánh hay không.



Hình 2.8 Phương pháp xử lý xung đột điều khiển

- Branch Prediction: phương pháp này dự đoán nhánh lệnh có khả năng được thực thi nhất bằng các thuật toán như BHT (Branch History Table), BTB (Branch Target Buffer), điều này dẫn đến việc phần cứng trở nên phức tạp và tăng mức tiêu thụ tài nguyên. Nếu dự đoán sai, chương trình sẽ khôi phục lại trạng thái đầu và thực thi tương tự như dùng stall (mất 2 chu kì), nhưng nếu đoán đúng sẽ cải thiện được hiệu năng khi không bị mất một chu kì nào.

## **2.3. Sơ lược về chuẩn mã hoá video H.264/AVC**

### **2.3.1. Giới thiệu**

Advanced Video Coding (AVC/H.264) [1] là một chuẩn nén video được phát triển như bước kế thừa của các chuẩn trước đó như MPEG-2 và MPEG-4 Part 2. H.264 được chuẩn hóa bởi Tổ chức Tiêu chuẩn hóa Quốc tế ISO/IEC và Liên minh Viễn thông Quốc tế ITU-T, dưới sự hợp tác của hai nhóm chuyên môn là MPEG và VCEG, và chính thức công bố vào năm 2003.

Mục tiêu chính của H.264 là cung cấp khả năng nén hiệu quả cao trong khi vẫn duy trì chất lượng hình ảnh và giảm băng thông truyền dẫn. Điểm nổi bật trong kiến trúc này là dùng biến đổi số nguyên (Integer Transform) thay vì biến đổi DCT giúp giảm độ phức tạp và dễ triển khai trên FPGA. H.264 còn áp dụng các cơ chế dự đoán nội khung đa hướng (4–9 hướng) và mã hóa entropy theo ngữ cảnh như CAVLC hoặc CABAC, giúp tối ưu hóa bitrate mà vẫn giữ độ trễ thấp – điều rất quan trọng đối với các hệ thống xử lý tín hiệu thời gian thực.

Mặc dù H.265/HEVC có khả năng nén vượt trội, nhưng độ phức tạp kiến trúc của nó gây nhiều khó khăn khi triển khai trên SoC hoặc FPGA nhúng. Việc sử dụng CTU 64x64, 35 hướng dự đoán nội khung, cùng mã hóa entropy CABAC bắt buộc dẫn đến việc thiết kế pipeline khó hơn và tiêu tốn nhiều tài nguyên hơn – khiến thiết kế phần cứng phức tạp, khó đạt timing và yêu cầu bộ nhớ lớn. Trong khi đó, H.264 sử dụng macroblock 16x16, hỗ trợ CAVLC dễ hiện thực, cùng cấu trúc biến đổi và lượng tử hóa đơn giản hơn, giúp dễ dàng tích hợp vào pipeline xử lý tuần tự, phù hợp với các hệ thống nhúng giới hạn tài nguyên nhưng yêu cầu hiệu suất cao.

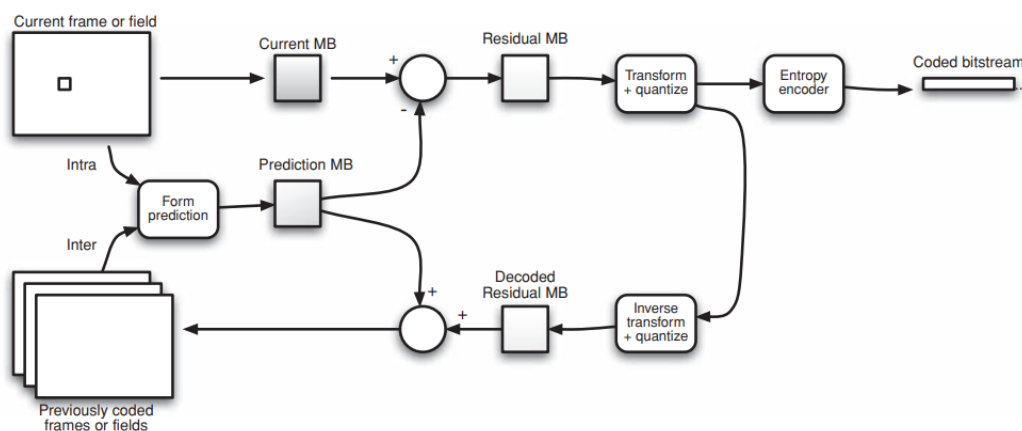
Bảng 2.1 Bảng so sánh các chuẩn nén video phổ biến

Tiêu chí	MPEG-2	MPEG-4 Part 2	H.264/AVC	H.265/HEVC
Năm ra đời	1995	1999	2003	2013
Khối xử lý	16x16	16x16	16x16	CTU (64x64)
Hướng dự đoán	1 hướng	2 hướng	4-9 hướng	35 hướng
Biến đổi	DCT 8x8	DCT 8x8	Integer 4x4/8x8	DCT (4x4-32x32)
Lượng tử hoá	Tuyến tính	Tuyến tính	QP phi tuyến tính	Adaptive QP phức tạp
Entropy Coding	Huffman	VLC	CAVLC/CABAC	Chỉ CABAC
Deblocking filter	Không có	Đơn giản	Có	Có + SAO
Tỉ lệ nén (so với MPEG-2)	1x	~1.2x	~1.5-2x	~2.5x

### 2.3.2. Nguyên lý hoạt động của bộ mã hoá video H.264/AVC

Nguyên lý của bộ mã hoá là dự đoán và nén sự dư thừa vì các pixel lân cận thường có khuynh hướng giống nhau. Hình 2.9 mô tả một quá trình mã hoá tiêu chuẩn. Đầu vào là một khung hình hiện tại và được chia thành các đơn vị macro-block. Với mỗi macro-block, bộ mã hoá sẽ tạo một khối dự đoán dựa trên thông tin các khối lân cận (intra) hoặc từ các khung hình đã được mã hoá trước đó (inter). Phần dư giữa khối hiện tại và khối dự đoán sẽ được biến đổi và lượng tử hoá để giảm kích thước dữ liệu. Phần dư lượng tử này được mã hoá entropy (bằng CAVLC hoặc CABAC) để tạo ra luồng bit đầu ra. Đồng thời, bộ mã hoá cũng thực hiện một vòng lặp giải mã nội bộ (inverse transform + cộng lại với prediction) nhằm tái tạo lại khối hình ảnh để lưu vào bộ tham chiếu, phục vụ cho quá trình dự đoán liên

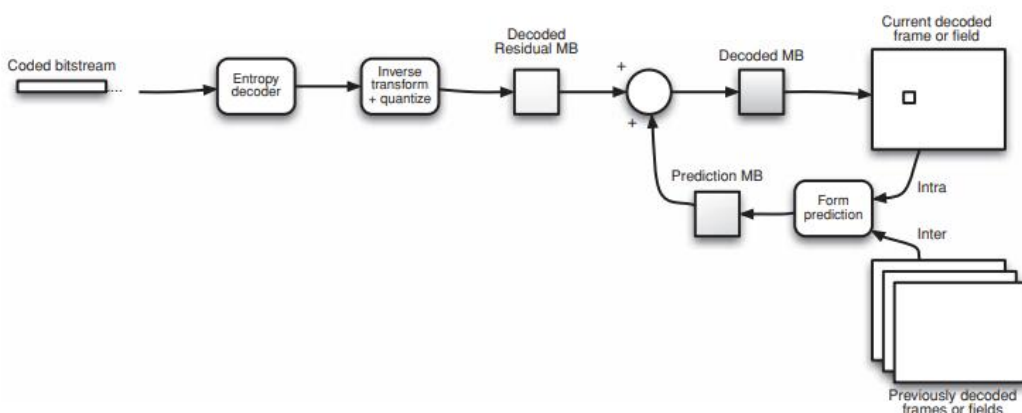
khung ở các macroblock tiếp theo. Cách làm này giúp giảm bitrate hiệu quả mà vẫn đảm bảo chất lượng khi giải mã.



Hình 2.9 Quá trình mã hoá video trong chuẩn H.264

Hình 2.10 minh hoạ một quá trình giải mã điển hình. Quá trình này bắt đầu từ coded bitstream được truyền vào bộ entropy decoder, nơi thực hiện giải mã CAVLC hoặc CABAC để thu được các hệ số lượng tử hoá. Các hệ số này tiếp tục đi qua bước giải lượng tử và nghịch biến đổi (Inverse transform + quantize) để khôi phục lại Residual MB — phần dư đã được mã hoá từ phía encoder.

Sau đó, bộ giải mã sẽ tái tạo khối dự đoán (Prediction MB) dựa trên thông tin nội khung (Intra) hoặc từ các khung hình đã giải mã trước đó (Inter). Prediction MB và Decoded Residual MB được cộng lại để tạo ra Decoded MB. Cuối cùng, các Decoded MB được lắp ghép để tái tạo thành khung hình đầy đủ.

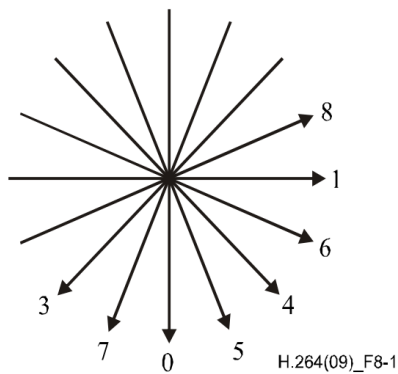


Hình 2.10 Quá trình giải mã video trong chuẩn H.264

### 2.3.2.1. Dự đoán (prediction)

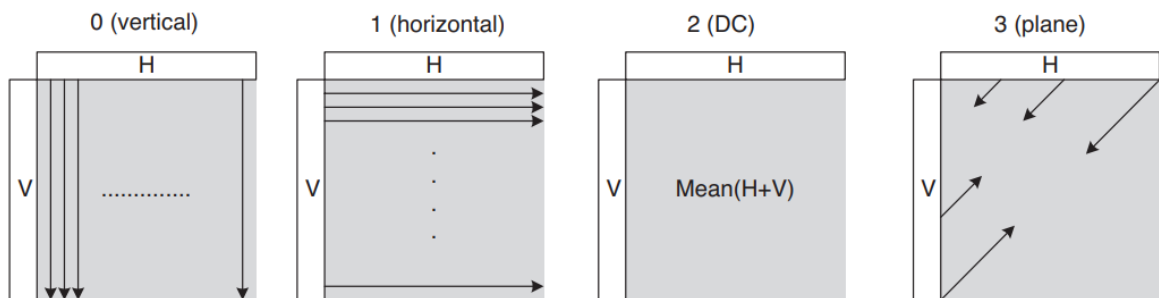
Dự đoán ở bộ H.264 sẽ dự đoán cả 3 kênh. Với kênh luma sẽ có 2 chế độ là intra 4x4 và 16x16, sau đó sẽ chọn ra chế độ có chi phí (RDO) thấp nhất để mã hoá, chỉ cần lưu lại phần dư và mode lựa chọn. Với phần chroma, sẽ chỉ có 4 chế độ dự đoán cho khối MB 8x8 cho mỗi thành phần Cr, Cb.

Dự đoán nội khung sử dụng các điểm ảnh lân cận đã mã hoá trong cùng khung hình để suy ra giá trị của khối hiện tại với kích thước linh hoạt từ 4x4, 16x16 pixel với nhiều hướng khác nhau dựa trên vị trí của khối hiện tại, tối đa 9 hướng với thành phần màu luma (hình 2.11) và 4 hướng đối với thành phần màu chroma (hình 2.12).



Intra4x4PredMode[ luma4x4BlkIdx ]	Name of Intra4x4PredMode[ luma4x4BlkIdx ]
0	Intra_4x4_Vertical (prediction mode)
1	Intra_4x4_Horizontal (prediction mode)
2	Intra_4x4_DC (prediction mode)
3	Intra_4x4_Diagonal_Down_Left (prediction mode)
4	Intra_4x4_Diagonal_Down_Right (prediction mode)
5	Intra_4x4_Vertical_Right (prediction mode)
6	Intra_4x4_Horizontal_Down (prediction mode)
7	Intra_4x4_Vertical_Left (prediction mode)
8	Intra_4x4_Horizontal_Up (prediction mode)

Hình 2.11 Các chế độ dự đoán nội khung cho thành phần Luma (Y)



Hình 2.12 Các chế độ dự đoán nội khung cho thành phần Chroma (Cr, Cb)

Tại mỗi khối, bộ mã hoá sẽ thử các chế độ dự đoán khả dụng để chọn ra chế độ tối ưu theo tiêu chí Rate-Distortion Optimization (RDO):  $J = D + \lambda \cdot R$ . Trong đó, D là độ biến dạng, R là số bit cần mã hóa, và  $\lambda$  là hệ số điều chỉnh phụ thuộc vào mức lượng tử hóa (QP).

### 2.3.2.2. Biến đổi (transform)

Mục tiêu chính của phép biến đổi trong mã hoá video là cô lập và biểu diễn rõ ràng các thành phần thông tin quan trọng, thường nằm ở dải tần số thấp (thành phần DC), đồng thời tách biệt hoặc làm suy giảm các thành phần ít quan trọng hơn như chi tiết nhỏ, nhiễu cao tần hoặc thông tin khó nhận biết bằng mắt người. Nhờ đó, năng lượng của tín hiệu sau biến đổi sẽ tập trung chủ yếu ở một số ít hệ số đầu, tạo điều kiện thuận lợi cho quá trình lượng tử hoá và mã hoá, giúp cải thiện hiệu quả nén mà vẫn đảm bảo chất lượng hình ảnh.

Trong đề tài này, bộ mã hoá sử dụng giải thuật Integer Transform thay vì phép biến đổi DCT thuần túy. Integer Transform là một biến thể gần đúng của DCT, được thiết kế để thực thi hiệu quả trên phần cứng nhúng hoặc FPGA, nơi mà phép tính dấu phẩy động (floating-point) là tốn kém. Biến đổi được thực hiện hoàn toàn trên các số nguyên, cho phép giảm độ phức tạp phần cứng và tiết kiệm tài nguyên tính toán, mặc dù có thể chấp nhận một sai số nhỏ về mặt chất lượng so với DCT chuẩn.

Phép biến đổi này được áp dụng trên các khối kích thước  $4 \times 4$ , với công thức cụ thể như sau:

$$Y = C \cdot X \cdot C^T \quad (2.1)$$

Với ma trận  $C = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix}$

Trong đó:

- X: ma trận trước biến đổi (ma trận dư thừa -residual).
- Y: ma trận sau biến đổi.
- C: ma trận biến đổi ( $C^T$  là ma trận chuyển vị của C).

### 2.3.2.3. Lượng tử hoá (quantization)

Mục tiêu của lượng tử hoá là giảm bớt độ chính xác dựa vào tham số nén QP bằng các chia ma trận sau biến đổi (TF) cho ma trận lượng tử hoá. Với QP càng cao, tỉ lệ nén càng lớn nhưng bù lại, chất lượng sẽ giảm.

Mỗi giá trị QP được chia thành 6 cấp độ (QP%6), mỗi cấp độ sẽ có 1 tập ma trận lượng tử hoá  $m()$  được ánh xạ theo 3 nhóm vị trí trên ma trận 4x4 (hình 2.13):

QP	$m(r, 0):$ $M_{f4}$ positions (0,0), (0,2), (2,0), (2,2)	$m(r, 1):$ $M_{f4}$ positions (1,1), (1,3), (3,1), (3,3)	$m(r, 2):$ Remaining $M_{f4}$ positions
0	13107	5243	8066
1	11916	4660	7490
2	10082	4194	6554
3	9362	3647	5825
4	8192	3355	5243
5	7282	2893	4559

Hình 2.13 Bảng tham chiếu cho ma trận lượng tử hoá

Sau khi có các hệ số  $m()$ , ma trận lượng tử hoá  $M_{f4}$  được tạo dựa trên bảng:

$$M_{f4} = \begin{bmatrix} m(QP, 0) & m(QP, 2) & m(QP, 0) & m(QP, 2) \\ m(QP, 2) & m(QP, 1) & m(QP, 2) & m(QP, 1) \\ m(QP, 0) & m(QP, 2) & m(QP, 0) & m(QP, 2) \\ m(QP, 2) & m(QP, 1) & m(QP, 2) & m(QP, 1) \end{bmatrix}$$

Công thức lượng tử hoá như sau:

$$Y = \text{round} \left( TF \circ m(QP\%6, n) \cdot \frac{1}{2^{15 + \text{floor}(QP/6)}} \right) \quad (2.2)$$

Trong đó:

- Hàm round(): làm tròn về số nguyên gần nhất.
- Hàm floor(): lấy phần nguyên nhỏ nhất.
- Y: ma trận sau lượng tử hoá.
- TF: ma trận sau biến đổi.
- QP: tham số lượng tử hoá.
- M(): ma trận lượng tử hoá.

### 2.3.2.4. Mã hoá CAVLC



CAVLC (Context-Adaptive Variable Length Coding) sử dụng mã Huffman với độ dài biến đổi, đồng thời thích ứng theo ngữ cảnh của từng khối dữ liệu như số lượng hệ số khác 0, vị trí hệ số cuối và mẫu giá trị trước đó. Mục tiêu là làm giảm tối đa kích thước dữ liệu mà làm giảm tối đa sự mất mát dữ liệu dựa vào chế độ nén bằng cách mã hoá các thông số:

- TotalCoeffs: số lượng hệ số khác 0
- TotalZeros: tổng các số 0
- T1s (trailing-ones): các hệ số  $\pm 1$  ở cuối
- run\_before: số lượng số 0 giữa các TotalCoeffs
- Level: các giá trị hệ số

#### **2.4. Sơ lược Directed Memory Access (DMA)**

DMA là một cơ chế cho phép thiết bị ngoại vi như bộ mã hoá video, Uart, ... truyền dữ liệu trực tiếp đến bộ nhớ hoặc ngược lại mà không cần CPU can thiệp từng bước giúp giảm tải cho CPU vì CPU chỉ cần cấu hình, không cần xử lý từng byte, ngoài ra giúp tăng tốc độ truyền dữ liệu do truyền theo burst và tối ưu băng thông hệ thống do không cần chờ CPU phản hồi từng lệnh lw/sw.

#### **2.5. Sơ lược về giao thức AMBA AXI**

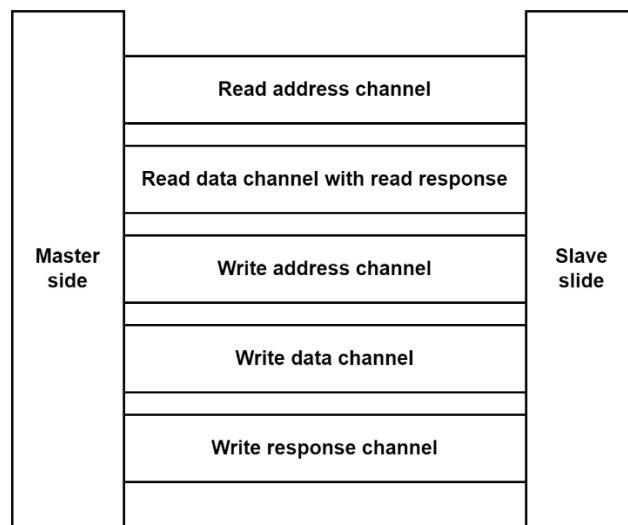
AMBA (Advanced Microcontroller Bus Architecture) là một chuẩn bus do ARM phát triển, dùng để kết nối các thành phần bên trong một SoC như CPU, DMA, bộ nhớ, ngoại vi... Đây là một trong những giao thức phổ biến trong hệ thống SoC vì có nhiều ưu điểm vượt trội, trong đó nổi bật như:

- Tách riêng pha truyền địa chỉ (address), thông tin điều khiển (control) với pha truyền dữ liệu (data).
- Hỗ trợ việc trao đổi dữ liệu unaligned và hỗ trợ write strobe.
- Hỗ trợ transaction theo cơ chế burst, chỉ cần phát địa chỉ đầu tiên của burst.
- Tách riêng kênh dữ liệu đọc (read) và kênh dữ liệu ghi (write).
- Hỗ trợ phát nhiều địa chỉ chồng lấn (multiple outstanding).
- Các transaction có thể hoàn thành không theo thứ tự (out-of-order)/

- Cho phép chèn thêm các tầng thanh ghi giúp timing của BUS nói riêng và của hệ thống tốt hơn.

Cấu trúc của một giao diện AXI gồm 5 kênh được mô tả qua hình 2.14:

- Kênh địa chỉ đọc AR: truyền thông tin địa chỉ và thông tin điều khiển của một transaction đọc từ master đến slave.
- Kênh dữ liệu đọc R: truyền dữ liệu đọc và thông tin response của một transaction từ slave đến master.
- Kênh địa chỉ ghi AW: truyền thông tin địa chỉ và thông tin điều khiển của một transaction ghi từ master đến slave.
- Kênh dữ liệu ghi W: truyền dữ liệu ghi từ master đến slave.
- Kênh phản hồi ghi B: truyền thông tin response của một transaction ghi từ slave đến master.

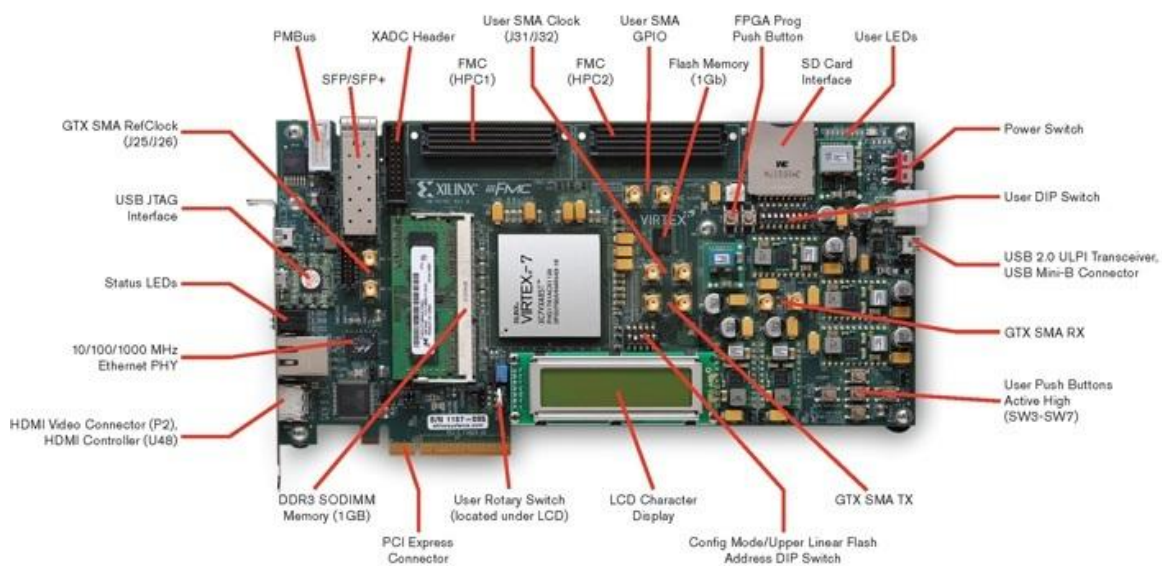


Hình 2.14 Minh họa 5 kênh giao tiếp của giao thức AXI

## 2.6. Sơ lược về board Xilinx FPGA Virtex-7 VC707

Xilinx VC707 là bo mạch phát triển sử dụng FPGA Virtex-7 XC7VX485T thuộc dòng high-performance của Xilinx, với ~485K LUTs, ~607K Flip-Flops, 37Mb bộ nhớ BRAM và 2800 DSP slices. Đây là nền tảng lý tưởng cho các ứng dụng cần xử lý tốc độ cao như mã hoá video, truyền thông mạng, hoặc phát triển hệ thống SoC tùy chỉnh. VC707 hỗ trợ giao tiếp bộ nhớ DDR3 1GB (72-bit có ECC), PCIe Gen2 x8, và một loạt giao tiếp ngoại vi như Gigabit Ethernet, USB-JTAG, UART, và FMC HPC (cho phép mở rộng với camera, ADC, DAC...).

Với thiết kế phần cứng mạnh mẽ và khả năng tương thích hoàn toàn với hệ sinh thái Vivado Design Suite, VC707 cho phép người dùng triển khai và kiểm thử các hệ thống phức tạp theo kiến trúc AXI với tốc độ cao. Các IP lõi như AXI DMA, DDR3 Memory Controller. Hình 2.15 minh họa board Xilinx Virtex-7 VC707 với các chú thích chức năng kèm theo.



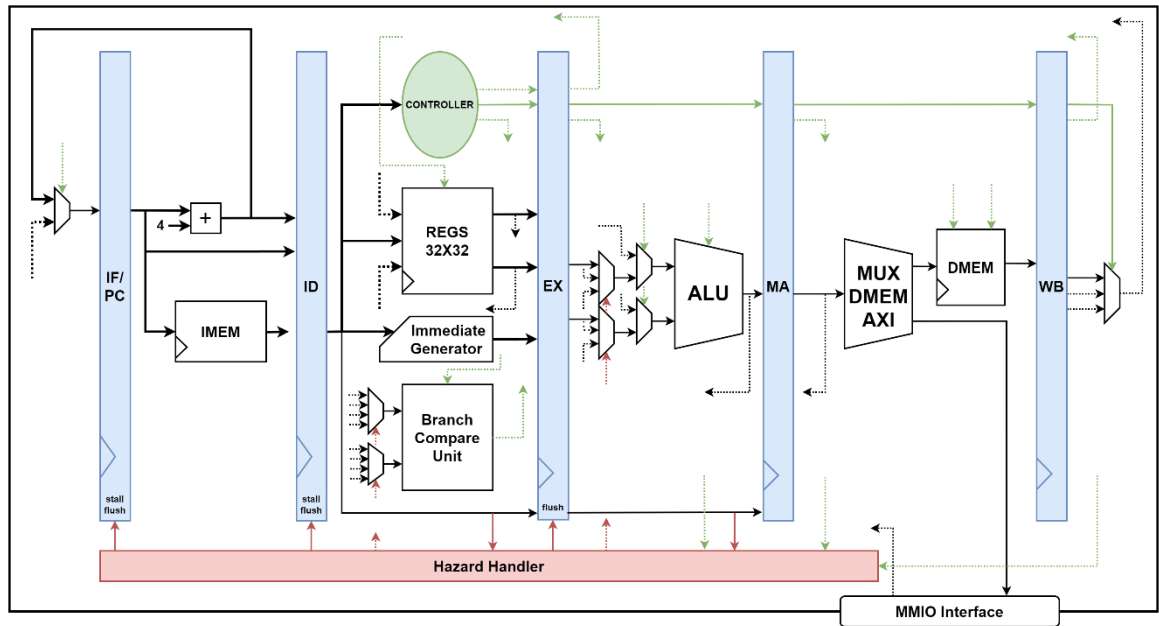
Hình 2.15 Minh họa board Virtex 7 – VC707

## Chương 3. HIỆN THỰC THIẾT KẾ

### 3.1. Thiết kế bộ xử lý RISC-V RV32I

#### 3.1.1. Kiến trúc tổng quát

Bộ xử lý có thiết kế pipeline 5 tầng với luồng dữ liệu và điều khiển được tổ chức rõ ràng, có thể xử lý hazard và giao tiếp với hệ thống ngoại vi thông qua MMIO Interface. Kiến trúc bộ xử lý RV32I được mô tả tổng quát qua Hình 3.1.



Hình 3.1 Kiến trúc RV32I pipeline

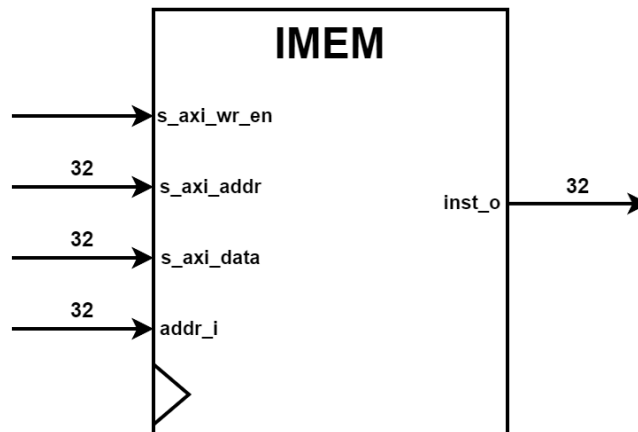
MMIO Interface dùng để giao tiếp với ngoại vi thông qua giao diện AXI-Lite, đây là một kỹ thuật ánh xạ các thanh ghi điều khiển của các ngoại vi vào không gian địa chỉ bộ nhớ của bộ xử lý. Thay vì sử dụng các lệnh đặc biệt để truy xuất thiết bị, chúng ta chỉ cần đọc/ghi vào địa chỉ cụ thể trong bộ nhớ, thao tác này tương đương với việc điều khiển phần cứng ngoại vi.

### 3.1.2. Thiết kế khối Datapath pipeline

#### 3.1.2.1. Tầng Fetch (IF)

Tầng IF chứa bộ đếm chương trình PC và logic cập nhật địa chỉ lệnh, đồng thời truy xuất bộ IMEM để truy xuất lệnh dựa vào giá trị PC. Tầng này có các tín hiệu stall và flush để xử lý các branch hazard và điều khiển luồng.

Chi tiết khối IMEM được mô tả qua Hình 3.2:



Hình 3.2 Khối IMEM.

Khối này sẽ có nhiệm vụ lấy lệnh (inst\_o) từ địa chỉ (addr\_i) được truy xuất từ giá trị thanh ghi PC. Ngoài ra, khối này còn có nhiệm vụ lưu lệnh (s\_axi\_data) vào địa chỉ (s\_axi\_addr) khi tín hiệu s\_axi\_wr\_en tích cực. Bộ tính hiệu này được dùng khi nạp chương trình ban đầu vào IMEM bằng MicroBlaze khi nạp xuống FPGA hoặc khi mô phỏng trên Vivado và dùng Xilinx AXI VIP để giả lập MicroBlaze.

#### 3.1.2.2. Tầng Decode (ID)

Tầng ID có nhiệm vụ giải mã lệnh và đọc thanh ghi nguồn từ bộ REG 32x32, đồng thời tạo giá trị tức thời Imm qua bộ Immediate Generator. Bên cạnh đó, bộ Branch Compare Unit cũng được xử lý sớm ở tầng này để tính toán các quyết định rẽ nhánh sớm hơn 1 chu kì. Ngoài ra, tầng này còn tương tác với bộ Controller để tạo các tín hiệu điều khiển pipeline cho cả bộ xử lý.

### 3.1.2.3. Tầng Execute (EX)

Tầng này thực hiện các phép toán số học, logic và tính toán địa chỉ bộ nhớ tích hợp các bộ MUX để chọn các giá trị phù hợp thông qua điều khiển của bộ Forwarding Unit. Chức năng của bộ ALU được mô tả thông qua bảng 3.1:

Bảng 3.1 Bảng mô tả chức năng bộ ALU

ALU_SEL	Công thức	Mô tả
0000	$A \& B$	Phép AND bit-wise
0001	$A \mid B$	Phép OR bit-wise
0010	$A + B$	Phép cộng số học
0011	$A - B$	Phép trừ số học
0100	$(A < B) ? 1 : 0$	So sánh có dấu
0101	$!(A \mid B)$	Phép NOR bit-wise
0110	$B \ll 12$	Load Upper Immediate
0111	$A \wedge B$	Phép XOR bit-wise
1000	$A \ll B[4:0]$	Dịch trái logic
1001	$A \gg B[4:0]$	Dịch phải logic
1010	$A + (B \ll 12)$	Add Upper Immediate to PC
1011	$(A < B) ? 1 : 0$	So sánh không dấu
1100	Dịch có bảo toàn dấu (sra)	Dịch phải số học
1111	no-op	Kết quả ALU mặc định bằng 0

#### 3.1.2.4. Tầng Memory (MA)

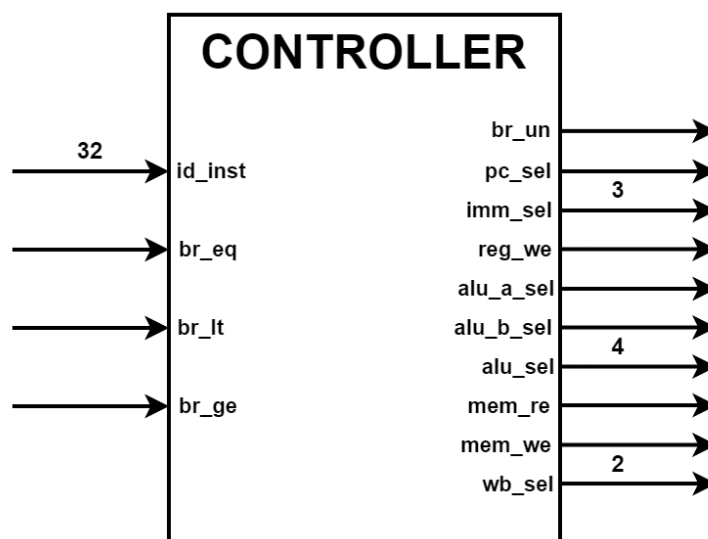
Tầng này có nhiệm vụ giao tiếp với bộ nhớ dữ liệu, cho phép lựa chọn truy xuất bộ nhớ nội (DMEM) hoặc thông qua ngoại vi MMIO (qua giao thức AXI-Lite) bằng các sử dụng bộ chọn MUX\_DMEN\_AXI. Tín hiệu khởi tạo giao dịch axi (axi\_init) cũng xuất phát ở tầng MA này, nhưng dữ liệu đọc từ ngoại vi về sẽ được lưu trực tiếp vào thanh ghi bên trong bộ REG\_32x32 ở tầng ID, do đó sẽ không xảy ra xung đột cấu trúc ở bộ nhớ nội DMEM.

#### 3.1.2.5. Tầng Write-Back (WB)

Tầng này có nhiệm vụ ghi kết quả từ ALU, từ bộ nhớ hoặc giá trị PC tiếp theo về bộ nhớ REG\_32x32 tùy vào lệnh được quyết định bởi bộ Controller.

#### 3.1.3. Thiết kế khối Controller

Bộ controller là một mạch tổ hợp có nhiệm vụ tính toán các tín hiệu điều khiển cho lệnh được nhận từ tầng IF trong suốt quá trình pipeline của lệnh này. Hình 3.3 mô tả input/output khối Controller trong đề tài này.



Hình 3.3 Khối Controller của bộ xử lý RV32I

Trong đó các tín hiệu của bộ Controller được mô tả qua bảng 3.2 bên dưới:

Bảng 3.2 Bảng mô tả chức năng khối Controller

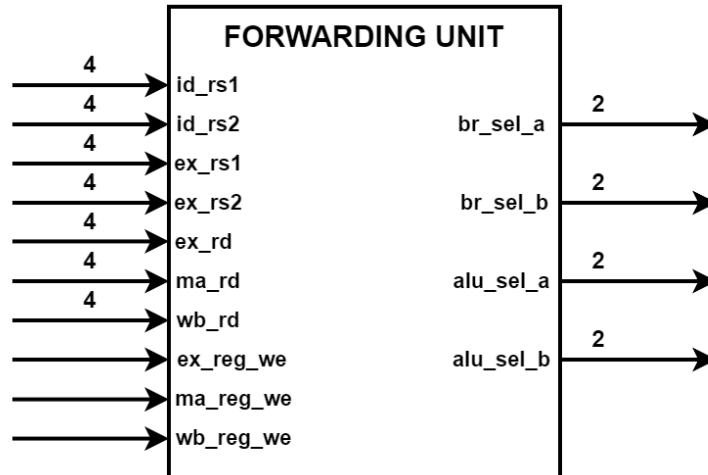
Tín hiệu	I/O	Width	Chức năng
Id_inst	Input	32	Lệnh nhận từ tầng IF
Br_eq	Input	1	Kết quả so sánh bằng từ khối Branch Compare
Br_lt	Input	1	Kết quả so sánh $rs1 < rs2$ từ khối Branch Compare
Br_ge	Input	1	Kết quả so sánh $rs1 \geq rs2$ từ khối Branch Compare
Br_un	Output	1	Chọn so sánh nhánh không dấu (BLTU, BGEU)
Pc_sel	Output	1	Chọn giá trị cho thanh ghi lệnh PC tiếp theo
Imm_sel	Output	3	Chọn cách lấy số tức thời dựa vào trường opcode
Reg_we	Output	1	Cho phép ghi vào bộ nhớ REG_32x32 hay không
Alu_a_sel	Output	1	Chọn đầu vào ALU A từ thanh ghi hoặc giá trị PC
Alu_b_sel	Output	1	Chọn đầu vào ALU B từ thanh ghi hoặc giá trị PC
Alu_sel	Output	4	Chọn chức năng cho ALU
Mem_we	Output	1	Cho phép ghi vào bộ nhớ DMEM hoặc MMIO
Mem_re	Output	1	Cho phép đọc từ bộ nhớ DMEM
Wb_sel	Output	2	Chọn giá trị lưu về REG_32x32



### 3.1.4. Bộ xử lý xung đột

#### 3.1.4.1. Khối Forwarding Unit

Nhiệm vụ chính của khối này là xác định và lựa chọn đầu vào thích hợp cho khối so sánh nhánh (Branch Compare) ở tầng ID và khối ALU ở tầng EX, đảm bảo rằng các toán hạng được sử dụng là giá trị mới nhất – ngay cả khi giá trị đó chưa kịp ghi về thanh ghi. Hình 3.4 mô tả input/output của khối Forwarding Unit.

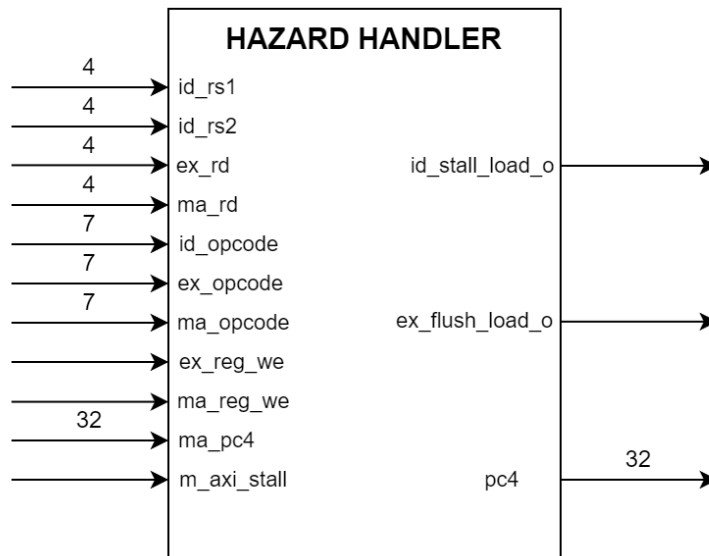


Hình 3.4 Khối Forwarding Unit

Cụ thể, khối Forwarding Unit sẽ so sánh các thanh ghi nguồn (rs1, rs2) của lệnh hiện tại với các thanh ghi đích (rd) của các lệnh đang nằm ở các tầng sau (EX, MA, WB). Nếu phát hiện sự trùng khớp và có tín hiệu ghi reg\_we tương ứng được kích hoạt, điều đó có nghĩa là giá trị mà lệnh hiện tại cần sử dụng đang được tính toán hoặc chuẩn bị ghi về. Khi đó, khối Forwarding sẽ chủ động chuyển tiếp (forward) giá trị từ tầng pipeline phía trước về cho lệnh hiện tại, thay vì chờ đọc từ tập thanh ghi Register File 32x32. Điều này giúp tránh sử dụng dữ liệu cũ, đảm bảo tính đúng đắn của phép toán và tránh được việc chèn lệnh nop (stall), giúp duy trì hiệu suất hoạt động cao cho hệ thống.

### 3.1.4.2. Khối Hazard Handler

Khối này có nhiệm vụ phát hiện và xử lý các trường hợp load-use hazard trong pipeline của bộ xử lý. Khi một lệnh tải dữ liệu từ bộ nhớ (ví dụ: lw) đang nằm ở tầng EX hoặc MA và lệnh kế tiếp đang ở tầng ID cần sử dụng kết quả từ thanh ghi đích đó, thì có khả năng xảy ra hazard do dữ liệu chưa sẵn sàng. Khối này sẽ kiểm tra sự trùng khớp giữa các thanh ghi nguồn (rs1, rs2) của lệnh hiện tại với các thanh ghi đích (rd) ở các tầng sau, đồng thời xác định loại lệnh (nhánh hoặc thường) để đưa ra quyết định stall hoặc flush. Hình 3.5 mô tả input/output của khối Hazard Handler.



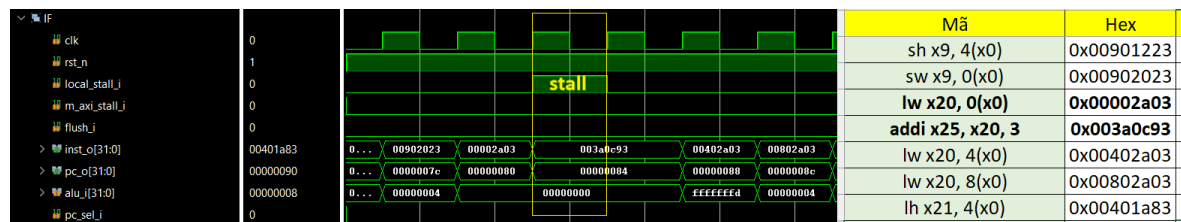
Hình 3.5 Khối Hazard Handler

Cụ thể, nếu phát hiện load-use hazard, khối sẽ kích hoạt tín hiệu `id_stall_load_o` để giữ tầng ID (không nạp lệnh mới) và `ex_flush_load_o` để xóa lệnh đang ở tầng EX. Ngoài ra, khi xảy ra giao dịch bộ nhớ thông qua AXI và xuất hiện tín hiệu `m_axi_stall_i`, khối này sử dụng latch để giữ giá trị PC tại tầng MA nhằm giữ địa chỉ lệnh kế tiếp. Trong thời gian giao dịch chưa hoàn tất, CPU sẽ tạm dừng và giữ nguyên giá trị PC. Ngay khi giao dịch kết thúc, giá trị PC sẽ được cập nhật lại ở tầng IF cho phép pipeline tiếp tục thực thi lệnh mới.

### 3.1.4.3. Một số waveform mô tả xung đột

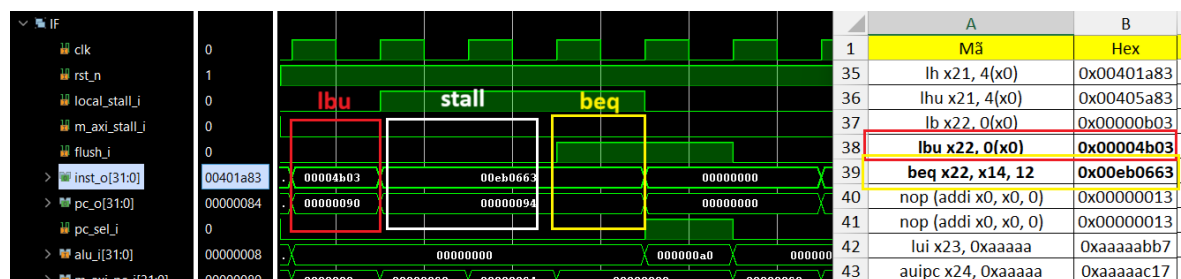
#### a. Xung đột dữ liệu

Hình 3.6 minh họa một xung đột khi lệnh R-type có sử dụng giá trị thanh ghi x20 và giá trị thanh ghi này đang được cập nhật ở lệnh lw phía trước, dẫn đến chúng ta cần phải stall một chu kì và sau đó forward dữ liệu trong thanh ghi này đến lệnh R-type hiện tại, tổng cộng tốn 1 chu kì cho xử lý trường hợp này.



Hình 3.6 Waveform minh họa xung đột load-hazard 1

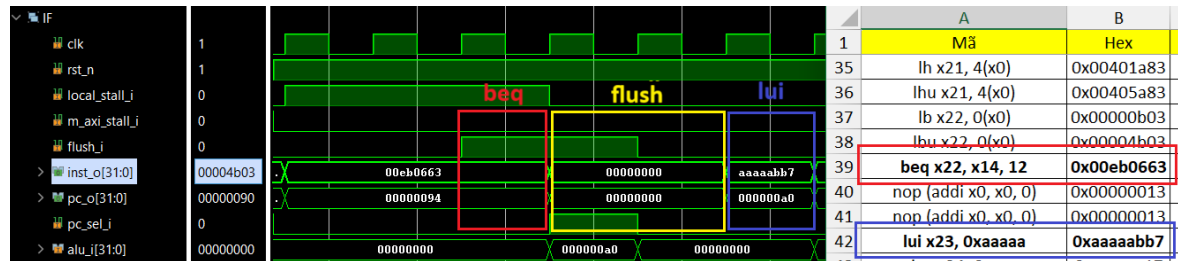
Hình 3.7 minh họa một xung đột khi lệnh B-type có sử dụng giá trị thanh ghi x22 và giá trị thanh ghi này đang được cập nhật ở lệnh lbu phía trước. Khác với lệnh R-type khi cần giá trị thanh ghi ở tầng EX, lệnh B-type cần giá trị thanh ghi ở tầng ID để tiến hành so sánh qua khối Branch Compare, nên bắt buộc chúng ta phải stall 2 chu kì và sau đó forward dữ liệu trong thanh ghi này đến lệnh B-type hiện tại, tổng cộng tốn 2 chu kì cho xử lý trường hợp này.



Hình 3.7 Waveform minh họa xung đột load-hazard 2

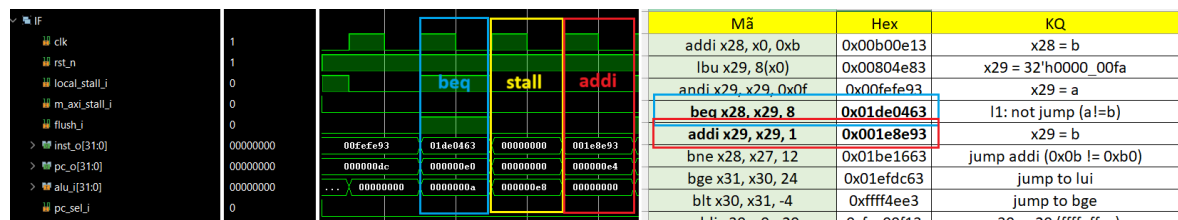
## b. Xung đột điều khiển

Hình 3.8 minh họa lệnh nhảy xảy ra và kết quả địa chỉ pc nhảy được tính ở tầng EX và chúng ta mất 2 chu kì để flush. Các lệnh nop chèn vào chỉ có ý nghĩa xem có lỗi trong quá trình nhảy hay không.



Hình 3.8 Waveform minh họa xung đột điều khiển 1

Hình 3.9 minh họa lệnh nhảy không xảy ra và kết quả so sánh được thực hiện ngay ở tầng ID và chúng ta chỉ mất 1 chu kì flush.



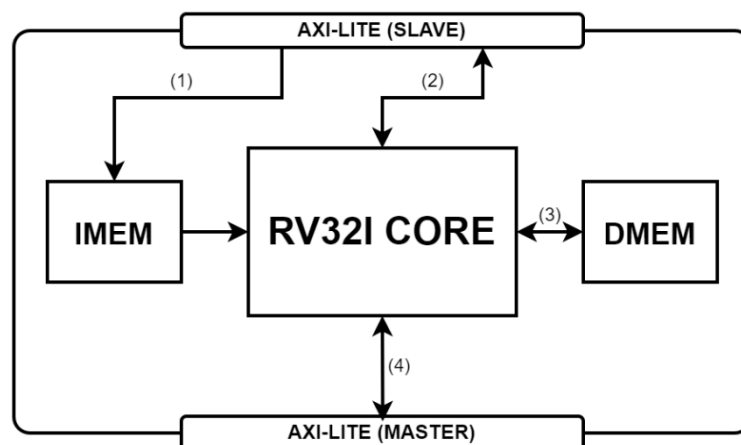
Hình 3.9 Waveform minh họa xung đột điều khiển 2

### 3.1.5. Thiết kế giao diện và đóng gói IP

#### 3.1.5.1. Kiến trúc bộ xử lý tích hợp các giao diện AXI-Lite

Do toàn bộ hệ thống (ngoại trừ bộ mã hoá video H.264) đều hoạt động trong cùng một miền xung nhịp (single clock domain) nên việc trao đổi dữ liệu giữa các thành phần sử dụng giao thức AXI-Lite không cần bộ đệm trung gian như FIFO hay các cơ chế đồng bộ tín hiệu. Các thanh ghi nội bộ được sử dụng để lưu trữ dữ liệu đọc/ghi từ giao diện slave giúp đảm bảo tính toàn vẹn và dễ dàng xử lý bên trong khối IP.

Bộ xử lý sẽ khởi tạo các giao dịch axi-lite-master khi có nhiệm vụ truy xuất ngoại vi thông qua các lệnh load/store. Các lệnh này không chỉ đọc/ghi dữ liệu mà còn có đóng vai trò ghi giá trị cấu hình vào không gian địa chỉ ngoại vi (memory-map I/O). Trong phạm vi đề tài này, các thiết bị ngoại vi bao gồm bộ mã hoá video H.264, bộ điều khiển DMA.

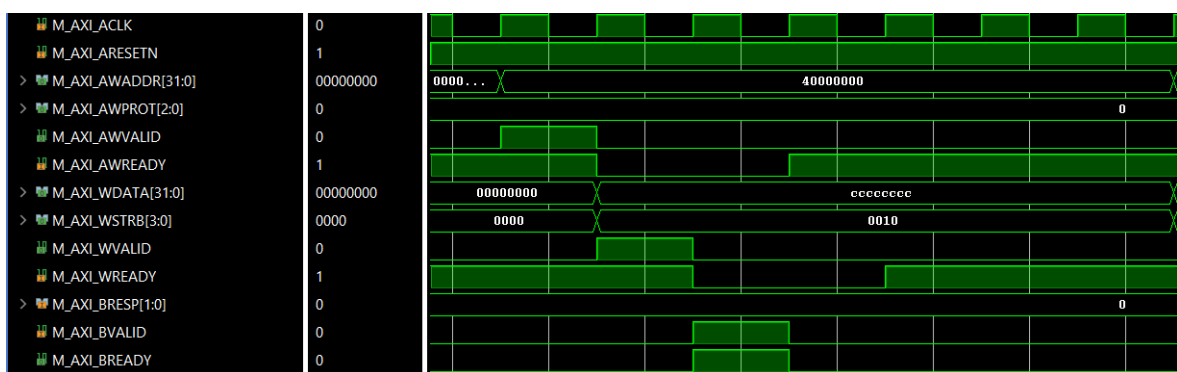


Hình 3.10 Kiến trúc IP bộ xử lý RV32I sau khi đóng gói

Hình 3.10 mô tả bộ xử lý RV32I sau khi đóng gói có tích hợp các giao diện AXI-Lite. Trong đó, phía Slave AXI-Lite có nhiệm vụ nhận yêu cầu đọc ghi từ bên ngoài (từ AXI VIP khi mô phỏng hoặc MicroBlaze khi triển khai trên FPGA). Mục đích của giao diện này là có thể ghi lệnh khởi tạo vào IMEM của bộ xử lý RV32I (1), đọc giá trị thanh ghi pc, lệnh hiện tại (2). Ngoài ra, phía Master AXI-Lite có nhiệm vụ khởi tạo giao dịch AXI khi có tín hiệu truy xuất bộ nhớ mà không phải là bộ nhớ nội DMEM (chi tiết vùng địa chỉ được mô tả ở bảng 3.6).

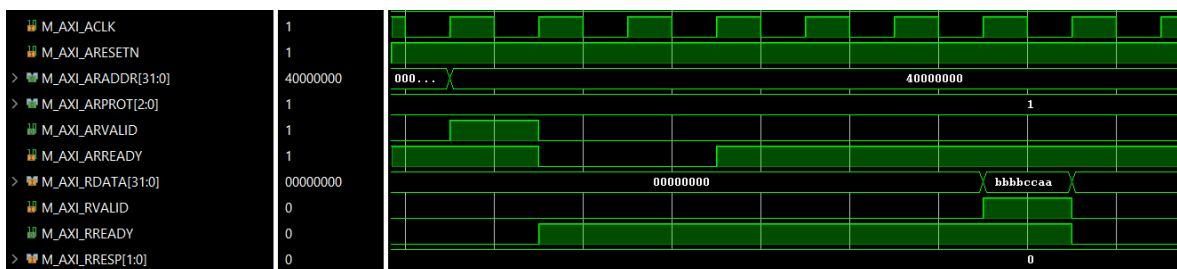
### 3.1.5.2. Cơ chế xử lý các lệnh store-load trong giao diện AXI-Lite

Khi có lệnh ghi ra ngoại vi, module này sẽ xử lý các lệnh sw/sh/sb thông qua việc điều khiển tín hiệu WSTRB. Với lệnh sw, tất cả 4 byte đều valid (strb=4'b1111). Lệnh sh chỉ kích hoạt 2 byte valid và tùy thuộc vào offset của lệnh ở bit thứ 2 (kèm điều kiện bit thứ nhất phải bằng 0). Tương tự với lệnh sb, chỉ kích hoạt 1 byte valid tương ứng với vị trí byte cần ghi. Hình 3.11 minh họa lệnh ghi ra ngoại vi sb x4, 1(x5), tín hiệu M\_AXI\_WSTRB=4'b0010 được xử lý chỉ có byte thứ 2 là valid.



Hình 3.11 Waveform minh ho giao dịch AXI ghi từ lệnh store-byte

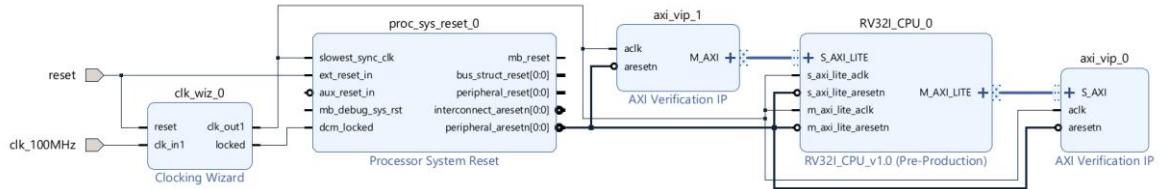
Khi có lệnh đọc từ ngoại vi, module này sẽ đọc toàn bộ 1 word (4 byte) với địa chỉ được word-align (2 bit thấp của địa chỉ được gán bằng 0). Sau khi nhận được dữ liệu từ slave, module sẽ trích xuất dữ liệu dựa trên loại lệnh như lw/lh/lhu/lb/lbu và read\_offset địa chỉ gốc. Hình 3.12 minh họa một giao dịch đọc AXI-Lite, trong đó kênh đọc luôn transfer đúng 1 word (32-bit) mỗi lần và không sử dụng tín hiệu AxSIZE như trong AXI-Full.



Hình 3.12 Waveform minh hoạ giao dịch AXI đọc từ lệnh load-half

### 3.1.5.3. Mô hình kiểm tra bộ xử lý sau đóng gói

Nhóm sử dụng 2 khối Xilinx AXI VIP với giao diện AXI-Lite để kiểm tra các chức năng của bộ xử lý. Block diagram mô tả CPU tích hợp các bộ kiểm tra được mô tả qua hình 3.13.



Hình 3.13 Mô hình kiểm tra bộ xử lý RV32I bằng AXI VIP

Trong đó, Master AXI-Lite đóng vai trò như một thiết bị khởi tạo giao tiếp, thực hiện các lệnh đọc/ghi với các burst đơn (single-beat transactions). Bên cạnh đó, Slave\_mem AXI-Lite được cấu hình như một thiết bị mô phỏng một mô hình bộ nhớ đơn giản, có nhiệm vụ phản hồi các yêu cầu đọc/ghi từ Master. Khi khởi tạo các agent này, bên trong sẽ tích hợp các protocol checker để kiểm tra các tín hiệu trong mỗi chu kỳ clk. Hình 3.14 mô tả một thông báo lỗi FATAL vi phạm protocol của AXI. Vấn đề này do truy xuất lệnh trong vùng IMEM không xác định (giá trị 32'hx). Dẫn tới việc truy xuất RDATA dựa vào lệnh INST này cũng không xác định tại thời điểm RVALID tích cực.

```
Time: 37225 ns Started: 37225 ns Scope: /tb_top_rv32i_wrapper/DUT/rv32i_test_i/axi_vip_1/inst/IF/PC File: C:/Xilinx/Vivado/2022.
Fatal: AXI4_ERRS_RDATA_X: When RVALID is high, a value of X on RDATA valid byte lanes is not permitted. Spec: section A3.2.2.
Time: 37225 ns Iteration: 1 Process: /tb_top_rv32i_wrapper/DUT/rv32i_test_i/axi_vip_1/inst/IF/PC//tb_top_rv32i_wrapper/DUT/rv3
```

Hình 3.14 Thông báo vi phạm từ protocol check của AXI VIP

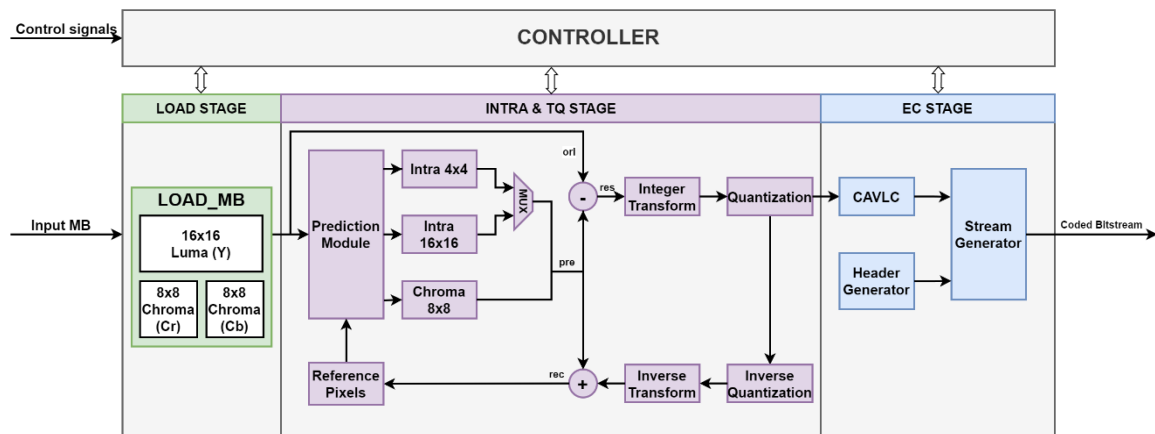
## 3.2. Thiết kế bộ mã hoá H.264/AVC

### 3.2.1. Kiến trúc lõi H.264 IE

Kiến trúc lõi H.264 Intra Encoder được mô tả như Hình 3.15. Với thiết kế pipeline xử lý theo từng MB 16x16 với 3 tầng chính:

- Tầng Load: có nhiệm vụ loại một MB từ bên ngoài.
- Tầng Intra & Transform-Quantization: có nhiệm vụ dự đoán, biến đổi và lượng tử hoá MB.
- Entrophy Coding (EC): có nhiệm vụ nén với phương pháp CAVLC.

Đầu vào file video YUV 4:2:0 sẽ được nhóm xử lý bằng phần mềm và tách theo từng MB 16x16 cả 3 kênh Y, U và V. Sau đó, dữ liệu được nạp vào bộ mã hoá khi có yêu cầu load MB từ bộ Controller. Đầu ra là bitstream 8-bit sẽ được lưu vào FIFO để truyền ra bên ngoài.



Hình 3.15 Kiến trúc bộ mã hoá video H.264 IE pipeline

### 3.2.2. Bộ LOAD\_MB\_16x16

#### 3.2.3.1. Tách video đầu vào bằng Python

Với chuẩn video đầu vào là video thô với format YUV 4:2:0. Trong đó Y (Luma) là thông tin độ sáng với độ phân giải đầy đủ, U (Cb) và V (Cr) là thông tin màu với độ phân giải giảm  $\frac{1}{2}$  theo cả chiều ngang và chiều dọc. Ví dụ 1 khung video FHD có độ phân giải 1920x1080 thì có tổng cộng  $(1920 \times 1080) \times 1 + ((1920 \times 1080) \times 0.5 \times 0.5) \times 2 = (1920 \times 1080) \times 1.5$  điểm ảnh, tương ứng với số pixel cũng như số byte (mã màu 8 bit).



Cấu trúc file YUV này được tổ chức theo cấu trúc:

File YUV = [Frame 1][Frame 2][Frame 3][...][Frame n]

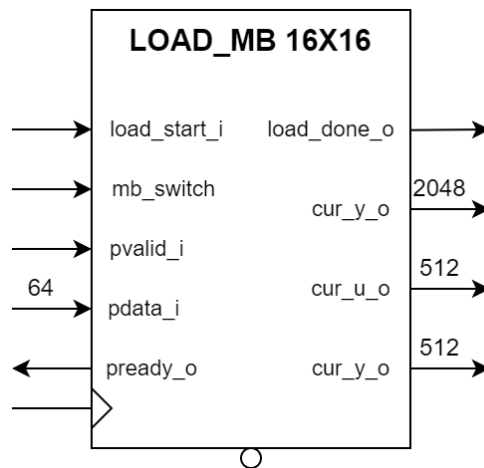
Với mỗi [Frame x] = [Y component][U component][V component]

Với cấu trúc video .yuv ban đầu được sắp xếp serial, nhóm sẽ tiến hành tách tuần tự các component dựa trên height và width nhập vào từ người dùng. Từ đó, nhóm sẽ chuyển về dạng ma trận  $M \times N$  và tách theo macro-block có dạng  $16 \times 16$  pixel. Ví dụ, với frame FHD  $1920 \times 1080$  ta có:  $1920/16 = 120$  MB theo chiều ngang;  $1080/16 = 68$  MB theo chiều dọc, sau khi sắp xếp lại bằng script python, mã hex có dạng:

Frame  $1920 \times 1080 = 120 \times 68$  macroblocks

MB(0,0)	MB(1,0)	MB(2,0)	...	MB(120,0)
MB(0,1)	MB(1,1)	MB(2,1)	...	MB(120,1)
...	...	...	...	...
MB(0,68)	MB(1,68)	MB(2,68)	...	MB(120,68)

### 3.2.3.2. Bộ LOAD\_MB



Hình 3.16 Khối Load MB 16x16

Hình 3.16 mô tả input/output của khối, nhiệm vụ của khối này là tải macro-block đầu vào bên ngoài khi có yêu cầu load từ bộ CONTROLLER. Bộ này sẽ load mỗi lần 64 bit, với tổng số lần load một MB cả 3 kênh màu là 48 lần, được kích hoạt bởi cặp tín hiệu handshake pvalid và pready. Với pvalid sẽ là tín hiệu từ FIFO

và khả dụng khi FIFO không rỗng, tín hiệu pready từ khối này báo hiệu sẵn sàng nhận dữ liệu khi chưa tải đủ 48 lần (đủ 1 MB). Chi tiết tín hiệu mô tả qua bảng 3.3.

Với số lần load từng kênh được tính như sau:

- Load kênh Y:  $(16*16*8) / 64 = 32$  (lần)
- Load kênh U:  $(8*8*8) / 64 = 8$  (lần)
- Load kênh V:  $(8*8*8) / 64 = 8$  (lần)

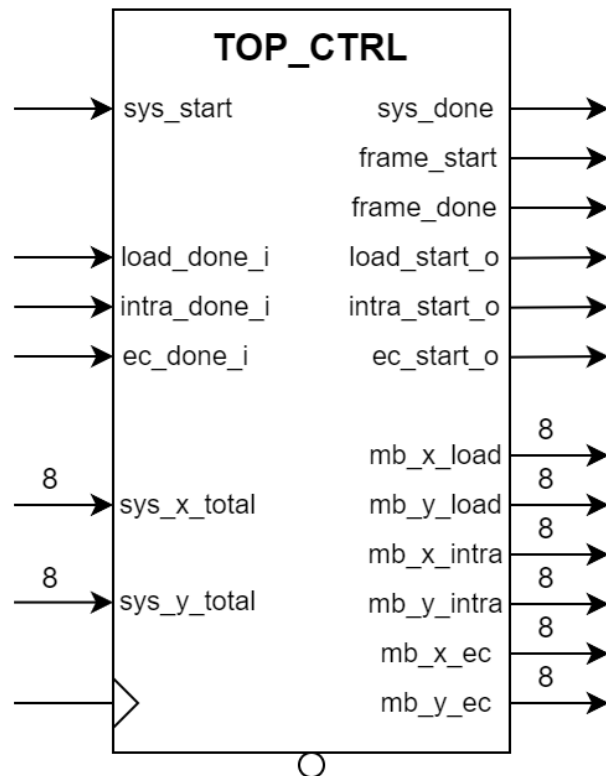
Bảng 3.3 Bảng mô tả tín hiệu khối Load MB 16x16

Tín hiệu	I/O	Width	Chức năng
Clk	Input	1	Xung clock
Rst_n	Input	1	Reset tích cực mức thấp
Load_start_i	Input	1	Tín hiệu bắt đầu load MB mới
Load_done_o	Output	1	Tín hiệu báo trạng thái load hoàn thành
Mb_switch	Input	1	Trigger chuyển MB từ load sang bộ xử lý
Pvalid_i	Input	1	Tín hiệu valid cho data đầu vào từ FIFO
Pdata_i	Input	64	Dữ liệu pixel đầu vào
Pready_o	Output	1	Tín hiệu ready chấp nhận dữ liệu
Cur_y_o	Output	2048	Dữ liệu luma Y (16x16) của MB hiện tại
Cur_u_o	Output	512	Dữ liệu chroma U (8x8) của MB hiện tại
Cur_v_o	Output	512	Dữ liệu chroma V (8x8) của MB hiện tại

### 3.2.3. Bộ CONTROLLER

Bộ controller này có nhiệm vụ điều khiển hoạt động của toàn bộ hệ thống encoder, có các nhiệm vụ:

- Điều phối pipeline xử lý MB.
- Quản lý trạng thái của toàn bộ hệ thống.
- Đồng bộ hoá các module con (load, intra, entropy coding).
- Theo dõi tiến độ của frame (frame start, frame done).



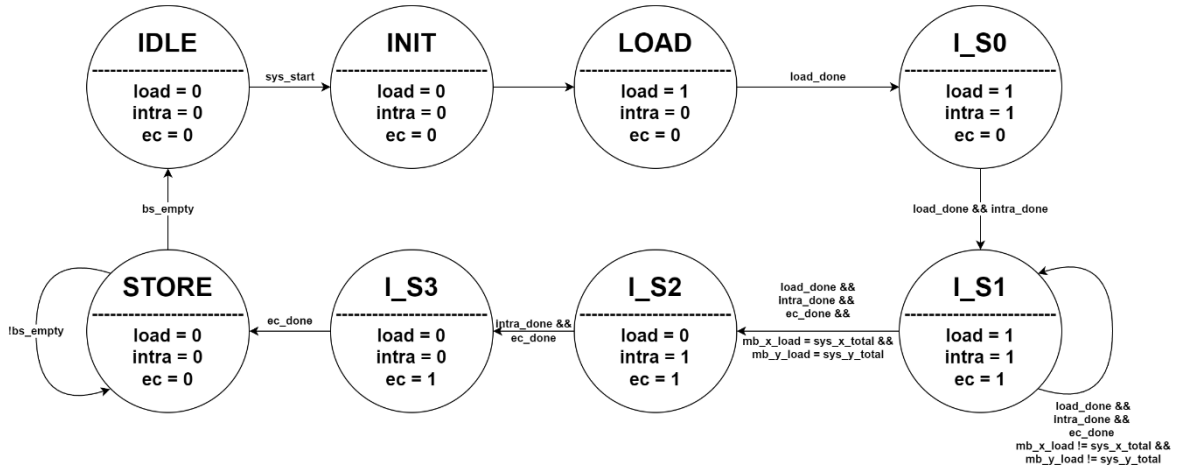
Hình 3.17 Khối H.264 Controller

Hình 3.17 minh họa input/output của khối H.264 Controller, khối này hoạt động theo cơ chế FSM và các tín hiệu được mô tả qua bảng 3.4.

Bảng 3.4 Bảng mô tả các tín hiệu của bộ H.264 Controller

<b>Tín hiệu</b>	<b>I/O</b>	<b>Width</b>	<b>Chức năng</b>
Clk	Input	1	Xung clock
Rst_n	Input	1	Reset tích cực mức thấp
Sys_start	Input	1	Kích hoạt mã hoá 1 khung hình
Sys_done	Output	1	Tín hiệu báo mã hoá xong 1 khung hình
Frame_start	Output	1	Kích hoạt mã hoá data sau slide header
Frame_done	Output	1	Báo hiệu đã mã hoá xong 1 khung hình
Load_done_i	Input	1	Báo hiệu load xong 1 MB
Load_start_o	Output	1	Kích hoạt load MB mới
Intra_done_i	Input	1	Báo hiệu dự đoán xong 1 MB
Intra_start_o	Output	1	Kích hoạt dự đoán 1 MB
Ec_done_i	Input	1	Báo hiệu mã hoá entropy trong 1 MB
Ec_start_o	Output	1	Kích hoạt mã hoá entropy 1 MB
Sys_x_total	Input	8	Nhận số MB theo trục X từ module top
Sys_y_total	Input	8	Nhận số MB theo trục Y từ module top
Mb_x_load	Output	8	Vị trí MB trục X hiện tại cho bộ load
Mb_y_load	Output	8	Vị trí MB trục Y hiện tại cho bộ load
Mb_x_intra	Output	8	Vị trí MB trục X hiện tại cho bộ intra
Mb_y_intra	Output	8	Vị trí MB trục Y hiện tại cho bộ intra
Mb_x_ec	Output	8	Vị trí MB trục X hiện tại cho bộ ec
Mb_y_ec	Output	8	Vị trí MB trục Y hiện tại cho bộ ec

Cơ chế FSM hoạt động của bộ Controller được mô tả qua Hình 3.18



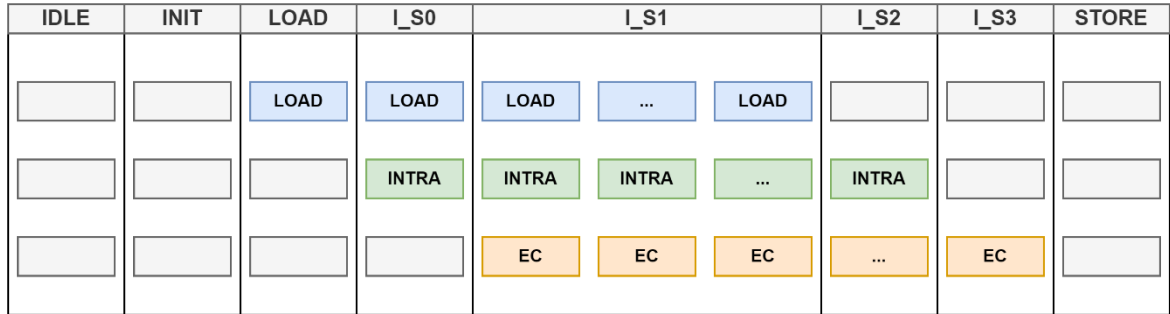
Hình 3.18 Sơ đồ biến đổi trạng thái trong bộ H.264 Controller

Bộ FSM trên mô tả quy trình mã hoá 1 khung hình, trong đó các trạng thái được giải thích thông qua bảng 3.5 bên dưới:

Bảng 3.5 Bảng mô tả chức năng các trạng thái của bộ H.264 Controller

Trạng thái	Mô tả
IDLE	Trạng thái khởi tạo ban đầu
INIT	Trạng thái trung gian khi bắt đầu mã hoá
LOAD	Trạng thái load MB đầu tiên
I_S0	Trạng thái load MB thứ 2 và bắt đầu dự đoán MB đầu tiên
I_S1	Trạng thái pipeline đầy đủ, có cả 3 giai đoạn load, intra và ec. Trạng thái này sẽ kéo dài nhất trong quá trình mã hoá. Chỉ chuyển trạng thái khi khối đã load đến khối MB cuối cùng trong khung ảnh
I_S2	Trạng thái chỉ còn dự đoán và entropy coding, tắt bộ load
I_S3	Trạng thái chỉ còn entropy coding MB cuối cùng trong khung
STORE	Trạng thái chỉ còn ghi bitstream và đợi cho đến khi ghi xong bs

Biểu đồ trạng thái hoạt động pipeline theo thời gian được minh họa qua hình 3.19. Với thời gian hoạt động chủ yếu ở trạng thái I\_S1, nơi cả 3 tầng load, intra và ec đều hoạt động các nhiệm vụ của mình ở những MB khác nhau và liên tiếp nhau cho đến MB cuối cùng trong khung ảnh.



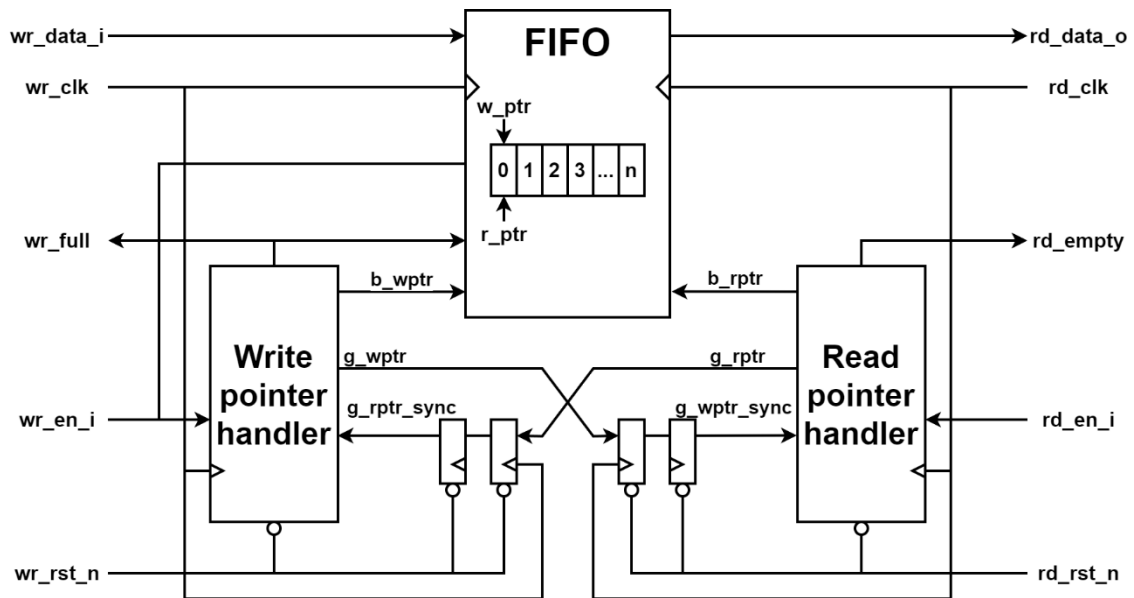
Hình 3.19 Biểu đồ trạng thái hoạt động pipeline theo thời gian

### 3.2.4. Giải pháp xử lý vấn đề Cross Domain Crossing (CDC)

#### 3.2.2.1. FIFO bất đồng bộ

Trong đề tài này, khối mã hóa H.264 hoạt động với tần số 62.5 MHz, khác biệt so với phần còn lại của hệ thống sử dụng tần số 100 MHz. Do đó, để đảm bảo việc truyền dữ liệu giữa hai miền xung nhịp này một cách an toàn, nhóm sử dụng các FIFO bất đồng bộ để tạm trữ và trao đổi dữ liệu giữa DMA và khối H.264.

FIFO bất đồng bộ được thiết kế để cho phép truyền dữ liệu tin cậy giữa hai clock domain khác nhau mà không cần đồng bộ toàn hệ thống — điều thường không khả thi trong thiết kế SoC phức tạp. FIFO này sử dụng cơ chế đồng bộ hóa con trỏ đọc/ghi thông qua 2FF synchronizer kết hợp với kỹ thuật mã hóa Gray, giúp giảm thiểu nguy cơ metastability khi chuyển thông tin trạng thái giữa hai miền. Kiến trúc chi tiết của asynchronous FIFO được minh họa qua hình 3.20.



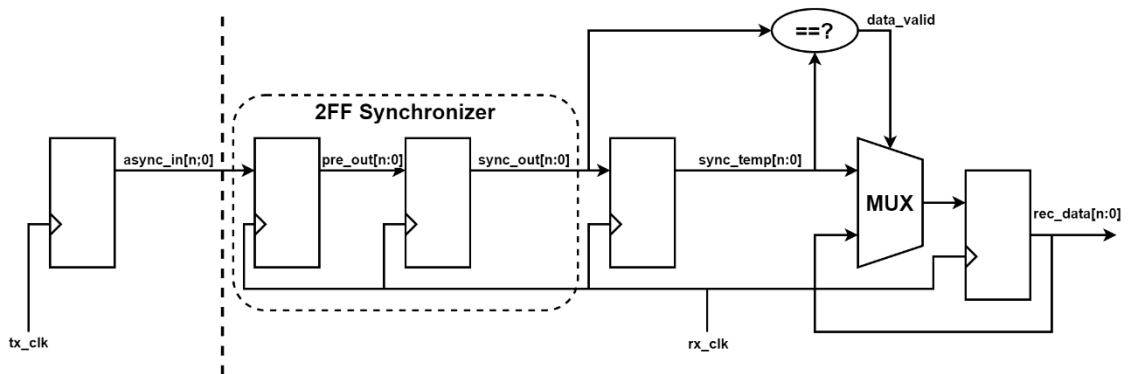
Hình 3.20 Kiến trúc khối FIFO bất đồng bộ

Trong đó gồm các khối:

- Write pointer handler: hoạt động trong miền xung ghi wr\_clk. Khối này quản lý con trỏ ghi, tạo tín hiệu wr\_full, và xử lý logic tăng con trỏ khi thỏa mãn điều kiện ghi (wr\_en\_i && !wr\_full).
- Read pointer handler: hoạt động trong miền xung đọc rd\_clk. Khối này xử lý việc tăng con trỏ đọc khi có yêu cầu đọc (rd\_en\_i), đồng thời sinh ra tín hiệu rd\_empty để báo FIFO trống.
- 2FF synchronizer: dùng để đồng bộ hóa các con trỏ giữa hai miền xung khác nhau, giúp kiểm tra chính xác trạng thái full hoặc empty. Do con trỏ là giá trị nhiều bit, nên trước khi đồng bộ, chúng được chuyển từ mã nhị phân sang mã gray code. Cách mã hóa này đảm bảo rằng tại mỗi thời điểm chỉ một bit thay đổi, giúp giảm thiểu khả năng metastability khi đi qua khối synchronizer.
- FIFO Memory: là bộ nhớ thực lưu trữ dữ liệu, hỗ trợ ghi và đọc song song từ hai clock domain khác nhau. Địa chỉ ghi/đọc được điều khiển bởi các khối pointer handler. Độ sâu của FIFO (depth) nên được thiết kế phù hợp với tỷ lệ tần số giữa hai miền xung để đảm bảo không bị tràn hoặc rỗng trong một

khoảng thời gian khi có sự lệch tốc độ giữa hai domain. Ở đề tài của nhóm, giá trị mặc định cho cả 2 bộ FIFO là 4KB, chấp nhận FIFO thường xuyên đầy (ghi từ miền tần số cao) và một FIFO thường xuyên rỗng (đọc từ miền tần số cao).

### 3.2.2.2. Phương pháp đồng bộ tín hiệu nhiều bit



Hình 3.21 Mạch nguyên lý phương pháp kiểm tra sự ổn định của giá trị nhận [10]

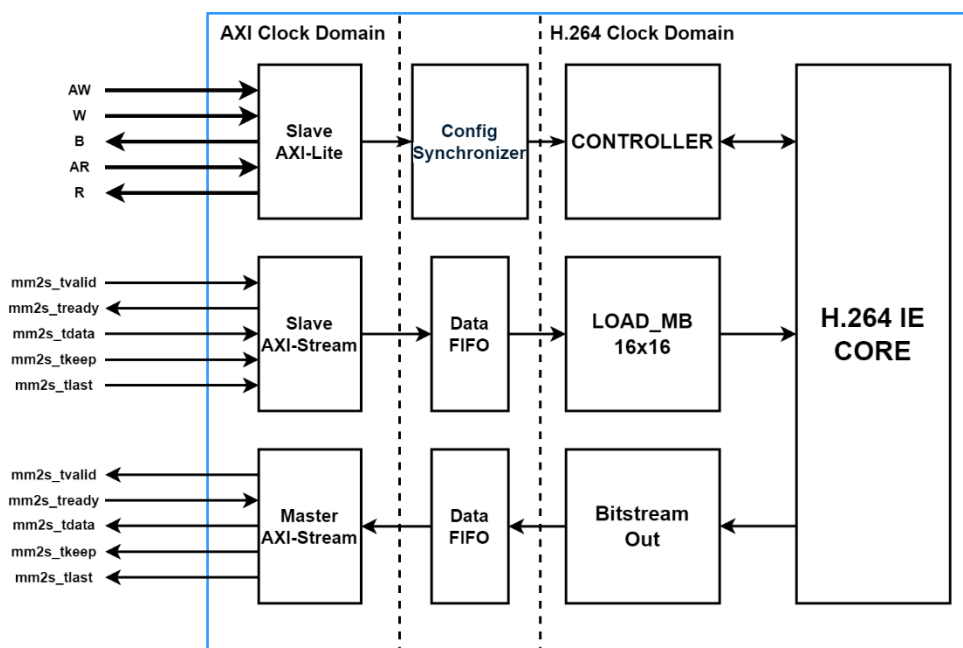
Hình 3.21 minh họa một trong những phương pháp đồng bộ tín hiệu nhiều bit. Một tầng FF được sử dụng sau mạch đồng bộ 2FF để lưu lại giá trị ngõ ra của mạch đồng bộ sync\_out theo từng xung clock rx\_clk. Ngõ ra của tầng FF này là sync\_temp sẽ được so sánh với giá trị ngõ ra của mạch đồng bộ, nếu 2 giá trị này bằng nhau thì đây chính là giá trị ổn định được truyền từ miền xung clock tx\_clk. Lúc này, ngõ ra bộ so sánh data\_valid sẽ tích cực để báo hiệu một giá trị hợp lệ cần được cập nhật. Lúc này thanh ghi rec\_data sẽ cập nhật giá trị, và đây chính là giá trị ổn định có thể sử dụng được.

Vì các thông số cấu hình bộ mã hoá được lưu trong các thanh ghi cấu hình ở giao diện slave axi-lite. Do đó, để lấy đúng các giá trị, nhóm cần 1 cơ chế đồng bộ nhiều bit (như số frame, height, width, qp, ...). Với điều kiện sử dụng phương pháp trên là các tín hiệu ngõ vào phải duy trì giá trị ổn định để h264\_clk có thể đọc đúng giá trị từ 2 lần xung clock trở lên.



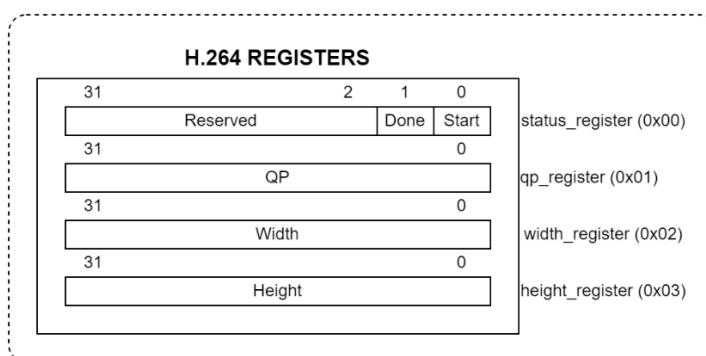
### 3.2.5. Đóng gói và tích hợp bộ mã hoá

Hình 3.22 mô tả IP bộ mã hoá H.264 có tích hợp 1 lõi mã hoá video H264 IE Core, nhóm sẽ tiến hành xây dựng cơ chế load dữ liệu MB đầu vào, thiết kế bộ controller và tích hợp các giao diện kèm cơ chế xử lý CDC để giải quyết vấn đề xung đột giữa 2 miền xung clock khác nhau.



Hình 3.22 Kiến trúc bộ mã hoá video H.264 sau khi đóng gói

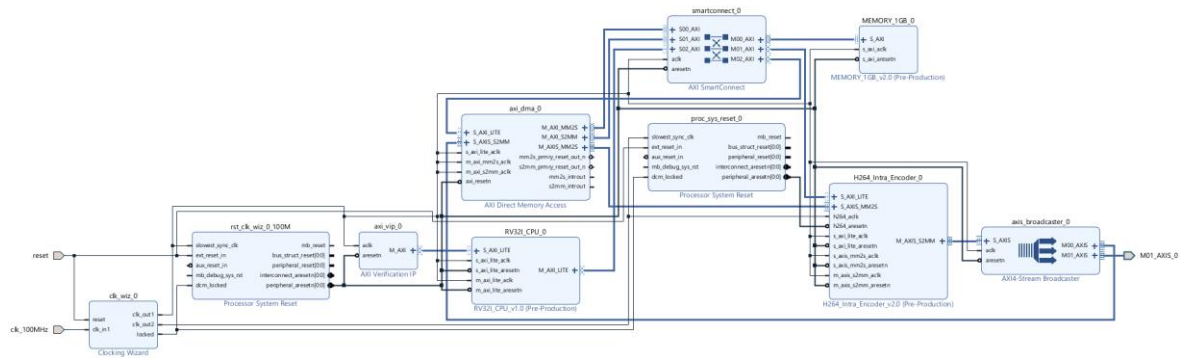
Với giao diện cấu hình AXI-Lite sẽ có các thanh ghi nội mô tả qua Hình 3.23, giá trị bên trong các thanh ghi này sẽ sử dụng các phương pháp đồng bộ đã nói ở trên để truyền tín hiệu điều khiển/cấu hình vào bên trong lõi mã hoá video H.264 IE Core và ngược lại. Tín hiệu done báo hiệu mã hoá xong 1 frame, và tín hiệu này sẽ được reset về 0 khi có tín hiệu start bắt đầu mã hoá frame mới.



Hình 3.23 Thanh ghi cấu hình của bộ mã hoá video H.264

### 3.3. Hiện thực hệ thống SoC cho mô phỏng trên Vivado

#### 3.3.1. Hệ thống SoC tổng quát



Hình 3.24 Hệ thống SoC mô phỏng

Hình 3.24 mô tả hệ thống mô phỏng đề tài SoC của nhóm thông qua thiết kế Block Design trên Vivado. Để nạp lệnh cho bộ xử lý RV32I, nhóm sẽ tích hợp một AXI VIP được cấu hình với vai trò là Master có giao diện AXI-Lite có nhiệm vụ khởi tạo các giao dịch ghi (write transactions) để nạp mã lệnh trực tiếp vào bộ nhớ IMEM bên trong bộ xử lý. Đồng thời nhóm sẽ tạo một IP Memory\_1G giả lập DDRRAM 1GB trên board để chứa dữ liệu video đầu vào cũng như dữ liệu đã được mã hoá bằng H.264 Intra Encoder IP.

Luồng dữ liệu đầu vào sẽ được truy xuất từ Memory\_1GB thông qua khối AXI DMA. Sau đó, DMA truyền dữ liệu này đến khối H.264 Intra Encoder thông qua giao diện AXI-Stream. Kết quả mã hoá được xuất ra cũng theo giao diện AXI-Stream về DMA và lưu lại vào Memory\_1GB. Địa chỉ bắt đầu đọc, địa chỉ bắt đầu ghi sẽ được cấu hình và quản lý bởi bộ xử lý RV32I.

Để kiểm tra chức năng của hệ thống, nhóm tích hợp thêm một IP axis\_broadcaster để tạo thêm một luồng dữ liệu bitstream trả về từ bộ mã hoá H.264 ra bên ngoài. Nhóm sẽ tiến hành bóc tách dữ liệu từ kênh axi-stream này để so sánh kết quả so với phần mềm.

Toàn bộ các kết nối giữa các IP đều được liên kết thông qua IP AXI SmartConnect – bản nâng cấp mới hơn AXI Interconnect với các giải thuật được

tích hợp như cơ chế xử lý phân xử, routing và có vai trò điều phối truy cập từ nhiều master như RV32I, DMA.

Để có thể cấu hình và điều khiển khối DMA và khối H.264, các thanh ghi điều khiển của các khối này cần được ánh xạ vào không gian địa chỉ bộ nhớ của CPU thông qua phương thức Memory-Mapped IO. Khi CPU thực hiện truy cập đọc/ghi đến các địa chỉ này, các giao dịch AXI tương ứng sẽ được tạo ra để kết nối tới các khối phân cứng đó

Bảng 3.6 dưới đây mô tả phân vùng không gian địa chỉ của hệ thống SoC được mô phỏng. Trong đó:

- Vùng DMEM nằm ở địa chỉ thấp, được sử dụng làm bộ nhớ dữ liệu nội, không tạo ra giao dịch AXI.
- Vùng Memory dùng để ánh xạ RAM chính dùng để mô phỏng DDR RAM 1GB trên board và có phát sinh giao dịch AXI.
- Các khối ngoại vi như DMA và H.264 cũng được ánh xạ vào vùng địa chỉ cao, mỗi khối chiếm 4KB, và đều kết nối thông qua AXI bus.
- Các vùng reserved sẽ không được sử dụng và dùng để mở rộng thêm các chức năng trong tương lai.

Bảng 3.6 Bảng mô tả không gian địa chỉ truy cập của bộ xử lý RV32I

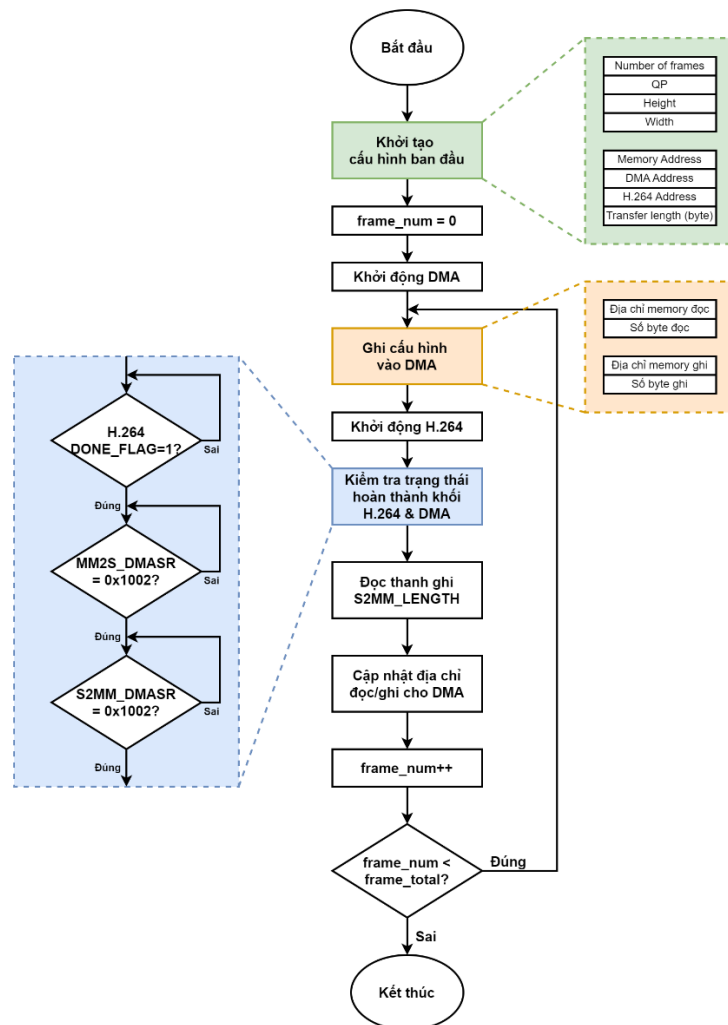
Name	Start address	End address	Size
DMEM	0x0000_0000	0x0000_0FFF	4KB
Reserved	0x0000_1000	0x3FFF_FFFF	~
Memory	0x4000_0000	0x7FFF_FFFF	1GB
H.264	0x8000_0000	0x8000_0FFF	4KB
DMA	0x8000_1000	0x8000_1FFF	4KB
Reserved	0x8000_2000	0xFFFF_FFFF	~

### 3.3.2. Giải thuật điều khiển bộ mã hoá

Giải thuật tổng thể:

- Khởi tạo & cấu hình hệ thống.
- Khởi tạo và cấu hình DMA (địa chỉ đọc/ghi kèm theo số byte truyền)
- Cấu hình bộ và khởi động bộ mã hoá H.264.
- Chạy chương trình chính theo cấu hình đã cài đặt.
- Thoát chương trình.

Hình 3.25 mô tả giải thuật bộ xử lý cấu hình và điều khiển bộ DMA, bộ mã hoá H.264 thông qua việc đọc/ghi vào các thanh ghi nội bên trong từng khối. Các phần bên dưới giải thích chi tiết chương trình mã giả mô tả các phần quan trọng trong giải thuật này.



Hình 3.25 Lưu đồ giải thuật mô tả chương trình SoC cho mã hoá video

### 3.3.2.1. Khởi tạo hệ thống

```
BEGIN
    // Thiết lập tham số encoding
    start_signal = 1
    num_frames = 50
    qp_parameter = 28
    frame_width = 1920
    frame_height = 1080

    // Thiết lập địa chỉ memory map
    input_memory_base = 0x40000000 // Vùng nhớ input
    output_memory_base = 0x60000000 // Vùng nhớ output
    h264_encoder_base = 0x80000000 // H.264 encoder registers
    dma_controller_base = 0x80001000 // DMA controller registers

    // Tính toán kích thước frame
    input_frame_size = 1920 × 1080 × 1.5 = 3110400 bytes // YUV420
    output_frame_size = 3110400 bytes // Compressed

    // Cấu hình H.264 encoder
    H264_QP_REG = qp_parameter
    H264_WIDTH_REG = frame_width
    H264_HEIGHT_REG = frame_height

    // Khởi tạo biến đếm
    current_frame = 0
    current_input_addr = input_memory_base
    current_output_addr = output_memory_base
END
```

### 3.3.2.2. Khởi động DMA Controller

```
FUNCTION initialize_dma()  
    // Khởi động kênh MM2S (Memory to Stream)  
    DMA_MM2S_CONTROL_REG = start_signal  
  
    // Khởi động kênh S2MM (Stream to Memory)  
    DMA_S2MM_CONTROL_REG = start_signal  
END FUNCTION
```

### 3.3.2.3. Cấu hình DMA Transfer

```
FUNCTION configure_dma_transfer()  
    // Cấu hình kênh MM2S (đọc từ memory)  
    DMA_MM2S_SRC_ADDR = current_input_addr  
    DMA_MM2S_LENGTH = input_frame_size  
  
    // Cấu hình kênh S2MM (ghi vào memory)  
    DMA_S2MM_DST_ADDR = current_output_addr  
    DMA_S2MM_LENGTH = input_frame_size // Buffer size  
END FUNCTION
```

### 3.3.2.4. Khởi động bộ mã hoá video H.264

```
FUNCTION start_h264_encoding()  
    // Gửi tín hiệu start tới H.264 encoder  
    H264_CONTROL_REG = start_signal  
END FUNCTION
```

### 3.3.2.5. Kiểm tra trạng thái hoàn thành mã hoá 1 frame H.264 (polling mode)

```
FUNCTION wait_for_h264_completion()
    done_flag = 0
    WHILE (done_flag == 0) DO
        // Đọc thanh ghi trạng thái H.264
        status_reg = H264_STATUS_REG

        // Kiểm tra bit[1] để xem đã done chưa
        done_flag = (status_reg AND 0x02) >> 1
        // Polling - chờ đến khi done
    END WHILE
END FUNCTION
```

### 3.3.2.6. Kiểm tra trạng thái hoàn thành của DMA

```
FUNCTION check_dma_status()
    WHILE (mm2s_status != 0x1002) DO
        // Đọc trạng thái kênh MM2S
        mm2s_status = DMA_MM2S_STATUS_REG
    END WHILE

    WHILE (s2mm_status != 0x1002) DO
        // Đọc trạng thái kênh S2MM
        s2mm_status = DMA_S2MM_STATUS_REG
    END WHILE
END FUNCTION
```

### 3.3.2.7. Đọc số byte thực tế đã ghi từ thanh ghi S2MM\_LENGTH

```
FUNCTION dma_read_s2mm_length()
    // Đọc số byte thực tế đã ghi (để cập nhật địa chỉ)
    actual_output_size = DMA_S2MM_LENGTH_REG
END FUNCTION
```

### 3.3.2.8. Cập nhật địa chỉ cho Memory

```
FUNCTION update_memory_addresses()
    // Tăng địa chỉ input cho frame tiếp theo
    current_input_addr = current_input_addr + input_frame_size

    // Tăng địa chỉ output dựa trên kích thước thực tế đã ghi
    current_output_addr = current_output_addr + actual_output_size
END FUNCTION
```

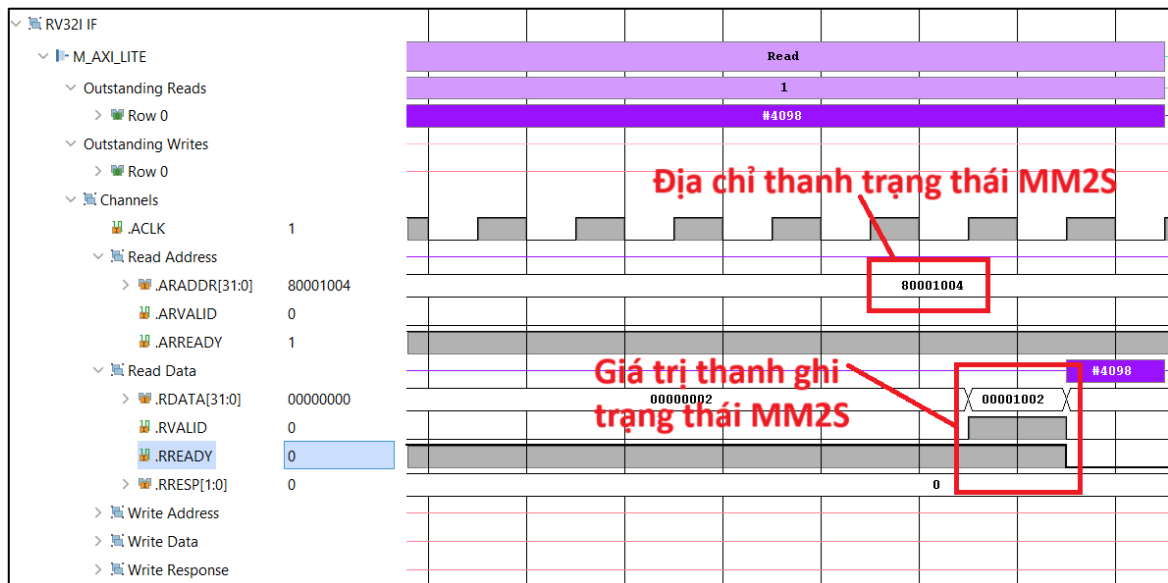
Số byte cấu hình kênh stream (s2mm) trả về Memory sẽ được ghi một giá trị lớn bằng với giá trị ghi cho kênh mms2, nhưng chúng ta sẽ không cần để tâm đến giá trị này bởi vì khi gửi transfer cuối cùng, packet truyền cuối cùng kèm tín hiệu TLAST bật lên báo cho DMA kết thúc frame truyền. Số byte đã truyền sẽ được lưu vào thanh ghi S2MM\_LENGTH (địa chỉ 0x58 tính từ địa chỉ cơ sở của DMA), chính là giá trị ‘actual\_output\_size’ ở đoạn mã giả ở trên.

Cách so sánh thanh ghi trạng thái của 2 kênh với giá trị ‘0x1002’ bởi vì bit[12] là bit IOC\_Irq (Interrupt on Complete), bit này sẽ được set về 1 khi một sự kiện ngắt được bật khi hoàn thành việc truyền. Còn bit[1] là bit Idle, đây là bit trạng thái của kênh và bit này chỉ ra rằng đã hoàn thành việc truyền và đang đợi tham số cấu hình thay đổi như địa chỉ, transfer\_length để kích hoạt lần truyền tiếp theo. Do đó chúng ta cần phải kiểm tra cả 2 thanh ghi trạng thái ở cả 2 kênh so với giá trị ‘0x1002’ để có thể đảm bảo việc đọc và truyền một frame đã hoàn thành.



### 3.3.3. Một số waveform mô tả hệ thống

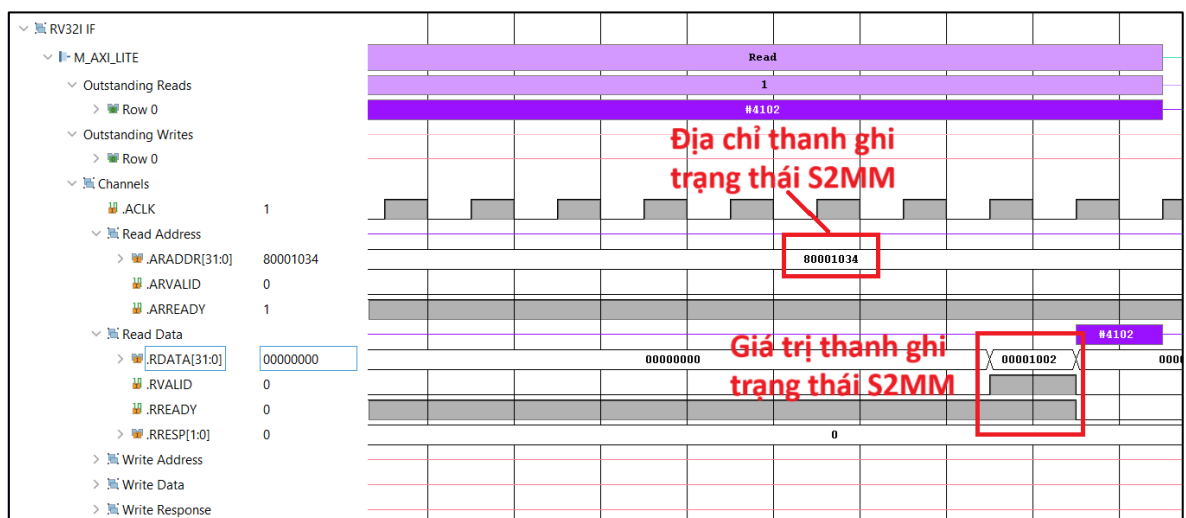
#### 3.3.2.1. Đọc thanh ghi trạng thái MM2S



Hình 3.26 Lệnh đọc từ bộ xử lý RV32I đến thanh ghi MM2S\_DMA\_SR

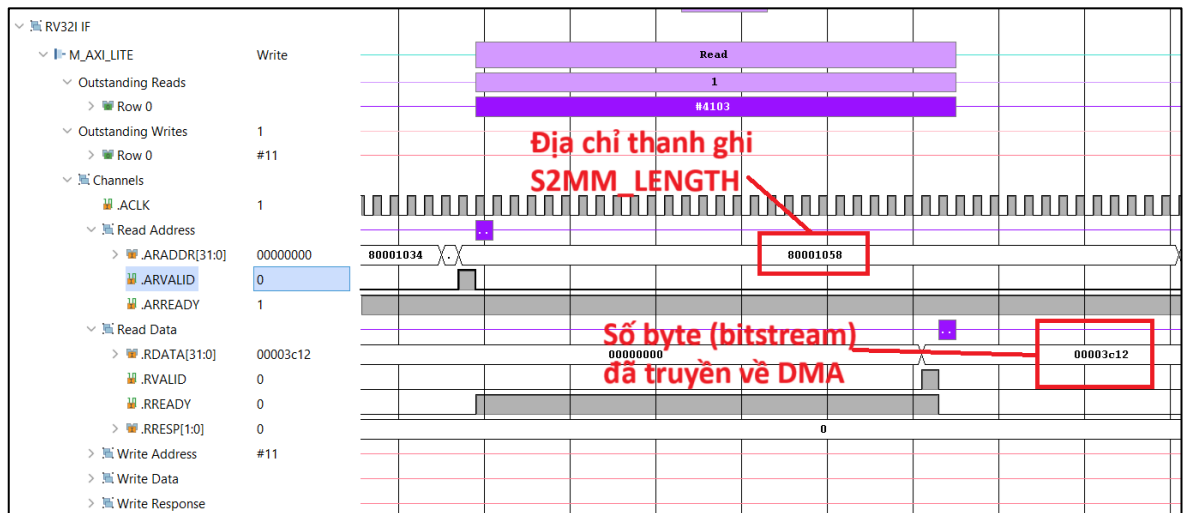
Hình 3.26 minh họa một lệnh đọc thanh ghi trạng thái MM2S từ DMA cho kết quả 0x1002 nghĩa là bit[12] = 1 (interrupt on complete) và bit [1] = 1 (trạng thái IDLE) chứng tỏ kênh MM2S đã hoàn thành nhiệm vụ truyền của mình.

#### 3.3.2.2. Đọc thanh ghi trạng thái S2MM

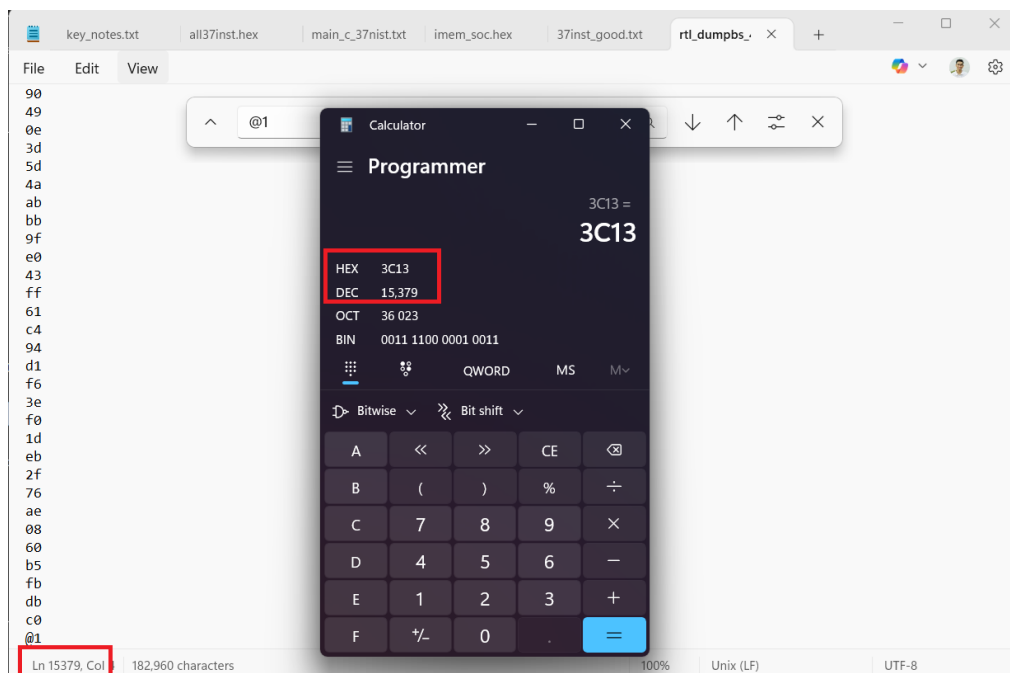


Hình 3.27 Lệnh đọc từ bộ xử lý RV32I đến thanh ghi S2MM\_DMA\_SR

Hình 3.27 minh hoạ một lệnh đọc thanh ghi trạng thái S2MM từ DMA cho kết quả 0x1002 nghĩa là bit[12] = 1 (interrupt on complete) và bit [1] = 1 (trạng thái IDLE) chứng tỏ kênh S2MM đã hoàn thành nhiệm vụ truyền của mình.



Hình 3.28 Lệnh đọc từ bộ xử lý RV32I đến thanh ghi S2MM\_LENGTH



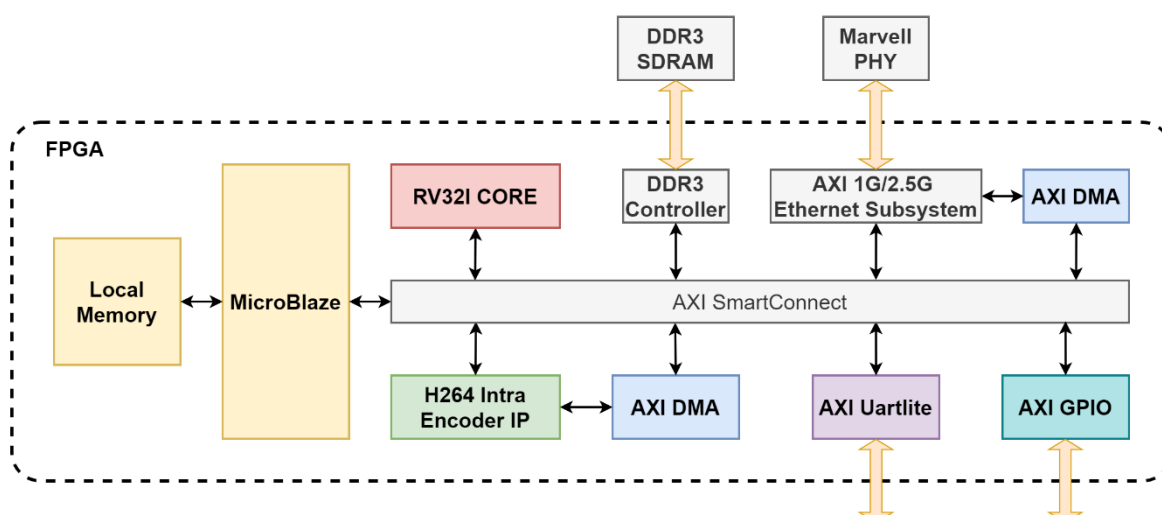
Hình 3.29 Kết quả so sánh số byte truyền từ DMA so với byte thực tế

Từ hình 3.28 và hình 3.29 cho thấy giá trị thanh ghi S2MM\_LENGTH và số byte bitstream mã hoá là giống nhau, cho thấy DMA đã truyền đúng dữ liệu mà không có mất mát nào xảy ra.

### 3.4. Hiện thực hệ thống SoC cho việc triển khai xuống FPGA

Hình 3.30 mô tả hệ thống SoC khi nhóm triển khai xuống FPGA. Khác với trên mô phỏng, khi triển khai xuống FPGA nhóm cần phải lưu dữ liệu video đầu vào ở DDR SDRAM (1GB) vật lý trên board vì tài nguyên FPGA là quá nhỏ (vài MB) không chứa được video. Do đó, nhóm cần phải tích hợp thêm IP tích hợp sẵn từ Xilinx là AXI 1G/2.5G Ethernet Subsystem để nhận và gửi dữ liệu từ bên PC vào FPGA và lưu vào RAM.

Để có thể cấu hình được IP này, nhóm sẽ sử dụng soft-core MicroBlaze để tiến hành cấu hình trên phần mềm Vitis sử dụng thư viện lwIP và một số thư viện hỗ trợ khác để có thể nhận packet từ cổng LAN. Ngoài ra, nhóm sẽ tiến hành viết script Python để tạo gói tin dạng Ethernet RAW có sử dụng thư viện socket để gửi đến FPGA.

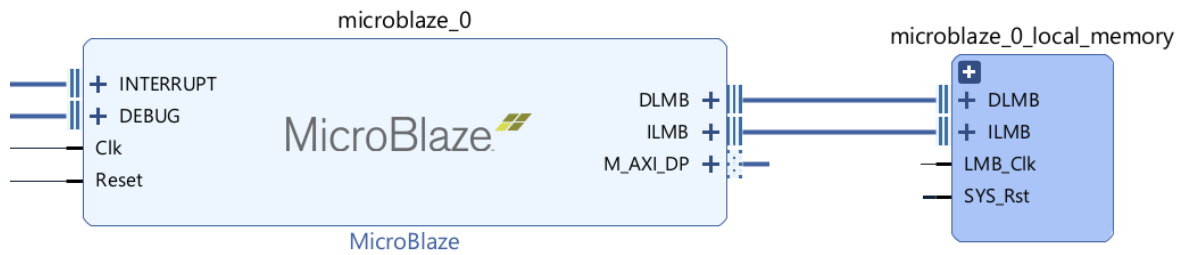


Hình 3.30 Kiến trúc SoC triển khai xuống FPGA

Trong đề tài này, nhóm sử dụng 2 bộ AXI DMA, 1 bộ với chế độ Directed Register dùng cho bộ xử lý RV32I cấu hình việc truyền dữ liệu video thô vào bộ mã hoá (MM2S) và dữ liệu đã mã hoá từ bộ H.264 (S2MM). Bộ AXI DMA còn lại sẽ có chế độ SG để cấu hình và truyền dữ liệu từ khối AXI 1G/2.5G Ethernet Subsystem về bộ nhớ DDR SDRAM.

### 3.5. Các IP của Xilinx được sử dụng trong Block Design

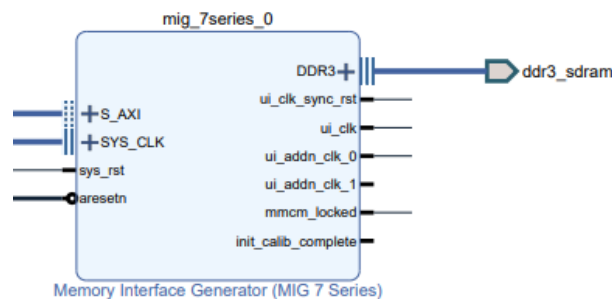
#### 3.5.1. MicroBlaze



Hình 3.31 Soft-core MicroBlaze trong Vivado Block Design

Trong đề tài của nhóm, soft-core MicroBlaze (hình 3.31) được tích hợp nhằm đóng vai trò là bộ điều khiển trung gian, cho phép nạp chương trình vào bộ xử lý RV32I thông qua môi trường Vitis IDE mà không cần tạo lại bitstream sau mỗi lần thay đổi mã lệnh. Ngoài ra, MicroBlaze còn được sử dụng để cấu hình và xử lý logic cho khối nhận dữ liệu từ PC qua Ethernet sử dụng thư viện lwIP, giúp đơn giản hoá quá trình triển khai hệ thống.

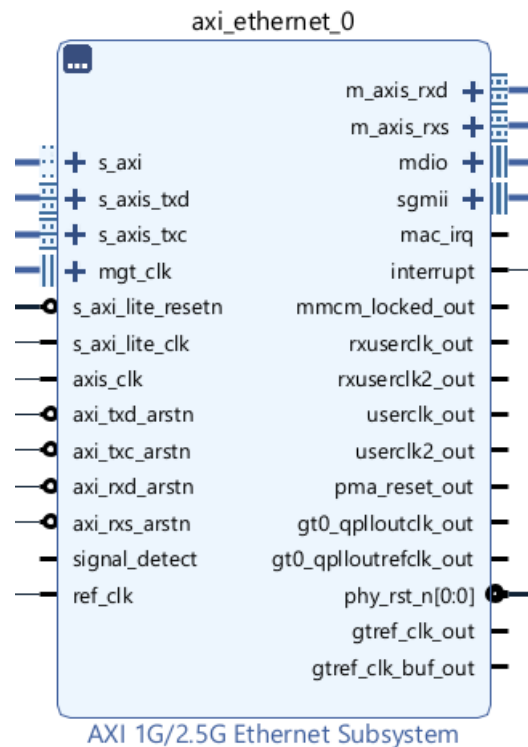
#### 3.5.2. Memory Intergrater Generator (MIG 7 series)



Hình 3.32 Memory Intergrater Generator trên Vivado Block Design

Khối Memory Interface Generator (MIG) (hình 3.32) được sử dụng để kết nối hệ thống SoC với bộ nhớ DDR3 SDRAM ngoài, đóng vai trò là giao diện chuyển đổi giữa bus AXI nội bộ và giao thức DDR3 vật lý. Trong đề tài của nhóm, MIG cho phép khối AXI DMA truy cập không gian bộ nhớ lớn với tốc độ cao để đọc/ghi dữ liệu video lớn.

### 3.5.3. Khối AXI 1G/2.5G Ethernet Subsystem

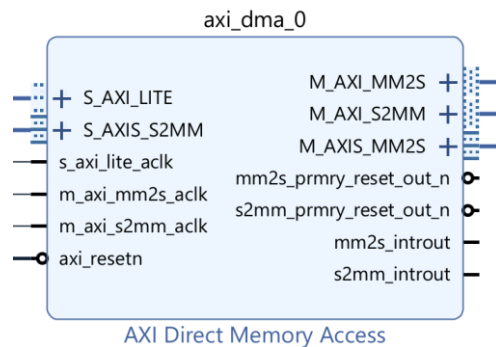


Hình 3.33 AXI Ethernet Subsystem trong Vivado Block Design

Khối AXI 1G/2.5G Ethernet Subsystem (Hình 3.33) được sử dụng làm giao diện truyền nhận Ethernet tốc độ cao giữa hệ thống SoC và chip PHY Marvell ngoài qua SGMII. Bên trong IP này gồm 3 khối chính, chi tiết như sau:

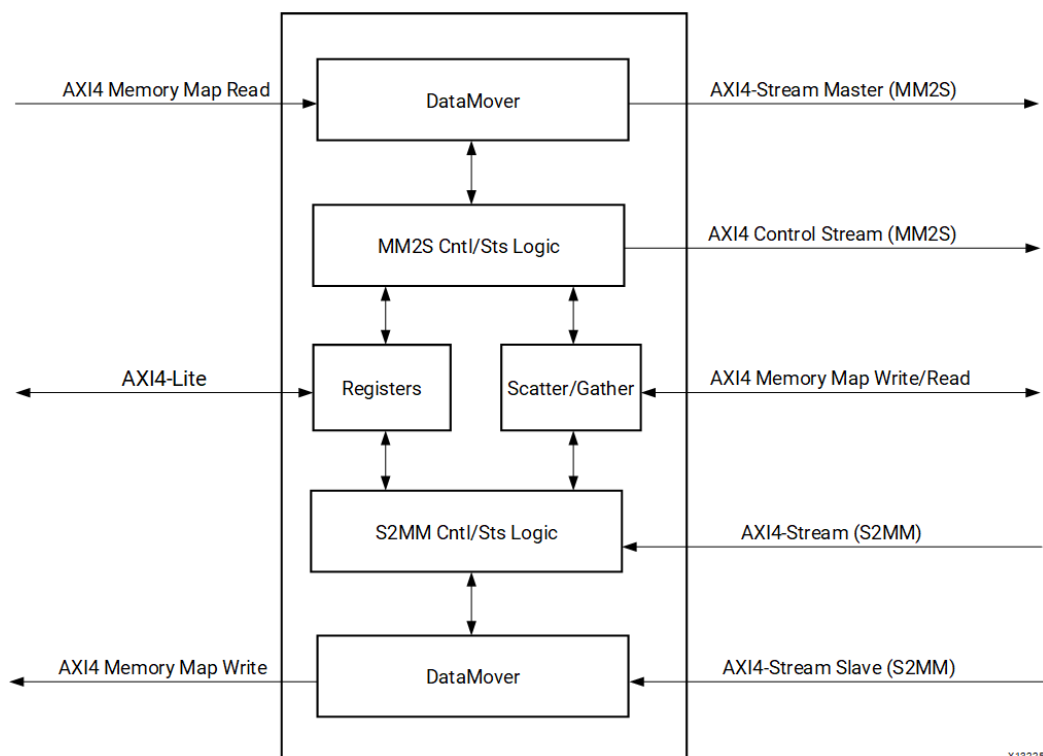
- Khối Tri-Mode Ethernet MAC (TEMAC): có nhiệm vụ đóng/gỡ khung Ethernet, kiểm soát truyền nhận frame theo chuẩn IEEE 802.3. Khối này có các giao diện như AXI-Lite để nhận cấu hình, giao diện AXI-Stream để trao đổi dữ liệu với DMA.
- Khối AXI Ethernet Buffer: có nhiệm vụ làm bộ đệm dữ liệu từ TEMAC sang luồng AXI-Stream và ngược lại.
- Khối PCS/PMA or SGMII Core: là giao diện tầng vật lý tương thích chuẩn SGMII, có nhiệm vụ chuyển đổi tín hiệu từ SGMII sang dạng mã hoá tín hiệu truyền thông serdes, giao tiếp trực tiếp với chip PHY Marvell 88E1111 qua các tín hiệu như sgmii, mdio, ...

### 3.5.4. Khối AXI DMA



Hình 3.34 Khối AXI DMA trong Vivado Block Design

AXI DMA (Hình 3.34) là thành phần không thể thiếu và có nhiệm vụ truyền dữ liệu giữa các khối xử lý (H.264 IP, Ethernet IP) và bộ nhớ DDR SDRAM thông qua bus AXI. Trong đó, IP có 2 chế độ là Directed Register và Scatter Gather. Sự khác nhau là Directed Register phải cấu hình trực tiếp thanh ghi điều khiển mỗi lần truyền dữ liệu, còn chế độ SG sẽ sử dụng 1 vùng RAM chứa Buffer Descriptors (BDs) nơi mô tả từng khối dữ liệu truyền như địa chỉ, độ dài, ...



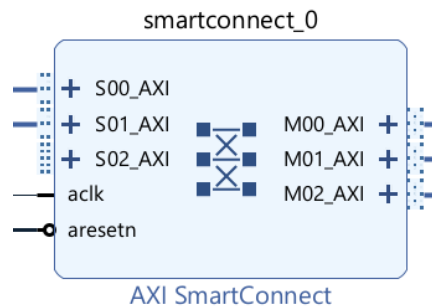
Hình 3.35 Kiến trúc khối AXI DMA [12]

Kiến trúc của bộ AXI DMA được mô tả qua Hình 3.35. Trong đó, có 2 đường dữ liệu là MM2S và S2MM. MM2S là đường truyền dữ liệu từ bộ nhớ đến khối ngoại vi, ngược lại S2MM là đường truyền dữ liệu từ khối ngoại vi đến bộ nhớ. Các thanh ghi về thông tin cấu hình và điều khiển sẽ được đọc/ghi qua giao diện AXI-Lite, thông tin về các thanh ghi được mô tả qua bảng 3.7.

Bảng 3.7 Bảng mô tả thanh ghi DMA trong chế độ Directed Register

Offset địa chỉ	Tên thanh ghi	Mô tả
00h	MM2S_DMACR	Thanh ghi điều khiển MM2SDMA
04h	MM2S_DMASR	Thanh ghi trạng thái MM2SDMA
08h-14h	Reserved	Không sử dụng
18h	MM2S_SA	Địa chỉ nguồn MM2S (32 bit thấp)
1Ch	MM2S_SA_MSB	Địa chỉ nguồn MM2S (32 bit cao)
28h	MM2S_LENGTH	MM2S Transfer length (bytes)
30h	S2MM_DMACR	Thanh ghi điều khiển S2MM DMA
34h	S2MM_DMASR	Thanh ghi trạng thái S2MM DMA
38h-44h	Reserved	Không sử dụng
48h	S2MM_DA	Địa chỉ đích S2MM (32 bit thấp)
4Ch	S2MM_DA_MSB	Địa chỉ đích S2MM (32 bit cao)
58h	S2MM_LENGTH	S2MM Buffer Length (bytes)

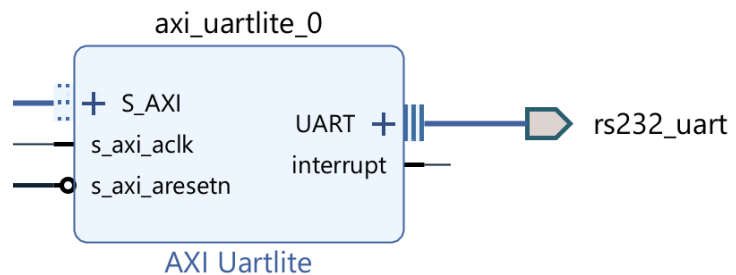
### 3.5.5. Khối AXI SmartConnect



Hình 3.36 Khối AXI SmartConnect Trong Vivado Block Design

AXI SmartConnect (Hình 3.36) đóng vai trò là interconnect logic thế hệ mới, thay thế cho các khối AXI Interconnect truyền thống. Khối này cung cấp kết nối N:M giữa nhiều master và slave AXI, hỗ trợ đầy đủ các kênh AXI4 (AR, AW, W, R, B) với khả năng xử lý song song, định tuyến động và hỗ trợ pipelining hiệu quả.

### 3.5.6. Khối AXI UartLite



Hình 3.37 Khối AXI Uartlite trên Vivado Block Design

Khối AXI Uartlite (Hình 3.37) được điều khiển bởi MicroBlaze để debug, truyền lệnh và các giao tiếp cơ bản với máy tính thông qua cổng COM (USB-UART). Khối này giao tiếp nối tiếp UART tiêu chuẩn hỗ trợ baud cấu hình được qua tham số, được điều khiển qua bus AXI-Lite. Ngoài ra, AXI UARTLite còn tích hợp cơ chế ngắt (interrupt), giúp tối ưu hiệu năng bằng cách tránh việc polling liên tục từ CPU giúp nâng cao hiệu quả xử lý và độ phản hồi trong quá trình giao tiếp.



## Chương 4. KẾT QUẢ VÀ ĐÁNH GIÁ

### 4.1. Bộ xử lý RISC-V RV32I

#### 4.1.1. Kiểm tra 37 lệnh cơ bản qua mô phỏng và triển khai trên FPGA

Bộ lệnh kiểm tra bao gồm 37 lệnh, chi tiết chương trình kiểm tra tập lệnh được trình bày thông qua Bảng 4.1:

- R-type: add, sub, xor, or, and, sll, srl, sra, slt, sltu
- I-type (imm): addi, xori, ori, andi, slli, srli, srai, slti, sltiu
- I-type (load): lb, lh, lw, lbu, lhu
- S-type: sb, sh, sw
- B-type: beq, bne, blt, bge, bltu, bgeu
- Các lệnh khác: jal, jalr, lui, auipc

Bảng 4.1 Chương trình kiểm tra 37 lệnh cơ bản của tập lệnh RV32I

Mã lệnh	Hex	KQ
addi x0, x0, 3	0x00300013	x0 = 0
addi x1, x0, 3	0x00300093	x1 = 3
addi x2, x1, 1	0x00108113	x2 = 4
add x3, x1, x2	0x002081b3	x3 = 7
sub x3, x1, x2	0x402081b3	x3 = -1 (ffff_ffff)
xor x4, x1, x2	0x0020c233	x4 = 7
or x5, x1, x2	0x0020e2b3	x5 = 7
and x6, x1, x2	0x0020f333	x6 = 0
addi x7, x0, 1	0x00100393	x7 = 1
sll x7, x7, x7	0x007393b3	x7 = 2
srl x8, x5, x7	0x0072d433	x8 = 1
addi x9, x9, -6	0xfffa48493	x9 = -6 (ffff_fffa)

sra x10, x9, x7	0x4074d533	x10 = -3 (ffff_fffe)
sra x10, x5, x7	0x4072d533	x10 = 1 (7>>2)
slt x11, x9, x10	0x00a4a5b3	x11 = 1 (-6<-3)
sltu x11, x9, x10	0x00a4b5b3	x11 = 0 (6>3)
xori x12, x11, 5	0x0055c613	x12 = 5
ori x13, x12, 7	0x00766693	x13 = 7
andi x14, x13, 8	0x0086f713	x14 = 0
slli x15, x12, 1	0x00161793	x15 = 10
srli x16, x15, 1	0x0017d813	x16 = 5
srai x17, x9, 1	0x4014d893	x17 = -3
srai x17, x16, 1	0x40185893	x17 = 2
slti x18, x17, 2	0x0028a913	x18 = 0
sltiu x19, x17, 15	0x00f8b993	x19 = 1
addi x14, x0, 0xfa	0x0fa00713	x14 = 32'h0000_00fa
sw x0, 8(x0)	0x00002423	M[8] = 32'h0
sw x0, 4(x0)	0x00002223	M[4] = 32'h0
sw x0, 0(x0)	0x00002023	M[0] = 32'h0
sb x9, 8(x0)	0x00900423	M[8] = 32'h0000_00fa
sh x9, 4(x0)	0x00901223	M[4] = 32'h0000_fffa
sw x9, 0(x0)	0x00902023	M[0] = 32'hffff_fffa
lw x20, 0(x0)	0x00002a03	x20 = 32'hffff_fffa (-6)
addi x25, x20, 3	0x003a0c93	x25 = 32'hffff_fffd (-3)

lw x20, 4(x0)	0x00402a03	x20 = 32'h0000_fffa
lw x20, 8(x0)	0x00802a03	x20 = 32'h0000_00fa
lh x21, 4(x0)	0x00401a83	x21 = 32'hffff_fffa
lhu x21, 4(x0)	0x00405a83	x21 = 32'h0000_fffa
lb x22, 0(x0)	0x00000b03	x22 = 32'hffff_fffa
lbu x22, 0(x0)	0x00004b03	x22 = 32'h0000_00fa
beq x22, x14, 12	0x00eb0663	jump to lui
nop (addi x0, x0, 0)	0x00000013	No-operation
nop (addi x0, x0, 0)	0x00000013	No-operation
lui x23, 0xaaaaa	0xaaaaabb7	x23 = 32'haaaaa000
auipc x24, 0xaaaaa	0xaaaaac17	x24 = 32'haaaaa + pc
jal x26, 20	0x01400d6f	x26 = pc4   jump to jalr
nop (addi x0, x0, 0)	0x00000013	No-operation
nop (addi x0, x0, 0)	0x00000013	No-operation
nop (addi x0, x0, 0)	0x00000013	No-operation
nop (addi x0, x0, 0)	0x00000013	No-operation
jalr x27, 28(x26)	0x01cd0de7	x27 = pc4   jump to addi
nop (addi x0, x0, 0)	0x00000013	No-operation
nop (addi x0, x0, 0)	0x00000013	No-operation
addi x28, x0, 0xb	0x00b00e13	x28 = b
lbu x29, 8(x0)	0x00804e83	x29 = 32'h0000_00fa
andi x29, x29, 0x0f	0x00fefe93	x29 = a

beq x28, x29, 8	0x01de0463	l1: not jump (a!=b)
addi x29, x29, 1	0x001e8e93	x29 = b
bne x28, x27, 12	0x01be1663	jump addi (0x0b != 0xb0)
bge x31, x30, 24	0x01efdc63	jump to lui
blt x30, x31, -4	0xffff4ee3	jump to bge
addi x30, x0, -20	0xfec00f13	x30 = -20 (ffff_ffec)
addi x31, x0, -15	0xff100f93	x31 = -15 (ffff_fff1)
bltu x31, x30, -16	0xffefe8e3	not jump to blt
bgeu x31, x30, -16	0xffeff8e3	jump to blt

Sau khi kiểm tra chức năng qua mô phỏng trên Vivado, nhóm tiến hành triển khai chương trình xuống FPGA, kết hợp với MicroBlaze viết chương trình nạp lệnh ban đầu cho bộ xử lý RV32I thông qua giao thức AXI-Lite. Sau đó đọc giá trị từ DMEM bên trong bộ xử lý RV32I và truyền qua Uart về máy tính và hiển thị lên terminal giá trị 32 thanh ghi.

```

Vitis Serial Console
Connected to: Serial ( COM5, 9600, 0, 8 )
RV32I Base Address: 0x44A00000

=====
RV32I CPU Control System Started
=====

1. Reset
2. Init RV32I 37 Instructions
3. Init RV32I SoC
4. Run video encoder program
5. Check Register
6. Print current PC
7. Exit

Please select an option (1-5): You selected: 2

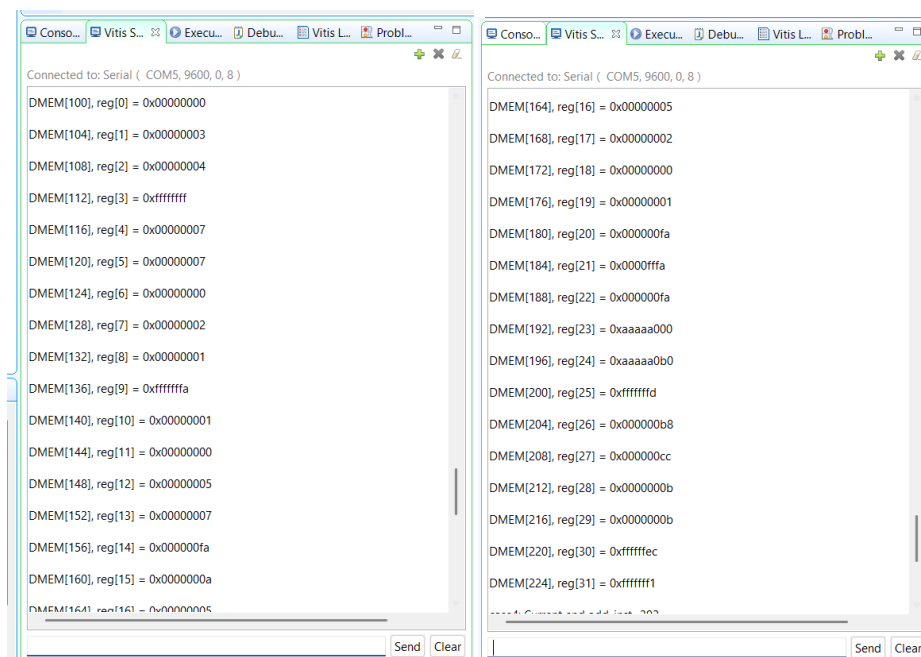
```

Hình 4.1 Minh họa chương trình nạp lệnh qua Vitis Serial Console

Kết quả Vitis Serial Console (hình 4.3) so với phần mềm giả lập RARS (hình 4.2) cho thấy các lệnh từ FPGA đều thực thi giống so với phần mềm, chỉ khác ở các thanh ghi x24, x26, x27 do địa chỉ PC bắt đầu ở các lệnh này là khác nhau.

Registers			Floating Point	Control and Status
Name	Number	Value		
zero	0	0x00000000		
ra	1	0x00000003		
sp	2	0x00000004		
gp	3	0xffffffff		
tp	4	0x00000007		
t0	5	0x00000007		
t1	6	0x00000000		
t2	7	0x00000002		
s0	8	0x00000001		
s1	9	0xfffffffffa		
a0	10	0x00000001		
a1	11	0x00000000		
a2	12	0x00000005		
a3	13	0x00000007		
a4	14	0x000000fa		
a5	15	0x0000000a		
a6	16	0x00000005		
a7	17	0x00000002		
s2	18	0x00000000		
s3	19	0x00000001		
s4	20	0x000000fa		
s5	21	0x0000ffa		
s6	22	0x000000fa		
s7	23	0xaaaa000		
s8	24	0xaaaa0b8		
s9	25	0xffffffffd		
s10	26	0x004000c0		
s11	27	0x004000d4		
t3	28	0x0000000b		
t4	29	0x0000000b		
t5	30	0xffffffffec		
t6	31	0xfffffffff1		
pc		0x0040010c		

Hình 4.2 Kết quả mô phỏng 37 lệnh qua phần mềm RARS



Hình 4.3 Kết quả thanh ghi đọc từ Vitis Serial Console

#### 4.1.2. Kiểm tra truy xuất ngoại vi qua MMIO Interface

Bảng 4.2 mô tả tập lệnh giả lập việc truy xuất ngoại vi thông qua MMIO. Tập lệnh này giả lập toàn bộ lệnh ghi sw, sh, sb được xử lý WSTRB khi phát sinh giao dịch ghi và các lệnh đọc lw, lh, lhu, lb, lbu khi phát sinh giao dịch đọc.

Bảng 4.2 Tập lệnh kiểm tra truy xuất ngoại vi qua giao diện MMIO

Mã lệnh	Mã hex	Kết quả
addi x1, x0, 3	0x00300093	x1=3
lui x2, 0xaaaab	0xaaaab137	x2=0xaaaa_b000
addi x2, x2, -1366	0xaaa10113	x2=0xaaaa_aaaa
lui x3, 0xbbbbc	0xbbbbc1b7	x3=0xbbbb_c000
addi x3, x3, -1093	0xbbb18193	x3=0xbbbb_bbbb
lui x4, 0xcccd	0xcccd237	x4=0xcccc_d000
addi x4, x4, -820	0xccc20213	x4=0xcccc_cccc
lui x5, 0x40000	0x400002b7	x5=0x4000_0000
sw x2, 0(x5)	0x0022a023	M[4000_0000] = 0xaaaa_aaaa
sh x3, 2(x5)	0x00329123	M[4000_0000] = 0xbbbb_aaaa
sb x4, 1(x5)	0x004280a3	M[4000_0000] = 0xbbbb_ccaa
lw x6, 0(x5)	0x0002a303	x6=0xbbbb_cca
lh x7, 0(x5)	0x00029383	x7=0xffff_ccaa
lhu x8, 2(x5)	0x0022d403	x8=0x0000_bbbb
lb x9, 3(x5)	0x00328483	x9=0xffff_ffbb
lbu x10, 1(x5)	0x0012c503	x10=0x0000_00cc

Hình 4.4 cho thấy kết giá trị thanh ghi x6, x7, x8, x9, x10 qua mô phỏng Vivado và phần mềm RARS là giống nhau cho thấy CPU xử lý các lệnh truy xuất ngoại vi chính xác.

Objects		Protocol Instances		Registers		Floating Point		Control and Status	
Name	Value			Name	Number			Value	
reg_32x32[31:0][31:0]	00000000,0000			zero	0			0x00000000	
> [12][31:0]	00000000			ra	1			0x00000003	
> [11][31:0]	00000000			sp	2			0xaaaaaaaa	
> [10][31:0]	000000cc			gp	3			0xbbbbbbbb	
> [9][31:0]	ffffffbb			tp	4			0xcccccccc	
> [8][31:0]	0000bbbb			t0	5			0x10010000	
> [7][31:0]	ffffccaa			t1	6			0xbbbbccaa	
> [6][31:0]	bbbcbcaa			t2	7			0xffffccaa	
> [5][31:0]	40000000			s0	8			0x0000bbbb	
> [4][31:0]	cccccccc			s1	9			0xffffffbb	
> [3][31:0]	bbbbbbbb			a0	10			0x000000cc	
> [2][31:0]	aaaaaaaa			a1	11			0x00000000	
> [1][31:0]	00000003			a2	12			0x00000000	
> [0][31:0]	00000000			a3	13			0x00000000	
> i[31:0]	X			a4	14			0x00000000	
				a5	15			0x00000000	
				a6	16			0x00000000	
				a7	17			0x00000000	
				s2	18			0x00000000	
				s3	19			0x00000000	
				s4	20			0x00000000	
				s5	21			0x00000000	
				s6	22			0x00000000	
				s7	23			0x00000000	
				s8	24			0x00000000	
				s9	25			0x00000000	
				s10	26			0x00000000	
				s11	27			0x00000000	
				t3	28			0x00000000	
				t4	29			0x00000000	
				t5	30			0x00000000	
				t6	31			0x00000000	
				pc				0x00400044	

Vivado

Phần mềm RARS

Hình 4.4 Kết quả so sánh thanh ghi giữa Vivado và phần mềm Rars

#### 4.1.3. Đánh giá tài nguyên sử dụng




Bảng 4.3 Bảng mô tả tài nguyên sử dụng của bộ xử lý RV32I

Tài nguyên	Đã sử dụng	Khả dụng	Tỉ lệ %
LUT	2511	303600	0.83
LUTRAM	176	130800	0.13
FF	1916	607200	0.32
BRAM	1	1030	0.10
IO	242	700	34.57

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.576 ns	Worst Hold Slack (WHS): 0.081 ns	Worst Pulse Width Slack (WPWS): 4.232 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 6273	Total Number of Endpoints: 6273	Total Number of Endpoints: 2063

All user specified timing constraints are met.

Hình 4.5 Kết quả timing sau khi gán constraint của bộ xử lý RV32I

			<b>Clock Summary</b>
Name	Waveform	Period (ns)	Frequency (MHz)
common	{0.000 5.000}	10.000	100.000

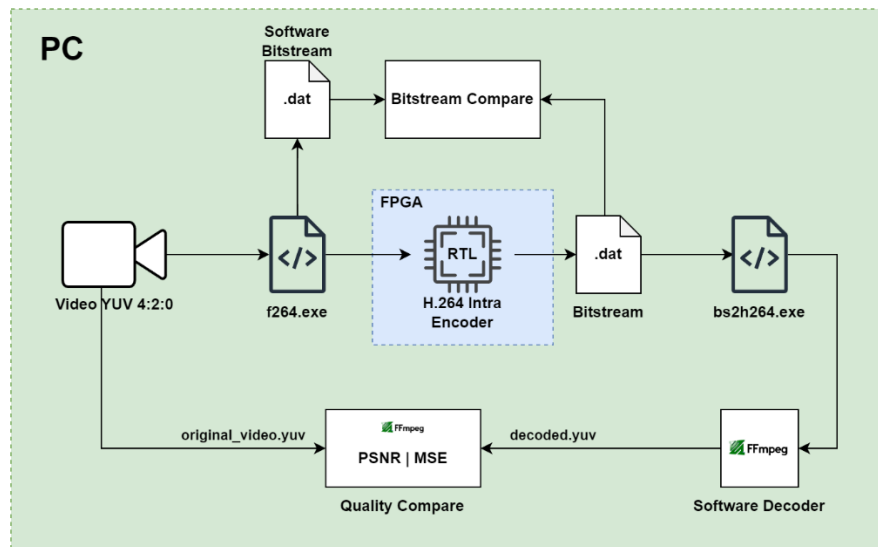
Hình 4.6 Mô tả constraint của bộ xử lý RV32I (100Mhz)



## 4.2. Bộ mã hoá video H264/AVC

### 4.2.1. Kịch bản kiểm tra

Với video YUV 4:2:0, nhóm sẽ sử dụng phần mềm Python do nhóm tự phát triển để phân tích các khung ảnh theo đúng yêu cầu đầu vào của bộ mã hoá với từng bộ macro-block 16x16 pixel. Sau đó, nhóm sẽ tiến hành nạp dữ liệu raw này vào IP bộ nhớ (giả lập DDRRAM 1GB để mô phỏng) hoặc truyền từ Ethernet từ PC vào FPGA và nạp vào DDRRAM 1GB trên board. Sau đó, hệ thống sẽ kích hoạt chế độ mã hoá video, kết quả của bộ mã hoá sẽ được kiểm tra so với dữ liệu được mã hoá bằng phần mềm, đồng thời sẽ được chuyển về chuẩn .264 dựa trên phần mềm. Cuối cùng, video sau khi được giải mã sẽ được đem so sánh chất lượng PSNR với video raw đầu vào thông qua phần mềm ffmpeg. Chi tiết được mô tả qua Hình 4.7:



Hình 4.7 Kịch bản kiểm tra bộ mã hoá video H.264/AVC

#### 4.2.2. Đánh giá hiệu suất

Nhóm tiến hành đánh giá bộ mã hoá H.264 với video FHD 1920x1080 có độ phức tạp cao với QP từ 18 đến 48. Hiện tại nhóm chỉ kiểm tra trên mô phỏng Vivado với tần số tối đa là 62.5Mhz và số frame hạn chế chỉ 50 frame nhưng đủ để bao phủ các thông số trung bình của bộ mã hoá. Chi tiết được mô tả qua bảng 4.4:

Bảng 4.4 Bảng kết quả hiệu suất của bộ mã hoá video H.264/AVC

QP	Số frame	FPS	Tỉ lệ nén	Thời gian nén	PSNR trung bình
18	50	31.714	26.92%	1.577 (s)	43.78 (dB)
28	50	34.96	10.19%	1.430 (s)	36.39 (dB)
38	50	37.93	3.47%	1.318 (s)	30.53 (dB)
48	50	39.69	1.30%	1.260 (s)	25.39 (dB)

Một số lưu ý:

- FPS được tính bằng cách tính số xung clock mà bộ H.264 sử dụng để mã hoá một khung hình từ khi có tín hiệu sys\_start cho đến khi tín hiệu sys\_done, thông số này không phản ánh FPS của toàn hệ thống mà chỉ phản ánh hiệu suất mã hoá riêng lẻ của bộ H.264.
- Tỉ lệ nén được tính bằng tỉ lệ giữa dung lượng file đã mã hoá (.264) và tỉ lệ video đầu vào (.yuv).
- PSNR được đo bằng phần mềm ffmpeg bằng cách so sánh điểm ảnh ở video được giải mã và video gốc ở cả 3 kênh y, u và v, sau đó được chia trung bình để lấy kết quả.

### 4.2.3. Đánh giá tài nguyên sử dụng

Bảng 4.5 Bảng mô tả tài nguyên của bộ mã hoá video H.264

Tài nguyên	Đã sử dụng	Khả dụng	Tỉ lệ %
LUT	34277	303600	11.29
LUTRAM	2912	130800	2.23
FF	13755	607200	2.27
BRAM	14.50	1030	1.41
DSP	108	2800	3.86
IO	236	700	33.71

Thông qua kết quả timing (Hình 4.8) và thông số constraint về clock (Hình 4.9) cho thấy bộ mã hoá này hoạt động tốt ở tần số 62.5Mhz và có các giao diện AXI hoạt động ở tần số 100Mhz đảm bảo việc tích hợp bộ mã hoá này vào hệ thống SoC của nhóm.

**Design Timing Summary**

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.075 ns	Worst Hold Slack (WHS): 0.074 ns	Worst Pulse Width Slack (WPWS): 4.232 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 47631	Total Number of Endpoints: 47631	Total Number of Endpoints: 16557

**All user specified timing constraints are met.**

Hình 4.8 Kết quả timing sau khi gán constraint của bộ mã hoá video H.264

**Clock Summary**

Name	Waveform	Period (ns)	Frequency (MHz)
axi_aclk	{0.000 5.000}	10.000	100.000
h264_aclk	{0.000 8.000}	16.000	62.500

Hình 4.9 Mô tả 2 clock domain của bộ mã hoá video H.264

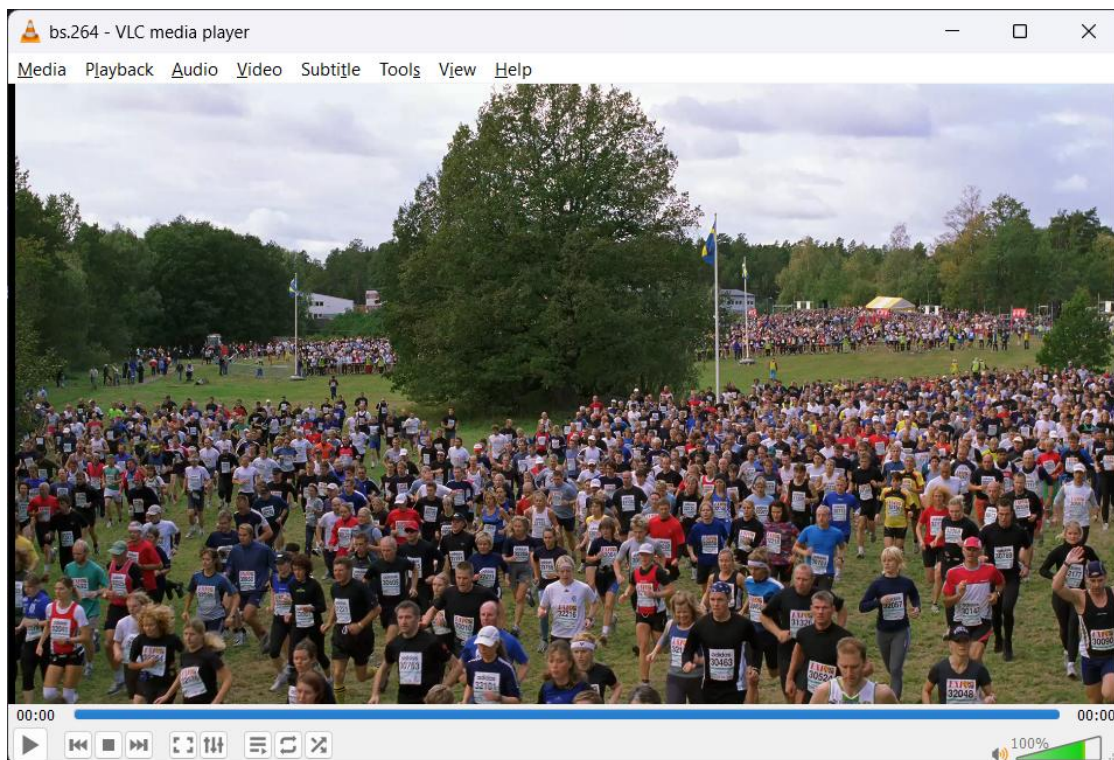
### 4.3. Kết quả thực thi của toàn bộ hệ thống SoC

Tính tới thời điểm hiện tại, nhóm chỉ mới triển khai thành công hệ thống SoC trên phần mềm mô phỏng Vivado, nên các kết quả bên dưới chỉ dừng lại ở việc mô phỏng waveform trên phần mềm Vivado. Hình 4.10 minh họa chương trình Vivado hiển thị ở TCL Console của toàn bộ hệ thống khi mô phỏng mã hoá 3 frame, đầu ra bitstream sẽ được sử dụng ở testbench để kiểm tra với kết quả phần mềm. Kết quả ‘MATCHED’ cho thấy byte từ phần cứng là giống với phần mềm, kết quả ‘MISMATCHED’ chỉ ra có sự sai khác so với phần mềm và dừng chương trình. Qua hình cho thấy luồng dữ liệu đầu vào, dữ liệu đầu ra của bộ mã hoá và vận chuyển bởi DMA là chính xác.

```
MATCHED: check_data(f7) == bs_data(f7)
MATCHED: check_data(22) == bs_data(22)
MATCHED: check_data(12) == bs_data(12)
MATCHED: check_data(3f) == bs_data(3f)
MATCHED: check_data(e2) == bs_data(e2)
MATCHED: check_data(51) == bs_data(51)
MATCHED: check_data(f4) == bs_data(f4)
MATCHED: check_data(7d) == bs_data(7d)
MATCHED: check_data(d4) == bs_data(d4)
MATCHED: check_data(b2) == bs_data(b2)
MATCHED: check_data(ef) == bs_data(ef)
MATCHED: check_data(59) == bs_data(59)
MATCHED: check_data(7e) == bs_data(7e)
MATCHED: check_data(48) == bs_data(48)
MATCHED: check_data(14) == bs_data(14)
MATCHED: check_data(de) == bs_data(de)
MATCHED: check_data(f6) == bs_data(f6)
MATCHED: check_data(f0) == bs_data(f0)
FRAME 2 DONE!
Executing Axi4 End Of Simulation checks
$finish called at time : 8388190 ns : File "D:/Workspaces/RTL/KLTN_sourcecode/rtl_2022_2/KLTN_FINAL/
run: Time (s): cpu = 00:02:27 ; elapsed = 00:10:10 . Memory (MB): peak = 1488.191 ; gain = 80.434
```

Hình 4.10 Kết quả mô phỏng hệ thống SoC

Sau khi kết thúc, chương trình sẽ sinh ra một file rtl.dat có format theo chương trình C từ nhà cung cấp. File này sẽ được phần mềm chuyển sang file dạng bs.264. Đây chính là file đã mã hoá theo chuẩn H.264 và mở được bằng phần mềm VLC (minh họa hình 4.11).



Hình 4.11 Kiểm tra video đã mã hoá .264 bằng phần mềm VLC

Ngoài ra, mô phỏng của nhóm sẽ tạo thêm một file summary kết quả FPS của toàn bộ quá trình (hình 4.12) mô tả thời gian hoàn thành, FPS của từng khung hình và của toàn bộ chuỗi khung video.

```

Frame 33: 35.001887 FPS -- Number of clk pulses: 1785618
Frame 34: 34.985899 FPS -- Number of clk pulses: 1786434
Frame 35: 35.003965 FPS -- Number of clk pulses: 1785512
Frame 36: 34.993539 FPS -- Number of clk pulses: 1786044
Frame 37: 34.927167 FPS -- Number of clk pulses: 1789438
Frame 38: 34.966737 FPS -- Number of clk pulses: 1787413
Frame 39: 35.002083 FPS -- Number of clk pulses: 1785608
Frame 40: 34.925879 FPS -- Number of clk pulses: 1789504
Frame 41: 34.918093 FPS -- Number of clk pulses: 1789903
Frame 42: 34.918952 FPS -- Number of clk pulses: 1789859
Frame 43: 34.980436 FPS -- Number of clk pulses: 1786713
Frame 44: 34.932418 FPS -- Number of clk pulses: 1789169
Frame 45: 34.958366 FPS -- Number of clk pulses: 1787841
Frame 46: 34.920200 FPS -- Number of clk pulses: 1789795
Frame 47: 34.930661 FPS -- Number of clk pulses: 1789259
Frame 48: 34.962238 FPS -- Number of clk pulses: 1787643
Frame 49: 34.936929 FPS -- Number of clk pulses: 1788938
AVERAGE FPS: 34.960943
ENCODING TIME: 1.430167

```

Hình 4.12 File tổng hợp kết quả mô phỏng dựa trên số xung clock

Để đánh giá hiệu suất, nhóm sử dụng phần mềm ffmpeg để giải mã video từ file .264 ở trên về chuẩn video .yuv. Sau đó, nhóm sẽ so sánh video decoded.yuv này so với video original.yuv gốc ban đầu, kết quả so sánh cho ra được mô tả khái quát qua hình 4.13:

```
n:37 mse_avg:14.65 mse_y:16.83 mse_u:11.00 mse_v:9.61 psnr_avg:36.47 psnr_y:35.87 psnr_u:37.72 psnr_v:38.30
n:38 mse_avg:14.86 mse_y:17.11 mse_u:11.08 mse_v:9.65 psnr_avg:36.41 psnr_y:35.80 psnr_u:37.69 psnr_v:38.29
n:39 mse_avg:14.93 mse_y:17.17 mse_u:11.15 mse_v:9.74 psnr_avg:36.39 psnr_y:35.78 psnr_u:37.66 psnr_v:38.24
n:40 mse_avg:15.03 mse_y:17.31 mse_u:11.18 mse_v:9.74 psnr_avg:36.36 psnr_y:35.75 psnr_u:37.65 psnr_v:38.25
n:41 mse_avg:14.95 mse_y:17.20 mse_u:11.15 mse_v:9.73 psnr_avg:36.38 psnr_y:35.77 psnr_u:37.66 psnr_v:38.25
n:42 mse_avg:14.91 mse_y:17.14 mse_u:11.18 mse_v:9.72 psnr_avg:36.40 psnr_y:35.79 psnr_u:37.65 psnr_v:38.25
n:43 mse_avg:14.74 mse_y:16.93 mse_u:11.10 mse_v:9.63 psnr_avg:36.45 psnr_y:35.85 psnr_u:37.68 psnr_v:38.29
n:44 mse_avg:14.73 mse_y:16.91 mse_u:11.11 mse_v:9.62 psnr_avg:36.45 psnr_y:35.85 psnr_u:37.67 psnr_v:38.30
n:45 mse_avg:14.72 mse_y:16.90 mse_u:11.10 mse_v:9.60 psnr_avg:36.45 psnr_y:35.85 psnr_u:37.68 psnr_v:38.31
n:46 mse_avg:14.54 mse_y:16.66 mse_u:11.04 mse_v:9.56 psnr_avg:36.51 psnr_y:35.91 psnr_u:37.70 psnr_v:38.33
n:47 mse_avg:14.86 mse_y:17.08 mse_u:11.18 mse_v:9.66 psnr_avg:36.41 psnr_y:35.81 psnr_u:37.65 psnr_v:38.28
n:48 mse_avg:14.69 mse_y:16.88 mse_u:11.04 mse_v:9.56 psnr_avg:36.46 psnr_y:35.86 psnr_u:37.70 psnr_v:38.33
n:49 mse_avg:14.85 mse_y:17.08 mse_u:11.14 mse_v:9.62 psnr_avg:36.41 psnr_y:35.80 psnr_u:37.66 psnr_v:38.30
n:50 mse_avg:14.89 mse_y:17.15 mse_u:11.12 mse_v:9.64 psnr_avg:36.40 psnr_y:35.79 psnr_u:37.67 psnr_v:38.29
```

Hình 4.13 Kết quả PSNR tính từ phần mềm ffmpeg

Nhóm sẽ đánh giá hiệu suất nén ảnh qua tiêu chí PSNR với công thức tính:

$$\text{PSNR} = 10 \cdot \log_{10} \frac{\text{MAX}_I^2}{\text{MSE}} \quad (4.1)$$

Với:

- PSNR: độ đo (đơn vị dB).
- MAX: giá trị pixel tối đa (ở đây là 255).
- MSE (Mean Square Error): độ lệch giữa pixel gốc và pixel tái tạo:

$$\text{MSE} = \frac{1}{m \cdot n} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i, j) - K(i, j)]^2 \quad (4.2)$$

- $I(i, j)$ : pixel tại vị trí  $(i, j)$  trong ảnh gốc.
- $K(i, j)$ : pixel tại vị trí  $(i, j)$  trong ảnh tái tạo.
- $M, n$ : kích thước ảnh.

#### 4.4. Đánh giá thiết kế và so sánh với các đề tài khác

##### 4.4.1. Đánh giá bộ xử lý RV32I

Bảng 4.6 Bảng so sánh thiết kế RV32I của nhóm với các đề tài khác

Nội dung so sánh	Thiết kế của nhóm	Thiết kế của tác giả [18]	Thiết kế của tác giả [19]
ISA	RISC-V	RISC-V	RISC-V
Phần mở rộng	RV32I	RV32I	RV32I
Tần số hoạt động tối đa	100Mhz trên Virtex-7	Dự kiến 100Mhz	Dự kiến 166,67Mhz
Pipeline	5 tầng	5 tầng	5 tầng
Tính năng nổi bật	Giao diện AXI-Lite (master, slave) Triển khai xuống FPGA thành công Tích hợp SoC gồm: DMA, H.264, bộ nhớ và hệ thống hoạt động đúng chức năng qua mô phỏng.	Kiến trúc supperscalar - Tích hợp vào SoC: Cache, BRAM, cầu AXI-APB , CDMA (Chưa kết nối với ngoại vi)	Kết nối được với System Cache

#### 4.4.2. Đánh giá bộ mã hoá video H.264

Bảng 4.7 So sánh thiết kế bộ mã hoá video H.264 so với các đề tài khác

Nội dung so sánh	Thiết kế của nhóm	Thiết kế của đề tài [6]	Thiết kế của đề tài [7]
Chế độ dự đoán	Intra	Intra	Intra
Bộ điều khiển	RISC-V RV32I tự phát triển	ARM A9 tích hợp sẵn	N/A
Nền tảng FPGA	Virtex-7 VC707	Zynq-7000	Zynq-7000/Artix-7 / Spartan-7
Tần số hoạt động tối đa	62.5Mhz sau implement	190Mhz sau synthesis	158–183 MHz (tùy cấu hình, đạt 2.3 Gbps ở 183 MHz)
Độ phân giải	1920x1080	640x460	1920x1080
FPS	34.96	21.69	N/A
PSNR trung bình	36.39 (dB) (ở QP=28)	38.19 (dB) (ở QP=28)	32.7 dB (QP = 23, cấu hình 9-mode Intra)
Tỉ lệ nén (QP=28)	10.19%	4.94%	2.7:1 ( $\approx 37\%$ ) tương đương với $\sim 4.4$ bpp từ 12 bpp gốc
Tính năng nổi bật	Giao diện AXI (lite, stream) Tích hợp vào hệ thống SoC thành công.	Giao diện AXI. Tích hợp SoC: VDMA, HDMI, USB	Có thể cấu hình chế độ dự đoán Hỗ trợ nhiều độ sâu bit (8, 10, 12 bit) Hỗ trợ pipeline



## **Chương 5. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN**

### **5.1. Kết luận**

- Nhóm đã xây dựng và mô phỏng thành công một hệ thống SoC tích hợp bộ xử lý RISC-V RV32I và bộ mã hoá video H.264 Intra-frame.
- Thiết kế và hiện thực thành công một bộ xử lý RISC-V RV32I hoạt động ổn định với tần số 100Mhz với kiến trúc pipeline 5 tầng, hỗ trợ đầy đủ 37 lệnh và giao diện truy xuất ngoại vi thông qua AXI-Lite.
- Hiểu được cơ chế hoạt động của bộ mã hoá H.264 thông qua mô phỏng Matlab.
- Tích hợp thành công lõi mã hoá H.264 vào hệ thống đạt tần số hoạt động tối đa 62,5 MHz có giao diện AXI và xử lý CDC để kết nối an toàn giữa 2 miền xung nhịp.

### **5.2. Hướng phát triển**

Về CPU RISC-V:

- Tích hợp branch prediction nhằm cải thiện hiệu suất xử lý nhánh.
- Bổ sung cơ chế ngắt và cờ trạng thái, hỗ trợ điều khiển ngoại vi.
- Thêm i-cache và d-cache để tăng tốc truy xuất bộ nhớ chương trình và dữ liệu.
- Áp dụng cơ chế phân quyền truy cập (protection), nâng cao tính an toàn trong hệ thống nhúng đa nhiệm.

Về bộ mã hóa H.264:

- Tối ưu logic và pipelining để tăng tần số hoạt động, qua đó nâng cao FPS.
- Nghiên cứu mở rộng sang inter-prediction để tăng hiệu quả nén toàn cảnh.
- Tích hợp thêm khối lọc Deblocking Filter, cải thiện chất lượng hình ảnh đầu ra.
- Nâng cấp từ CAVLC lên CABAC (Context-Adaptive Binary Arithmetic Coding) để đạt tỷ lệ nén tốt hơn ở cùng chất lượng.

## TÀI LIỆU THAM KHẢO

- [1] Iain E. G. Richardson, “H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia”, Second Edition. England: John Wiley & Sons, Ltd, 2010.
- [2] Joint Video Team (JVT) of ITUT VCEG and ISO/IEC MPEG, “Draft ITUT Recommendation and Final Draft International Standard of Joint Video Specification (ITUT Rec. H.264 and ISO/IEC 1449610 AVC),” May 2003
- [3] ITU-T Rec. H.263, “Video Codec for Low Bit Rate Communication”, 1996.
- [4] Lam, D.K.; Nguyen, P.T.A.; Tran, T.A. Hardware Architecture for Realtime HEVC Intra Prediction. Electronics 2023, 12, 1705  
<https://doi.org/10.3390/electronics12071705>.
- [5] Nejmeddine Bahri, Imen Werda, Thierry Grandpierre, Mohamed Ali Ben Ayed, Nouri Masmoudi, et al.. Optimizations for real-time implementation of H264/AVC video encoder on DSP processor. International Review on Computers and Software (IRECOS), 2013.
- [6] Z. Li, J. Li, Y. Zhao, C. Rong and J. Ma, "A SoC Design and Implementation of H.264 Video Encoding System Based on FPGA," 2014 Sixth International Conference on Intelligent Human-Machine Systems and Cybernetics, Hangzhou, China, 2014, pp. 321-324, doi: 10.1109/IHMSC.2014.179.
- [7] Azam Tayyebi, Darrin Hanna, Bryant Jones, “Parametrized low-complexity hardware architecture of an H.264-based video encoder for FPGAs,” Microprocessors and Microsystems, Volume 105, 2024, 105017, ISSN 0141-9331, <https://doi.org/10.1016/j.micpro.2024.105017>.
- [8] The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2”, Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, May 2017.
- [9] “[Background] SoC - System on a Chip,” [Background] SoC - System on a Chip ~ VLSI TECHNOLOGY, 2018.

<https://nguyenquanicd.blogspot.com/2018/05/backgroundsoc-system-on-chip.html>  
(truy cập 17/02/2025).

[10] Q. N. Nguyen, “Multi-clock design – Bài 3: Kỹ thuật đồng bộ tín hiệu nhiều bit,” Nguyễn Quân – IC Design, Feb. 2020. [Online]. Available:  
<https://nguyenquanicd.blogspot.com/2020/02/multi-clock-design-bai-3-ky-thuat-ong.html> (truy cập 17/2/2025).

[11] Xilinx Inc., UG887: VC707 Evaluation Board for the Virtex-7 FPGA, 2019,  
<http://www.xilinx.com/>.

[12] Xilinx Inc., PG021: AXI DMA LogiCORE IP Product Guide, 2024  
<http://www.xilinx.com/>.

[13] ARM Ltd., "AMBA AXI Protocol Specification", Version 2.2. England:  
ARM Holdings, 2023.

[14] Xilinx, “AXI Verification IP (PG267),” Product Guide, v1.0, Mar. 2017.  
[Online]. Available:  
[https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_vip/v1\\_0/pg267-axi-vip.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_vip/v1_0/pg267-axi-vip.pdf)

[15] Xilinx, “AXI 1G/2.5G Ethernet Subsystem (PG138),” Product Guide, v7.2,  
Apr. 2021. [Online]. Available:  
[https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_ethernet/v7\\_2/pg138-axi-ethernet.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_ethernet/v7_2/pg138-axi-ethernet.pdf)

[16] Xilinx, “AXI UART Lite (PG142),” Product Guide, v2.0, Oct. 2020.  
[Online]. Available:  
[https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_uartlite/v2\\_0/pg142-axi-uartlite.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_uartlite/v2_0/pg142-axi-uartlite.pdf)

[17] Xilinx, “AXI GPIO (PG144),” Product Guide, v2.0, Sep. 2020. [Online].  
Available:  
[https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_gpio/v2\\_0/pg144-axi-gpio.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf)

- [18] Phan Lê Min, Võ Phan Hoàng Kha, “Mô phỏng và hiện thực hệ thống SoC với RISC-V 32 bit CPU trên FPGA,” Trường Đại học Công nghệ Thông tin, Hồ Chí Minh, 2024.
- [19] Vũ Duy Di Đan, Phạm Quang Hải, “Mô phỏng và hiện thực 32-bit RISC-V CPU trên FPGA,” Trường Đại học Công Nghệ Thông Tin, Hồ Chí Minh, 2022.
- [20] [xk264 - H.264 Encoder IP Core], OpenASIC Organization, GitHub repository: <https://github.com/openasic-org/xk264>, truy cập ngày 17/02/2025.
- [21] A. Shriram, \*RISC-V Instruction Card\*. [Online]. Available: [https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/notebooks/RISCV/RISCV\\_CARD.pdf](https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/notebooks/RISCV/RISCV_CARD.pdf)