

**Đại học Quốc gia Thành phố Hồ Chí Minh**  
**Trường Đại học Bách Khoa**  
**Khoa Khoa học và Kỹ thuật Máy tính**



## **Luận Văn Tốt Nghiệp**

**Đề tài:** Tìm hiểu bộ xử lý RISC-V và ánh xạ một hiện thực của nó xuống FPGA

Hội đồng : Kỹ Thuật Máy Tính  
Giáo viên hướng dẫn : TS. Phạm Quốc Cường  
Giáo viên phản biện : PGS. TS. Trần Ngọc Thịnh

Nhóm sinh viên thực hiện : Nguyễn Lương Phúc Vinh - 1514063  
Bùi Lương Bảo - 1510146

Ngày 14 tháng 6 năm 2019

# Lời cam đoan

Nhóm cam đoan mọi điều được ghi trong báo cáo, cũng như mã nguồn là do nhóm tự thực hiện - trừ các kiến thức tham khảo có trích dẫn cũng như mã nguồn mẫu do chính nhà sản xuất cung cấp, hoàn toàn không sao chép từ bất cứ nguồn nào khác. Nếu lời cam đoan trái với sự thật, nhóm xin chịu mọi trách nhiệm trước Khoa và Nhà trường.

Nhóm sinh viên thực hiện đề tài

# Lời cảm ơn

Sau bốn năm theo học tại khoa Khoa học Kỹ thuật Máy tính trường Đại học Bách khoa TP. Hồ Chí Minh, em đã được áp dụng tất cả những kiến thức đã học để thực hiện luận văn tốt nghiệp, đề tài cuối cùng mang tính quyết định kết quả học tập và quá trình tốt nghiệp.

Em xin chân thành cảm ơn quý thầy cô trong Khoa đã mang đến những kiến thức bổ ích trong suốt quãng đời sinh viên, đặc biệt là thầy Phạm Quốc Cường đã giúp đỡ em trong quá trình thực hiện cũng như quá trình hoàn thiện luận văn. Em cũng gửi lời cảm ơn đến một số bạn bè trong Khoa đã hỗ trợ giúp em tiết kiệm được thời gian tìm hiểu đề tài.

Em đã cố gắng hết sức để tránh các sai sót, nhưng nếu có phát hiện thì mong quý thầy cô góp ý để em càng hoàn thiện đề tài hơn. Cuối cùng em xin gửi lời chúc sức khỏe và cảm ơn chân thành nhất.

Nhóm sinh viên thực hiện đề tài

# Mục lục

## Lời cam đoan

## Lời cảm ơn

i

## Mục Lục Hình

v

## Danh Sách Bảng

v

## Thuật ngữ & từ viết tắt

viii

### 1 Giới thiệu đề tài

1

1.1	Tóm tắt kết quả đề cương luận văn . . . . .	1
1.2	Mục tiêu và phạm vi đề tài . . . . .	3
1.2.1	Lý thuyết . . . . .	3
1.2.2	Hiện thực . . . . .	3
1.3	Sơ lược về RISC-V . . . . .	3
1.3.1	Kiến trúc tập lệnh RISC . . . . .	3
1.3.2	Phần cứng mã nguồn mở . . . . .	4
1.3.3	RISC-V . . . . .	6
1.3.4	PULP và PULPino . . . . .	7
1.4	Giới thiệu tinh chỉnh lõi phần cứng . . . . .	8

### 2 Tổng quan về lõi RI5CY

13

2.1	Giới thiệu và mô hình . . . . .	13
2.2	Các thành phần trong core RI5CY . . . . .	14

### 3 Các thao tác với ZedBoard

21

3.1	Tổng quan mô hình và các thành phần của ZedBoard . . . . .	23
3.2	Thiết lập phần cứng cho ZedBoard . . . . .	24

ii

3.3	Các tính năng của ZedBoard . . . . .	27
3.3.1	Thao tác với GPIO Switches và LEDs . . . . .	27
3.3.2	Thao tác với OLED . . . . .	28
3.3.3	Thao tác với VGA và HDMI . . . . .	29
3.3.4	Thao tác với Ethernet . . . . .	29
3.3.5	Thao tác với SD Card . . . . .	30
<b>4</b>	<b>Hiện thực ánh xạ lõi RI5CY lên ZedBoard</b>	<b>31</b>
4.1	Lab 1: Build hệ điều hành và core RI5CY để boot lên ZedBoard . .	31
4.2	Lab 2: Compile và chạy ứng dụng lên ZedBoard . . . . .	33
4.3	Tạo một ứng dụng riêng để nạp lên ZedBoard . . . . .	33
4.4	GPIO với Zedboard . . . . .	33
<b>5</b>	<b>Kết quả thực hiện</b>	<b>37</b>
5.1	Kết quả khi build lõi RI5CY . . . . .	37
5.2	Kết quả khi chạy thử tập lệnh cơ sở . . . . .	41
5.3	Kết quả khi build ứng dụng . . . . .	44
<b>6</b>	<b>Kết luận</b>	<b>46</b>
6.1	Đánh giá kết quả . . . . .	46
6.1.1	Kết quả đã thực hiện được . . . . .	46
6.1.2	Hạn chế . . . . .	46
6.2	Hướng phát triển . . . . .	47
<b>7</b>	<b>Phụ lục A</b>	<b>48</b>
<b>8</b>	<b>Phụ lục B</b>	<b>53</b>
8.1	Lab1: Build hệ điều hành và core RI5CY để boot lên ZedBoard . .	53
8.1.1	Giới thiệu . . . . .	53
8.1.2	Build RI5CY core . . . . .	55
8.1.3	Chuẩn bị SD Card . . . . .	56
8.1.4	Boot hệ điều hành lên ZedBoard . . . . .	60
8.2	Lab2: Compile và chạy ứng dụng lên ZedBoard . . . . .	61
8.2.1	Thực hành . . . . .	62
8.3	Lab3: Tạo một ứng dụng riêng để nạp lên ZedBoard . . . . .	64
8.3.1	Giới thiệu . . . . .	64
8.3.2	Thực hành . . . . .	64

8.4	Lab4: GPIO với Zedboard . . . . .	66
8.4.1	Giới thiệu . . . . .	66
8.5	Thực hành . . . . .	67
8.6	Interrupt . . . . .	70
8.6.1	Giới thiệu . . . . .	70
8.6.2	External interrupt . . . . .	71
8.6.3	Timer Interrupt . . . . .	73
	Tài liệu tham khảo	76

# Danh sách hình vẽ

1.1	Speedup của RI5CY so với các lõi khác [15] . . . . .	2
1.2	So sánh RI5CY và ARM Cortex M4 [15] . . . . .	2
1.3	Biểu đồ chi phí cho việc phát triển chip . . . . .	5
1.4	So sánh sự thành công giữa phần mềm mã nguồn mở và phần cứng mã nguồn mở . . . . .	6
1.5	Sơ đồ khối của PULPino . . . . .	7
1.6	Các tùy chỉnh về bộ nhớ . . . . .	9
1.7	Các tùy chỉnh về chế độ và kiến trúc tập lệnh . . . . .	10
1.8	Các tùy chỉnh về debug . . . . .	10
1.9	Các tùy chỉnh về ngắt . . . . .	11
1.10	Các tùy chỉnh về tiên đoán rẽ nhánh . . . . .	11
1.11	Minh họa của việc sau khi tinh chỉnh core trên SiFive, tiếp theo ta chỉ cần tải các file này về và nạp xuống dưới board để chạy. . . . .	12
2.1	Kiến trúc của lõi RI5CY . . . . .	13
2.2	Cách hoạt động của bộ LSU . . . . .	15
2.3	Hoạt động của bộ LSU khi có memory chưa ghi xong . . . . .	15
2.4	RI5CY Pipeline . . . . .	17
3.1	Zedboard Zynq-7000 ARM/FPGA . . . . .	21
3.2	Sơ đồ khối của ZedBoard . . . . .	23
3.3	ZedBoard Bank Assignment . . . . .	24
3.4	Thiết lập các jumper để boot bằng thẻ SD . . . . .	25
3.5	Thiết lập Tera Term để vào hệ điều hành trên SD Card . . . . .	26
3.6	Màn hình terminal sau khi boot . . . . .	26
3.7	Sơ đồ khái niệm các tính năng của ZedBoard . . . . .	27
3.8	Thao tác với SD Card . . . . .	30

4.1	Quy trình ánh xạ lõi RI5CY . . . . .	32
4.2	Quy trình nạp một ứng dụng tự viết lên board . . . . .	34
4.3	Thư mục ips đã được cập nhật và chứa các lõi được pulpino sử dụng, đồng thời, ipstools cũng được clone về để hỗ trợ make . . . . .	34
5.1	Kết quả chạy thử một số lệnh cơ sở . . . . .	43
5.2	Build ứng dụng với RI5CY toolchain . . . . .	44
5.3	Nạp file spi_stim.txt với ./spiloader . . . . .	45
5.4	Demo gpio điều khiển led bằng switch . . . . .	45
8.1	Sơ đồ khối của PULPino . . . . .	54
8.2	Kiến trúc của lõi RI5CY . . . . .	54
8.3	Cross Platform ToolChain giữa máy host và board mạch . . . . .	62
8.4	Output sau khi thực hiện . . . . .	63
8.5	Quy trình nạp một ứng dụng tự viết lên board . . . . .	64
8.6	ZedBoard Block Diagram . . . . .	66
8.7	Event lines . . . . .	70
8.8	IER register . . . . .	70
8.9	ECP register . . . . .	71
8.10	ICP register . . . . .	71
8.11	INTEN register . . . . .	71
8.12	INTSTATUS register . . . . .	72

# Danh sách bảng

2.1	Instruction Fetch Signals . . . . .	14
2.2	LSU Signals . . . . .	15
2.3	Control and Status Register Map . . . . .	18
5.1	Report BlackBoxes . . . . .	37
5.2	Report Cell Usage . . . . .	38
5.3	Report Instance Areas . . . . .	39
5.4	Utilization Report . . . . .	39
5.5	So sánh kết quả build với tài nguyên của Zynq-7000 XC7Z020 . . . . .	40
5.6	Một số lệnh cơ sở RV32I . . . . .	41
7.1	Hardware Loops Operations . . . . .	48
7.2	Hardware Loops Encoding . . . . .	49
7.3	ALU Bit Manipulation Operations . . . . .	49
7.4	ALU Bit Manipulation Encoding . . . . .	50
7.5	Immediate Branching Operations . . . . .	50
7.6	Immediate Branching Encoding . . . . .	50
7.7	General ALU Operations . . . . .	51
7.8	General ALU Encoding . . . . .	52
8.1	Thiết lập chế độ boot trên ZedBoard . . . . .	60
8.2	Các Jumper dùng cho việc boot hệ điều hành . . . . .	60

# Thuật ngữ & từ viết tắt

<b>AXI</b> .....	Advanced Extensible Interface
<b>FPU</b> .....	Floating Point Unit
<b>JTAG</b> .....	Joint (European) Test Access Group
<b>LPC FMC</b> ...	Low Pin Count FPGA Mezzanine Card
<b>QSPI</b> .....	Queued Serial Peripheral Interface
<b>PS</b> .....	Processing System
<b>PL</b> .....	Programmable Logic
<b>RISC</b> .....	Reduced Instruction Set Computers
<b>CSR</b> .....	Control Status Register
<b>LSP</b> .....	Line Spectrum Pair
<b>BUFG</b> .....	Global Clock Simple Buffer
<b>CARRY4</b> ....	Fast Carry Logic with Look Ahead
<b>DSP48E1</b> ....	48-bit Multi-Functional Arithmetic Block
<b>LUT1</b> .....	1-Bit Look-Up table
<b>LUT2</b> .....	2-Bit Look-Up table
<b>LUT3</b> .....	3-Bit Look-Up Table
<b>LUT4</b> .....	4-Bit Look-Up Table

**LUT5** ..... 5-Input Look-Up Table  
**LUT6** ..... 6-Input Look-Up Table  
**MUXF7** ..... 2-to-1 Look-Up Table Multiplexer for 7-input function  
**MUXF8** ..... 2-to-1 Look-Up Table Multiplexer for 8-input function  
**FDCE** ..... D Flip-Flop with Clock Enable and Asynchronous Clear  
**FDPE** ..... D Flip-Flop with Clock Enable and Asynchronous Preset  
**FDRE** ..... D Flip-Flop with Clock Enable and Synchronous Reset  
**IBUF** ..... Input Buffer  
**OBUF** ..... Output Buffer

# 1 Giới thiệu đề tài

## 1.1 Tóm tắt kết quả đề cương luận văn

Trong giai đoạn đề cương, nhóm đã hoàn thành được các tiêu chí đã đưa ra như sau:

- Hiểu được RISC-V là gì và ý nghĩa của nó đối với phần cứng mã nguồn mở.
- Các công cụ hỗ trợ RISC-V như ngôn ngữ Chisel, bộ mô phỏng Firesim và các development kits dùng để hiện thực được RISC-V.
- Tìm hiểu về ba lõi phổ biến của RISC-V là Rocket Core, BOOM của Rocket Chip và RI5CY của PULP theo kiến trúc tập lệnh, kiến trúc lõi, hiệu năng, độ phức tạp... từ đó so sánh và chọn được lõi thích hợp để đưa vào đề tài luận văn.

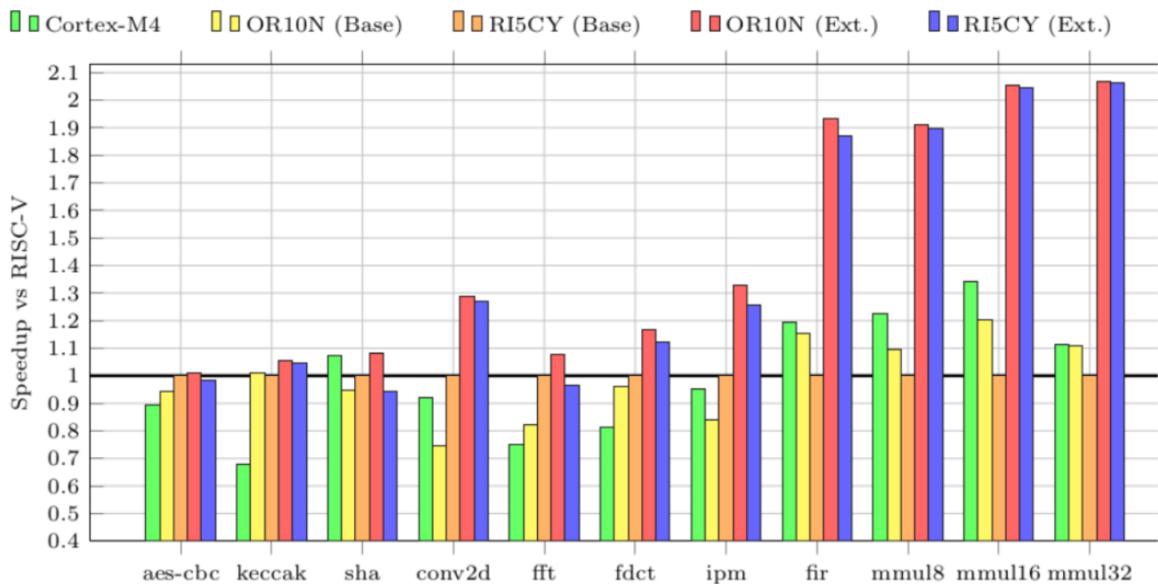
Từ các kết quả trên, nhóm quyết định chọn lõi RI5CY để hiện thực cho giai đoạn luận văn, dựa vào một số tính năng nổi trội của RI5CY:

- **Hỗ trợ đầy đủ kiến trúc 32-bit:** RI5CY hỗ trợ tương đối đầy đủ kiến trúc 32-bit của RISC-V; bao gồm: RV32I, RV32M, RV32F,... Ngoài ra còn hỗ trợ các phần mở rộng như dấu chấm động, ALU, bộ load/store post-increment, và hardware loops.
- **Chi phí cho board rẻ:** chỉ với tầm 25\$, ta có thể sở hữu một mạch để chạy RI5CY, kích cỡ arduino, so với các core khác như Rocket, BOOM, yêu cầu

## 1.1. Tóm tắt kết quả đề cương luận văn

phần cứng hỗ trợ kiến trúc 64-bit, chi phí cho một board như vậy rất đắt, như đã minh họa ở trên (tầm 599\$ để có ZedBoard chạy Rocket).

- **Biên dịch, mô phỏng đơn giản:** ta chỉ cần có GCC và kiến thức về ngôn ngữ C để có thể chạy các sản phẩm của PULP, công cụ mô phỏng có thể sử dụng ModelSim để chạy mô phỏng.
- **Hiệu năng tốt, kích thước core nhỏ:** so sánh với các board cùng ISA 32-bit, RI5CY có speedup tương đối tốt (sử dụng bộ compiler GCC 5.2 trong quá trình so sánh) với kích thước và hiệu năng tiêu thụ nhỏ.



Hình 1.1: Speedup của RI5CY so với các lõi khác [15]

	RI5CY	ARM Cortex M4 <sup>2</sup>	
Technology	65 nm	90nm	65 nm
Conditions	25°C, 1.2V	25°C, 1.2V	25°C, 1.2V
Dynamic Power [ $\mu W / MHz$ ]	17.5	32.82	23.2 <sup>1</sup>
Area [ $mm^2$ ]	0.050	0.119	0.062 <sup>1</sup>

Hình 1.2: So sánh RI5CY và ARM Cortex M4 [15]

Trong giai đoạn luận văn nhóm sẽ thực hiện công việc ánh xạ xuống bảng mạch và tiến hành tinh chỉnh lõi RI5CY.

## 1.2 Mục tiêu và phạm vi đề tài

### 1.2.1 Lý thuyết

- Tìm hiểu về lõi RI5CY.
- Tìm hiểu về ZedBoard và các thao tác cơ bản với ZedBoard.
- Tìm hiểu về PULPino để hỗ trợ việc ánh xạ RI5CY xuống ZedBoard.
- Tìm hiểu về cách tinh chỉnh một core RISC-V.

### 1.2.2 Hiện thực

- Thực hiện ánh xạ lõi RI5CY xuống ZedBoard.
- Chạy thử nghiệm các câu lệnh để kiểm tra thử hiện thực.
- Thủ nghiệm tinh chỉnh với lõi RI5CY.

## 1.3 Sơ lược về RISC-V

### 1.3.1 Kiến trúc tập lệnh RISC

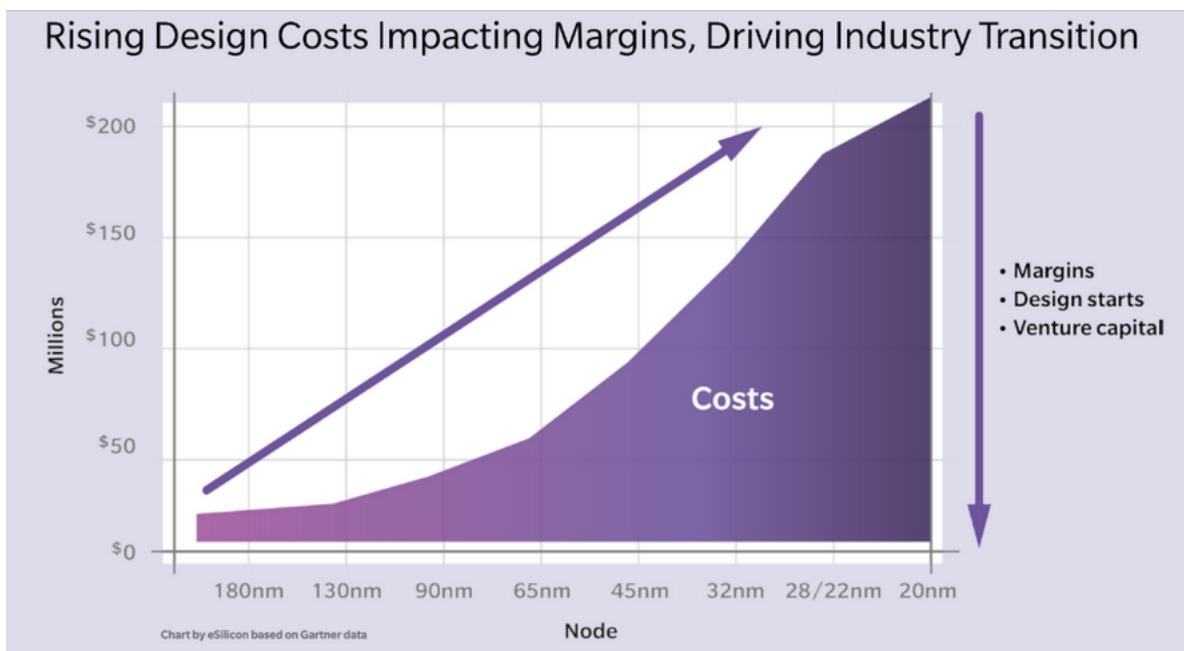
RISC (Reduced Instruction Set Computers) là một kiến trúc vi xử lý thiết kế theo hướng đơn giản hóa tập lệnh, trong đó thời gian thực thi tất cả các lệnh đều như nhau. Khác với hướng tiếp cận của CISC, RISC cố gắng giảm số lượng thao tác trên một câu lệnh nên câu lệnh sẽ trở nên đơn giản. Vì xử lý RISC nhấn mạnh tính đơn giản và hiệu quả. Các thiết kế RISC khởi đầu với tập lệnh thiết yếu và vừa đủ. RISC tăng tốc độ xử lý bằng cách giảm số chu kỳ đồng hồ trên một lệnh. Mục đích của RISC là tăng tốc độ hiệu dụng bằng cách chuyển việc thực hiện các tác vụ không thường xuyên vào phần mềm, những tác vụ phổ biến do phần cứng thực hiện nhằm tăng hiệu năng của máy tính. Vì xử lý RISC thường phù hợp với các ứng dụng điều khiển hay nhúng như máy in laser, máy in đa chức năng. Vì xử lý RISC cũng rất phù hợp với các ứng dụng như xử lý ảnh, robot và đồ họa nhờ có mức tiêu thụ điện thấp, thực thi nhanh chóng.

Đặc điểm của RISC:

- Các lệnh đơn giản: các kiến trúc RISC sử dụng các lệnh đơn giản hơn với độ dài cố định và không có các lệnh kết hợp load/store với số học.
- Ít kiểu dữ liệu: RISC hỗ trợ một vài kiểu dữ liệu đơn giản một cách hiệu quả và các kiểu dữ liệu kết hợp/phức tạp được tổng hợp từ chúng.
- Các chế độ định địa chỉ (addressing mode) đơn giản: RISC dùng các chế độ định địa chỉ đơn giản và các lệnh có chiều dài cố định để tạo điều kiện cho việc xử lý song song (pipelining). Chế độ định địa chỉ bộ nhớ gián tiếp không được cung cấp.
- Các thanh ghi mục đích chung giống nhau: thiết kế RISC cho phép bất kỳ thanh ghi nào cũng có thể dùng trong bất kỳ ngữ cảnh nào, đơn giản hóa thiết kế trình biên dịch.
- Kiến trúc Harvard: các thiết kế RISC thường sử dụng mô hình bộ nhớ Harvard, các dòng lệnh và các luồng dữ liệu được tách ra.
- Một số bộ xử lý phổ biến dựa trên kiến trúc RISC: ARM, SuperH, MIPS, SPARC, DEC, Alpha, PA-RISC, PIC, và PowerPC của IBM. [1]

#### 1.3.2 Phần cứng mã nguồn mở

Để hỗ trợ các lĩnh vực đang tăng trưởng nhanh như IoT, Edge Computing, Machine Learning, không chỉ cần có phần mềm hỗ trợ mà còn cả phần cứng, tuy nhiên phần cứng và các chip phục vụ mục đích chung (general purpose) sẽ không thể nào đáp ứng hiệu năng cần thiết cho các ứng dụng đang phát triển này. Trong sự tăng trưởng liên tục trong thị trường công nghệ hiện nay, chip tự đặt hay chip tùy chỉnh sẽ đóng vai trò chủ yếu hơn là sử dụng nhiều con chip dùng cho mục đích chung. Sử dụng chip tự đặt sẽ cho chúng ta nhiều sự lựa chọn để tích hợp vào ứng dụng, với các thông số tùy biến, chúng ta có thể thiết kế ứng dụng một cách đa dạng hơn và không bị gò bó như khi sử dụng chip dùng cho mục đích chung. Tuy nhiên ngành kinh tế thiết kế chip hiện tại không thích hợp để xây dựng các chip riêng, nó hướng tới xây dựng các chip nói chung, như biểu đồ sau.

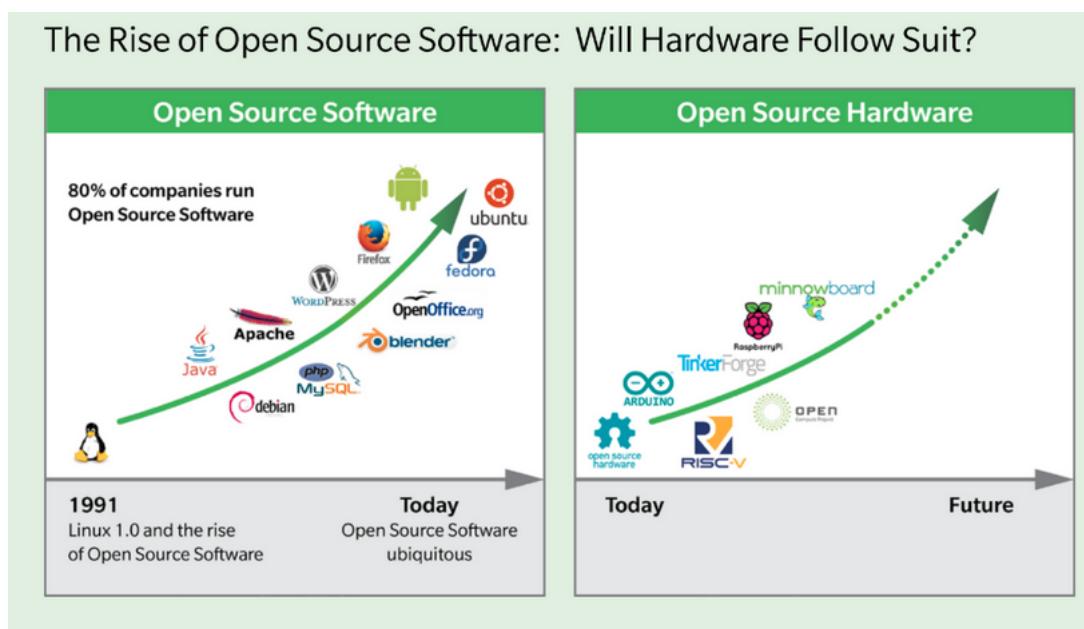


Hình 1.3: Biểu đồ chi phí cho việc phát triển chip

Biểu đồ trên cho thấy, để xây một vi xử lý 22nm hiện đại sẽ tốn cả trăm triệu đô la. Đó là lý do tại sao ngày càng ít người muốn tự đặt mua chip, chưa kể đến việc, ta sẽ phải tốn thời gian và khả năng phải đẩy lùi sản phẩm sau ngày dự kiến tung ra thị trường. Cho nên để tự đặt các con chip cho ứng dụng, chúng ta có nguy cơ thua lỗ rất nhiều hoặc hiệu năng của chip chỉ ở mức vừa hoặc thấp. [2]

Như Jerome Nadel, giám đốc công ty Rambus nói "chúng tôi thấy rằng các khoản đầu tư vào phần mềm tiếp tục phát triển, nhưng trường hợp này không tương tự với phần cứng. Lợi nhuận hao mòn ngày càng nghiêm trọng, và việc đầu tư vào chi phí thiết kế và chế tạo rất tốn kém, đến nỗi khái niệm *Xây một, hái mười* không còn có nghĩa nữa trong khi chi phí tăng và lợi nhuận giảm". Để chống lại xu hướng này, ông nói, lựa chọn duy nhất phụ thuộc vào các ASIC và FPGA tự đặt, tổng quát hơn, là tương lai của phần cứng mã nguồn mở, sự thành công đã được chứng minh cho các thiết bị nhớ (như hình bên dưới nhưng cần thời gian để tiếp cận được phía server). [3]

Sự phát triển mạnh mẽ và đa dạng của các phần mềm mã nguồn mở đã đem lại thành công cho các dự án phần mềm, khoảng 80% công ty trên thế giới sử dụng phần mềm mã nguồn mở. Một trong những thành công của phần mềm mã nguồn



Hình 1.4: So sánh sự thành công giữa phần mềm mã nguồn mở và phần cứng mã nguồn mở

mở đó là Instagram, chỉ với 13 nhân viên, phát triển thành công ty trị giá 1 tỷ đô la Mỹ và sau đó được mua lại bởi facebook, một trong những thành công của Instagram là vào các công nghệ mã nguồn mở ở đẳng sau hỗ trợ, làm nền tảng cho công ty phát triển. Tuy nhiên đối với bên phần cứng, liệu có công ty nào trị giá 1 tỷ đô la Mỹ chỉ với 13 nhân viên?

Để có được thành công như phần mềm, phần cứng phải noi theo và hướng tới mục tiêu mã nguồn mở. RISC-V là một trong số những nhãn hiệu đang hướng tới mục tiêu này. RISC-V là một tổ chức phi lợi nhuận, phi bản quyền, cũng như là một tập lệnh kiến trúc mã nguồn mở, tương thích với nhiều hệ thống tính toán quy mô khác nhau từ vi mô (vi điều khiển) tới vĩ mô (Siêu máy tính), đang được phổ biến rộng rãi trong cả công nghiệp lẫn giáo dục.

#### 1.3.3 RISC-V

RISC-V là 1 kiến trúc tập lệnh miễn phí cho phép 1 kỷ nguyên mới của bộ xử lí thông qua cộng tác tiêu chuẩn. Được thành lập vào năm 2015, nền tảng RISC-V bao gồm hơn 100 tổ chức thành viên xây dựng nên cùng với những cộng đồng về

phần cứng và phần mềm hợp tác đổi mới.

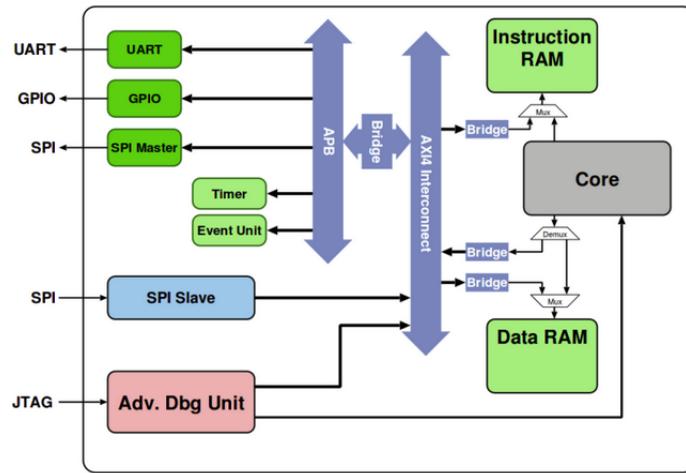
Sinh ra trong môi trường học thuật và nghiên cứu, RISC-V mang đến 1 cấp độ mới về những phần cứng và phần mềm tự do mở rộng kiến trúc, mở đường cho 50 năm tiếp theo của thiết kế và đổi mới máy tính.

Tên RISC-V được lựa chọn để thể hiện kiến trúc tập lệnh RISC quan trọng đời thứ 5 của đại học California Berkeley, từ V được sử dụng để biểu hiện "sự đa dạng"(variations) và "vector", tập lệnh này hỗ trợ cho nhiều nghiên cứu về kiến trúc, bao gồm các thiết bị gia tốc dữ liệu song song, đây cũng là mục tiêu cụ thể nhất của thiết kế kiến trúc tập lệnh. [4]

#### 1.3.4 PULP và PULPino

Nền tảng PULP (Parallel Ultra Low Power) là một nền tảng đa lõi đạt được hiệu quả năng lượng hàng đầu và có hiệu suất cao với nhiều tính năng có thể điều chỉnh.[5]

Mục đích của PULP là đáp ứng nhu cầu tính toán của các ứng dụng IoT yêu cầu xử lý linh hoạt các luồng dữ liệu được tạo bởi nhiều cảm biến, như gia tốc kế, camera độ phân giải thấp, microphone arrays, ...



Hình 1.5: Sơ đồ khái niệm của PULPino

## **1.4. Giới thiệu tinh chỉnh lõi phần cứng**

---

Trái ngược với MCU lõi đơn, kiến trúc song song với mức năng lượng thấp có thể lập trình được cho phép đáp ứng yêu cầu tính toán của các ứng dụng mà không vượt quá công suất cỡ một vài mW điển hình là các hệ thống chạy bằng pin thu nhỏ.

Hơn thế nữa PULP còn hỗ trợ OpenMP, OpenCL và OpenVX hỗ trợ việc phát triển, điều chỉnh và sửa lỗi ứng dụng.

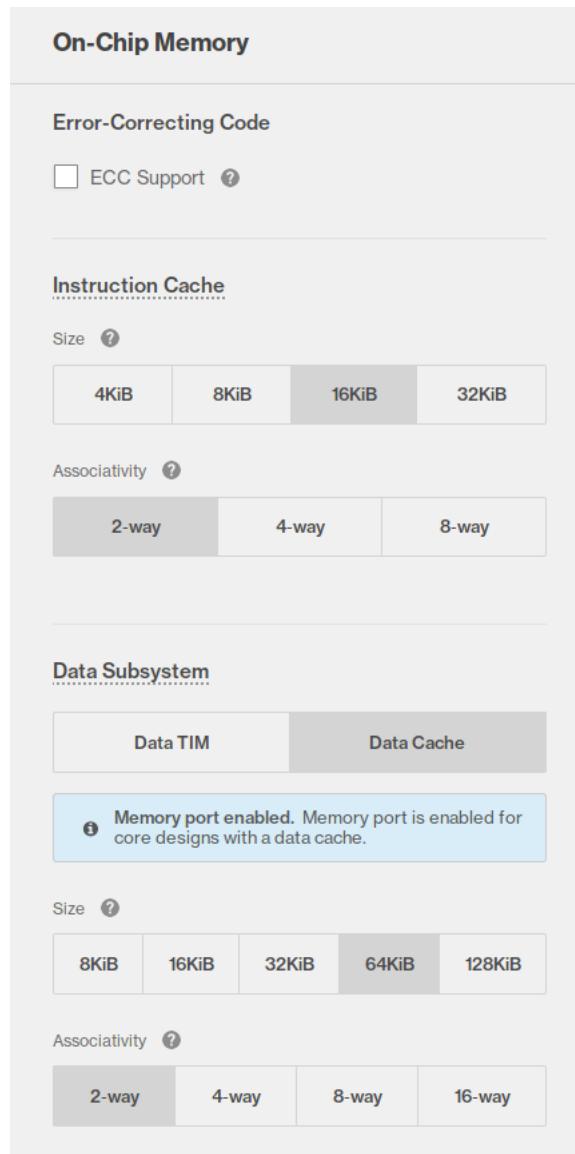
PULPino là một sản phẩm của PULP-platform, hướng tới việc xây dựng các kiến trúc vi xử lý tiết kiệm năng lượng để sử dụng trong các lĩnh vực như IoT, Wearables... PULPino là một hệ thống vi xử lý đơn lõi mã nguồn mở, dựa trên các lõi RISC-V 32-bit được phát triển tại đại học ETH Zurich ở Thụy Điển, ngoài RI5CY, ta có thể điều chỉnh sử dụng lõi Zero-RI5CY thay thế.

## **1.4 Giới thiệu tinh chỉnh lõi phần cứng**

Tinh chỉnh lõi phần cứng, hay còn được gọi phát triển chip tự đặt riêng, là việc chúng ta tùy chỉnh các cấu hình mặc định của lõi để phục vụ các mục đích nhất định đã đề ra, các thành phần có thể được hiệu chỉnh bao gồm:

- Bộ nhớ On-chip: tùy chỉnh kích thước của Instruction Cache và lựa chọn sử dụng và tùy chỉnh kích thước của Data Cache
- Chế độ và kiến trúc tập lệnh: tùy chỉnh có hay không có quyền người dùng (user-level privilege) và các extensions của kiến trúc tập lệnh thông thường (thêm hoặc không thêm bộ nhân, xử lý dấu chấm động).
- Debug: thiết lập số breakpoint và tùy chọn có debug DMA, performance counters cho mục đích đo hiệu suất.

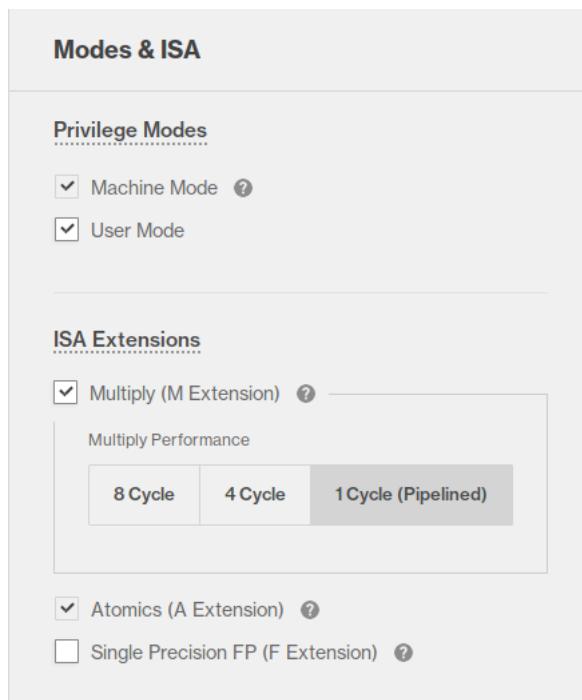
## 1.4. Giới thiệu tinh chỉnh lõi phần cứng



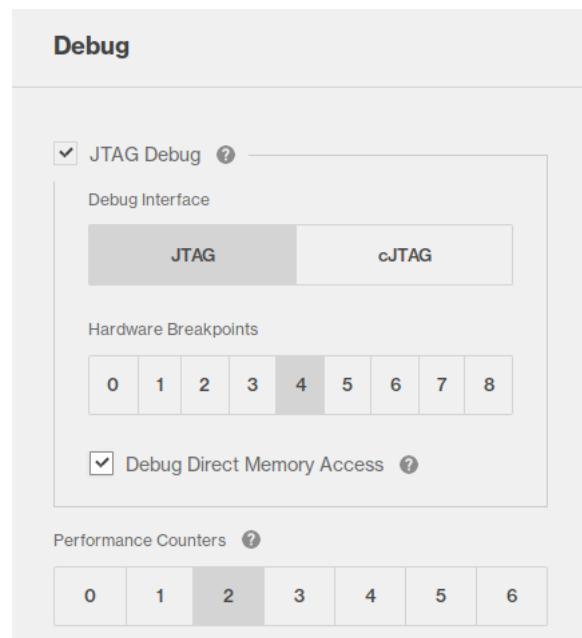
Hình 1.6: Các tùy chỉnh về bộ nhớ

- Ngắt: thiết lập ngắt cục bộ và ngắt toàn cục
- Tiên đoán rẽ nhánh: thiết lập bộ nhớ cho việc tiên đoán rẽ nhánh như buffer và stack của địa chỉ trả về.

#### 1.4. Giới thiệu tinh chỉnh lõi phần cứng

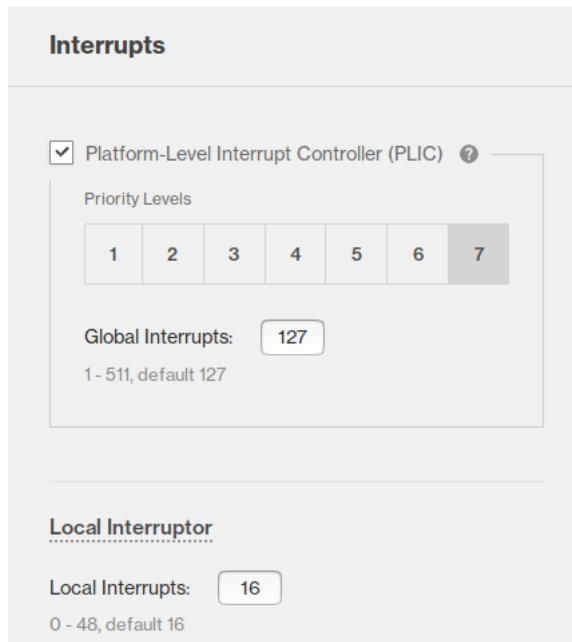


Hình 1.7: Các tùy chỉnh về chế độ và kiến trúc tập lệnh

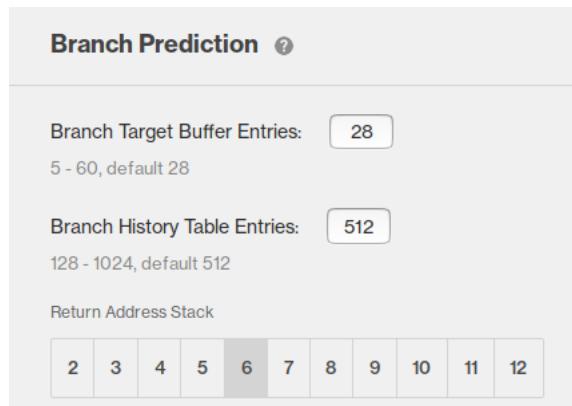


Hình 1.8: Các tùy chỉnh về debug

## 1.4. Giới thiệu tinh chỉnh lõi phần cứng



Hình 1.9: Các tùy chỉnh về ngắt

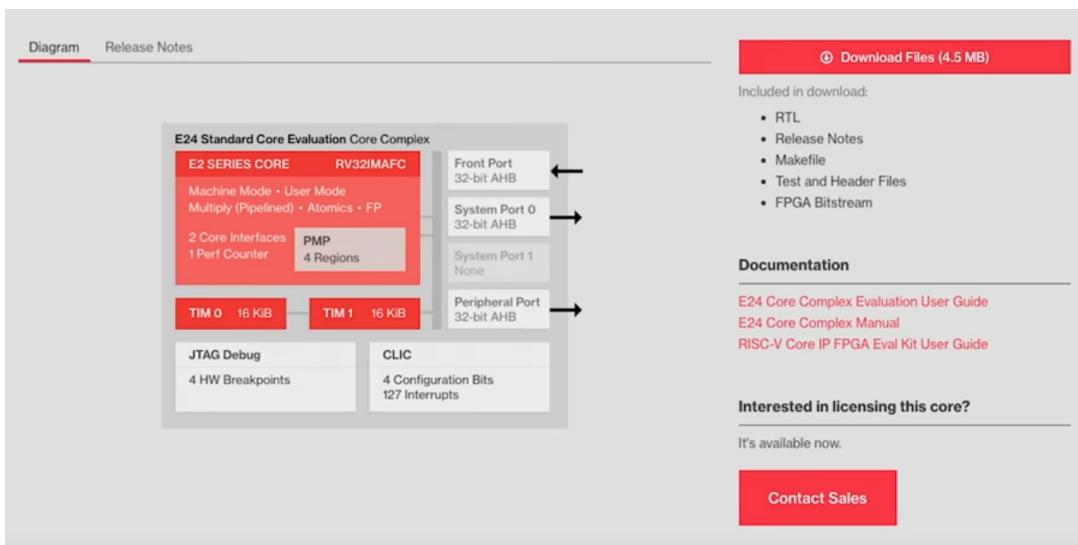


Hình 1.10: Các tùy chỉnh về tiên đoán rẽ nhánh

Sau khi thực hiện các tinh chỉnh cần thiết, ta sẽ tiến hành xử lý và sinh ra các file cần thiết để nạp xuống board bao gồm: RTL, makefile, và fpga images.

Việc tinh chỉnh giúp ta có thể lập trình các lõi theo yêu cầu xuống board, các quá trình này hoàn toàn ta tự thực hiện được, đối lập với làm việc với các nhà cung cấp (sản phẩm chip tự đặt được sở hữu trí tuệ) khi mà các giai đoạn có thể sẽ rườm rà như việc kí NDA và phải qua một số chi phí nhất định để có thể tiến

## 1.4. Giới thiệu tinh chỉnh lõi phần cứng



Hình 1.11: Minh họa của việc sau khi tinh chỉnh core trên SiFive, tiếp theo ta chỉ cần tải các file này về và nạp xuống dưới board để chạy.

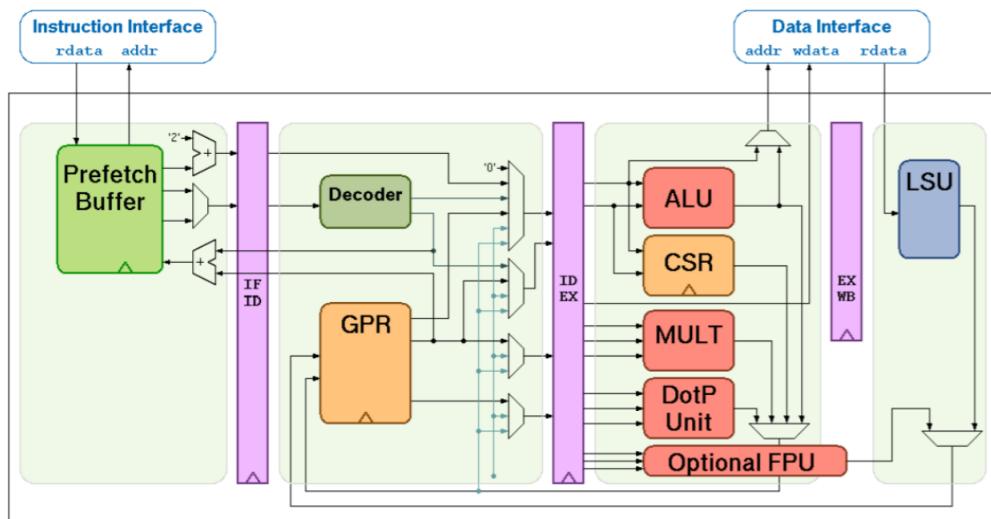
hành thử nghiệm các phần mềm trên chip tự đặt.

Hiện tại việc thiết kế lõi chỉ hoạt động giới hạn ở bộ xử lý, nhưng trong tương lai không xa, việc tích hợp với các bên sở hữu trí tuệ thứ ba như đồ họa (graphics controller) và cho phép khách hàng / người dùng tự xây dựng cả một hệ thống SoC một cách thuận tiện thông qua giao diện web.[6]

## 2 Tổng quan về lõi RI5CY

### 2.1 Giới thiệu và mô hình

RI5CY là một lõi trật tự đơn lệnh với IPC gần về 1, hỗ trợ hoàn toàn tập lệnh số nguyên (RV32), các lệnh nén (RV32C), tập lệnh nhân mở rộng (RV32M). Và có thể thiết lập để sử dụng tập lệnh dấu chấm động mở rộng (RV32F). RI5CY còn được hỗ trợ để thực hiện thêm các thao tác như: loop phần cứng, post-increment load, lệnh store và 1 số lệnh trong bộ ALU không có trong kiến trúc tập lệnh RISC-V tiêu chuẩn.



Hình 2.1: Kiến trúc của lõi RI5CY

## 2.2 Các thành phần trong core RI5CY

- Instruction Fetch:

Bộ nạp lệnh của core cung cấp 1 lệnh cho ID stage (Instruction Decode) mỗi chu kì nếu bộ nhớ hoặc bộ nhớ đệm của lệnh có thể phục vụ 1 lệnh mỗi chu kì. Địa chỉ lệnh phải ở dạng half-word-aligned (padding với các bit 0 để đủ 32 bits) để hỗ trợ compressed instruction.

Các signals được sử dụng cho bộ nạp lệnh được mô tả theo bảng 2.1

Bảng 2.1: Instruction Fetch Signals

Signal	Direction	Description
instr_req_o	output	Request ready, must stay high until instr_gnt_i is high for one cycle
instr_addr_o[31:0]	output	Address
instr_rdata_i[31:0]	input	Data read from memory
instr_rvalid_i	input	instr_rdata_i holds valid data when instr_rvalid_i is high. This signal will be high for exactly one cycle per request
instr_gnt_i	input	The other side accepted the request. instr_addr_o may change in the next cycle

- Load-Store-Unit (LSU)

Bộ LSU đảm nhiệm việc truy cập bộ nhớ dữ liệu. Hỗ trợ load và store với 32 bits, 16 bits và 8 bits.

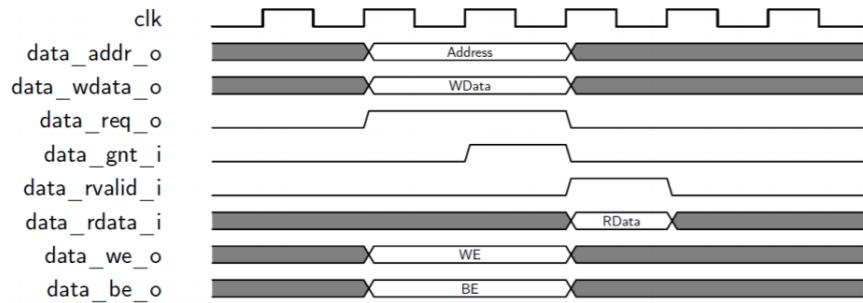
Các signal sử dụng cho bộ LSU được mô tả theo bảng 2.2.

### Giao tiếp giữa LSU và memory

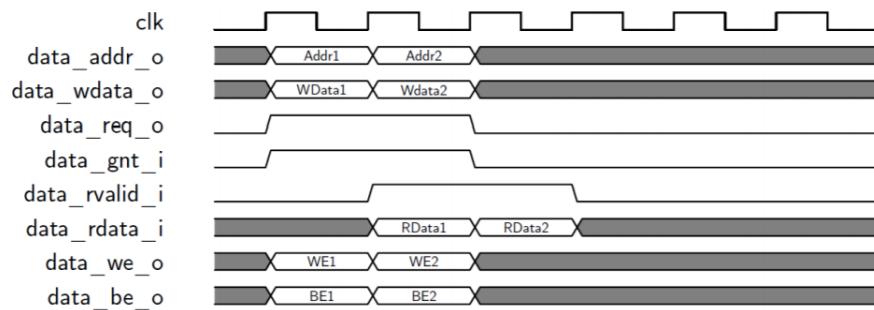
Khi data\_req\_o ở mức cao cho biết bộ LSU sẵn sàng cho việc đọc ghi và việc ghi chỉ xảy ra khi data\_gnt\_i ở mức cao cho biết memory sẵn sàng còn khi muốn đọc dữ liệu ra thanh ghi data\_rdata\_i ta phải set data\_rvalid\_i ở mức cao. (hình 2.2)

Khi data\_gnt\_i ở mức thấp có nghĩa là memory đang còn xử lý dữ liệu thì bộ LSU sẽ chờ trong 1 hoặc 1 vài chu kì cho tới khi memory xử lý xong và set data\_gnt\_i ở mức cao. (hình 2.3)

## 2.2. Các thành phần trong core RI5CY



Hình 2.2: Cách hoạt động của bộ LSU



Hình 2.3: Hoạt động của bộ LSU khi có memory chưa ghi xong

Bảng 2.2: LSU Signals

Signal	Direction	Description
data_req_o	output	Request ready, must stay high until data_gnt_i is high for one cycle
data_addr_o[31:0]	output	Address
data_we_o	output	Write Enable, high for writes, low for reads. Sent together with data_req_o
data_be_o[3:0]	output	Byte Enable. Is set for the bytes to write/read, sent together with data_req_o
data_wdata_o[31:0]	output	Data to be written to memory, sent together with data_req_o
data_rdata_i[31:0]	input	Data read from memory
data_rvalid_i	input	data_rdata_i holds valid data when data_rvalid_i is high. This signal will be high for exactly one cycle per request.
data_gnt_i	input	The other side accepted the request. data_addr_o may change in the next cycle

### Post-Incrementing Load and Store Instructions

Thực hiện load/store với bộ nhớ dữ liệu đồng thời tăng địa chỉ cơ sở với 1 giá trị offset, khi truy cập bộ nhớ thì sẽ sử dụng địa chỉ cơ sở.

Việc này giúp giảm bớt số lượng lệnh cần để thực thi code với cách truy cập dữ liệu thông thường (thường có trong loops). Cách này cho phép việc tăng địa chỉ được nhúng trong các lệnh truy cập bộ nhớ và loại bỏ các lệnh đặc biệt khi làm việc với con trỏ. Cùng với Hardware Loop Extension, các câu lệnh này cho phép giảm đáng kể chi phí cho vòng lặp.

- Multiply-Accumulate

RI5CY sử dụng 1 bộ nhân 32 bits x 32 bits đơn chu kì với kết quả là 32 bits, hỗ trợ tập lệnh của RISC-V M.

Phép nhân với kết quả trên 32 bits (upper-word) cần 4 chu kì để tính toán. Lệnh chia và các lệnh còn lại mất từ 2 đến 32 chu kì (dựa theo giá trị toán hạng).

Ngoài ra RI5CY còn hỗ trợ cho việc tính lũy thừa và phép nhân 16 bits với kết quả được dịch bit tùy ý.

- PULP ALU Extensions (xem thêm bảng 7.3 7.4 7.5 7.6 7.7 7.8)

RI5CY hỗ trợ bộ ALU nâng cao cho phép thực hiện nhiều lệnh trong tập lệnh cơ sở với 1 lệnh duy nhất cho phép tăng hiệu năng của core. Ví dụ các lệnh có thể đưa vào trong 1 lệnh duy nhất: các phép thao tác với bit, min, max, ...

Bộ ALU cũng hỗ trợ saturating, cắt, chuẩn hóa cho phép các phép tính với số thực hiệu quả hơn.

- Floating Point Unit (FPU)

Bộ FPU cho phép thực hiện tất cả các phép tính số thực của RISC-V được định nghĩa trong RV32F ISA. Bộ FPU này là phần mở rộng.

- Hardware Loop Extensions (xem thêm bảng 7.1 7.2)

Để tăng hiệu năng của loops, RI5CY hỗ trợ hardware loops cho phép thực thi 1 đoạn code nhiều lần mà không cần branches hay cập nhập bộ đếm. Hardware loops không cần stall để nhảy đến lệnh đầu tiên của vòng lặp.

## 2.2. Các thành phần trong core RI5CY

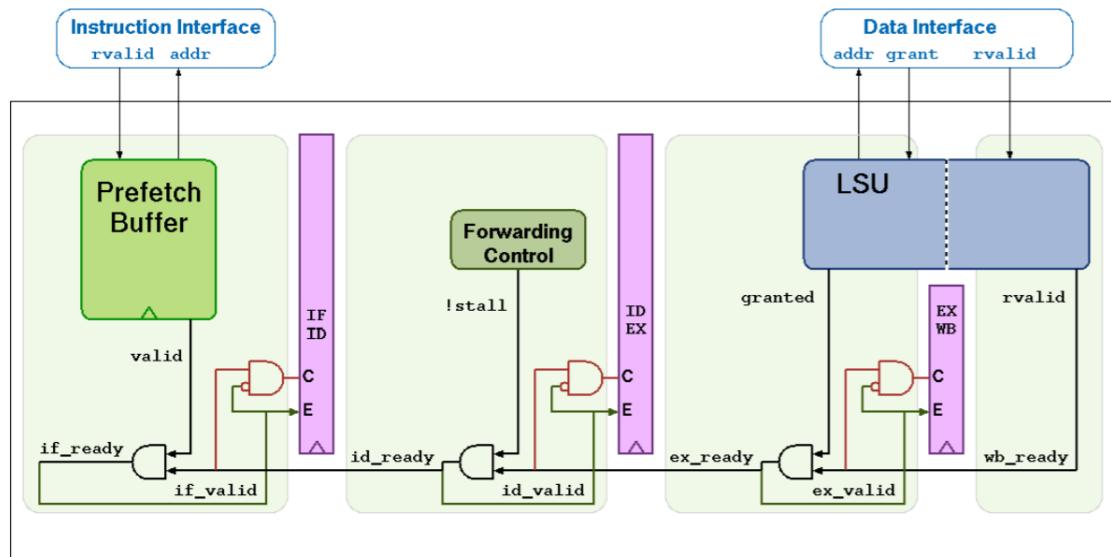
Một hardware loops được xác định bằng địa chỉ đầu tiên và kết thúc của vòng lặp với 1 bộ đếm giảm dần mỗi lần thực hiện vòng lặp. Ngoài ra RI5CY còn có 2 thanh ghi hỗ trợ hardware loops với các hardware loops lồng nhau, mỗi thanh ghi chứa có thể chứa 3 giá trị trong các flip-flop riêng biệt.

- Pipeline

RI5CY có pipeline độc lập hoàn toàn có nghĩa là khi dữ liệu được chuyển qua các giai đoạn của pipeline nó sẽ không chịu ảnh hưởng của bất kì stall không cần thiết nào.

Các signal chính cho các giai đoạn pipeline được mô tả ở hình 2.4. Các signal này được truyền từ phải sang trái. Mỗi giai đoạn pipeline có 2 tín hiệu điều khiển là enable và clear. Tín hiệu enable cho phép core và giai đoạn pipeline thực hiện tới lệnh kế tiếp. Tín hiệu clear loại bỏ lệnh khỏi giai đoạn pipeline. Giai đoạn pipeline sẽ clear nếu tín hiệu ready từ giai đoạn tiếp theo ở mức cao và tín hiệu valid ở mức thấp. Nếu tín hiệu valid thì tín hiệu enable sẽ ở mức cao.

Mỗi giai đoạn pipeline độc lập với giai đoạn trước nó có nghĩa là nó có thể kết thúc thực thi mà không cần quan tâm đến giai đoạn trước nó có stall hay không. Mặt khác 1 lệnh chỉ có thể qua giai đoạn pipeline tiếp theo chỉ khi giai đoạn đó sẵn sàng để nhận lệnh mới.



Hình 2.4: RI5CY Pipeline

## 2.2. Các thành phần trong core RI5CY

---

- Register File

RI5CY sử dụng thanh ghi 32 bits từ x1 đến x31. Thanh ghi x0 là thanh ghi đặc biệt mang giá trị 0 và chỉ có thể đọc. Có 2 kiểu:

- Latch-based (cho ASIC).
- Flip-flop (cho FPGA synthesis).

Ngoài ra khi lựa chọn core với bộ FPU thì còn có thêm 32 thanh ghi f0-f31.

- Control and Status Registers

RI5CY không cung cấp tất cả các thanh ghi điều khiển và trạng thái được chỉ định trong RISC-V và chỉ giới hạn ở những thanh ghi mà hệ thống PULP cần nhằm giảm bớt chi phí.

Bảng 2.3: Control and Status Register Map

CSR Address				Hex	Name	Acc.	Description
11:10	9:8	7:6	5:0				
00	11	00	000000	0x300	MSTATUS	R/W	Machine Status
00	11	00	000101	0x305	MTVEC	R	Machine Trap-Vector Base Address
00	11	01	000001	0x341	MEPC	R/W	Machine Exception Program Counter
00	11	01	000010	0x342	MCAUSE	R/W	Machine Trap Cause
01	11	00	0xxxxx	0x780-0x79F	PCCRs	R/W	Performance Counter Counter Registers
01	11	10	100000	0x7A0	PCER	R/W	Performance Counter Enable
01	11	10	100001	0x7A1	PCMR	R/W	Performance Counter Mode
01	11	10	110xxxx	0x7B0-0x7B7	HWLP	R/W	Hardware Loop Registers
11	00	00	010000	0xC10	PRIVLV	R	Privilege Level
00	00	00	010100	0x014	UHARTID	R	Hardware Thread ID
11	11	00	010100	0xF14	MHARTID	R	Hardware Thread ID

- Performance Counters

Bộ đếm hiệu năng trong RI5CY được đặt trong các thanh ghi điều khiển và trạng thái và có thể truy cập bằng lệnh csrr và csrwr.

- Exceptions and Interrupts: RI5CY hỗ trợ ngắt và xử lý ngoại lệ các lệnh không hợp lệ.

Ngắt chỉ có thể được kích hoạt và tắt trên cơ sở toàn cục, không thực hiện bật tắt riêng lẻ. Giả định rằng có một bộ điều khiển sự kiện / ngắt bên ngoài lõi thực hiện masking và đếm các dòng ngắt. Việc khởi động ngắt toàn cục được thực hiện bởi thanh ghi MSTATUS của thanh ghi điều khiển trạng thái.

Xử lý ngoại lệ cho các lệnh không hợp lệ luôn hoạt động và không thể tắt được.

## 2.2. Các thành phần trong core RI5CY

---

RI5CY không hỗ trợ xử lý ngắn / ngoại lệ lồng vào nhau nhau. Các ngoại lệ bên trong trình xử lý ngắn / ngoại lệ sẽ gây ra một ngoại lệ khác, vì thế các ngoại lệ xảy ra trong trình xử lý ngoại lệ, ví dụ trước khi lưu các thanh ghi MEPC và MESTATUS, sẽ khiến các thanh ghi đó bị ghi đè lên. Đối với ngắn trong khi xử lý ngắn / ngoại lệ, thiết lập này mặc định sẽ tắt, nhưng có thể được kích hoạt nếu người dùng muốn.

Khi thực hiện một lệnh mret, lỗi nhảy vào bộ đếm chương trình được lưu trong MEPC của thanh ghi điều khiển trạng thái và phục hồi giá trị MPIE của thanh ghi MSTATUS thành IE. Khi bước vào trình xử lý ngắn / ngoại lệ, lỗi đặt giá trị MEPC thành giá trị bộ đếm chương trình hiện tại và lưu giá trị hiện tại của MIE vào trong MPIE của thanh ghi MSTATUS.

- Debug Unit
- Instruction Set Extensions
  - Các tập lệnh Post-Incrementing Load & Store

Các tập lệnh post-incrementing (tăng địa chỉ sau khi thực hiện) load & store thực hiện các lệnh load và store tương ứng, đồng thời tăng địa chỉ được sử dụng để truy cập bộ nhớ. Vì cách thức là post-incrementing nên địa chỉ cơ sở được sử dụng để truy cập và địa chỉ được sau khi tăng được ghi vào tệp tin thanh ghi. Một số phiên bản các tập lệnh sử dụng tức thời các thanh ghi và một số khác sử dụng các thanh ghi như là offset. Địa chỉ cơ sở luôn xuất phát từ một thanh ghi.

- Vòng lặp phần cứng

RI5CY hỗ trợ 2 mức vòng lặp phần cứng lồng nhau. Vòng lặp phải được thiết lập trước khi vào thân vòng lặp. Vì thế, có hai phương thức, hoặc là sử dụng các lệnh dài đặt riêng các địa chỉ bắt đầu và kết thúc của vòng lặp và số lần lặp, hoặc sử dụng lệnh ngắn thực hiện tất cả điều này trong một lệnh. Lệnh ngắn có phạm vi giới hạn cho số lượng lệnh được chứa trong một vòng lặp và vòng lặp phải bắt đầu trong lệnh kế tiếp lệnh thiết lập.

Một vòng lặp phần cứng phải tuân theo các ràng buộc sau:

- \* Có Tối thiểu 2 lệnh trong thân vòng lặp.
- \* Bộ đếm vòng lặp phải lớn hơn 0, vì thân vòng lặp luôn được truy cập ít nhất một lần.

## 2.2. Các thành phần trong core RI5CY

---

### – ALU

Phần mở rộng ALU được chia thành nhiều nhóm nhỏ hòa hợp với nhau:

- \* Các lệnh thao túng bit hữu dụng để làm việc với các bit đơn lẻ, hoặc nhóm các bit trong một word.
- \* Các lệnh ALU chung hợp nhất các chuỗi được sử dụng phổ biến thành một lệnh và vì thế tăng hiệu năng của các kernel sử dụng các chuỗi đó.
- \* Các lệnh rẽ nhánh tức thời hữu dụng trong việc só sánh một thanh ghi với một giá trị tức thời trước khi chọn để theo hoặc không theo một nhánh.

### – Vectorial

Các câu lệnh vector thực hiện các toán tử với thao tác tương tự SIMD (đơn lệnh, đa thiết bị) trên nhiều thành phần sub-word cùng một lúc. Các lệnh đầy thực hiện được nhờ việc phân đoạn đường dẫn dữ liệu thành các phần nhỏ hơn khi thực hiện các thao tác dùng 8 hoặc 16 bits.

Các câu lệnh vector hỗ trợ ở 2 dạng:

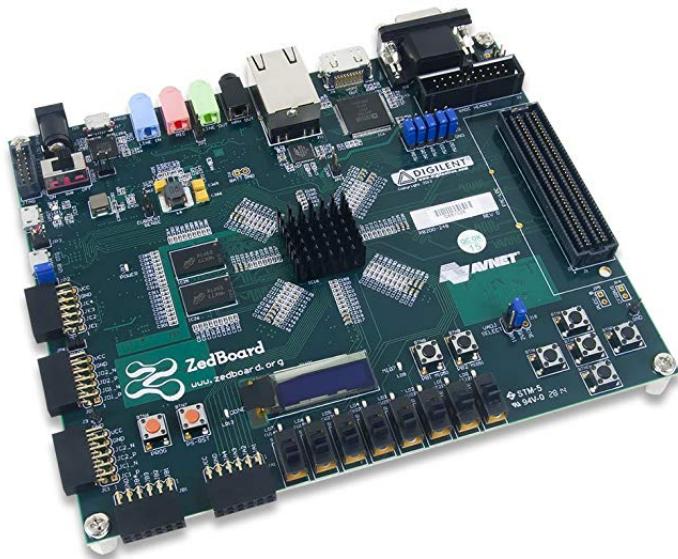
- \* 8-bit, để thực hiện 4 thao tác ở 4 bytes bên trong 32-bit word cùng một lúc.
- \* 16-bit, để thực hiện 2 thao tác ở 2 half-word bên trong 32-bit word cùng một lúc.

Ngoài ra, có ba chế độ ảnh hưởng đến toán hạng thứ hai:

- \* Chế độ bình thường, hoạt động theo kiểu vector-vector. Cả hai toán hạng, từ rs1 và rs2, được coi là vector của byte hoặc half-word.
- \* Chế độ sao chép vô hướng (.sc), hoạt động theo kiểu vecto-scalar. Toán hạng 1 được coi là một vecto, trong khi toán hạng 2 được coi là scalar và được nhân rộng hai hoặc bốn lần để tạo thành một vecto hoàn chỉnh. LSP được sử dụng cho mục đích này.
- \* Chế độ sao chép vô hướng tức thời (.sci), hoạt động theo kiểu vector-scalar. Toán hạng 1 được coi là vecto, trong khi toán hạng 2 được coi là scalar và xuất phát từ biến tức thời. Biến tức thời có thể là số âm mở rộng hoặc số không mở rộng, tùy thuộc vào hoạt động. Nếu không được chỉ định, biến thức thời sẽ là số âm mở rộng.[7]

### 3 Các thao tác với ZedBoard

ZedBoard là một bảng mạch chi phí thấp để chạy và phát triển Xilinx Zynq-7000 All Programmable SoC. Bảng mạch này bao gồm tất cả các thành phần thiết yếu để tạo các thiết kế Linux, Android, Windows hoặc các hệ điều hành hay hệ điều hành thời gian thực (RTOS) khác.Thêm vào đó, một số đầu nối mở rộng cho người dùng dễ tiếp cận với hệ thống xử lý và các I/O luận lý có thể lập trình được. ZedBoard được hỗ trợ bởi trang web cộng đồng zedboard.org, nơi người dùng có thể cộng tác với các kỹ sư khác cũng làm việc với các thiết kế Zynq.



Hình 3.1: Zedboard Zynq-7000 ARM/FPGA

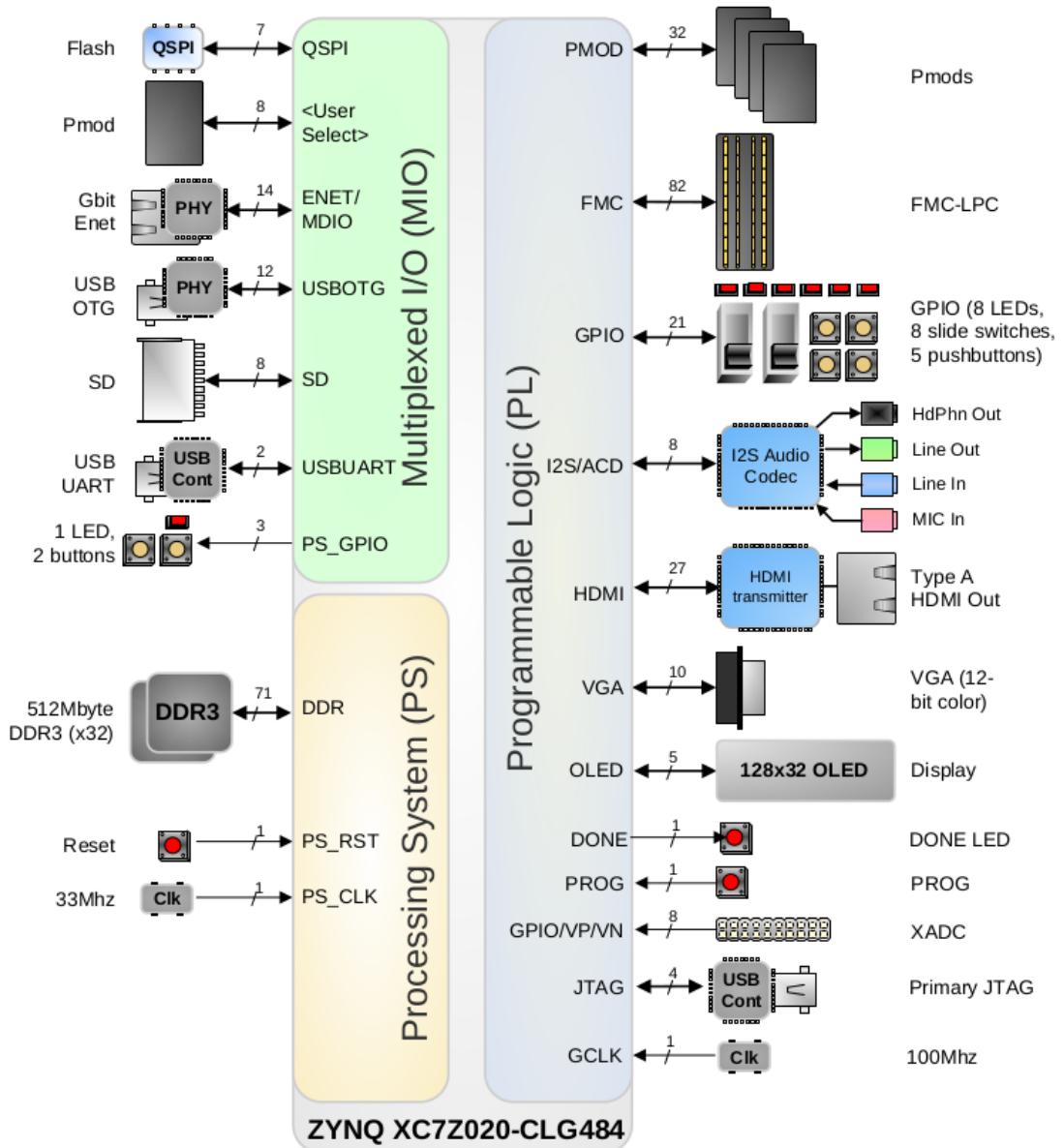
---

Các tính năng:

- Zynq-7000 All Programmable SoC mô hình XC7Z020-CLG484-1.
  - Thiết lập chính sử dụng QSPI.
  - Các thiết lập phụ hỗ trợ: Cascaded JTAG, thẻ SD.
- Bộ nhớ:
  - 512 MB DDR3.
  - 256 MB Quad-SPI Flash.
- Các giao diện:
  - Lập trình USB-JTAG Onboard sử dụng mạch Digilent SMT1 tương ứng.
  - Cổng Ethernet 10/100/1G.
  - USB OTG 2.0 và USB-UART.
  - Thẻ SD.
  - Một LPC FMC.
  - Một AMS Header.
  - Hai nút reset (1 PS, 1 PL).
  - Mở rộng PS & PL I/O.
  - Đa hiển thị (1080p HDMI, 8-bit VGA, 128x32 OLED).

### 3.1 Tổng quan mô hình và các thành phần của ZedBoard

Các sơ đồ đặc tả các thành phần của ZedBoard được hiển thị như bên dưới:[7]

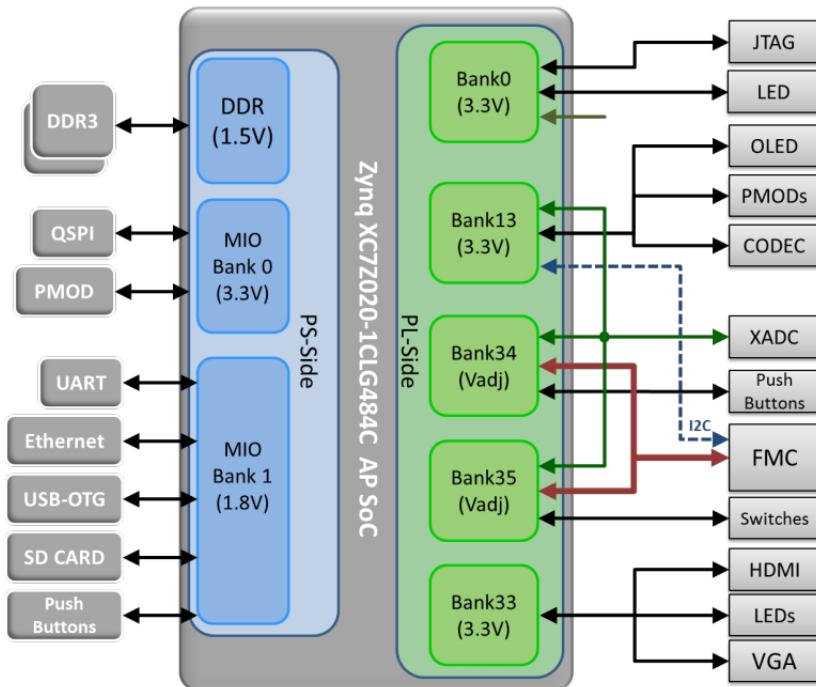


Hình 3.2: Sơ đồ khối của ZedBoard

### 3.2. Thiết lập phần cứng cho ZedBoard

Các thành phần của ZedBoard được chia thành 3 khối khác nhau, mỗi khối giữ một chức năng riêng lẻ:

- Khối xử lý (PS): Xử lý các tín hiệu nhận vào từ Reset, CLK, và từ memory.
- Khối luận lý có thể lập trình (PL): gồm các cổng luận lý như switches, GPIO, LED, display output.
- Khối đầu vào đầu ra ghép kênh (MIO): gồm các thành phần IO như Flash, LED, USB-OTG, SD, UART.



Hình 3.3: ZedBoard Bank Assignment

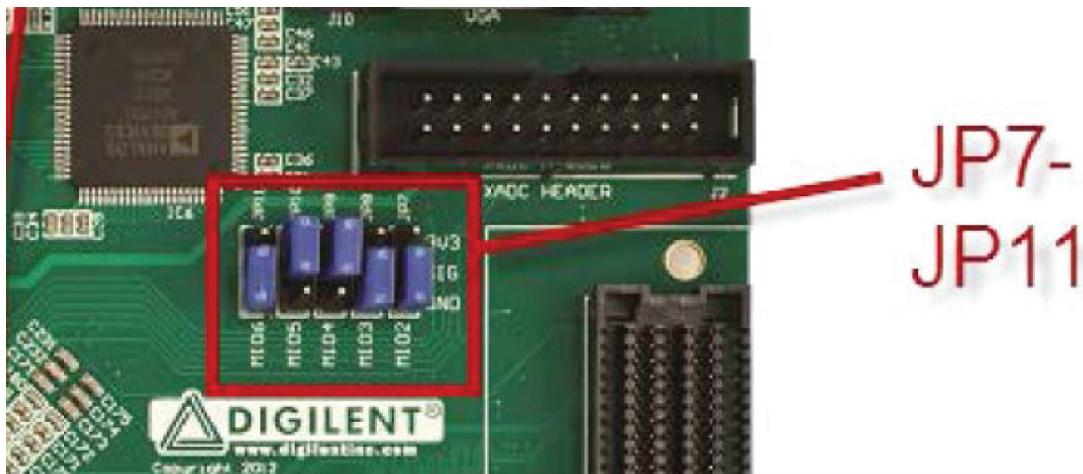
## 3.2 Thiết lập phần cứng cho ZedBoard

Đây là bước thiết lập đầu tiên để chúng ta có thể sử dụng và thao tác các tính năng ngoại vi tích hợp sẵn của ZedBoard.

- Bước 1: Kết nối nguồn 12V vào jack cắm (J20).

### **3.2. Thiết lập phần cứng cho ZedBoard**

- Bước 2: Kết nối cổng USB-UART của ZedBoard (J14) có nhãn UART vào PC bằng cáp MicroUSB.
- Bước 3: Cắm thẻ SD 4GB được cung cấp vào khe SD (J12) nằm ở mặt dưới của ZedBoard. Thẻ này cung cấp chương trình Linux để demo các tính năng của ZedBoard.
- Bước 4: Thiết lập boot cho ZedBoard để boot bằng thẻ SD bằng cách thiết lập các chân JP7-JP11 và MIO0 (JP6) như hình sau.

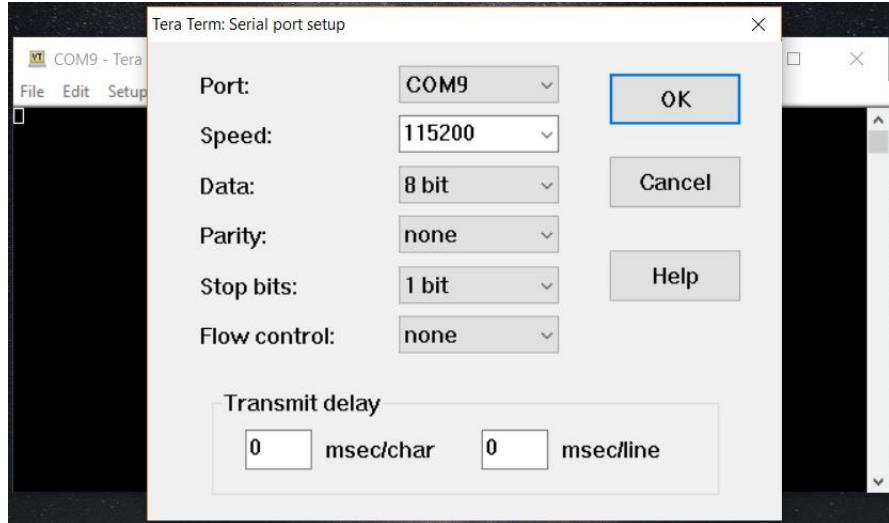


Hình 3.4: Thiết lập các jumper để boot bằng thẻ SD

- Bước 5: Bật switch nguồn (SW8 về trạng thái ON). ZedBoard sẽ được khởi động và đèn LED tín hiệu nguồn màu lục (LD13) sẽ sáng.
- Bước 6: Đợi khoảng 15s, đèn LED tín hiệu hoàn thành màu lam (LD12) sẽ sáng và hình ảnh mặc định sẽ được hiển thị trên OLED (DISP1).
- Bước 7: Tải và cài đặt driver UART để kết nối tới ZedBoard tại cypress: Microsoft Certified USB UART Driver

### 3.2. Thiết lập phần cứng cho ZedBoard

- Bước 8: Sử dụng phần mềm Terminal (như Tera Term) để kết nối tới ZedBoard, thiết lập Serial như dưới đây.



Hình 3.5: Thiết lập Tera Term để vào hệ điều hành trên SD Card

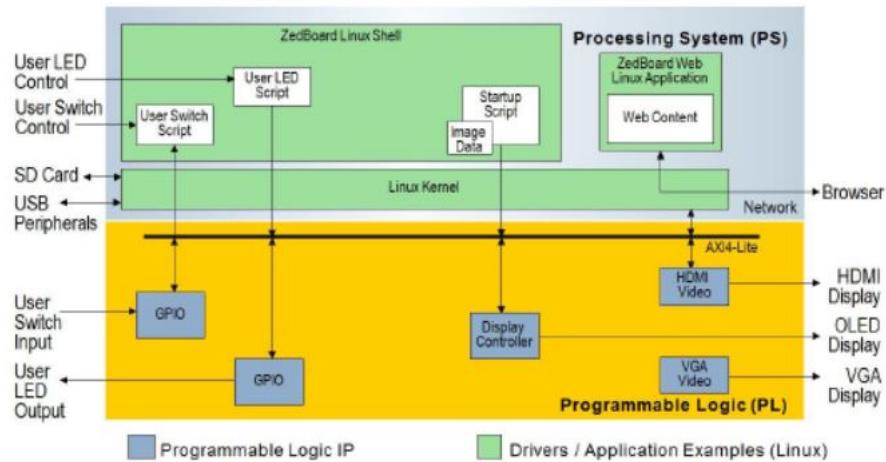
- Bước 9: Lúc này ta đã vào được terminal của linux chạy trên ZedBoard và có thể bắt đầu khám phá các tính năng của Zed Board.

```
[ 1.310000] UFS: Mounted root (ext4 filesystem) on device 1:0.
[ 1.310000] Freeing init memory: 152K
Starting rcS
++ Mounting filesystem
++ Setting up mdev
++ Configure static IP 192.168.1.10
[ 1.500000] GEM: lp->tx_bd ffdfb000 lp->tx_bd_dma 18fc0d000 lp->tx_skb d807028
[ 1.510000] GEM: lp->rx_bd ffdfc000 lp->rx_bd_dma 18fce000 lp->rx_skb d807038
[ 1.510000] GEM: MAC 0x00350a00, 0x00002201, 00:0a:35:00:01:22
[ 1.520000] GEM: phydev d8b6b400, phydev->phy_id 0x1410dd1, phydev->addr 0x0
[ 1.520000] eth0, phy_addr 0x0, phy_id 0x01410dd1
[ 1.530000] eth0, attach [Marvell 88E1510] phy driver
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting dropbear (<ssh>) daemon
++ Starting OLED Display
[ 1.570000] pmodoled-gpio-spi [zed_oled] SPI Probing
++ Exporting LEDs & SWs
rcS Complete
zynq>
```

Hình 3.6: Màn hình terminal sau khi boot

### 3.3 Các tính năng của ZedBoard

Thiết kế hệ thống cho các tính năng cơ bản của ZedBoard được thể hiện như sau.



Hình 3.7: Sơ đồ khái niệm các tính năng của ZedBoard

Từ các tính năng đơn giản này, các thiết kế đa dạng có thể được xây dựng tùy theo yêu cầu của người dùng.

Thư mục /usr/bin chứa các scripts được sử dụng xuyên suốt bài lab này để minh họa các tính năng của ZedBoard.

#### 3.3.1 Thao tác với GPIO Switches và LEDs

- Câu lệnh `read_sw` được sử dụng để đọc trạng thái của switchers người dùng (SW0-SW7), trước tiên ta chỉnh trạng thái mong muốn sau đó chạy `read_sw` script. Trạng thái của các switches sẽ được biểu diễn ở dạng số hệ 16 và hệ 10.

Script `read_sw` xử lý công việc đọc trạng thái GPIO từ `sys/class/gpio/gpio$sw/value` sysfs nodes. Thay đổi vị trí của switch thì các giá trị GPIO sẽ được đọc lại khi chạy `read_sw` script

- Câu lệnh `write_led` được sử dụng để thay đổi trạng thái của các LEDs (LD0-LD7), sử dụng `write_led + giá trị đặc tả trạng thái đèn LED`, ví dụ

write\_led 1 hoặc write\_led 0x1 sẽ làm sáng đèn led LD0.

Script write\_led xử lý công việc ghi giá trị trạng thái vào sys/class/gpio/gpio\$led/value sysfs nodes. Thay đổi trạng thái của LED bằng cách chạy write\_led script với giá trị khác.

- Ta có thể tự tạo script để thao tác với cả switches và LEDs; switch nào được set lên 1, thì sẽ sáng đèn tương ứng; khi bật hết tất cả các switch thì đèn sẽ chớp lần lượt từ phải sang trái.

---

```

1      #!/bin/sh
2
3      BLINK=1
4      while true
5          do
6              TEMP=`/usr/bin/read_sw | cut -f 2 -d ' '`
7              if [ $TEMP -eq 255 ]
8                  then
9                      /usr/bin/write_led $BLINK
10                     if [ $BLINK -eq 128 ]
11                         then
12                             BLINK=1
13                         else
14                             BLINK=$((BLINK*2))
15                         fi
16                         sleep 0.5
17                     else
18                         /usr/bin/write_led $TEMP
19                         fi
20             done

```

---

#### 3.3.2 Thao tác với OLED

- Chạy câu lệnh unload\_oled để tắt màn hình OLED. Khi lệnh này được gọi, module driver của OLED là pmodoled-gpio.ko sẽ bị gỡ khỏi kernel, khiến cho OLED bị tắt và không còn hiển thị logo Digilent.

### **3.3. Các tính năng của ZedBoard**

---

- Để khởi động lại màn hình OLED, chạy lệnh load\_oled. Lệnh này sẽ chèn module driver của OLED ngược lại vào kernel, và sẽ khởi động lại OLED trong quá trình. Tiếp theo, mã nguồn hình ảnh logo ở /root/logo.bin được chuyển đến node OLED ở /dev/zed\_oled và driver sẽ hiệu chỉnh OLED để hiển thị logo Digilent.

#### **3.3.3 Thao tác với VGA và HDMI**

- Thiết lập các bước để boot vào hệ điều hành như trên, sau đó cắm dây VGA có khả năng hiển thị ít nhất 640x480 pixel vào J10, có nhãn VGA, màn hình LCD sẽ sáng lên và hiển thị logo Digilent.
- Tương tự với HDMI, sử dụng dây HDMI có khả năng cho ra độ phân giải tối thiểu 1080p60 vào J9, có nhãn HDMI-OUT.

#### **3.3.4 Thao tác với Ethernet**

- Cắm dây Ethernet vào cổng Zedboard Gigabit Ethernet (J11) và đầu kia của dây vào máy tính.
- Vào Control Panel -> Network and Sharing Center để chỉnh lại địa chỉ IP cho máy.
- Vào IPv4 và thiết lập như hình bên dưới.
- Vào ZedBoard kiểm tra lại địa chỉ IP của board, mặc định là 192.168.1.10.
- Từ máy tính mở trình duyệt và truy cập vào địa chỉ 192.168.1.10, trang web được nhúng sẵn trong ZedBoard sẽ hiện lên.
- Ngoài ra, ta còn có thể thao tác với SSH và FTP sau khi đã kết nối qua Ethernet để truy cập từ xa và truyền file.

### 3.3.5 Thao tác với SD Card

- Thẻ SD được liệt kê dưới dạng các khối MMC /dev/mmcblk0 và phân vùng chính của thẻ được định danh là /dev/mmcblk0p1.
- Mount phân vùng chính của thẻ SD được liệt kê vào điểm mount /mnt bằng lệnh mount /dev/mmcblk0p1 /mnt.
- Phân vùng chính của thẻ SD đã được mount vào hệ thống file root ở /mnt, cho phép thao tác đọc và ghi file vào SD card, mặc định thẻ SD card sử dụng định dạng file FAT32.
- Phân vùng chính của thẻ SD nên được un-mount trước khi tắt board.

```

zynq> ls -l /dev/mmcblk0*
brw-rw---- 1 root 0 179, 0 Jan 1 00:00 /dev/mmcblk0
brw-rw---- 1 root 0 179, 1 Jan 1 00:00 /dev/mmcblk0p1
zynq> mount /dev/mmcblk0p1 /mnt
zynq> cd /mnt
zynq> ls -l
total 10368
-rwxr--r-- 1 root 0 4312256 Jul 13 2012 BOOT.BIN
-rwxr--r-- 1 root 0 2779 Jul 13 2012 README
drwxr-xr-x 2 root 0 32768 Mar 29 2019 System Volume Information
-rwxr--r-- 1 root 0 5817 Jul 13 2012 devicetree_rändisk.dtb
-rwxr--r-- 1 root 0 3694108 Jul 13 2012 rändisk8M.image.gz
-rwxr--r-- 1 root 0 2479640 Jul 13 2012 zImage
  
```

Hình 3.8: Thao tác với SD Card

**Lưu ý:** Nếu thiết bị không thể được unmount do lỗi "Device or resource busy", hãy bảo đảm là không có tệp tin hoặc thư mục nào đang mở, tốt nhất là cd /root và sau đó unmount.

# 4 Hiện thực ánh xạ lõi RI5CY lên ZedBoard

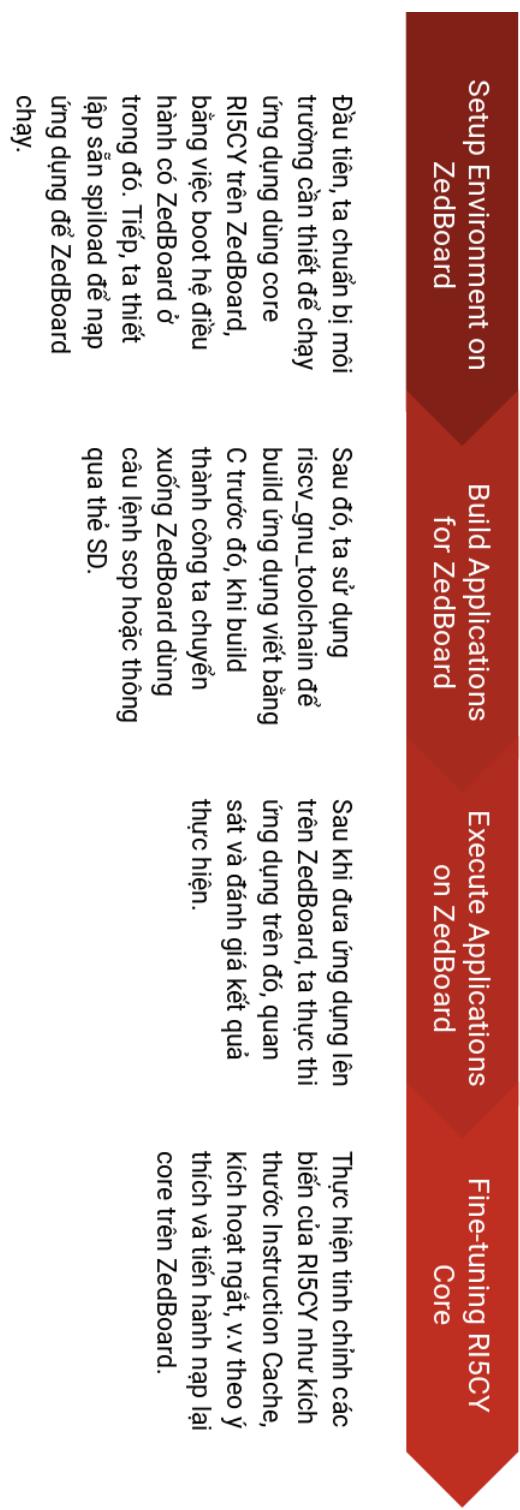
Quy trình hiện thực như được đặc tả thông qua flow như hình 4.1. Để thể hiện chi tiết các flow trên, nhóm đã làm 4 bài thực hành riêng lẻ, mô tả chi tiết các bước tiến hành và các yêu cầu, kiến thức cần nắm để hiểu về các bài thực hành, tổng quan về các bài thực hành như sau:

## 4.1 Lab 1: Build hệ điều hành và core RI5CY để boot lên ZedBoard

Ở bài lab này, chúng ta sẽ thiết lập các thông số cần thiết để xây dựng một hệ điều hành với lõi RI5CY để chạy được trên ZedBoard, khi có được hệ điều hành này thì ta có thể chạy các ứng dụng viết bằng C với lõi RI5CY, nội dung của bài lab này bao gồm:

- Giới thiệu lại PULPino và RI5CY.
- Các công cụ và môi trường cần thiết lập để tiến hành build.
- Chuẩn bị SD Card để thiết lập các image cho ZedBoard lên đó.
- Boot hệ điều hành lên trên ZedBoard bằng SD, và giới thiệu các cách boot khác.

#### 4.1. Lab 1: Build hệ điều hành và core RI5CY để boot lên ZedBoard



Hình 4.1: Quy trình ánh xạ lỗi RI5CY

## 4.2 Lab 2: Compile và chạy ứng dụng lên ZedBoard

Sau khi đã chạy được hệ điều hành lên ZedBoard, ta tiến hành compile thử ứng dụng demo (có thể sử dụng helloworld cho đơn giản) được PULPino cung cấp sẵn để kiểm tra lại các thiết lập có đúng hay không và đồng thời tạo file spi\_stim để chạy trên ZedBoard, nội dung cụ thể là:

- Giới thiệu toolchain sử dụng để build ứng dụng: riscy\_gnu\_toolchain; các khái niệm về GNU Toolchain và cross-compiler.
- Compile ứng dụng với toolchain nêu trên Chạy thử ứng dụng đó trên ZedBoard.

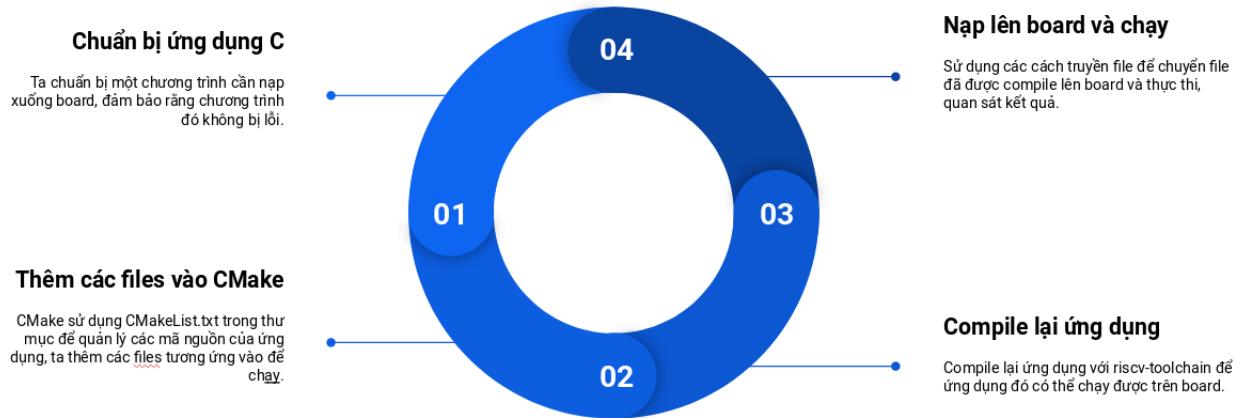
## 4.3 Lab 3: Tạo một ứng dụng riêng để nạp lên ZedBoard

Ở bài lab này, ta sẽ tìm hiểu cách để đưa ứng dụng C ta tự viết xuống board chạy thử, nội dung bao gồm:

- Quy trình nạp một ứng dụng tự viết lên board (đặc tả như hình 4.2).
- Các thao tác để nạp một ứng dụng tự viết lên board.

## 4.4 Lab 4: GPIO với Zedboard

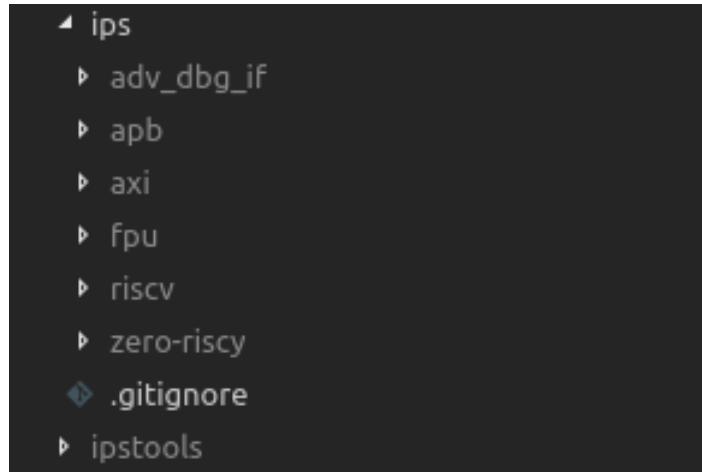
- Giới thiệu về GPIO của ZedBoard, các chân tín hiệu và chân Zynq pin.
- Các khai báo hàm để sử dụng GPIO trên ZedBoard.
- Code minh họa việc điều khiển LED bằng Switch.



Hình 4.2: Quy trình nạp một ứng dụng tự viết lên board

### Tinh chỉnh lõi RI5CY

Việc tinh chỉnh được tiến hành trước khi ta sử dụng lệnh *make* trong thư mục *pulpino/fpga* và sau khi chạy đoạn script *update-ips.py* trong thư mục *pulpino*. Sau khi đoạn script *update-ips.py* được thực hiện, thư mục *ips* được cập nhật các lõi mà *pulpino* sử dụng: *ri5cy* và *zero-ri5cy*.



Hình 4.3: Thư mục *ips* đã được cập nhật và chứa các lõi được *pulpino* sử dụng, đồng thời, *ipstools* cũng được clone về để hỗ trợ *make*

Sau đó ta chạy vào thư mục *riscv*, đây là thư mục lõi gốc của RI5CY, chứa các file *rtl*, ta thực hiện hiệu chỉnh các file trên để có thể xây được một core theo ý

muốn. Ngoài ra ta còn có thể tinh chỉnh được một số thiết lập của PULPino trong thư mục rtl của pulpino. Một số thiết lập minh họa như sau:

- Instruction Memory: điều chỉnh giá trị tham số `RAM_SIZE` trong file `instr_ram_wrap.sv` và `sp_ram_wrap.sv` (mặc định là 32768 bytes) của repo pulpino.

---

```

1 module instr_ram_wrap
2 #(
3     parameter RAM_SIZE = 32768,           // in bytes
4     parameter ADDR_WIDTH = $clog2(RAM_SIZE) + 1, // one bit more
5     parameter DATA_WIDTH = 32
6 )()
```

---



---

```

1 module sp_ram_wrap
2 #(
3     parameter RAM_SIZE = 32768,           // in bytes
4     parameter ADDR_WIDTH = $clog2(RAM_SIZE),
5     parameter DATA_WIDTH = 32
6 )()
```

---

- Instruction Cache: điều chỉnh giá trị tham số `DEPTH` trong file `riscv_fetch_fifo.sv` khi update ips xong.

---

```

1 module riscv_fetch_fifo
2 (
3     input logic      clk,
4     input logic      rst_n,
5     .....
6 );
7 localparam DEPTH = 4; // must be 3 or greater
```

---

#### 4.4. Lab 4: GPIO với Zedboard

---

- Kích hoạt ngắt lồng: thiết đặt tham số `PULP_SECURE` là 1 trong file `riscv_int_controller.sv` khi update ips xong.

---

```
1 module riscv_int_controller
2 #(
3     parameter PULP_SECURE = 0
4 )
```

---

Sau khi hiệu chỉnh các biến mình mong muốn, ta tiến hành make lại từ đầu như bài Lab 1 và nạp lên board để tiến hành test.

# 5 Kết quả thực hiện

## 5.1 Kết quả khi build lõi RI5CY

Sau khi xây dựng lõi RI5CY trên nền tảng PULPino ta thu được các thông tin của lõi như sau.[11]

- BlackBox của RI5CY trên model xc7z020clg484-1 và số cell sử dụng .

Bảng 5.1: Report BlackBoxes

	BlackBox name	Instances
1	xilinx_fp_fma	1
2	xilinx_mem_8192x32	2

Ta thấy BlackBox mà PULPino sử dụng là xilinx\_mem\_8192x32, là IP mã nguồn mở mà Xilinx cung cấp để build hệ điều hành chứa lõi RI5CY để nạp lên Zedboard.

## 5.1. Kết quả khi build lõi RI5CY

---

Bảng 5.2: Report Cell Usage

	Cell	Count
1	xilinx_fp_fma	1
2	xilinx_mem_8192x32	1
3	xilinx_mem_8192x32_1	1
4	BUFG	3
5	CARRY4	1079
6	DSP48E1	7
7	DSP48E1_1	1
8	LUT1	1067
9	LUT2	2617
10	LUT3	2009
11	LUT4	2114
12	LUT5	3956
13	LUT6	11928
14	MUXF7	995
15	MUXF8	390
16	FDCE	11204
17	FDPE	202
18	FDRE	2
19	IBUF	54
20	OBUF	89

Bảng 5.2 cho ta thấy số lượng logic cell mà PULPino với lõi RI5CY sử dụng.

## 5.1. Kết quả khi build lõi RI5CY

Bảng 5.3: Report Instance Areas

	Instance	Module	Cells
1	top		37817
2	pulpino_i	pulpino_top	37363
69	cs_registers_i	riscv_cs_registers	1560
70	debug_unit_i	riscv_debug_unit	1186
71	ex_stage_i	riscv_ex_stage	3401
72	alu_i	riscv_alu	465
75	fpu_i	fpu_private	2653
76	fpu_core	fpu_core	1535
88	id_stage_i	riscv_id_stage	8745
89	controller_i	riscv_controller	319
90	hwloop_regs_i	riscv_hwloop_regs	296
92	registers_i	riscv_register_file	6346
93	if_stage_i	riscv_if_stage	2348
96	load_store_unit_i	riscv_load_store_unit	204

Bảng 5.3 đưa ra cụ thể số logic cell dùng cho các module cụ thể của lõi.

- Các tài nguyên sử dụng cho PULPino với lõi RI5CY.

Bảng 5.4: Utilization Report

Instance	Module	Total LUTs	Logic LUTs	FFs	RAMB36	DSP48 Blocks
pulpino	(top)	21771	21771	11567	16	10
(pulpino)	(top)	262	262	0	0	0
pulpino_i	pulpino_top	21509	21509	11567	0	10
ex_stage_i	riscv_ex_stage	3102	3102	516	0	10
alu_i	riscv_alu	202	202	107	0	0
fpu_core	fpu_core	1204	1204	75	0	2
data_mem	sp_ram_wrap	19	19	1	8	0

Bảng này cho ta thấy các tài nguyên thực thể mà PULPino sử dụng như số LUT số flip-flop số block ram và số lượng DSP mà PULPino và lõi RI5CY sử dụng.

## 5.1. Kết quả khi build lõi RI5CY

---

- So sánh kết quả build với tài nguyên của Zedboard (Zynq-7000 XC7Z020).[14]

Bảng 5.5: So sánh kết quả build với tài nguyên của Zynq-7000 XC7Z020

	PULPino	Zynq-7000 XC7Z020	Percent
Logic Cells	37,817	85,000	44.5%
Look-Up Tables (LUTs)	21,771	53,200	40.9%
Flip-Flops	11,567	106,400	10.9%
Total Block RAM	576 Kb	4.9 Mb	11.8%
DSP Slices	10	220	4.5%

**Kết luận:** So sánh kết quả build với tài nguyên của Zedboard sử dụng ta thấy Zedboard (Zynq-7000 XC7Z020) đủ để hiện thực ánh xạ của core RI5CY sử dụng PULPino platform.

## 5.2 Kết quả khi chạy thử tập lệnh cơ sở

Để kiểm tra core có hoạt động đúng không nhóm kiểm tra với 1 số lệnh thuộc tập lệnh cơ sở RV32I như bảng sau:

Bảng 5.6: Một số lệnh cơ sở RV32I

Format	Name	Pseudocode
LUI rd, imm	Load Upper Immediate	$rd \leftarrow imm$
ADD rd, rs1, rs2	Add	$rd \leftarrow sx(rs1) + sx(rs2)$
SUB rd, rs1, rs2	Subtract	$rd \leftarrow sx(rs1) - sx(rs2)$
AND rd, rs1, rs2	And	$rd \leftarrow ux(rs1) \wedge x(rs2)$
OR rd, rs1, rs2	Or	$rd \leftarrow ux(rs1) \vee ux(rs2)$
XOR rd, rs1, rs2	Xor	$rd \leftarrow ux(rs1) \oplus ux(rs2)$
SLL rd, rs1, rs2	Shift Left Logical	$rd \leftarrow ux(rs1) \ll rs2$
SRL rd, rs1, rs2	Shift Right Logical	$rd \leftarrow ux(rs1) \gg rs2$
SRA rd, rs1, rs2	Shift Right Arithmetic	$rd \leftarrow sx(rs1) \gg rs2$
BEQ rs1, rs2, offset	Branch Equal	if $rs1 = rs2$ then $pc \leftarrow pc + offset$
BNE rs1, rs2, offset	Branch Not Equal	if $rs1 \neq rs2$ then $pc \leftarrow pc + offset$
BLT rs1, rs2, offset	Branch Less Than	if $rs1 < rs2$ then $pc \leftarrow pc + offset$
BGE rs1, rs2, offset	Branch Greater than Equal	if $rs1 \geq rs2$ then $pc \leftarrow pc + offset$
JAL rd, offset	Jump and Link	$rd \leftarrow pc + length(inst)$ $pc \leftarrow pc + offset$
J offset	Jump	$pc \leftarrow pc + offset$
LB rd, offset(rs1)	Load Byte	$rd \leftarrow s8[rs1 + offset]$
LH rd, offset(rs1)	Load Half	$rd \leftarrow s16[rs1 + offset]$
LW rd, offset(rs1)	Load Word	$rd \leftarrow s32[rs1 + offset]$
SB rs2, offset(rs1)	Store Byte	$u8[rs1 + offset] \leftarrow rs2$
SH rs2, offset(rs1)	Store Half	$u16[rs1 + offset] \leftarrow rs2$
SW rs2, offset(rs1)	Store Word	$u32[rs1 + offset] \leftarrow rs2$

Để thực hiện việc kiểm tra hoạt động của các lệnh trên nhóm sử dụng thư viện bench.c trong /pulpino/sw/libs/ và ARM cross compiler để tiến hành build ứng dụng nạp xuống board.

## 5.2. Kết quả khi chạy thử tập lệnh cơ sở

---

Ta có các testcase như sau:

```
1  testcase_t testcases[] = {  
2      {.name = "lui", .test = check_lui},  
3      {.name = "add", .test = check_add},  
4      {.name = "sub", .test = check_sub},  
5      {.name = "and", .test = check_and},  
6      {.name = "or", .test = check_or},  
7      {.name = "xor", .test = check_xor},  
8      {.name = "sll", .test = check_sll},  
9      {.name = "srl", .test = check_srl},  
10     {.name = "sra", .test = check_sra},  
11     {.name = "beq", .test = check_beq},  
12     {.name = "bne", .test = check_bne},  
13     {.name = "blt", .test = check_blt},  
14     {.name = "bge", .test = check_bge},  
15     {.name = "jal", .test = check_jal},  
16     {.name = "jump", .test = check_jump},  
17     {.name = "lb", .test = check_lb},  
18     {.name = "lh", .test = check_lh},  
19     {.name = "lw", .test = check_lw},  
20     {.name = "sb", .test = check_sb},  
21     {.name = "sh", .test = check_sh},  
22     {.name = "sw", .test = check_sw},  
23     {0 , 0}  
24 };
```

---

Sử dụng hàm run\_suite của thư viện bench cho phép ta thực thi lần lượt các hàm trong testcases.

---

```
1  unsigned int run_suite(testcase_t *tests);
```

---

Sau đó ta cần hiện thực các hàm kiểm tra và so sánh kết quả output với kết quả ta mong muốn. Ví dụ sau đây là với lệnh load upper immediate.

## 5.2. Kết quả khi chạy thử tập lệnh cơ sở

```
1 void check_lui(testresult_t *result, void (*start)(), void
2 (*stop)()){
3     unsigned int a[5] = {0x00000000, 0x12003400, 0xABAB1234,
4         0x8678CDAB, 0x2579ABCF};
5     unsigned int exp = 0x1FFA000;
6
7     for(int i = 0; i < 5; i++){
8         asm volatile ("lui %[a], 0x1FFA;" :
9             [a] "+r" (a[i]));
10        check_uint32(result, "lui", a[i], exp);
11    }
12 }
```

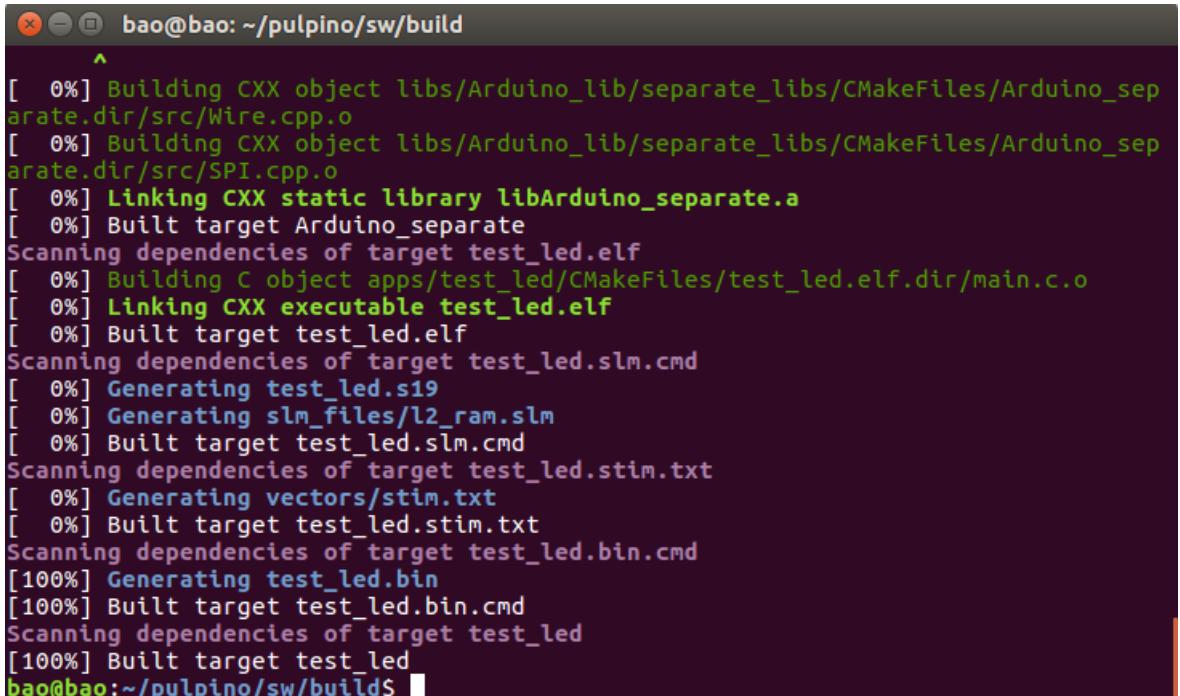
Source code hiện thực của chương trình kiểm thử ở [đây](#). Compile và nạp xuống board bằng spiload ta có kết quả sau (cách build tham khảo thêm ở lab 3).

```
PULPino: == test: lui -> success, nr. of errors: 0
PULPino: == test: add -> success, nr. of errors: 0
PULPino: == test: sub -> success, nr. of errors: 0
PULPino: == test: and -> success, nr. of errors: 0
PULPino: == test: or -> success, nr. of errors: 0
PULPino: == test: xor -> success, nr. of errors: 0
PULPino: == test: sll -> success, nr. of errors: 0
PULPino: == test: srl -> success, nr. of errors: 0
PULPino: == test: sra -> success, nr. of errors: 0
PULPino: == test: beq -> success, nr. of errors: 0
PULPino: == test: bne -> success, nr. of errors: 0
PULPino: == test: bge -> success, nr. of errors: 0
PULPino: == test: bge -> success, nr. of errors: 0
PULPino: == test: blt -> success, nr. of errors: 0
PULPino: == test: jump -> success, nr. of errors: 0
PULPino: == test: jal -> success, nr. of errors: 0
PULPino: == test: lb -> success, nr. of errors: 0
PULPino: == test: lh -> success, nr. of errors: 0
PULPino: == test: lw -> success, nr. of errors: 0
PULPino: == test: sb -> success, nr. of errors: 0
PULPino: == test: sh -> success, nr. of errors: 0
PULPino: == test: sw -> success, nr. of errors: 0
PULPino: === SUMMARY: SUCCESS
```

Hình 5.1: Kết quả chạy thử một số lệnh cơ sở

### 5.3 Kết quả khi build ứng dụng

Dưới đây là kết quả của nhóm sau khi build ứng dụng điều khiển led bằng switch viết bằng C. Ứng dụng này nhóm tự viết dựa trên thư viện gpio.h và cách gắn chân gpio của Zedboard và nhóm đặt tên ứng dụng là test\_led (tham khảo thêm ở lab 4).



```
bao@bao:~/pulpino/sw/build
[ 0%] Building CXX object libs/Arduino_lib/separate_libs/CMakeFiles/Arduino_separate.dir/src/Wire.cpp.o
[ 0%] Building CXX object libs/Arduino_lib/separate_libs/CMakeFiles/Arduino_separate.dir/src/SPI.cpp.o
[ 0%] Linking CXX static library libArduino_separate.a
[ 0%] Built target Arduino_separate
Scanning dependencies of target test_led.elf
[ 0%] Building C object apps/test_led/CMakeFiles/test_led.elf.dir/main.c.o
[ 0%] Linking CXX executable test_led.elf
[ 0%] Built target test_led.elf
Scanning dependencies of target test_led.slm.cmd
[ 0%] Generating test_led.s19
[ 0%] Generating slm_files/l2_ram.slm
[ 0%] Built target test_led.slm.cmd
Scanning dependencies of target test_led.stim.txt
[ 0%] Generating vectors/stim.txt
[ 0%] Built target test_led.stim.txt
Scanning dependencies of target test_led.bin.cmd
[100%] Generating test_led.bin
[100%] Built target test_led.bin.cmd
Scanning dependencies of target test_led
[100%] Built target test_led
bao@bao:~/pulpino/sw/build$
```

Hình 5.2: Build ứng dụng với RI5CY toolchain

### 5.3. Kết quả khi build ứng dụng

Sau khi sinh ra file spi\_stim.txt ta tiến hành chuyển file đó qua SD card và nạp với ./spiloader, kết quả thu được như hình dưới.

```
# ./spiloader -t5 spi_stim.txt
Device has been reset
Sending block addr 00000000 with 256 entries
Sending block addr 000003FC with 256 entries
Sending block addr 000007F8 with 256 entries
Sending block addr 00000BF4 with 256 entries
Sending block addr 00000FF0 with 256 entries
Sending block addr 000013EC with 213 entries
Sending block addr 00100000 with 256 entries
Sending block addr 001003FC with 59 entries
Starting device
Waiting for EOC...
Timeout reached!
Stopped after 5.790

#
```

Hình 5.3: Nạp file spi\_stim.txt với ./spiloader

Bây giờ ta tiến hành bật tắt switch và kiểm tra trạng thái của đèn led.



Hình 5.4: Demo gpio điều khiển led bằng switch

# 6 Kết luận

## 6.1 Đánh giá kết quả

### 6.1.1 Kết quả đã thực hiện được

Nhóm đã thực hiện được một số mục tiêu đề ra ở giai đoạn đề cương luận văn như sau:

- Tìm hiểu cơ bản xung quanh core RI5CY về kiến trúc, thành phần và tập lệnh.
- Tìm hiểu mã nguồn mở của core RI5CY và ánh xạ hiện thực của nó lên Zedboard.
- Tìm hiểu một số tùy chỉnh cho core.

### 6.1.2 Hạn chế

Một số hạn chế của nhóm trong quá trình thực hiện luận văn:

- Mã nguồn phụ thuộc nhiều vào repo của git nên khó cho việc tinh chỉnh.
- Một số tinh chỉnh của nhóm còn trên lý thuyết do gần đây repo của git xảy ra vấn đề khiến nhóm không thể update ips nên chưa thể make lại và kiểm chứng.

## **6.2 Hướng phát triển**

- Chạy thử các ứng dụng nâng cao và xử lý nhiều để đánh giá hiệu năng như các giải thuật sort.
- So sánh và đánh giá speedup khi sử dụng core RI5CY gốc và core RI5CY sau khi đã tinh chỉnh.
- Tìm hiểu thêm về cơ chế đa lõi của các platform khác.

## 7 Phụ lục A

Dưới đây là các bảng giới thiệu về 1 số tập lệnh của RI5CY.

Bảng 7.1: Hardware Loops Operations

Mnemonic	Description
<b>Long Hardware Loop Setup instructions</b>	
lp.starti L, uimmL	lpstart[L] = PC + (uimmL « 1)
lp.endi L, uimmL	lpPEND[L] = PC + (uimmL « 1)
lp.count L, rs1	lpCOUNT[L] = rs1
lp.counti L, uimmL	lpCOUNT[L] = uimmL
<b>Short Hardware Loop Setup Instructions</b>	
lp.setup L, rs1, uimmL	lpstart[L] = pc + 4 lpPEND[L] = pc + (uimmL « 1) lpCOUNT[L] = rs1
lp.setupi L, uimmS, uimmL	lpstart[L] = pc + 4 lpPEND[L] = pc + (uimmS « 1) lpCOUNT[L] = uimmL

Bảng 7.2: Hardware Loops Encoding

31	20	19	15	14	12	11	10	7	6	0	
uimmL[11:0]	rs1	funct3	0000	L		opcode					
uimmL[11:0]	00000	000	0000	L	111	10111	lp.starti L, uimmL				
uimmL[11:0]	00000	001	0000	L	111	10111	lp.endi L, uimmL				
0000 0000 0000	src1	010	0000	L	111	10111	lp.count L, rs1				
uimmL[11:0]	00000	011	0000	L	111	10111	lp.counti L, uimmL				
uimmL[11:0]	src1	100	0000	L	111	10111	lp.setup L, rs1, uimmL				
uimmL[11:0]	uimmS[4:0]	101	0000	L	111	10111	lp.setupi L, uimmS, uimmLL				

Bảng 7.3: ALU Bit Manipulation Operations

Mnemonic	Description
p.extract rD, rs1, Is3, Is2	rD = Sext((rs1 & ((1 « Is3) – 1) « Is2) » Is2) Note: Is3 + Is2 must be $\leq 32$
p.extractu rD, rs1, Is3, Is2	rD = Zext((rs1 & ((1 « Is3) – 1) « Is2) » Is2) Note: Is3 + Is2 must be $\leq 32$
p.extractr rD, rs1, rs2	rD = Sext((rs1 & ((1 « rs2[9:5]) – 1) « rs2[4:0]) » rs2[4:0]) Note: rs2[9:5]+ rs2[4:0] must be $\leq 32$
p.extractur rD, rs1, rs2	rD = Zext((rs1 & ((1 « rs2[9:5]) – 1) « rs2[4:0]) » rs2[4:0]) Note: rs2[9:5]+ rs2[4:0] must be $\leq 32$
p.insert rD, rs1, Is3, Is2	rD = rD   (rs1[Is3:0] « Is2) Note: Is3 + Is2 must be $\leq 32$ , the rest of the bits of rD are passed through and are not modified
p.insertr rD, rs1, rs2	rD = rD   (rs1[Is3:0] « rs2[4:0]) Note: rs2[9:5]+ rs2[4:0] must be $\leq 32$ , the rest of the bits of rD are passed through and are not modified
p.bclr rD, rs1, Is3, Is2	rD = rs1 & (((1 « (Is3+1)) – 1) « Is2) Note: Is3 + Is2 must be $\leq 32$
p.bclrr rD, rs1, rs2	rD = rs1 & (((1 « (rs2[9:5]+1)) – 1) « rs2[4:0]) Note: rs2[9:5]+ rs2[4:0] must be $\leq 32$
p.bset rD, rs1, Is3, Is2	rD = rs1   (((1 « (Is3+1)) – 1) « Is2) Note: Is3 + Is2 must be $\leq 32$
p.bsetr rD, rs1, rs2	rD = rs1   (((1 « (rs2[9:5]+1)) – 1) « rs2[4:0]) Note: rs2[9:5]+ rs2[4:0] must be $\leq 32$
p.ff1 rD, rs1	rD = bit position of the first bit set in rs1, starting from LSB. If bit 0 is set, rD will be 0. If only bit 31 is set, rD will be 31. If rs1 is 0, rD will be 32.
p.fl1 rD, rs1	rD = bit position of the last bit set in rs1, starting from MSB. If bit 31 is set, rD will be 31. If only bit 0 is set, rD will be 0. If rs1 is 0, rD will be 32.
p.clb rD, rs1	rD = count leading bits of rs1 Note: This is the number of consecutive 1's or 0's from MSB. Note: If rs1 is 0, rD will be 0.
p.cnt rD, rs1	rD = Population count of rs1, i.e. number of bits set in rs1
p.ror rD, rs1, rs2	rD = Population count of rs1, i.e. number of bits set in rs1

Bảng 7.4: ALU Bit Manipulation Encoding

31	30	29	25	24	20	19	15	14	12	11	7	6	0
			f2	Is3[4:0]	Is2[4:0]		rs1	funct3	rD	opcode			
11			Luimm5[4:0]	Iuimm5[4:0]		src	000	dest	011 0011	p.extract rD, rs1, Is3, Is2			
11			Luimm5[4:0]	Iuimm5[4:0]		src	001	dest	011 0011	p.extractu rD, rs1, Is3, Is2			
11			Luimm5[4:0]	Iuimm5[4:0]		src	010	dest	011 0011	p.insert rD, rs1, Is3, Is2			
11			Luimm5[4:0]	Iuimm5[4:0]		src	011	dest	011 0011	p.bclr rD, rs1, Is3, Is2			
11			Luimm5[4:0]	Iuimm5[4:0]		src	100	dest	011 0011	p.bset rD, rs1, Is3, Is2			
10			00000	src2		src1	000	dest	011 0011	p.extractr rD, rs1, rs2			
10			00000	src2		src1	001	dest	011 0011	p.extractur rD, rs1, rs2			
10			00000	src2		src1	010	dest	011 0011	p.insertr rD, rs1, rs2			
10			00000	src2		src1	011	dest	011 0011	p.bclrr rD, rs1, rs2			
10			00000	src2		src1	100	dest	011 0011	p.bsetr rD, rs1, rs2			
31	25	24	20	19	15	14	12	11	7	6	0		
			funct7	rs2	rs1	funct3	rD	opcode					
000 0100			src2	src1	101	dest	011 0011	p.ror rD, rs1, rs2					
000 1000			00000	src1	000	dest	011 0011	p.ff1 rD, rs1					
000 1000			00000	src1	001	dest	011 0011	p.fl1 rD, rs1					
000 1000			00000	src1	010	dest	011 0011	p.clb rD, rs1					
000 1000			00000	src1	011	dest	011 0011	p.cnt rD, rs1					

Bảng 7.5: Immediate Branching Operations

Mnemonic	Description
p.beqimm rs1, Imm5, Imm12	Branch to PC + (Imm12 « 1) if rs1 is equal to Imm5. Imm5 is signed.
p.bneimm rs1, Imm5, Imm12	Branch to PC + (Imm12 « 1) if rs1 is not equal to Imm5. Imm5 is signed.

Bảng 7.6: Immediate Branching Encoding

31	25	24	20	19	15	14	12	11	7	6	0
			Imm12	Imm5	rs1	funct3	Imm12		opcode		
[12]	[10:5]	[4:0]	src1	010	[4:1]	[11]	110	0011	p.beqimm rs1, Imm5, Imm12		
[12]	[10:5]	[4:0]	src1	011	[4:1]	[11]	110	0011	p.bneimm rs1, Imm5, Imm12		

---

Bảng 7.7: General ALU Operations

Mnemonic	Description
p.abs rD, rs1	$rD = rs1 < 0 ? -rs1 : rs1$
p.slet rD, rs1, rs2	$rD = rs1 \leq rs2 ? 1 : 0$
p.sletu rD, rs1, rs2	$rD = rs1 \leq rs2 ? 1 : 0$
p.min rD, rs1, rs2	$rD = rs1 < rs2 ? rs1 : rs2$
p.minu rD, rs1, rs2	$rD = rs1 < rs2 ? rs1 : rs2$
p.max rD, rs1, rs2	$rD = rs1 < rs2 ? rs2 : rs1$
p.maxu rD, rs1, rs2	$rD = rs1 < rs2 ? rs2 : rs1$
p.exths rD, rs1	$rD = \text{Sext}(rs1[15:0])$
p.exthz rD, rs1	$rD = \text{Zext}(rs1[15:0])$
p.extbs rD, rs1	$rD = \text{Sext}(rs1[7:0])$
p.extbz rD, rs1	$rD = \text{Zext}(rs1[7:0])$
p.clip rD, rs1, Is2	if $rs1 \leq -2^{\hat{I}s2-1}$ , $rD = -2^{\hat{I}s2-1}$ else if $rs1 \geq 2^{\hat{I}s2-1}-1$ , $rD = 2^{\hat{I}s2-1}-1$ else $rD = rs1$
p.clipr rD, rs1, rs2	if $rs1 \leq -(rs2+1)$ , $rD = -(rs2+1)$ , else if $rs1 \geq rs2$ , $rD = rs2$ , else $rD = rs1$
p.clipu rD, rs1, Is2	if $rs1 \leq 0$ , $rD = 0$ , else if $rs1 \geq 2^{\hat{I}s2-1}-1$ , $rD = 2^{\hat{I}s2-1}-1$ else $rD = rs1$
p.clipur rD, rs1, rs2	if $rs1 \leq 0$ , $rD = 0$ , else if $rs1 \geq rs2$ , $rD = rs2$ , else $rD = rs1$
p.addN rD, rs1, rs2, Is3	$rD = (rs1 + rs2) \gg Is3$ Note: Arithmetic shift right. Setting Is3 to 2 replaces former p.avg
p.adduN rD, rs1, rs2, Is3	$rD = (rs1 + rs2) \gg Is3$ Note: Logical shift right. Setting Is3 to 2 replaces former p.avg
p.addRN rD, rs1, rs2, Is3	$rD = (rs1 + rs2 + 2^{\hat{I}s3-1}) \gg Is3$
p.adduRN rD, rs1, rs2, Is3	$rD = (rs1 + rs2 + 2^{\hat{I}s3-1}) \gg Is3$
p.addNr rD, rs1, rs2	$rD = (rD + rs1) \gg rs2[4:0]$
p.adduNr rD, rs1, rs2	$rD = (rD + rs1) \gg rs2[4:0]$
p.addRNR rD, rs1, rs2	$rD = (rD + rs1 + 2^{\hat{I}s2[4:0]}) \gg rs2[4:0]$
p.adduRNR rD, rs1, rs2	$rD = (rD + rs1 + 2^{\hat{I}s2[4:0]-1}) \gg rs2[4:0]$
p.subN rD, rs1, rs2, Is3	$rD = (rs1 - rs2) \gg Is3$
p.subuN rD, rs1, rs2, Is3	$rD = (rs1 - rs2) \gg Is3$
p.subRN rD, rs1, rs2, Is3	$rD = (rs1 - rs2 + 2^{\hat{I}s3-1}) \gg Is3$
p.subuRN rD, rs1, rs2, Is3	$rD = (rs1 - rs2 + 2^{\hat{I}s3-1}) \gg Is3$
p.subNr rD, rs1, rs2	$rD = (rD - rs1) \gg rs2[4:0]$
p.subuNr rD, rs1, rs2	$rD = (rD - rs1) \gg rs2[4:0]$
p.subRNR rD, rs1, rs2	$rD = (rD - rs1 + 2^{\hat{I}s2[4:0]-1}) \gg rs2[4:0]$
p.subuRNR rD, rs1, rs2	$rD = (rD - rs1 + 2^{\hat{I}s2[4:0]-1}) \gg rs2[4:0]$

Bảng 7.8: General ALU Encoding

31	25	24	20	19	15	14	12	11	7	6	0	
funct7	rs2	rs1	funct3	rD	opcode							
000 0010	00000	src1	000	dest	011 0011	p.abs rD, rs1						
000 0010	src2	src1	010	dest	011 0011	p.slet rD, rs1, rs2						
000 0010	src2	src1	011	dest	011 0011	p.sletu rD, rs1, rs2						
000 0010	src2	src1	100	dest	011 0011	p.min rD, rs1, rs2						
000 0010	src2	src1	101	dest	011 0011	p.minu rD, rs1, rs2						
000 0010	src2	src1	110	dest	011 0011	p.max rD, rs1, rs2						
000 0010	src2	src1	111	dest	011 0011	p.maxu rD, rs1, rs2						
000 1000	00000	src1	100	dest	011 0011	p.exths rD, rs1						
000 1000	00000	src1	101	dest	011 0011	p.exthz rD, rs1						
000 1000	00000	src1	110	dest	011 0011	p.extbs rD, rs1						
000 1000	00000	src1	111	dest	011 0011	p.extbz rD, rs1						

31	25	24	20	19	15	14	12	11	7	6	0	
funct7	Is2[4:0]	rs1	funct3	rD	opcode							
000 1010	Iuimm5[4:0]	src1	001	011 0011	011 0011	p.clip rD, rs1, Is2						
000 1010	Iuimm5[4:0]	src1	010	011 0011	011 0011	p.clipu rD, rs1, Is2						
000 1010	src2	src1	010	011 0011	011 0011	p.clipr rD, rs1, Is2						
000 1010	src2	src1	010	011 0011	011 0011	p.clipur rD, rs1, Is2						

31	30	29	25	24	20	19	15	14	12	11	7	6	0
f2	Is3[4:0]	rs2	rs1	funct3	rD	opcode							
00	Luimm5[4:0]	src2	src1	010	101 1011	011 0011	p.addN rD, rs1, rs2, Is3						
10	Luimm5[4:0]	src2	src1	010	101 1011	011 0011	p.adduN rD, rs1, rs2, Is3						
00	Luimm5[4:0]	src2	src1	110	101 1011	011 0011	p.addRN rD, rs1, rs2, Is3						
10	Luimm5[4:0]	src2	src1	110	101 1011	011 0011	p.adduRN rD, rs1, rs2, Is3						
00	Luimm5[4:0]	src2	src1	011	101 1011	011 0011	p.subN rD, rs1, rs2, Is3						
10	Luimm5[4:0]	src2	src1	011	101 1011	011 0011	p.subuN rD, rs1, rs2, Is3						
00	Luimm5[4:0]	src2	src1	111	101 1011	011 0011	p.subRN rD, rs1, rs2, Is3						
10	Luimm5[4:0]	src2	src1	111	101 1011	011 0011	p.subuRN rD, rs1, rs2, Is3						
01	Luimm5[4:0]	src2	src1	010	101 1011	011 0011	p.addNr rD, rs1, rs2						
11	00000	src2	src1	010	101 1011	011 0011	p.adduNr rD, rs1, rs						
01	00000	src2	src1	110	101 1011	011 0011	p.addRNR rD, rs1, rs						
11	00000	src2	src1	110	101 1011	011 0011	p.adduRNR rD, rs1, rs2						
11	00000	src2	src1	110	101 1011	011 0011	p.adduRNR rD, rs1, rs2						
01	00000	src2	src1	011	101 1011	011 0011	p.subNr rD, rs1, rs2						
11	00000	src2	src1	011	101 1011	011 0011	p.subuNr rD, rs1, rs2						
01	00000	src2	src1	111	101 1011	011 0011	p.subRNr rD, rs1, rs2						
11	00000	src2	src1	111	101 1011	011 0011	p.subuRNr rD, rs1, rs2						

## 8 Phụ lục B

Dưới đây là các bài lab hướng dẫn sử dụng.

### 8.1 Lab1: Build hệ điều hành và core RI5CY để boot lên ZedBoard

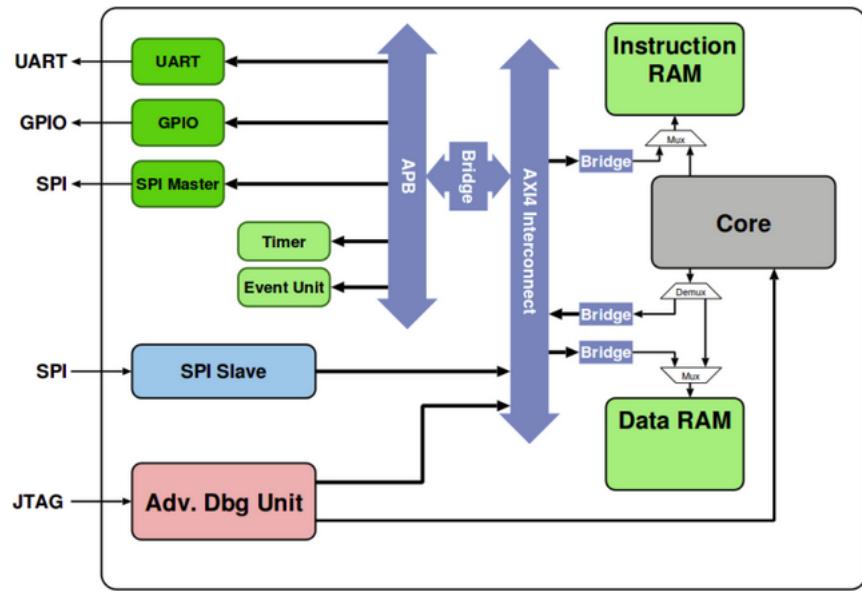
#### 8.1.1 Giới thiệu

Ở bài lab này, chúng ta sẽ tìm hiểu về cách build hệ điều hành và core RI5CY để boot lên Zed Board, ta sẽ sử dụng PULPino để làm nền tảng để nạp, vì PULPino hỗ trợ hệ điều hành và core để synthesized và nạp lên ZedBoard

PULPino là một sản phẩm của PULP-platform, hướng tới việc xây dựng các kiến trúc vi xử lý tiết kiệm năng lượng để sử dụng trong các lĩnh vực như IoT, Wearables... PULPino là một hệ thống vi xử lý đơn lõi mã nguồn mở, dựa trên các lõi RISC-V 32-bit được phát triển tại đại học ETH Zurich ở Thụy Điển, ngoài RI5CY, ta có thể điều chỉnh sử dụng lõi Zero-RI5CY thay thế.

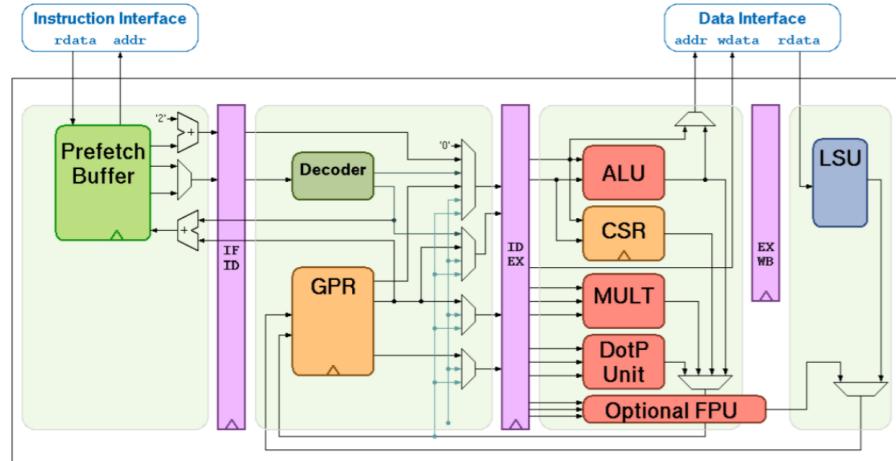
RI5CY là một lõi trật tự đơn lệnh với IPC gần về 1, hỗ trợ hoàn toàn tập lệnh số nguyên (RV32), các lệnh nén (RV32C), tập lệnh nhân mở rộng (RV32M). Và có thể thiết lập để sử dụng tập lệnh dấu chấm động mở rộng (RV32F). RI5CY còn được hỗ trợ để thực hiện thêm các thao tác như: loop phần cứng, post-increment

## 8.1. Lab1: Build hệ điều hành và core RI5CY để boot lên ZedBoard



Hình 8.1: Sơ đồ khái niệm của PULPino

load, lệnh store và 1 số lệnh trong bộ ALU không có trong kiến trúc tập lệnh RISC-V tiêu chuẩn.



Hình 8.2: Kiến trúc của lõi RI5CY

## 8.1. Lab1: Build hệ điều hành và core RI5CY để boot lên ZedBoard

Để tiến hành được các flow như trên, ta cần chuẩn bị môi trường và một số công cụ như sau:

- Nền tảng hệ điều hành sử dụng: Linux 64-bit.
- Các tool cần sử dụng: ri5cy\_gnu\_toolchain, Xilinx SDK + Vivado 2015.1 WebPack, và tcsh.
- Các gói ARM cross compiler: libc6:i386, libstdc++6:i386, zlib1g:i386.
- SD card để boot hệ điều hành lên ZedBoard (nhóm sử dụng SD Card 16 GB).
- Cài đặt gmake và chuẩn bị repo của pulpino.

---

```
1 $ sudo ln -s /usr/bin/make /usr/bin/gmake
2 $ git clone https://github.com/pulp-platform/pulpino
```

---

Ta có 3 cách khác nhau để boot hệ điều hành pulpino lên board: thông qua SD Card, boot bằng JTAG, và boot bằng QSPI, trong đó phương pháp sử dụng SD Card để boot hệ điều hành vì cách boot này thuận tiện cho việc thay đổi config của hệ điều hành (chỉ cần thay đổi trên SD Card), ít tốn công chuẩn bị.

### 8.1.2 Build RI5CY core

Trước tiên, ta thiết lập các biến môi trường cần thiết để build lõi RI5CY, có các biến tùy chỉnh như sau:

- **BOARD**: ta giữ mặc định là "zedboard".
- **XILINX\_PATH**: ta giữ mặc định là "xc7z020clg484-1".
- **XILINX\_BOARD**: ta giữ mặc định là "em.avnet.com:zynq:zed:c".
- **USE\_ZERO\_RISCV**: ta set về 0 để sử dụng lõi RI5CY.
- **RI5CY\_RV32F**: tùy chọn set về 1 để sử dụng hỗ trợ dấu chấm động cho lõi RI5C.

## **8.1. Lab1: Build hệ điều hành và core RI5CY để boot lên ZedBoard**

---

Thực hiện các câu lệnh sau để áp dụng các setting.

---

```
1 $ sudo tcsh
2 $ source /opt/Xilinx/Vivado/2015.1/settings64.csh # Lay cac bien moi
   truong cua Vivado va Xilinx SDK
3 $ setenv RI5CY_RV32F=1 # o day ta co the lua chon core de
   synthesize khac nhu USE_ZERO_RISCV=1 hoac ZERO_RV32M cho
   zero-riscy,..
```

---

Tiến hành build core và hệ điều hành.

---

```
1 $ make all # mat khoang 3 tieng voi Vivado WebPacks
```

---

### **8.1.3 Chuẩn bị SD Card**

Kiểm tra SD kích thước và xác định SD Card đã cắm vào máy (ở đây tên thiết bị của nhóm là /dev/mmcblk0).

---

```
1 $ dmesg
```

---

Sử dụng **dd** Xóa sector đầu tiên (fdisk không thể xóa được các byte đầu thuộc sector đầu tiên của thẻ nhớ sau khi lưu bảng phân vùng)

---

```
1 $ dd if=/dev/zero of=/dev/mmcblk0 bs=1024 count=1
```

---

Tính giá trị new\_cylinder dựa trên kích thước của SD Card. Giá trị của nhóm là 1937, ta có thể lấy các thông tin qua lệnh fdisk( (x = new\_cylinders = <SD\_card\_size>/8825280)).

---

```
1 $ sudo fdisk -l /dev/mmcblk0
```

---

## 8.1. Lab1: Build hệ điều hành và core RI5CY để boot lên ZedBoard

Phân vùng lại SD card với 1 vùng là 200mb cho boot và còn lại để chứa dữ liệu với giá trị number of cylinder là **x**, tham khảo phụ lục A để biết thêm chi tiết.

---

```
1 $ sudo fdisk /dev/mmcblk0
```

---

Bây giờ tiến hành chỉnh các phân vùng, cylinders, heads của SD Card.

---

```
1 Command (m for help): x
2 Expert command (m for help): h
3 Number of heads (1-256, default 30): 255
4 Expert command (m for help): s
5 Number of sectors (1-63, default 29): 63
6 Expert command (m for help): c
7 Number of cylinders (1-1048576, default 2286): <x>
8 Command (m for help): r
```

---

Bây giờ ta có thể tạo phân vùng.

---

```
1 Command (m for help): n
2 Partition type:
3 p primary (0 primary, 0 extended, 4 free)
4 e extended
5 Select (default p): p
6 Partition number (1-4, default 1): 1
7 First sector (2048-15759359, default 2048):
8 Using default value 2048
9 Last sector, +sectors or +size{K,M,G} (2048-15759359, default
10 15759359): +200M
11 Command (m for help): n
12 Partition type:
13 p primary (1 primary, 0 extended, 3 free)
14 e extended
15 Select (default p): p
```

## 8.1. Lab1: Build hệ điều hành và core RI5CY để boot lên ZedBoard

```
16 Partition number (1-4, default 2): 2
17 First sector (411648-15759359, default 411648):
18 Using default value 411648
19 Last sector, +sectors or +size{K,M,G} (411648-15759359, default
20 15759359):
Using default value 15759359
```

---

Tiếp theo, set cờ boot và ID cho các phân vùng.

```
1 Command (m for help): a
2 Partition number (1-4): 1
3
4 Command (m for help): t
5 Partition number (1-4): 1
6 Hex code (type L to list codes): c
7 Changed system type of partition 1 to c (W95 FAT32 (LBA))
8
9 Command (m for help): t
10 Partition number (1-4): 2
11 Hex code (type L to list codes): 83
```

---

Kiểm tra bảng phân vùng và ghi các thay đổi.

```
1 Command (m for help): p
2
3 Disk /dev/sdb: 8068 MB, 8068792320 bytes
4 249 heads, 62 sectors/track, 1020 cylinders, total 15759360 sectors
5 Units = sectors of 1 * 512 = 512 bytes
6 Sector size (logical/physical): 512 bytes / 512 bytes
7 I/O size (minimum/optimal): 512 bytes / 512 bytes
8 Disk identifier: 0x920c958b
9
10 Device Boot Start End Blocks Id System
11 /dev/sdb1 * 2048 411647 204800 c W95 FAT32 (LBA)
```

---

## 8.1. Lab1: Build hệ điều hành và core RI5CY để boot lên ZedBoard

```
12      /dev/sdb2 411648 15759359 7673856 83 Linux
13
14      Command (m for help): w
15      The partition table has been altered!
16
17      Calling ioctl() to re-read partition table.
18
19      WARNING: If you have created or modified any DOS 6.x
20      partitions, please see the fdisk manual page for additional
21      information.
22      Syncing disks.
```

---

Tạo filesystems trên các partiions mới.

---

```
1      $ sudo mkfs.vfat -F 32 -n boot /dev/mmcblk0p1
2      $ sudo mkfs.ext4 -L root /dev/mmcblk0p2
```

---

Copy trong thư mục ..../pulpino/fpga/sw/sd\_image các file BOOT.bin, device-tree.dtb và ulimage vào /mnt/boot. Unmount thẻ SD và cắm lại.

---

```
1      $ umount /mnt/boot
```

---

Giải nén file rootfs.tar trong thư mục .../pulpino/fpga/sw/sd\_image vào phân vùng còn lại của SD card. Lúc này SD card đã sẵn sàng để đưa lên Zedboard.

Compile ứng dụng spiloader trong thư mục .../pulpino/fpga/sw/apps/spiloader và copy vào SD card. Ứng dụng này cho phép Linux sử dụng cơ chế SPI để ghi trực tiếp vào bộ nhớ lệnh và dữ liệu của Pulpino.

---

```
1      $ make
```

---

## 8.1. Lab1: Build hệ điều hành và core RI5CY để boot lên ZedBoard

### 8.1.4 Boot hệ điều hành lên ZedBoard

Chuyển về boot SD Card bằng việc set lên 3.3V các chân JP9 và JP10 và set đất các JP7, JP8, và JP11, và shorten chân JP6.

Ngoài ra ta có thể thiết lập để boot theo kiểu khác như JTAG và QSPI bằng cách thiết lập các chân MIO như bảng sau:

Bảng 8.1: Thiết lập chế độ boot trên ZedBoard

Xilinx TRM	MIO[6] Boot_Mode[4]	MIO[5] Boot_Mode[0]	MIO[4] Boot_Mode[2]	MIO[3] Boot_Mode[1]	MIO[2] Boot_Mode[3]
JTAG mode					
Cascaded JTAG					0
Independent JTAG					1
Boot Devices					
JTAG		0	0	0	
Quad-SPI		1	0	0	
SD Card		1	1	0	
PLL Modes					
PLL Used	0				
PLL Bypassed	1				
Bank Voltages					
MIO Bank 500				3.3V	
MIO Bank 501				1.8V	

Bảng 8.2: Các Jumper dùng cho việc boot hệ điều hành

JP6	PS_MIO0 Pull-Down	Short	Install for SD card boot on CES silicon
JP7	Boot_Mode[3]/MIO[2]	GND - Cascaded JTAG	JTAG Mode. GND cascades PS and PL JTAG chains. VCC makes JTAG chains independent
JP8	Boot_Mode[0]/MIO[3]		
JP9	Boot_Mode[1]/MIO[4]	110 - SD card	Boot Device Select
JP10	Boot_Mode[2]/MIO[5]		
JP11	Boot_Mode[4]/MIO[6]	GND - PLL Used	PLL Select. GND uses PS PLLs. VCC bypasses internal PS PLLs and PL JTAG chains. VCC

Các thiết lập của các chân như sau:

- MIO[2]/Boot\_Mode[3]: chọn chế độ boot cho JTAG.
- MIO[5:3]/Boot\_Mode[2:0]: chọn boot mode.
- MIO[6]/Boot\_Mode[4]: cho phép xài PLL.
- MIO[8:7]/Vmode[1:0]: cố định trên ZedBoard, không chỉnh được.

### Một số chú ý trong quá trình thiết lập

1. Lỗi FPGA awk: symbol lookup error: awk: undefined symbol: mpfr\_z\_sub khi thực thi lệnh make trong thư mục pulpino.

- Lỗi này xảy ra do 2 nguyên nhân: phiên bản Vivado sử dụng khác 2015.1 hay 2015.4.

Khắc phục: nếu phiên bản sử dụng là 2016.x thì sửa dòng sau trong /opt/Xilinx/Vivado/2016.4/bin/loader

```
1      # OS=$(awk '/DISTRIB_ID=/' /etc/*-release | sed
      's/DISTRIB_ID=//' | tr '[upper:]' '[lower:]')
2      OS=$(cat /etc/*-release | grep DISTRIB_ID|sed
      's/DISTRIB_ID=//' | tr '[upper:]' '[lower:]')
```

Còn với 2015.1 thì lỗi này xảy ra khi make lần 2, nguyên nhân là do đường dẫn LD\_LIBRARY\_PATH trong settings64.sh gây xung đột với awk của hệ thống khi set lại lần 2, để khắc phục ta tìm đến /opt/Xilinx/Vivado/2015.1/settings64.csh và để trống LD\_LIBRARY\_PATH.

2. Tốt nhất là phải dùng tcsh để set môi trường và thực hiện lệnh make, khi sử dụng sh và lệnh export thì pulpino sẽ không hiểu và bỏ qua các thiết lập hệ thống như RI5CY\_RV32F.

## 8.2 Lab2: Compile và chạy ứng dụng lên ZedBoard

Trước tiên ta cần phải chuẩn bị ri5cy\_gnu\_toolchain tải từ [https://github.com/pulp-platform/ri5cy\\_gnu\\_toolchain](https://github.com/pulp-platform/ri5cy_gnu_toolchain).

RI5CY\_gnu\_toolchain là một biến thể của riscv\_gnu\_toolchain được mở rộng ra để hỗ trợ nhiều hơn cho các core của PULPino, các core được hỗ trợ:

- RI5CY: compile toolchain với make.
- RI5CY\_FPU (hỗ trợ dấu chấm động): compile với biến môi trường RISCV\_RV32F=1.

## 8.2. Lab2: Compile và chạy ứng dụng lên ZedBoard

- ZeroRI5CY: compile toolchain với biến môi trường `ZERORISCY = 1`
- MicroRI5CY: compile toolchain với biến môi trường `MICRORISCY = 1`.

GNU Toolchain là tập hợp các công cụ lập trình theo GNU Project, nhằm giúp phát triển phần mềm và hệ điều hành, ở đây riscv\_gnu\_toolchain sẽ hỗ trợ phát triển các ứng dụng C và C++ lên trên ZedBoard.

Cross-platform toolchains giúp ta compile các ứng dụng khác platform (như máy tính chạy linux và vi xử lý ARM, các platform này khác binaries), trong trường hợp này, toolchain của chúng ta chạy x86 platforms và có thể tạo được các binaries chạy trên ARM Cores.



Hình 8.3: Cross Platform ToolChain giữa máy host và board mạch

### 8.2.1 Thực hành

Trước tiên compile ri5cy\_gnu\_toolchain, vào thư mục chứa ri5ct\_gnu\_toolchain gõ

```
1 $ make
```

Thêm biến môi trường cho ri5cy\_gnu\_toolchain, ta có thể sửa trong `./bash_rc` hoặc dùng `export`.

```
1 $ export PATH="..../ri5cy_gnu_toolchain-master/install/bin/":${PATH}
```

## 8.2. Lab2: Compile và chạy ứng dụng lên ZedBoard

Tạo thư mục build và copy file cmake\_configure.riscv.gcc.sh vào trong tại thư mục ..../pulpino/sw/, vào thư mục build và thực thi script này để tạo môi trường compile. (nếu dùng lõi RI5CY\_RV32F có thể dùng cmake\_configure.riscvfloat.gcc.sh).

---

```
1 $ chmod +x cmake_configure.riscv.gcc.sh
2 $ ./ cmake_configure.riscv.gcc.sh
```

---

Hệ thống sẽ tạo ra các file cần thiết để build một ứng dụng. Bây giờ compile ứng dụng helloworld có sẵn được viết trên C.

---

```
1 $ sudo make helloworld
```

---

Sau khi compile thành công ta đưa vào trong SD\_card và tiến hành chạy ứng dụng với file spi\_stim.txt trong thư mục ... /pulpino/sw/build/apps/helloworld/slm\_files.

---

```
1 $ ./spiloader -t5 spi_stim.txt # -tx voi x la so giay cho sau khi hoan
tat
```

---

Kết quả như hình dưới

```
# ./spiloader -t20 spi_stim.txt
Device has been reset
Sending block addr 00000000 with 256 entries
Sending block addr 000003FC with 256 entries
Sending block addr 000007F8 with 256 entries
Sending block addr 00000BF4 with 256 entries
Sending block addr 00000FF0 with 256 entries
Sending block addr 000013EC with 176 entries
Sending block addr 00100000 with 256 entries
Sending block addr 001003FC with 58 entries
Starting device
Waiting for EOC...
PULPino: Hello World!!!!
Timeout reached!
Stopped after 20.563
```

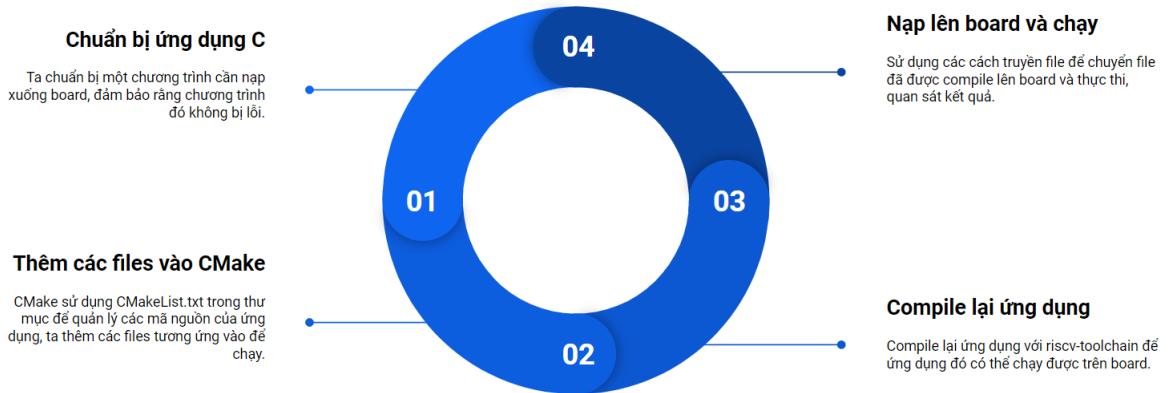
Hình 8.4: Output sau khi thực hiện

**Bài tập:** Tìm hiểu chuyển file xuống SD card dùng UART, JTAG hoặc SCP.

## 8.3 Lab3: Tạo một ứng dụng riêng để nạp lên ZedBoard

### 8.3.1 Giới thiệu

Ở các bài lab trước, chúng ta đã thiết lập hoàn chỉnh ZedBoard lõi RI5CY và chạy thử nghiệm các code mẫu trên đó. Trong bài này, chúng ta sẽ nạp và chạy chương trình do chúng ta viết trên board. Các bước làm được thể hiện như hình dưới đây.



Hình 8.5: Quy trình nạp một ứng dụng tự viết lên board

### 8.3.2 Thực hành

Trước tiên ta sẽ Thêm 1 ứng dụng mới:

- Ta đưa thư mục chứa source của ứng dụng đó vào thư mục .../pulpino/sw/apps.
  - Giả sử với ứng dụng tên là “hello” với source là main.c thì ta thêm vào CmakeLists.txt như sau:

---

1            add\_application(hello main.c)

---

### **8.3. Lab3: Tạo một ứng dụng riêng để nạp lên ZedBoard**

---

- Giả sử với ứng dụng tên là “hello” với source bao gồm main.c và helper.c:

---

```
1      set(SOURCES main.c helper.c)
2      add_application(hello "${SOURCES}")
```

---

- Giả sử với ứng dụng tên là “hello” với source gồm main.c và thư mục con sub\_dir:

---

```
1      add_application(hello main.c SUBDIR "sub_dir")
```

---

- Ta thêm file CMakeLists.txt để khai báo cho tool về ứng dụng.

Sau đó ta thêm dòng sau cung để cấp cho tool biết thư mục chứa ứng dụng vào file CmakeLists.txt trong thư mục .../pulpino/sw.

---

```
1      add_subdirectory(hello)
```

---

Cuối cùng ta tiến hành lại các bước như bài lab trước với ứng dụng compile là ứng dụng ta mới thêm vào.

---

```
1      make <ten_ung_dung>
```

---

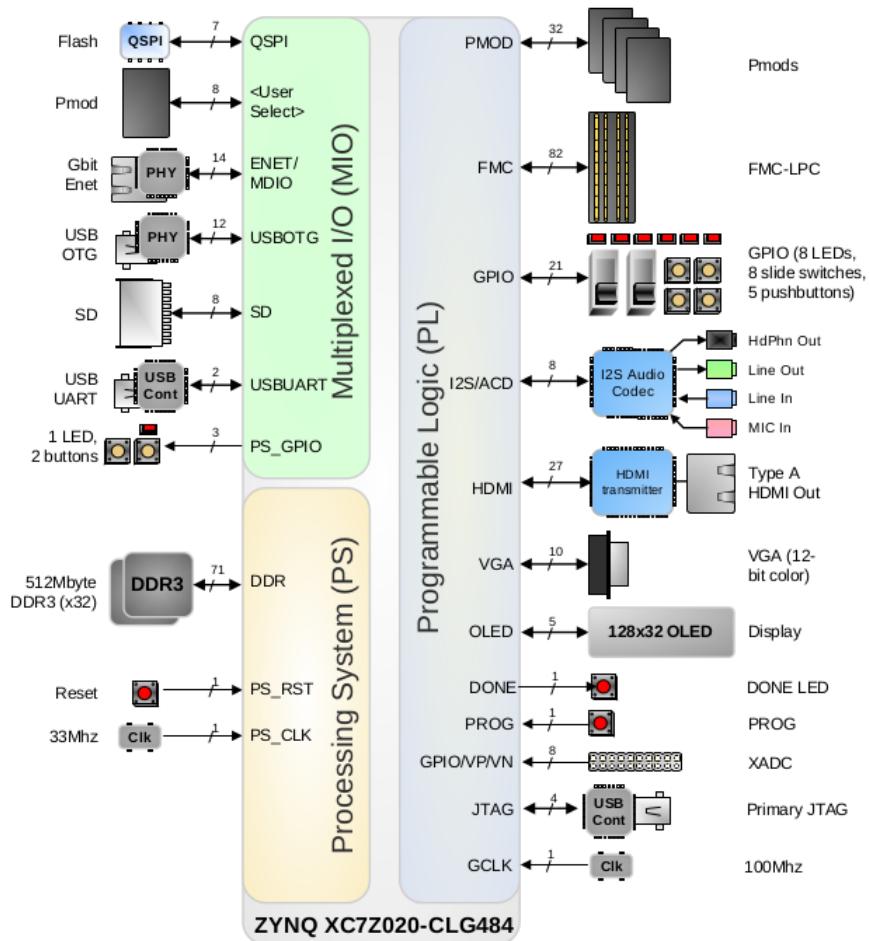
#### **Bài tập:**

- Hiện thực 1 phép nhân ma trận số nguyên 5x5.
- Hiện thực 1 giải thuật sort bất kì đã học.

## 8.4 Lab4: GPIO với Zedboard

### 8.4.1 Giới thiệu

GPIO của Zedboard (LEDs, buttons, switches) được điều khiển qua bộ Programmable Logic (PL).



Hình 8.6: ZedBoard Block Diagram

User push button: Zedboard cung cấp 7 buttons, 5 buttons bên PL (Programmable Logic) và 2 bên PS (Processing System).

## 8.5 Thực hành

Tạo thư mục test\_led trong .../pulpino/sw/apps/. Tạo file main.c và CmakeLists.txt, thêm dòng dưới đây khai báo chương trình cho tool biết.

---

```
1 add_application(test_led main.c)
```

---

Trong main.c ta khai báo thư viện sau. Trong đó gpio.h chứa các lệnh cho phép thiết lập kiểm tra và điều khiển IO.

---

```
1 #include <stdio.h>
2 #include "string_lib.h"
3 #include "utils.h"
4 #include "gpio.h"
5 #include "uart.h"
```

---

Khai báo GPIO:

---

```
1 #define SW_0_BIT  (1 << 0)
2 #define SW_1_BIT  (1 << 1)
3 #define SW_2_BIT  (1 << 2)
4 #define SW_3_BIT  (1 << 3)
5 #define SW_4_BIT  (1 << 4)
6 #define SW_5_BIT  (1 << 5)
7 #define SW_6_BIT  (1 << 6)
8 #define SW_7_BIT  (1 << 7)

9
10 #define LED_0      8
11 #define LED_1      9
12 #define LED_2      10
13 #define LED_3      11
14 #define LED_4      12
15 #define LED_5      13
16 #define LED_6      14
17 #define LED_7      15
```

18

```

19 #define BTN_C_BIT (1 << 16)
20 #define BTN_D_BIT (1 << 17)
21 #define BTN_L_BIT (1 << 18)
22 #define BTN_R_BIT (1 << 19)
23 #define BTN_U_BIT (1 << 20)

```

---

Ta sử dụng các hàm sau:

```

1 void set_pin_function(int pinnumber, int function); // thiet lap
2   chuc nang
3 void set_gpio_pin_direction(int pinnumber, int direction); //thiet
4   lap in hay out
5 void set_gpio_pin_value(int pinnumber, int value); //set gia tri
6   output
7 int get_gpio_pin_value(int pinnumber); //lay gia tri chan GPIO

```

---

Code điều khiển LED bằng Switch:

```

1
2 void waste_time() {
3   int i;
4   for(i = 0; i < 10000; i++) asm volatile("nop");
5 }
6
7 void init_gpio(){
8   int i;
9   for(i = 0; i < 8; i++) {
10     set_gpio_pin_direction(i, DIR_IN);
11     set_pin_function(i, FUNC_GPIO);
12   }
13
14   for(i = 8; i < 16; i++) {
15     set_gpio_pin_direction(i, DIR_OUT);
16     set_pin_function(i, FUNC_GPIO);
17   }
18
19   for(i = 16; i < 20; i++) {

```

```
20         set_gpio_pin_direction(i, DIR_IN);
21         set_pin_function(i, FUNC_GPIO);
22     }
23 }
24
25 void check_sw(){
26     if(get_gpio_pin_value(SW_1_BIT) == 1)
27         set_gpio_pin_value(LED_1, 1);
28     else set_gpio_pin_value(LED_1, 0);
29 }
30
31 int main(){
32     init_gpio();
33     while(1){
34         check_sw();
35         waste_time();
36     }
37 }
```

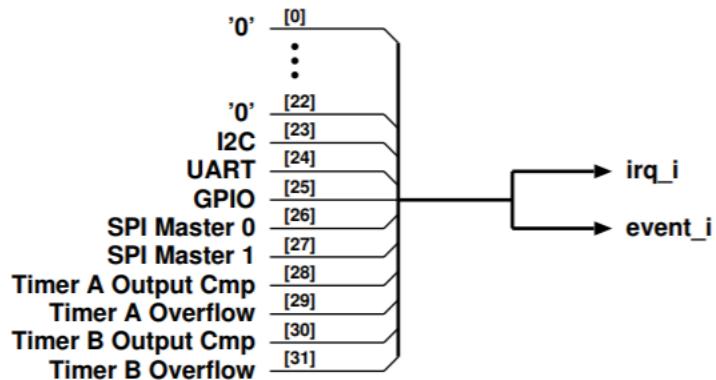
Compile ứng dụng và đưa file spi\_stim.txt xuống sd\_card chạy và kiểm tra kết quả. Source code có thể tải tại [đây](#).

**Bài tập:** Sử dụng button điều khiển các led chạy (khoảng 3 cách chạy led).

## 8.6 Interrupt

### 8.6.1 Giới thiệu

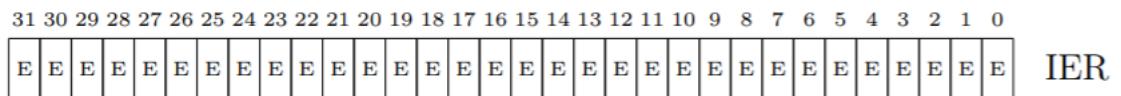
PULPino cung cấp một bộ event và interrupt unit hỗ trợ 32 lines vector ngắn và ngắn với tối đa 32 input lines.[13]



Hình 8.7: Event lines

Ta quan tâm đến các thanh ghi sau:

- IER (Interrupt Enable)

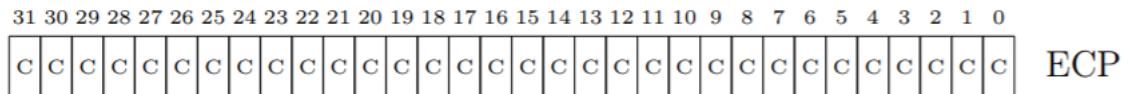


Hình 8.8: IER register

Thanh ghi cho phép ngắt bằng cách set bit ở line mình muốn sử dụng.

## 8.6. Interrupt

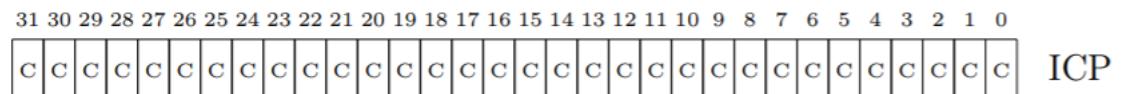
- ECP (Event Clear Pending)



Hình 8.9: ECP register

Thanh ghi cho phép clear pending event bằng cách set bit với line tương ứng.

- ICP (Interrupt Clear Pending)



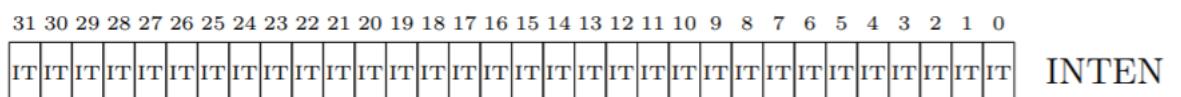
Hình 8.10: ICP register

Thanh ghi cho phép clear pending interrupt bằng cách set bit với line tương ứng.

### 8.6.2 External interrupt

Ngắt ngoài với GPIO ta cần quan tâm đến thanh ghi sau:

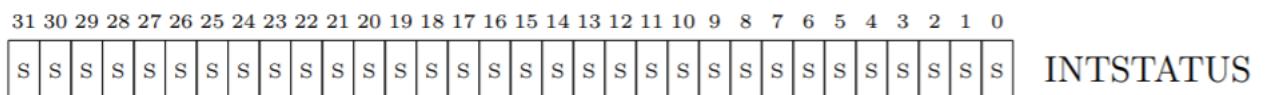
- INTEN (Interrupt Enable)



Hình 8.11: INTEN register

Đây là thanh ghi cho phép ngắn và lựa chọn cách ngắn như thế nào bằng cách set bit INTEN[0] và INTEN[1]. Có 4 lựa chọn ngắn:

- INTEN[1:0] = 0x0: Level 1.
  - INTEN[1:0] = 0x1: Level 0.
  - INTEN[1:0] = 0x2: Ngắt cạnh lên.
  - INTEN[1:0] = 0x3: Ngắt cạnh xuống.  
  - INTSTATUS (Interrupt Status)



Hình 8.12: INTSTATUS register

Thanh ghi trạng thái ngắt cho biết trạng thái ngắt của mỗi GPIO line và sẽ reset khi được đọc.

PULPino hỗ trợ thư viện gpio ta đã tìm hiểu ở bài lab 4 và trong bài lab này ta sẽ sử dụng các hàm sau.

```
1 void set_gpio_pin_irq_type(int pinnumber, int type);
2 void set_gpio_pin_irq_en(int pinnumber, int enable);
3 int get_gpio_irq_status();
```

Thiết lập cho phép ngắt button ở cạnh lên.

```
1 set_gpio_pin_direction(17, DIR_IN);
2 set_pin_function(17, FUNC_GPIO);
3
4 ECP = 0xFFFFFFFF;
5 IER = 1 << 25;
6 ICP = 1 << 25;
7 int_enable();
```

```

9     set_gpio_pin_irq_en(17, 1);
10    set_gpio_pin_irq_type(17, 0x2);

```

---

Mỗi khi xảy ra ngắt ta sẽ xóa cờ ngắt và in ra màn hình số lần bấm nút.

---

```

1 void ISR_GPIO (void){
2     ICP = 1 << 25;
3     if(get_gpio_irq_status() == 0x20000){
4         count++;
5         printf("Button pressed: %d times !!!\n", count);
6     }
7 }

```

---

Compile chương trình rồi nạp xuống Zedboard và kiểm tra kết quả. (Code hiện thực tại [đây](#))

### 8.6.3 Timer Interrupt

Trong bài lab này ta sẽ sử dụng timer A của PULPino để blink led. Một số thanh ghi ta cần quan tâm với timer A:

- TIRA (timer A register): Thanh ghi đếm số chu kì.
- TPRA (timer A control register): thanh ghi điều khiển cho phép hoặc tắt timer A bằng cách set bit TPRA[0] và thiết lập giá trị prescaler với TPRA[5:3].
- TOCRA (timer A output compare register): thanh ghi chứa giá trị kết hợp với prescaler để tính ra chu kì cho mỗi lần kích hoạt ngắt theo công thức sau:

$$Int\_Cycle = \frac{1}{Timer\_clock\_freq} \cdot \frac{1}{(TOCRA + 1) * (Prescaler + 1)}$$

## 8.6. Interrupt

---

Thiết lập timer A và kích hoạt ngắt khoảng mỗi 1s (giá trị TOCRA là nhóm tính tương đối vì không có datasheet cụ thể cho tần số clock của timer PULPino).

---

```
1 ECP = 0xFFFFFFFF;
2 IER = 1 << 29;
3 ICP = 1 << 29;
4 int_enable();
5
6 stop_timer();
7 reset_timer();
8 TOCRA = 5500000;
9 start_timer();
```

---

Mỗi khi ngắt ta tiến hành xóa cờ ngắt và blink led.

---

```
1 void ISR_TA_CMP (void){
2     ICP = (1 << 29);
3     a = !a;
4     set_gpio_pin_value(11, a);
5 }
```

---

Compile chương trình và nạp xuống Zedboard kiểm tra kết quả. (code hiện thực tại [đây](#))

**Bài tập:** Sử dụng ngắt ngoài và timer để blink nhiều led khoảng mỗi 1s với 2-3 cách blink và đổi cách blink bằng button.

# Tài liệu tham khảo

- [1] Lê Nguyễn Trường Giang, Nguyễn Ngọc Phương, Diệp Hưng, Quách Đình Hoàng, Nguyễn Văn Kiên, Nguyễn Hữu Lộc, Tiểu luận môn kiến trúc máy tính nâng cao tìm hiểu về CISC và RISC. September 9th, 2015.
- [2] Yunsup Lee (2018), Keynote: Designing the Next Billion Chips: How RISC-V is Revolutionizing Hardware. *Retrieved on Oct 12th, 2018*
- [3] Nicole Hemsoth (May 16th, 2016), Can Open Source Hardware Crack Semiconductor Industry Economics?. *Retrieved from Can Open Source Hardware Crack Semiconductor Industry Economics? Oct 21st, 2018*
- [4] Andrew Water, Krste Asanović, SiFive Inc. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, University of California, Berkeley, May 7, 2017.
- [5] Jean-Luc Aufranc (CNSOFT), April 6, 2016, PULPino Open Source RISC-V MCU is Designed for IoT and Wearables, *Retrieved on May 2nd, 2019*
- [6] Linus Sebastian, Dec 5th 2018, Design Your Own CPU!!!, *Retrieved on April 18th 2019*
- [7] Andreas Traber, Michael Gautschi, Pasquale Davide Schiavone, RI5CY: User Manual, *Retrieved on March 10th 2019*
- [8] AVNET, Zedboard Getting Started Guide version 7.0, AVNET Inc., 31st Jan 2014
- [9] Pulpino platform, *Retrieved on April 2nd 2019*
- [10] RI5CY toolchain, *Retrieved on April 20th 2019*

- [11] Vivado log, PULPino utilization log
- [12] Synthesis report
- [13] Andreas Traber, Michael Gautschi.PULPino datasheet. ETH zurich
- [14] Zynq-7000 Product Selection guide.
- [15] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Hau-gou, Eric Flamand, Frank K. Gürkaynak, Luca Benini. PULPino: A small single-core RISC-V SoC. ETH zurich