|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ĐẠI HỌC BÁCH KHOA HÀ NỘI  **TRƯỜNG ĐIỆN – ĐIỆN TỬ**  logo_128Ảnh có chứa văn bản, đồ họa véc-tơ  Mô tả được tạo tự động  BÁO CÁO MÔN HỌC  Đồ án thiết kế III  **Đề tài:**  Xây dựng kiến trúc SoC với nền tảng LiteX  Nhóm sinh viên thực hiện:   |  |  | | --- | --- | | Họ, tên | MSSV | | Trần Đức Tâm | 20172799 | | Nguyễn Tất Đạt | 20172455 |   Giảng viên hướng dẫn: Nguyễn Đức Minh |

MỤC LỤC

[DANH MỤC HÌNH VẼ 3](#_Toc111532411)

[DANH MỤC BẢNG BIỂU 3](#_Toc111532412)

[Mở đầu 4](#_Toc111532413)

[CHƯƠNG 1. Đặt vấn đề 5](#_Toc111532414)

[1.1 Động lực 5](#_Toc111532415)

[1.2 Mục tiêu 5](#_Toc111532416)

[1.3 Một số hệ thống đã được triển khai 6](#_Toc111532417)

[1.3.1 Mạch PCIe 3G SDI với kit Artix7, PCIe Gen2 X4 6](#_Toc111532418)

[1.3.2 Linux on LiteX with VexRiscv 6](#_Toc111532419)

[1.3.3 NetTV2 6](#_Toc111532420)

[1.3.4 Fomu 7](#_Toc111532421)

[1.3.5 Betrusted.io 7](#_Toc111532422)

[CHƯƠNG 2. Tìm hiểu chung 8](#_Toc111532423)

[2.1 Tổng quan về nền tảng LiteX 8](#_Toc111532424)

[2.1.1 Migen 9](#_Toc111532425)

[2.1.2 Cấu trúc của LiteX SoC 10](#_Toc111532426)

[2.1.3 Mạch FPGA được hỗ trợ 12](#_Toc111532427)

[2.1.4 Hệ sinh thái LiteX cores 12](#_Toc111532428)

[2.1.5 Công cụ tổng hợp 14](#_Toc111532429)

[2.2 Thiết lập phần mềm cho SoC với LiteX 14](#_Toc111532430)

[2.2.1 LiteX BIOS 14](#_Toc111532431)

[2.2.2 Bootloader 15](#_Toc111532432)

[2.2.3 Linux 16](#_Toc111532433)

[2.2.4 Quá trình boot 16](#_Toc111532434)

[CHƯƠNG 3. Thực hiện 18](#_Toc111532435)

[3.1 Cài đặt LiteX và môi trường linux cho LiteX 18](#_Toc111532436)

[3.2 Cấu hình và xây dựng SoC 19](#_Toc111532437)

[3.2.1 Giới thiệu về mạch FPGA sử dụng 19](#_Toc111532438)

[3.2.2 Phân tích chức năng 20](#_Toc111532439)

[3.2.3 Thực hiện 21](#_Toc111532440)

[3.3 Tùy chỉnh phần mềm chạy trên BIOS 31](#_Toc111532441)

[3.4 Boot Linux 34](#_Toc111532442)

[3.4.1 Chuẩn bị công cụ 34](#_Toc111532443)

[3.4.2 Boot linux chạy trên SoC lên FPGA 35](#_Toc111532444)

[3.4.3 Chạy chương trình trên BIOS 37](#_Toc111532445)

[KẾT LUẬN 40](#_Toc111532446)

[TÀI LIỆU THAM KHẢO 40](#_Toc111532447)

DANH MỤC HÌNH VẼ

[Hình 2.1 Cấu trúc chương trình LiteX 8](#_Toc111532505)

[Hình 2.2 Sơ đồ khối của board FPGA 13](#_Toc111532506)

[Hình 2.2 Cấu trúc phân cấp phần mềm và phần cứng 14](#_Toc111532507)

[Hình 3.1 Board FPGA KC705 19](#_Toc111532508)

[Hình 3.2 Nhóm lệnh trên BIOS 31](#_Toc111532509)

[Hình 3.3 Format phân vùng bootable cho thẻ SD 35](#_Toc111532510)

[Hình 3.4 Nạp bitstream lên board KC705 36](#_Toc111532511)

[Hình 3.5 Nạp thành công SoC lên board thông qua cổng JTAG 36](#_Toc111532512)

[Hình 3.6 Linux chạy trên kiến trúc LiteX Vexriscv 37](#_Toc111532513)

[Hình 3.7 Terminal BIOS 38](#_Toc111532514)

[Hình 3.8 Gợi ý lệnh trên BIOS 38](#_Toc111532515)

[Hình 3.9 Thực thi lệnh packet2snn trên BIOS 38](#_Toc111532516)

[Hình 3.10 Kiểm tra kết quả 39](#_Toc111532517)

DANH MỤC BẢNG BIỂU

[Bảng 2.1 Các IP core được cung cấp bởi LiteX 13](#_Toc111532575)

Mở đầu

Đồ án này hướng đến việc làm cho việc phát triển một bộ vi xử lý RISC-V trở nên đơn giản và dễ tiếp cận hơn. Đề tài sẽ trình bày về một môi trường mã nguồn mở LiteX, giúp hỗ trợ việc xây dựng một kiến trúc System on Chip (SoC) cùng hệ sinh thái của LiteX. Kiến trúc SoC sẽ có thể tích hợp nhiều thành phần được cung cấp bởi LiteX hoặc được tích hợp thêm từ những Intellectual Property (IP) core bên ngoài. Phần cứng sau đó sẽ được nạp vào kit FPGA và chạy hệ điều hành linux trên kiến trúc phần cứng đó.

Đề tài này sẽ giúp ta tạo ra các SoC có thể được sử dụng làm nền tảng cho việc nghiên cứu và học tập. Đây có thể sẽ là một bản phác thảo để tích hợp nhiều chức năng và bộ xử lý khác trong tương lai.

# Đặt vấn đề

Trong những năm gần đây RISC-V đã trở thành một kiến trúc phổ biến cho giới học thuật và ngành công nghiệp chip, vi mạch. Việc tích hợp 1 kiến trúc tập lệnh như RISC-V vào SoC song khá phức tạp và khó tiếp cận. Khi xây dựng một thiết bị ngoại vi sử dụng tài nguyên lõi xử lý của hệ thống thì việc tích hợp nó với kiến trúc SoC của ta cũng là một thách thức lớn. Một công cụ giúp hỗ trợ để dễ dàng tiếp cận hơn là rất cần thiết cho việc nghiên cứu về ngành nghề này.

## Động lực

RISC-V là một lựa chọn phổ biến vì kiến trúc tập lệnh về bản chất có thể mở rộng và tùy chỉnh. Đây cũng là một kiến trúc mở và sẽ ít có khả năng bị vi phạm bản quyền sở hữu trí tuệ. Cùng với sự thành công của ARM, RISC nói chung đã thu hút được nhiều sự chú ý hơn, tạo cơ hội tốt cho RISC-V gia nhập thị trường. Hơn nữa, ngành công nghiệp AI đang ngày một phát triển mạnh mẽ, số lượng chip trên thế giới cũng đang rất khan hiếm đòi hỏi sự phát triển của ngành vi mạch. Các tài nguyên cần thiết để giải quyết các vấn đề dữ liệu lớn đã cho ta thấy 1 kiến trúc có thể tùy chỉnh là rất quan trọng.

Mặc dù các lõi RISC-V có thể phổ biến là vậy, song việc xây dựng một hệ thống hoàn chỉnh sử dụng các lõi này vẫn đòi hỏi rất nhiều kinh nghiệm và nỗ lực. Và để có thể tương tác với người dùng cùng những hệ thống khác, hệ thống của ta cần tích hợp một lượng lớn các thiết bị ngoại vi. Việc hỗ trợ các thiết bị như thế là vô cùng thách thức. Nền tảng LiteX được ra đời để giải quyết những vấn đề này. Ta có thể tập trung vào việc thiết kế các IP phục vụ chức năng của mình và sử dụng lại những thành phần sẵn có mà LiteX cung cấp để đạt hiệu quả tốt nhất. LiteX cũng hỗ trợ rất nhiều mạch FPGA khác nhau giúp cho việc lựa chọn và xây dụng SoC trên FPGA trở nên dễ dàng hơn.

## Mục tiêu

Đồ án này sẽ giới thiệu về LiteX, nền tảng xây dựng SoC cùng với các tiện ích mà nó cung cấp để tạo điều kiện thuận lợi cho việc xây dựng SoC hoàn chỉnh. Mọi triển khai sẽ thực hiện trên môi trường hệ điều hành Unix, cụ thể là trên Linux-Ubuntu.

Nội dung chính của công việc sẽ là xây dựng được một SoC được tùy chỉnh để phục vụ chức năng mong muốn. Ta có thể thêm bớt, sử dụng các thành phần có sẵn mà LiteX cung cấp hoặc tích hợp một IP core bên ngoài sao cho phù hợp với thiết kế của mình.

Ngoài ra, đồ án cũng điều chỉnh để boot linux trên kiến trúc phần cứng đã tạo ra để có thể thực hiện các ứng dụng nhúng trong tương lai. Hệ thống tương lai dự kiến sẽ có thể hoạt động độc lập vì vậy việc chạy linux trên kiến trúc SoC sẽ không cần thông qua máy chủ mà tự hệ thống có thể điều khiển. Vì vậy đề tài hướng đến việc sử dụng thẻ nhớ SD để lưu trữ tập file linux. Triển khai sẽ được thực hiện trên mạch Kintex-7 FPGA KC705 của hãng Xilinx.

## Một số hệ thống đã được triển khai

### Mạch PCIe 3G SDI với kit Artix7, PCIe Gen2 X4

Kiến trúc được xây dựng với LiteX. Nó sử dụng thành phần có sẵn được LiteX cung cấp là LitePCIe để giao tiếp với máy chủ , lõi TripleRateSDI của Xinlinx sẽ được đóng gói cùng với các module datapath để sử dụng hiệu quả bus PCIe mà không cần sử dụng thêm bất kỳ bộ đệm nào.

Dự án này là độc quyền nhưng LiteX ban đầu được phát triển để tạo ra các hệ thống như này và các dự án này cũng sản sinh ra các tính năng/lõi mã nguồn mở để ta sử dụng.

### Linux on LiteX with VexRiscv

Dự án này mô tả cách tạo ra một kiến trúc SoC hỗ trợ Linux với lõi CPU VexRiscv. Một kiến trúc SoC được tạo bởi LiteX cùng với các IP core được viết bằng Migen Python DSL cung cấp bởi LiteX. Tất cả các thành phần được sử dụng để tạo SoC đều là mã nguồn mở và cùng với tính linh hoạt của Spinal HDL/Migen cho phép dễ dàng nhắm đến sử dụng các thiết bị FPGA khác nhau.

Dự án cũng trình bày cách các HDL bậc cao cho phép các tính năng mới và bổ sung cho nhau.

### NetTV2

NetTV2 là 1 mạch thu/phát HDMI dựa trên mạch FPGA Artix7 của Xinlinx. Kiến trúc SoC thực hiện một khung hiển thị từ kit Raspberry Pi 3B+ theo kết nối HDMI. SoC được xây dựng bằng LiteX, lõi CPU Vexriscv được dùng để điều khiển, LiteDRAM là bộ điều khiển DDR3 dùng để lưu trữ chương trình firmware cho CPU, bộ đệm HDMI và LiteVideo để thu/phát HDMI.

Kiến trúc SoC libre sử dụng đầu nối PCIe Gen2 X4 của bo mạch với LitePCIe để tạo thiết bị thu/phát HDMI.

### Fomu

Fomu là một mạch FPGA nhỏ để phù hợp với dự án về cổng USB. Kiến trúc SoC được xây dựng bằng LiteX, sử dụng 1 bản tùy chỉnh của lõi VexRiscv để điều khiển, cùng với lõi ValentyUSB cho phép FPGA giao tiếp với máy chủ và mô phỏng các thiết vị USB.

Fomu trình bày cách các thiết kế iCE40 có thể nhỏ và mạnh tới mức nào. Fomu có thể chạy Micropython/Zephyr trên CPU RISC-V của nó trong khi cho phép debug phần mềm qua GDB và debug phần cứng qua cổng USB; cùng với đó có thể sửa đổi, tự lập trình lại trực tiếp từ USB với Foboot.

### Betrusted.io

Betrusted là một dự án về máy tính bảo mật, một máy tính mà cho phép truy cập từ bất cứ ai nhưng chỉ được niêm phong bởi người dùng. Hơn thế nữa, đấy còn là một hệ thống hoàn thiện với màn hình và bàn phím, đủ khả năng để thực hiện các tương tác bảo mật bởi người dùng.

Dự án được xây dựng từ 2 kiến trúc SoC FPGA tạo bởi LiteX: kiến trúc SoC Embeded Controller trên iCE40 và kiến trúc SoC Secure Domain trên Spartan7.

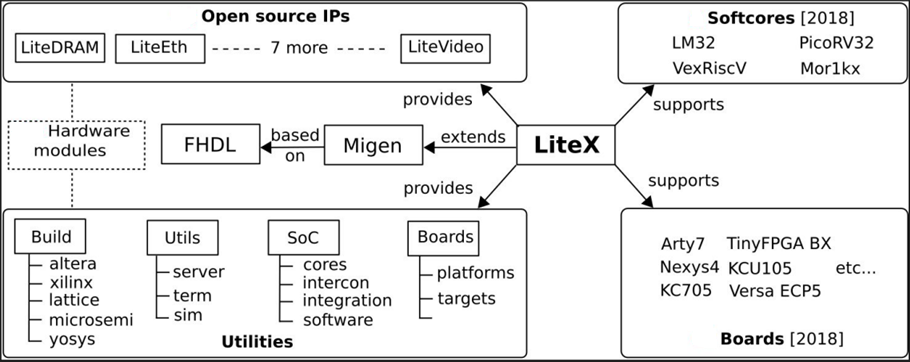
# Tìm hiểu chung

Chương này sẽ giới thiệu tổng quan về SoC và công cụ LiteX giúp tổng hợp SoC FPGA. Ngoài ra cũng đề cập về các tiện ích cần thiết để điều chỉnh mà mô tả một kiến trúc mới phù hợp yêu cầu.

## Tổng quan về nền tảng LiteX

LiteX là một nền tảng mã nguồn mở cung cấp các thư viện và tiện ích cần thiết để xây dựng nên một kiến trúc SoC thiết kế trên FPGA. LiteX hỗ trợ rất nhiều softcore CPU và các thành phần ngoại vi thiết yếu khác nhau mà không phụ thuộc vào vấn đề bản quyền. Điểm đặc biệt là các thành phần của LiteX được mô tả bằng ngôn ngữ bậc cao DSL (Domain Specific Language) là Migen Python, giúp dễ tiếp cận và đơn giản hóa thiết kế đối với những ai ít kinh nghiệm về phần cứng hơn. Cũng vì được viết bằng Python mà ta có thể sử dụng các tính năng hướng đối tượng của ngôn ngữ này.

LiteX trang bị RTL cho các thiết bị ngoại vi của nó, một hệ thống xây dựng và một thiết bị đầu cuối để kết nối và nạp phần mềm xuống FPGA, các tiện ích để điều chỉnh bộ nhớ RAM and một bộ phân tích logic. Migen có thể sinh ra Verilog code, vì vậy có thể kết nối (System)Verilog RTL với khối được viết bằng Migen. Điều này cho phép người dùng kết hợp các lõi CPU với các thiết bị ngoại vi. Đây cũng là một nền tảng thiết kế độc lập nên việc dùng kiến trúc hiện có để đưa lên một mạch FPGA mới sẽ dễ dàng hơn. Hình dưới đây mô tả cấu trúc chương trình của LiteX:



Hình 2.1 Cấu trúc chương trình LiteX

Các thành phần của LiteX được mô tả cụ thể sau đây.

### Migen

Migen là một chương trình viết bằng Python với mục đích tự động hóa quá trình thiết kế mạch tích hợp. Thiết kế được viết bằng Python dựa vào các thư viện có sẵn của Migen, sau đó được dịch ra Verilog để tổng hợp và tạo thành bitstream nạp lên mạch FPGA. Migen sử dụng cấu trúc lớp (class) của Python để mô tả thiết kế, trong đó có 2 lớp cơ bản được sử dụng là Module và Signals. Signals có thể coi là 1 kết nối (net) giữa các cổng của các khối trong thiết kế. Lớp Module mô tả cấu trúc của mạch tích hợp, có thể bao gồm mạch tuần tự, mạch tổ hợp, các khối con, ... Đặc biệt là Migen cho phép tích hợp các thiết kế được viết bằng Verilog hoặc VHDL thông qua thuộc tính Specials của lớp Module.

Migen được phát triền để cải thiện những điểm hạn chế của các ngôn ngữ mô tả phần cứng thông thường như Verilog và VHDL. Migen sử dụng hai khái niệm chính là mệnh đề tổ hợp (combinatorial statement) và mệnh đề đồng bộ (synchronous statement) thay cho mô hình theo sự kiện như always@(\*) trong Verilog. Điều này giúp việc thiết kế mạch theo hành vi dễ dàng hơn đồng thời giảm thiếu những sai sót trong quá trình lập trình với các ngôn ngữ thông thường. Việc tận dụng class trong Python cũng giúp quá trình phân cấp và kết nối các khối trong thiết kế thuận tiện hơn.

|  |
| --- |
| from migen import \*  from migen.fhdl import verilog  class Blinker(Module):  def \_\_init\_\_(self, sys\_clk\_freq, period):  self.led = led = Signal()  # # #  toggle = Signal()  counter\_preload = int(sys\_clk\_freq\*period/2)  counter = Signal(max=counter\_preload + 1)  self.comb += toggle.eq(counter == 0)  self.sync += \  If(toggle,  led.eq(~led),  counter.eq(counter\_preload)  ).Else (  counter.eq(counter - 1)  )  # Create a 10Hz blinker from a 100MHz system clock.  blinker = Blinker(sys\_clk\_freq = 100e6, period = 1e-1) print(verilog.convert(blinker, {blinker.led})) |

Đoạn code trên mô tả 1 khối chức năng được viết bằng Python Migen. Thiết kế này gồm 1 bộ đếm sẽ kích hoạt đèn led khi đếm về 0. Cấu trúc của chương trình như sau:

* Lớp Blinker được khai báo kế thừa từ lớp Module. Lớp này tương đương với khai báo Module trong Verilog hay entity trong VHDL
* “Signal” là thành phần cơ bản trong các thiết kế viết bằng Migen, giống như wire/reg trong Verilog. Lớp này được Migen định nghĩa lại để việc khai báo đơn giản hơn. Tham số hoặc biểu thức đưa vào sẽ là độ rộng bit của tín hiệu.
* “eq” là phương thức gán giá trị cho một tín hiệu.
* Thiết kế được phân chia thành nhóm mạch tổ hợp, xác định bằng câu lệnh “self.comb += {expr}”, và nhóm mạch tuần tự, được xác định bằng câu lệnh “self.sync += {expr}”.
* Lệnh “If” thể hiện câu lệnh điều kiện, với cấu trúc như trong Verilog gồm điều kiện (toggle == 1) và các lệnh thực thi (led.eq(~led))

### Cấu trúc của LiteX SoC

System on Chip là hệ thống mạch tích hợp bao gồm các thành phần cần thiết để tích hợp trên 1 con chip duy nhất. Các thành phần này có thể bao gồm số (digial), mạch tương tự (analog) và sự pha trộn giữa 2 mạch này (mixed-signal). SoC ngày nay thường có khả năng xử lý chạy rất nhiều ứng dụng âm thanh, hình ảnh, giải mã tín hiệu,… thậm chí bao gồm cả những chức năng xử lý dữ liệu lớn phục vụ cho AI.

LiteX SoC bao gồm các thành phần chính sau:

* + 1. Bộ xử lý

Bộ xử lý là thành phần quan trọng trong SoC, nó thực thi các lệnh của chương trình và điều khiển hoạt động của toàn bộ SoC. Bộ xử lý của LiteX SoC có thể gồm một hoặc nhiều lõi RISCV. Các nhân này được viết bằng Verliog hoặc VHDL và là các dự án mã nguồn mở như VexRISCV, LM32, PicoRV32, ... Các bộ xử lý này được tích hợp sẵn trong LiteX và người dùng có thể lựa chọn bộ xử lý, số nhân và cấu hình các thành phần khác dựa theo bộ xử lý đã chọn.

* + 1. Hệ thống bus

Bus là thành phần quan trọng kết nối tất cả thành phần còn lại trong soc. Dữ liệu và các lệnh sẽ được truyền qua lại giữa bộ xử lý và bộ nhớ, ngoại vi, ... thông qua bus. Hệ thống bus mặc định trong LiteX Soc là Wishbone bus nhưng người dùng có thể thay đổi thành các loại bus được hỗ trợ khác như AXI, Avalon,…

Wishbone bus là một hệ thống bus mã nguồn mở được thiết kế để chuẩn hóa việc kết nối các khối IP core với nhau, hỗ trợ cho quá trình học tập, nghiên cứu dễ dàng hơn. Mỗi khối IP core khi kết nối với hệ thống bus chỉ cần có một khối giao diện giải mã các tìn hiệu từ bus thành tín hiệu điều khiển cho IP core.

* + 1. Bộ nhớ

Với các SoC ASIC, bộ nhớ chính của hệ thống thường được thiết kế và tích hợp trực tiếp trên die khi sản xuất. Với FPGA, các hãng sản suất sẽ tích hợp 1 bộ nhớ tốc độ cao (1 thanh RAM) bên ngoài FPGA. LiteX sẽ coi bộ nhớ này như là bộ nhớ của SoC và thiết kế một bộ khối riêng (sẽ được nói đến trong phần 5.1.4) để kết nối và sử dụng nó.

Cấu trúc của bộ nhớ cảu SoC nói chung hay của LiteX nói riêng sẽ gồm một phần là các thanh ghi điều điều khiển và thanh ghi đa chức năng, một phần là bộ nhớ chương trình và dữ liệu. Ngoài ra trên FPGA, sẽ có một phần bộ nhớ được dùng để lưu trữ cài đặt ban đầu (bitstream, chương trình, ...) của FPGA. Người dùng LiteX hoàn toàn có thể thay đổi, chỉnh sửa cấu trúc bộ nhớ tùy theo kiến trúc của SoC với các thư viện bộ nhớ có sẵn.

* + 1. Ngoại vi và I/O

LiteX hướng tới việc xây dựng các hệ thống trên FPGA nên LiteX có hỗ trợ rất nhiều các giao thức kết nối thông dụng như UART, I2C, SPI, JTAG, … cho phép SoC kết nối với nhiều thiết bị ngoại vi cùng một lúc. Các giao thức này đã được LiteX thiết kết sẵn bằng Migen nên việc thêm, bớt hoặc cấu hình lại rất đơn giản.

Để sử dụng các giao thức kết nối thì người dùng cũng phải khai báo các chân I/O pin sử dụng trên FPGA. Việc này thông thường sẽ được thực hiện khi tổng hợp thiết kế trên phần mềm do nhà sản xuất mạch FPGA cung cấp. LiteX cũng cho phép người dùng khai báo sẵn các chân I/O sử dụng khi thiết kế và LiteX sẽ tự động thêm nó vào công cụ tổng hợp.

### Mạch FPGA được hỗ trợ

Hiện tại đã có hơn 120 mạch FPGA được LiteX hỗ trợ, chúng đến từ nhiều hãng sản xuất nổi tiếng như Xilinx, Intel, Lattice,… hoặc là những dự án mạch FPGA mã nguồn mở như Fomu, NeTV2, OrangeCrab,… Ngoài những mạch được hỗ trợ sẵn từ LiteX, người dùng cũng có thể tự xây dựng SoC trên một mạch FPGA mới một cách dễ dàng nhờ lớp cấu trúc LiteX-board được định nghĩa trên LiteX.

Để sử dụng mạch FPGA với LiteX, người dùng cần tạo 2 tệp lưu trong thư mục *…/LiteX\_boards/platforms* và *…/LiteX\_boards/targets*. Tệp trong thư mục *platforms* cung cấp định nghĩa các thành phần trên mạch FPGA gồm chân I/O sử dụng, clocks, các tính năng trên mạch, phương thức tải bitstream, … Đây là tệp thông tin quan trọng nên người dùng cần tìm hiểu và khai báo chỉnh xác theo cấu trúc mẫu của LiteX. Tệp trong thư mục *targets* sẽ chứa thiết kế của người dùng. LiteX sẽ sử dụng 2 tệp này trong quá trình tổng hợp thiết kế và tải thiết kế lên mạch FPGA. Thiết kế logic trong targets sau khi được tổng hợp sẽ kết nối những chức năng được sử dụng với những khối tính năng trên mạch FPGA. Mô tả chi tiết nội dung sẽ được nêu trong chương thực hiện.

### Hệ sinh thái LiteX cores

Với những hệ thống mạch FPGA phức tạp, có rất nhiều khối tính năng được thêm trên mạch để tối ưu logic sử dụng bên trong FPGA, có thể kể đến RAM, bộ nhớ ngoài, giao tiếp PCIe, Ethernet,... Các khối chức năng này cho phép người dùng sử dụng luôn mà không cần phải thiết kế logic cho chức năng đó trong FPGA, nhưng ngược lại, người dùng phải thiết kế thêm logic để kết nối với khối chức năng bên ngoài FPGA.

Để khắc phục nhược điểm này, LiteX phát triển một hệ sinh thái các khối IP core bằng Migen để thực hiện việc giao tiếp FPGA với các ngoại vi trên mạch. Các khối IP core này được thiết kế một cách tổng quát nhất dựa trên cấu trúc, cách kết nối của module có trên mạch FPGA đã được hỗ trợ. Từ đó người dùng có thể sử dụng hoặc tùy chỉnh để chạy trên các mạch FPGA mà LiteX chưa hỗ trợ sẵn.

Diagram

Description automatically generated

Hình 2.2 Sơ đồ khối của board FPGA

Thông tin về các IP core được liệt kê trong bảng dưới đây:

Bảng 2.1 Các IP core được cung cấp bởi LiteX

|  |  |
| --- | --- |
| IP core | Mô tả |
| LiteDRAM | Khối điều khiển Dynamic RAM, hỗ trợ các chuẩn SDRAM, DDR2/3/4. |
| LiteEth | Khối Ethernet, triển khai ở lớp vật lý cho đến lớp MAC(HW/SW interface). |
| LitePCIe | Khối PCIe, điều khiển kết nối PCIe giữa FPGA và máy tính chủ. |
| LiteSATA | Khối SATA, hỗ trợ kết nối với ổ cứng bằng giao thức SATA với tốc độ có thể đạt 6.0 GBps |
| LiteSDCard | Khối thẻ SD, hỗ trợ kết nối với thẻ SD, tốc độ đọc/ghi có thể đạt 55/54 MB/s |
| LiteICLink | Inter-Chip Communication, hỗ trợ kết nối giữa các bộ xử lý (chip-to-chip hoặc FPGA-to-FPGA), thường có trên các mạch có sẵn bộ xử lý và FPGA (như Xilinx Ultrascale) |
| LiteSPI | Hỗ trợ tải dữ liệu bằng SPI/QSPI lên flash memory có sẵn trên mạch FPGA |
| LiteScope | Hỗ trợ debug thông qua công cụ LiteScope |

### Công cụ tổng hợp

Để tạo thành một quy trình thiết kế hoàn chỉnh từ thiết kế đến triển khai trên mạch thì LiteX cho phép người dùng truy cập và sử dụng các phần mềm tổng hợp do hãng sản xuất FPGA cung cấp như Xilinx Vivado, Altera(Intel) Quartus, ... Các công việc như tạo dự án, thêm rằng buộc (constraints) cho thiết kế, tổng hợp có thể được thực hiện trên LiteX bằng cách thêm các thư viện hỗ trợ các phần mềm tổng hợp. Người dùng hoàn toàn có thể kết hợp sử dụng phần mềm tổng hợp trên LiteX hoặc trực tiếp trên GUI của phần mềm để đạt hiệu quả nhất.

Sau khi tổng hợp, thiết kế được tải lên FPGA cũng bằng các phần mềm trên. LiteX cũng có thể điều khiển phần mềm thực hiện việc này. Bên cạnh đó, nếu người dùng muốn lập trình phần mềm cho FPGA thì LiteX cũng có công cụ biên dịch vào cũng thông qua phần mềm tổng hợp để tải lên FPGA.

## Thiết lập phần mềm cho SoC với LiteX

Cấu trúc các lớp phần mềm và phần cứng của một hệ thống SoC được mô tả như hình 2.2. Các lớp thấp nhất sẽ là lớp phần cứng đại diện cho các thành phần ảnh hưởng nhiều nhất đến toàn bộ hệ thống. Lớp thấp nhất của phần mềm là BIOS, cho phép ta cấu hình và tương tác với phần cứng. Bootloader sẽ được chạy dựa trên kiến trúc tập lệnh RISC-V và là nền tảng để chạy các lớp cao hơn là hệ điều hành và ứng dụng.

Table

Description automatically generated

Hình 2.2 Cấu trúc phân cấp phần mềm và phần cứng

### LiteX BIOS

Khi khởi động CPU, chương trình đầu tiên được thực thi sẽ là nhóm chương trình BIOS. LiteX có định một vài chương trình cơ bản ở BIOS giúp tương tác với phần cứng và các thiết bị ngoài vi. Các chương trình này sẽ được nạp xuống bộ nhớ trong của FPGA. Mặc định rằng hệ thống sẽ cố gắng thực hiện boot vào hệ điều hành trước, nếu không có phương thức boot nào được thực hiện thì hệ thống sẽ vào giao diện của bios. Từ BIOS cũng có thể thực hiện boot vào hệ điều hành bằng các chương trình BIOS đã được định nghĩa sẵn.

LiteX cung cấp 3 phương thức để nạp bootloader. Đầu tiên nó sẽ luôn ưu tiên thực hiện serial boot thông qua cổng UART. Đây là phương thức đơn giản và phổ biến nhất. Tuy nhiên việc boot thông qua UART khá tốn thời gian và phụ thuộc vào baud rate được ta chỉ định. Phương thức thứ 2 khá hay và có thể được áp dụng cho nhiều hệ thống hoạt động đập lập là sdcard boot, thực hiện bằng việc sử dụng thẻ sd được kết nối với board. Boot qua thẻ SD sẽ rất thuận tiện và nhanh chóng. Nếu ta ví việc boot qua serial giống với việc đi bộ thì boot thông qua SD card sẽ giống như ta đang phóng xe máy với tốc độ 60km/h. Cùng với sự tiện dụng đó thì đây là một cách thức tương đối phức tạp ở quá khứ, hiện tại sau quá trình xây dựng thì LiteX đã hoàn thiện và đảm bảo có thể thực hiện cách thức này mà không gặp vấn đề gì. Tất cả những gì ta cần đó là chuẩn bị 1 thẻ SD đã được tùy chỉnh bootable. Cuối cùng, phương thức nhanh nhất đó là boot thông qua Ethernet. Đồ án này chưa được thực hiện và thử nghiệm phương thức này vì vậy Ethernet sẽ không được đề cập chi tiết.

Bên cạnh các tùy chọn về boot, LiteX BIOS cũng cung cấp các chương trình cho phép tương tác với phần cứng như đọc ghi vào bộ nhớ, khởi tạo giá trị cho LED, kiểm tra hệ thống… Thêm vào đó, ta cũng có thể tự định nghĩa một chương trình ở BIOS và xây dựng cùng với kiến trúc SoC để thực hiện việc kiểm tra chức năng của phần cứng mà ta tùy chỉnh. Mọi công việc đều khá dễ thực hiện với sự tiện dụng của LiteX.

### Bootloader

Bootloader là bộ chương trình giúp CPU chạy được hệ điều hành hoặc các ứng dụng khác. Đề tài này thực hiện chạy linux trên kiến trúc phần cứng được xây dựng.

U-boot, một bootloader có mã nguồn mở được sử dụng rộng rãi trong các hệ thống nhúng nhỏ. Nó hỗ trợ rất nhiều kiến trúc, bao gồm 68k, ARM, Blackfin, Microblaze, MIPS, Nios, SuperH, PPC, RISC-V và x86. Khi hoạt động nó sẽ khởi tạo phần cứng và nạp các thành phần của hệ điều hành linux (kernel, rootfs, device tree) lên RAM và trao quyền lại cho linux kernel.

### Linux

Đề tài này sẽ chỉ sử dụng buildroot để tạo 1 phiên bản hệ điều hành linux nhỏ gọn. Buildroot là một build-system Linux nhúng, gồm một tập các makefile và các patch để tạo ra một cross-compilation toolchain và root file-system cho hệ thống linux mà ta muốn xây dựng. Các thành phần của linux có thể được tùy chỉnh thông qua giao diện menuconfig của buildroot.

Trong buildroot có một số thành phấn giúp ta chạy thiết lập một hệ điều hành:

* Makefile: chứa các script để build OS với các config khác nhau
* Config.in: chứa các option giúp ta config dễ dàng hơn
* .config: file log giúp lưu lại các cài đặt config của ta

Sau khi build, ta sẽ có đầy đủ thành phần cơ bản của một hệ điều hành: kernel, root file-system, bootloader và các thành phần phụ trợ khác.

### Quá trình boot

* Boot ROM:

Khi hệ thống khởi động lần đầu tiên, hoặc reset. Quyền kiểm soát hệ thống sẽ thuộc về reset vector, nó là một đoạn mã assembly được định nghĩa khi xây dựng SoC và LiteX đã định nghĩa sẵn. Sau đó reset vector sẽ trỏ tới địa chỉ vùng nhớ chứa các đoạn mã khởi động đầu tiên, cụ thể là boot rom. Nếu không có reset vector thì bộ xử lý sẽ không biết nên thực thi bắt đầu từ đâu.

Chức năng chính của boot rom đấy chính là sao chép nội dung trong file "**MLO**" hay còn được gọi là Second Program Loader (**SPL**) - chương trình tải phụ vào IRAM và excute nó. SPL sau đó tiếp tục setup các thành phần cần thiết như DRAM controler, eMMC, Sau đó load U-boot vào RAM.

* Bootloader:

Sau khi được load vào RAM, u-boot sẽ thực hiện việc relocation. Di dời đến địa chỉ **relocaddr** của RAM (Thường là địa chỉ cuối của RAM) và nhảy đến mã của u-boot sau khi di dời.

* Linux kernel:

Sau khi nhận được quyền kiểm soát và các kernel parameters từ u-boot. Kernel sẽ thực hiện mount hệ thống file system (**Rootfs**) và cho chạy tiến trình **Init** trên RAM. Đây là tiến trình được chạy đầu tiên khi hệ thống khởi động thành công và chạy cho tới khi hệ thống kết thúc. Tiến trình **Init** sẽ khởi tạo toàn bộ các tiến trình con khác trên user space, các applications tương tác trực tiếp với người dùng. Lúc này, hệ thống của chúng ta đã hoàn toàn sẵn sàng cho việc sử dụng.

# Thực hiện

## Cài đặt LiteX và môi trường linux cho LiteX

Trước khi thực hiện đề tài, ta cần cài đặt môi trường LiteX và linux trên LiteX.

* Đầu tiên cần cài đặt một môi trường hệ điều hành linux (Ubuntu). Mọi công việc sẽ được triển khai trên đây. Khuyến nghị việc sử dụng bản Ubuntu ổn định và vẫn được hỗ trợ. Đề tài này được thực hiện trên phiên bản Ubuntu 20.04.
* Tiếp đó là cài đặt mã nguồn LiteX được công khai trên github:
  + Cài đặt Migen và các thành phần của LiteX:

|  |
| --- |
| $ wget https://raw.githubusercontent.com/enjoy-digital/LiteX/master/LiteX\_setup.py  $ chmod +x LiteX\_setup.py  $ ./LiteX\_setup.py --init --install --user --config=full |

* + Cài đặt các thư viện cần thiết:

|  |
| --- |
| $ sudo apt install libevent-dev libjson-c-dev build-essential device-tree-compiler wget git python3-setuptools |

* + Tùy vào board FPGA sử dụng mà cũng cần cài đặt tool phát triển của hãng đó. Đề tài này sử dụng board của hãng Xilinx vì vậy cần cài đặt Vivado
  + Cài đặt toolchain biên dịch chèo cho RISC-V

|  |
| --- |
| $ pip3 install meson ninja  $ wget https://static.dev.sifive.com/dev-tools/riscv64-unknown-elf-gcc-8.1.0-2019.01.0-x86\_64-linux-ubuntu14.tar.gz  $ tar -xvf riscv64-unknown-elf-gcc-8.1.0-2019.01.0-x86\_64-linux-ubuntu14.tar.gz  export PATH=$PATH:$PWD/riscv64-unknown-elf-gcc-8.1.0-2019.01.0-x86\_64-linux-ubuntu14/bin/ |

* Cài đặt sbt và scala: Cài đặt phiên bản javac tùy chọn rồi cài đặt scala. Để cài đặt sbt làm theo sau:

|  |
| --- |
| $ sudo apt-get update  $ sudo apt-get install apt-transport-https curl gnupg -yqq  $ echo "deb https://repo.scala-sbt.org/scalasbt/debian all main" | sudo tee /etc/apt/sources.list.d/sbt.list  $ echo "deb https://repo.scala-sbt.org/scalasbt/debian /" | sudo tee /etc/apt/sources.list.d/sbt\_old.list  $ curl -sL "https://keyserver.ubuntu.com/pks/lookup?op=get&search=0x2EE0EA64E40A89B84B2DF73499E82A75642AC823" | sudo -H gpg --no-default-keyring --keyring gnupg-ring:/etc/apt/trusted.gpg.d/scalasbt-release.gpg --import  $ sudo chmod 644 /etc/apt/trusted.gpg.d/scalasbt-release.gpg  $ sudo apt-get update  $ sudo apt-get install sbt |

* Cuối cùng, cài đặt môi trường linux on LiteX: tại thư mục LiteX

|  |
| --- |
| $ git clone https://github.com/LiteX-hub/linux-on-LiteX-vexriscv  $ cd linux-on-LiteX-vexriscv  $ sudo apt install libtool automake pkg-config libusb-1.0-0-dev  $ git clone https://github.com/ntfreak/openocd.git  $ cd openocd  $ ./bootstrap  $ ./configure --enable-ftdi  $ make  $ sudo make install |

## Cấu hình và xây dựng SoC

### Giới thiệu về mạch FPGA sử dụng

Diagram, schematic

Description automatically generated

Hình 3.1 Board FPGA KC705

Thiết kế SoC sẽ được triển khai trên board mạch KC705. Đây là một mạch FPGA đã được LiteX hỗ trợ, nên ta có thể chạy thiết kế vào tải lên board mà không cần lo lắng về việc khai báo thông tin của FPGA với LiteX.

KC705 là board thuộc dòng Kintex 7 được cung cấp bởi hang Xilinx. Board cung cấp các tính năng phổ biến cho nhiều hệ thống xử lý nhúng, bao gồm bộ nhớ DDR3 SODIMM, giao diện 8-lane PCIe, bộ Ethernet PHY tri-mode, I/O, giao diện UART, Các tính năng khác có thể được thêm vào bằng việc sử dụng thẻ FPGA Mezzanine Cards (FMCs) được cung cấp trên board. Các thành phần của board kc705 được minh họa ở hình 2.1.

Cụ thể các đặc tính của board như sau:

* 1 GB bộ nhớ DDR3 SODIMM
* 128 MB bộ nhớ Linear Byte Peripheral Interface (BPI) flash
* 128 MB bộ nhớ Quad Serial Peripheral Interface (SPI) flash
* Bộ kết nối thẻ Secure Digital (SD)
* Cổng USB, JTAG
* Khối tạo xung
* Bộ truyền nhận GTX
* Bộ kết nối đầu cuối PCIe
* Bộ kết nối SFP+
* Ba mode tốc độ 10/100/1000 Ethernet PHY
* Cổng chuyển đổi USB – UART
* Bộ giao mã HDMI
* Cổng giao tiếp I2C
* LEDs trạng thái: power, ethernet, init, don
* I/O
* Công tắc
* Bộ kết nối VITA 57.1 FMC HPC
* Bộ kết nối VITA 57.1 FMC LPC
* Khối quản lý công suất
* Hỗ trợ XADC
* Ngoài ra còn vài tính năng tùy chọn khác

### Phân tích chức năng

Với mục đích đã nêu ở chương đặt vấn đề, ta sẽ cấu hình SoC với các chức năng như sau:

* Gồm bộ xử lý có thể chạy được Linux, gồm 2 – 4 nhân
* Gồm các giao thức kết nối ngoại vi: UART, SPI, USB, Ethernet.
* Sử dụng các chức năng trên mạch FPGA: RAM, thẻ SD, Ethernet, USB JTAG, GPIO.
* Tối ưu logic sử dụng của FPGA.

### Thực hiện

#### Khai báo các thành phần sử dụng trên board mạch FPGA

Như đã trình bày trong mục tổng quan về LiteX, việc khai báo các thành phần trên mạch FPGA sẽ được viết trong file xilinx\_kc705.py trong thư mục platform.

|  |
| --- |
| …/litex-boards/platform/Xilinx\_kc705.py |
| #  # This file is part of LiteX-Boards.  #  # Copyright (c) 2013-2014 Sebastien Bourdeauducq <sb@m-labs.hk>  # Copyright (c) 2014-2019 Florent Kermarrec <florent@enjoy-digital.fr>  # Copyright (c) 2015 Yann Sionneau <ys@m-labs.hk>  # SPDX-License-Identifier: BSD-2-Clause  from LiteX.build.generic\_platform import \*  from LiteX.build.xilinx import XilinxPlatform  from LiteX.build.openocd import OpenOCD  # IOs -------------------------------------------------------------------------------------  \_io = [  # Clk / Rst  ("clk200", 0,  Subsignal("p", Pins("AD12"), IOStandard("LVDS")),  Subsignal("n", Pins("AD11"), IOStandard("LVDS"))  ),  ("clk156", 0,  Subsignal("p", Pins("K28"), IOStandard("LVDS\_25")),  Subsignal("n", Pins("K29"), IOStandard("LVDS\_25"))  ),  ("cpu\_reset", 0, Pins("AB7"), IOStandard("LVCMOS15")),  # Leds  ("user\_led", 0, Pins("AB8"), IOStandard("LVCMOS15")),  ("user\_led", 1, Pins("AA8"), IOStandard("LVCMOS15")),  ...  # Buttons  ("user\_btn\_c", 0, Pins("G12"), IOStandard("LVCMOS25")),  ("user\_btn\_n", 0, Pins("AA12"), IOStandard("LVCMOS15")),  ...  # Switches  ("user\_dip\_btn", 0, Pins("Y29"), IOStandard("LVCMOS25")),  ("user\_dip\_btn", 1, Pins("W29"), IOStandard("LVCMOS25")),  ...  # I2C  ("i2c", 0,  Subsignal("scl", Pins("K21")),  Subsignal("sda", Pins("L21")),  IOStandard("LVCMOS25")),  # Serial  ("serial", 0,  Subsignal("cts", Pins("L27")),  Subsignal("rts", Pins("K23")),  Subsignal("tx", Pins("K24")),  Subsignal("rx", Pins("M19")),  IOStandard("LVCMOS25")  ),  # DDR3 SDRAM  ("ddram", 0,  Subsignal("a", Pins(  "AH12 AG13 AG12 AF12 AJ12 AJ13 AJ14 AH14",  "AK13 AK14 AF13 AE13 AJ11 AH11 AK10 AK11"),  IOStandard("SSTL15")),  ...  ),  # SPIFlash  ("spiflash", 0,  Subsignal("cs\_n", Pins("U19")),  #Subsignal("clk", Pins("")), # Accessed through STARTUPE2  Subsignal("mosi", Pins("P24")),  Subsignal("miso", Pins("R25")),  Subsignal("wp", Pins("R20")),  Subsignal("hold", Pins("R21")),  IOStandard("LVCMOS25"),  ),  # SDCard  ("spisdcard", 0,  Subsignal("clk", Pins("AB23")),  Subsignal("cs\_n", Pins("AC21")),  Subsignal("mosi", Pins("AB22"), Misc("PULLUP")),  Subsignal("miso", Pins("AC20"), Misc("PULLUP")),  Misc("SLEW=FAST"),  IOStandard("LVCMOS25")  ),  # GMII Ethernet  ("eth\_clocks", 0,  Subsignal("tx", Pins("M28")),  Subsignal("gtx", Pins("K30")),  Subsignal("rx", Pins("U27")),  IOStandard("LVCMOS25")  ),  ("eth", 0,  Subsignal("rst\_n", Pins("L20")),  Subsignal("int\_n", Pins("N30")),  Subsignal("mdio", Pins("J21")),  Subsignal("mdc", Pins("R23")),  Subsignal("rx\_dv", Pins("R28")),  Subsignal("rx\_er", Pins("V26")),  Subsignal("rx\_data", Pins("U30 U25 T25 U28 R19 T27 T26 T28")),  Subsignal("tx\_en", Pins("M27")),  Subsignal("tx\_er", Pins("N29")),  Subsignal("tx\_data", Pins("N27 N25 M29 L28 J26 K26 L30 J28")),  Subsignal("col", Pins("W19")),  Subsignal("crs", Pins("R30")),  IOStandard("LVCMOS25")  ),  # LCD  ("lcd", 0,  Subsignal("db", Pins("AA13 AA10 AA11 Y10")),  Subsignal("e", Pins("AB10")),  Subsignal("rs", Pins("Y11")),  Subsignal("rw", Pins("AB13")),  IOStandard("LVCMOS15")  ),  ...  # PCIe  ("pcie\_x1", 0,  Subsignal("rst\_n", Pins("G25"), IOStandard("LVCMOS25")),  Subsignal("clk\_p", Pins("U8")),  Subsignal("clk\_n", Pins("U7")),  Subsignal("rx\_p", Pins("M6")),  Subsignal("rx\_n", Pins("M5")),  Subsignal("tx\_p", Pins("L4")),  Subsignal("tx\_n", Pins("L3"))  ), ...  )  ...  ]  # Platform --------------------------------------------------------------------------------  class Platform(XilinxPlatform):  default\_clk\_name = "clk156"  default\_clk\_period = 1e9/156.5e6  def \_\_init\_\_(self, toolchain="vivado"):  XilinxPlatform.\_\_init\_\_(self, "xc7k325t-ffg900-2", \_io, \_connectors, toolchain=toolchain)  self.add\_platform\_command("""  set\_property CFGBVS VCCO [current\_design]  set\_property CONFIG\_VOLTAGE 2.5 [current\_design]  """)  self.toolchain.bitstream\_commands = ["set\_property BITSTREAM.CONFIG.SPI\_BUSWIDTH 4 [current\_design]"]  self.toolchain.additional\_commands = ["write\_cfgmem -force -format bin -interface spix4 -size 16 -loadbit \"up 0x0 {build\_name}.bit\" -file {build\_name}.bin"]  def create\_programmer(self):  return OpenOCD("openocd\_xc7\_ft2232.cfg", "bscan\_spi\_xc7a325t.bit")  def do\_finalize(self, fragment):  XilinxPlatform.do\_finalize(self, fragment)  self.add\_period\_constraint(self.lookup\_request("clk200", loose=True), 1e9/200e6)  self.add\_period\_constraint(self.lookup\_request("eth\_clocks:rx", loose=True), 1e9/125e6)  self.add\_period\_constraint(self.lookup\_request("eth\_clocks:tx", loose=True), 1e9/125e6)  self.add\_platform\_command("set\_property DCI\_CASCADE {{32 34}} [get\_iobanks 33]") |

Cấu trúc của file gồm:

* Khai báo các python package hỗ trợ việc phần mềm tổng hợp và tải thiết kế lên FPGA: generic\_platform, xilinx: lựa chọn phần mềm tổng hợp, openOCD: phần mềm mở để tải bitstream lên FPGA.
* Khai báo chân I/O pin sử dụng trên FPGA dựa theo datasheet do hãng sản xuất cung cấp. Các chân I/O có thể không dùng trong thiết kế nhưng vẫn được khai báo cho việc sử dụng lại FPGA với thiết kế khác. Người dùng có thể xác định dễ dàng mục dích của I/O pin như: clock; GPIO cho led, nút bấm, công tắc; các giao thức ngoại vi: I2C, UART; các module chức năng trên board mạch: RAM, SPI flash, thẻ SD, Ethernet, PCIe,...
* Khai báo thông tin FPGA với phần mềm tổng hợp trong lớp Platform: tần số xung clock, tên chip FPGA, điện áp ra, công cụ tổng hợp, ...
* Thêm các rằng buộc tổng hợp, cách tải thiết kế.

#### Cấu hình cho SoC

Cấu trúc SoC được mô tả trong file xilinx\_kc705.py trong thư mục *targets,* bao gồm các phần:

* Các thư viện phần mềm LiteX, các thư viện IP core, ...
* Các lớp mô tả thiết kế SoC, gần giống như trong file viết bằng Verilog.
* Hàm thực thi các lệnh của LiteX với thiết kế.

Với SoC trong đề tài này, mô tả thiết kế như sau:

|  |
| --- |
| …/litex-boards/targets/Xilinx\_kc705.py |
| #!/usr/bin/env python3  #  # This file is part of LiteX-Boards.  #  # Copyright (c) 2014-2015 Sebastien Bourdeauducq <sb@m-labs.hk>  # Copyright (c) 2014-2020 Florent Kermarrec <florent@enjoy-digital.fr>  # Copyright (c) 2014-2015 Yann Sionneau <ys@m-labs.hk>  # SPDX-License-Identifier: BSD-2-Clause  import os  from migen import \*  from LiteX\_boards.platforms import xilinx\_kc705  from LiteX.soc.cores.clock import \*  from LiteX.soc.integration.soc\_core import \*  from LiteX.soc.integration.builder import \*  from LiteX.soc.cores.led import LedChaser  from litedram.modules import MT8JTF12864  from litedram.phy import s7ddrphy  from liteeth.phy import LiteEthPHY  from litepcie.phy.s7pciephy import S7PCIEPHY  from litepcie.software import generate\_litepcie\_software  # CRG ------------------------------------------------------------------------------  class \_CRG(Module):  def \_\_init\_\_(self, platform, sys\_clk\_freq):  self.rst = Signal()  self.clock\_domains.cd\_sys = ClockDomain()  self.clock\_domains.cd\_sys4x = ClockDomain()  self.clock\_domains.cd\_idelay = ClockDomain()  # # #  self.submodules.pll = pll = S7MMCM(speedgrade=-2)  self.comb += pll.reset.eq(platform.request("cpu\_reset") | self.rst)  pll.register\_clkin(platform.request("clk200"), 200e6)  pll.create\_clkout(self.cd\_sys, sys\_clk\_freq)  pll.create\_clkout(self.cd\_sys4x, 4\*sys\_clk\_freq)  pll.create\_clkout(self.cd\_idelay, 200e6)  platform.add\_false\_path\_constraints(self.cd\_sys.clk, pll.clkin) # Ignore sys\_clk to pll.clkin path created by SoC's rst.  self.submodules.idelayctrl = S7IDELAYCTRL(self.cd\_idelay)  # BaseSoC --------------------------------------------------------------------------  class BaseSoC(SoCCore):  def \_\_init\_\_(self, sys\_clk\_freq=int(125e6), with\_ethernet=False, with\_led\_chaser=True, with\_spi\_flash=False, with\_pcie=False, with\_sata=False, \*\*kwargs):  platform = xilinx\_kc705.Platform()  # CRG ----------------------------------------------------------------------  self.submodules.crg = \_CRG(platform, sys\_clk\_freq)  # SoCCore ------------------------------------------------------------------  SoCCore.\_\_init\_\_(self, platform, sys\_clk\_freq, ident="LiteX SoC on KC705", \*\*kwargs)  # DDR3 SDRAM ---------------------------------------------------------------  if not self.integrated\_main\_ram\_size:  self.submodules.ddrphy = s7ddrphy.K7DDRPHY(platform.request("ddram"),  memtype = "DDR3",  nphases = 4,  sys\_clk\_freq = sys\_clk\_freq)  self.add\_sdram("sdram",  phy = self.ddrphy,  module = MT8JTF12864(sys\_clk\_freq, "1:4"),  l2\_cache\_size = kwargs.get("l2\_size", 8192)  )  # Ethernet -----------------------------------------------------------------  if with\_ethernet:  self.submodules.ethphy = LiteEthPHY(  clock\_pads = self.platform.request("eth\_clocks"),  pads = self.platform.request("eth"),  clk\_freq = self.clk\_freq)  self.add\_ethernet(phy=self.ethphy)  # SPI Flash ----------------------------------------------------------------  if with\_spi\_flash:  from litespi.modules import N25Q128A13  from litespi.opcodes import SpiNorFlashOpCodes as Codes  self.add\_spi\_flash(mode="4x", module=N25Q128A13(Codes.READ\_1\_1\_4), rate="1:1", with\_master=True)  # PCIe ---------------------------------------------------------------------  if with\_pcie:  self.submodules.pcie\_phy = S7PCIEPHY(platform, platform.request("pcie\_x4"),  data\_width = 128,  bar0\_size = 0x20000)  self.add\_pcie(phy=self.pcie\_phy, ndmas=1)  # SATA ---------------------------------------------------------------------  if with\_sata:  from LiteX.build.generic\_platform import Subsignal, Pins  from litesata.phy import LiteSATAPHY  # IOs  \_sata\_io = [  # SFP 2 SATA Adapter / https://shop.trenz-electronic.de/en/TE0424-01-SFP-2-SATA-Adapter  ("sfp2sata", 0,  Subsignal("tx\_p", Pins("H2")),  Subsignal("tx\_n", Pins("H1")),  Subsignal("rx\_p", Pins("G4")),  Subsignal("rx\_n", Pins("G3")),  ),  ]  platform.add\_extension(\_sata\_io)  # RefClk, Generate 150MHz from PLL.  self.clock\_domains.cd\_sata\_refclk = ClockDomain()  self.crg.pll.create\_clkout(self.cd\_sata\_refclk, 150e6)  sata\_refclk = ClockSignal("sata\_refclk")  platform.add\_platform\_command("set\_property SEVERITY {{Warning}} [get\_drc\_checks REQP-52]")  # PHY  self.submodules.sata\_phy = LiteSATAPHY(platform.device,  refclk = sata\_refclk,  pads = platform.request("sfp2sata"),  gen = "gen2",  clk\_freq = sys\_clk\_freq,  data\_width = 16)  # Core  self.add\_sata(phy=self.sata\_phy, mode="read+write")  # Leds ---------------------------------------------------------------------  if with\_led\_chaser:  self.submodules.leds = LedChaser(  pads = platform.request\_all("user\_led"),  sys\_clk\_freq = sys\_clk\_freq)  # Build ----------------------------------------------------------------------------  def main():  from LiteX.soc.integration.soc import LiteXSoCArgumentParser  parser = LiteXSoCArgumentParser(description="LiteX SoC on KC705")  target\_group = parser.add\_argument\_group(title="Target options")  target\_group.add\_argument("--build", action="store\_true", help="Build design.")  target\_group.add\_argument("--load", action="store\_true", help="Load bitstream.")  target\_group.add\_argument("--sys-clk-freq", default=125e6, help="System clock frequency.")  target\_group.add\_argument("--with-ethernet", action="store\_true", help="Enable Ethernet support.")  target\_group.add\_argument("--with-spi-flash", action="store\_true", help="Enable SPI Flash (MMAPed).")  target\_group.add\_argument("--with-pcie", action="store\_true", help="Enable PCIe support.")  target\_group.add\_argument("--driver", action="store\_true", help="Generate PCIe driver.")  target\_group.add\_argument("--with-sata", action="store\_true", help="Enable SATA support (over SFP2SATA).")  builder\_args(parser)  soc\_core\_args(parser)  args = parser.parse\_args()  soc = BaseSoC(  sys\_clk\_freq = int(float(args.sys\_clk\_freq)),  with\_ethernet = args.with\_ethernet,  with\_spi\_flash = args.with\_spi\_flash,  with\_pcie = args.with\_pcie,  with\_sata = args.with\_sata,  \*\*soc\_core\_argdict(args)  )  builder = Builder(soc, \*\*builder\_argdict(args))  if args.build:  builder.build()  if args.driver:  generate\_litepcie\_software(soc, os.path.join(builder.output\_dir, "driver"))  if args.load:  prog = soc.platform.create\_programmer()  prog.load\_bitstream(builder.get\_bitstream\_filename(mode="sram"))  if \_\_name\_\_ == "\_\_main\_\_":  main() |

Thư viện LiteX bao gồm:

* from migen import \*: cài đặt chương trình theo cấu trúc của migen
* from LiteX\_boards.platforms import xilinx\_kc705: from migen import \*: tải file khai báo thông tin FPGA
* from LiteX.soc.... chứa các thành phần chỉnh của SoC: clock tạo các miền xung đồng hồ trong soc, soc\_core chứa các bộ xử lý được hỗ trợ, builder chứa phương thức tổng hợp thiết kế
* các lệnh còn lại: tổng các thư viện LiteX core sử dụng theo thông tin trên board như RAM, Ethernet, PCIe(nếu dùng)

Mô tả thiết kế SoC

* Lớp \_CRG mô tả các miền xung clock sử dùng trong SoC: cd\_sys là miền clock chỉnh của hệ thống có tần số 125MHz; cd\_sys4x đc tạo từ clock chỉnh, có tần số cao gấp 4 lần clock chính, dùng để lấy mẫu các tín hiện truyền dẫn,...; cd\_idelay dùng cho core liteDRAM
* Lớp BaseSoC mô tả cấu trúc của SoC. Ở đây, lớp này chỉ khai báo thêm các IP core sử dụng bởi đây là lớp kế thừa từ lớp SoCCore. Lớp SoCCore sẽ chứa khai báo về bộ xử lý, cấu trúc bus, các thành phần chỉnh do LiteX cài đặt sẵn.

Hàm thực thi

* Phần đầu mô tả các tham số khi chạy file xilinx\_kc705 bằng Python: ta có thể thêm các tham số như sau để thêm/bớt module trong thiết kế.
* soc = BaseSoC(...) tạo một thực thể của lớp BaseSoC đã định nghĩa. builder = Builder(soc, \*\*builder\_argdict(args)) đây là chương trình sẽ dịch thiết kế ( thực thể soc) từ Migen Python thành file viết bằng Verilog
* Các tham số build, driver, load sẽ lần lượt yêu cầu LiteX thực hiện việc tổng hợp thiết kế, xây dụng driver (nếu FPGA kết nối với máy tỉnh chủ bằng PCIe) và tải bitstream (sau khi tổng hợp) lên FPGA.

## Tùy chỉnh phần mềm chạy trên BIOS

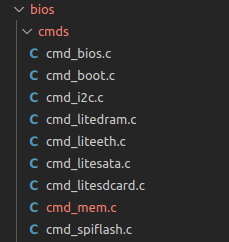
Thư mục software trong LiteX là nơi chứa định nghĩa cho các chương trình baremetal có thể chạy trên BIOS mà không cần hệ điều hành. Để có thể tương tác với phần cứng của kiến trúc được triển khai trên FPGA, ta có thể viết chương trình BIOS, đọc ghi trực tiếp vào tập thanh ghi CSR của SoC.

Cấu trúc phần mềm trong LiteX sẽ bao gồm:

* Các thư viện cần thiết: libbase, libc, liblitesdcard,…
* Nhóm chương trình boot-rom
* Makefile để build chương trình cùng với kiến trúc phần cứng

Ta sẽ tùy chỉnh các chương trình boot-room ở thư mục bios và sửa file Makefile nếu cần thiết. Ngoài các thư viện được cung cấp, LiteX cũng đã định nghĩa sẵn một số header hữu ích cho các chương trình BIOS. Trong đó bao gồm các hàm về khởi tạo lệnh, xây dựng lệnh và hiển thị lên màn hình bios. Ta chủ yếu sẽ tùy chỉnh các lệnh này ở thư mục cmd.

Tùy vào chức năng mà nhóm lệnh được chia thành 10 nhóm và định nghĩa ở từng file riêng biệt:



Hình 3.2 Nhóm lệnh trên BIOS

Đề tài này sẽ chỉ tùy chỉnh về các lệnh đọc ghi vào bộ nhớ trong file cmd\_mem.c. Chương trình được xây dựng là một lệnh tạo tín hiệu điều khiển vào một khối chức năng trong phần cứng. Chương trình sẽ cần truyền 3 tham biến là 3 con trỏ trỏ vào địa chỉ bộ nhớ là các thanh ghi CSR có khả năng thực hiện chức năng phần cứng. Trong đó một tham biến đóng vai trò là đầu vào của khối phần cứng và 2 tham biến còn lại đóng vai trò đầu ra của khối phần cứng được chương trình đọc lại để phân tích và hiển thị dữ liệu.

|  |
| --- |
| /**\*\***  **\* Command "packet2snn"**  **\* Transfer packet to SNN**  **\*/** |
| static void packet\_handler(int nb\_params, char \*\*params)  {  char \*c;  unsigned int \*addr\_state;  unsigned int \*addr\_complete;  void \*addr\_start;  if (nb\_params < 3) {  // >0< >1< >2<  printf("packet2snn <address\_start> <address\_state> <address\_complete> ");  return;  }  printf("Start \n");  //Transfer  //start  addr\_start = (void \*)strtoul(params[0], &c, 0);  if (\*c != 0) {  printf("Incorrect address");  return;  }  //state  addr\_state = (unsigned int \*)strtoul(params[1], &c, 0);  if (\*c != 0) {  printf("Incorrect address");  return;  }  //complete  addr\_complete = (unsigned int \*)strtoul(params[2], &c, 0);  if (\*c != 0) {  printf("Incorrect address");  return;  }  \*(uint32\_t \*)addr\_start = 1;  printf("Process start! ...\n");  if(\*addr\_state != 0) \*(uint32\_t \*)addr\_start = 0;  while (\*addr\_complete != 1)  {  switch (\*addr\_state)  {  case 1:  printf("\nState: LOAD\n");  break;  case 2:  printf("\nState: COMPUTE\n");  break;  case 4:  printf("\nState: WAIT\_END\n");  break;  default:  printf("\nState: IDLE\n");  break;  }  }  printf("End process!");  }  define\_command(packet2snn, packet\_handler, "Transfer packet to SNN", MEM\_CMDS); |

* Chương trình sẽ yêu cầu người dùng nhập đúng địa chỉ và số lượng tham biến
* Khi nhập nhỏ hơn số tham biến yêu cầu sẽ đưa ra gợi ý về cấu trúc của lệnh
* strlout là một hàm trả về về giá trị của tham biến truyền vào
* Sau các bước ban đầu trên ta sẽ định nghĩa cho lệnh sao cho phù hợp với yêu cầu của mỗi chương trình. Ở đây chương trình sẽ thực hiện đọc giá trị trạng thái của khối phần cứng từ 2 thanh ghi và thực hiện hiển thị hoặc kết thúc chương trình.
* Cuối cùng, defind\_command là một hàm được cung cấp để ta thêm 1 lệnh tùy chỉnh vào nhóm lệnh chỉ định. Cụ thể như với lệnh trên:
* Tên lệnh thực thi sẽ là packet2snn
* Nội dụng thực thi được lấy từ hàm packet\_handler do ta vừa định nghĩa
* Mô tả cho lệnh: "Transfer packet to SNN"
* Đưa lệnh này vào nhóm lệnh đọc ghi bộ nhớ MEM\_CMDS

## Boot Linux

### Chuẩn bị công cụ

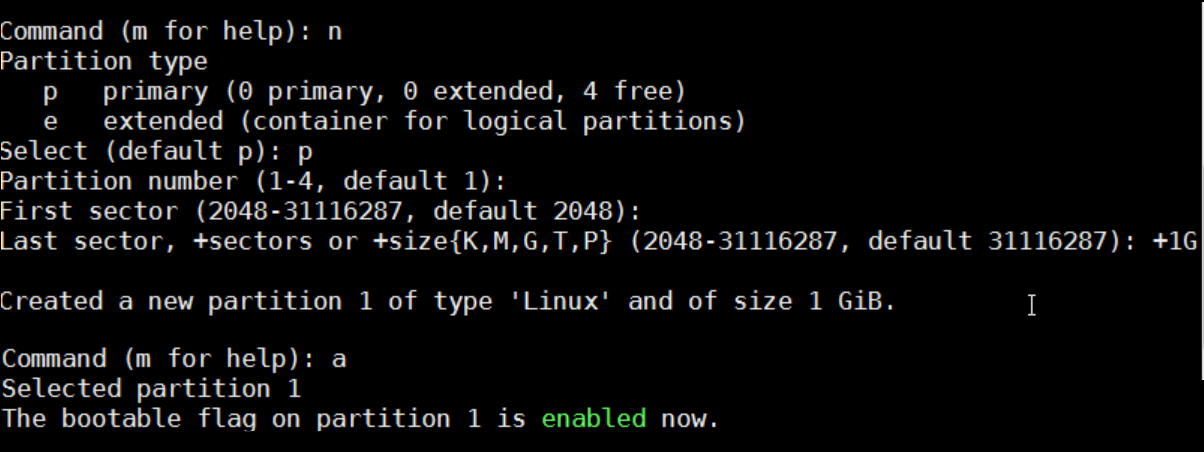
1. Tùy chỉnh phương pháp boot

Các thành phần của phiên bản Linux cần được chuẩn bị trước. Có thể sử dụng các bản image có sẵn hoặc tự tạo bằng buildroot. Sau đó ta sẽ hướng dẫn LiteX đường dẫn đến tập image này để bootloader nạp vào RAM. Đề tài này nhắm đến việc hoạt động độc lập nên sẽ sử dụng phương thức boot thông qua thẻ SD (sdcard boot). Thẻ SD đươc điều chỉnh để có phân vùng lưu trữ linux image là bootable. Đầu tiên ta sẽ gắn thẻ SD vào máy tính đang sử dụng và xác định tên device của thẻ SD, thông thường sẽ có dạng mmc... Trong trường hợp ở đồ án này, tên device của thẻ SD là mmcblk0. Sau đó trên terminal, thực hiên các lệnh sau:

|  |
| --- |
| $ sudo fdisk /dev/mmcblk0 |

Terminal sau đó sẽ chuyển sang giao diện format disk. Nếu thẻ có sẵn phân vùng nào đó, có thể xóa phân vùng đó bằng cách nhập “d”. Ta sẽ tiến hành tạo 1 phân vùng bootalbe:

* Nhập n để tạo 1 phân vùng mới
* Chọn p để là primary partition
* Chọn “partition number” và “first sector” là mặc định (Nhấn Enter)
* “Last sector”: chọn dung lượng cho phân vùng. Ví dú muốn 1Gb nhập “+1G”
* Nhập a để đặt phân vùng là bootable
* Cuối cùng nhập w để lưu lại.



Hình 3.3 Format phân vùng bootable cho thẻ SD

1. Thiết lập device driver cho board

Board FPGA được kết nối với máy tính chủ thông qua dây JTAG và UART. Kết nối thông qua JTAG được dùng để nạp bitstream (kiến trúc SoC đã xây dựng) lên FPGA, còn UART là cổng nạp chương trình bootrom vào bộ nhớ ROM của SoC. Hai kết nối này được gioa tiếp với máy chủ thông qua 2 dây chuyển đổi cổng USB và sẽ được máy chủ nhận biết với tên device là ttyUSBx (trong đó x có thể là 1 số 1,2,3,…). Ta cần cấp quyền execute cho cái driver này bằng lệnh ”chmod”:

|  |
| --- |
| $ chmod 777 /dev/ttyUSBx |

Để thuận tiện cho quá trình nhận diện thiết bị, ta có thể cài đặt Udev rules. Trong thư mục linux-on-LiteX-vexriscv, thực hiện:

|  |
| --- |
| $ git clone https://github.com/LiteX-hub/LiteX-buildenv-udev  $ cd LiteX-buildenv-udev  $ make install  $ make reload  $ sudo cp <path to>/LiteX/openocd/contrib/60-openocd.rules /etc/udev/rules.d  $ udevadm control –reload-rules |

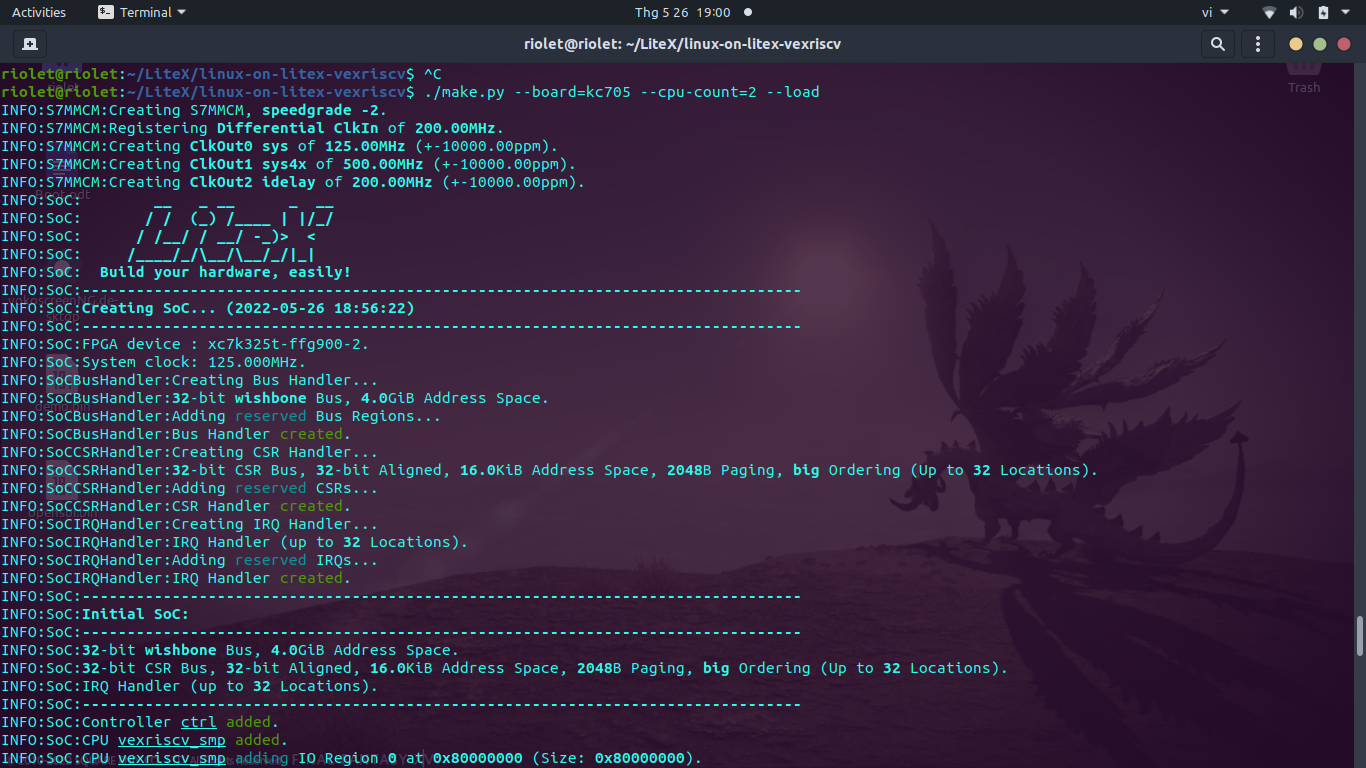
### Boot linux chạy trên SoC lên FPGA

Các công việc chuẩn bị đã hoàn thành, ta sẽ bắt đầu thực nghiệm trên board. Kiến trúc SoC trước tiên cần được nạp lên board. Trên terminal tại thư mục linux-on-LiteX-vexriscv, thưc hiện lệnh sau đây:

|  |
| --- |
| $ ./make.py –cpu-count=2 –board=kc705 --load |

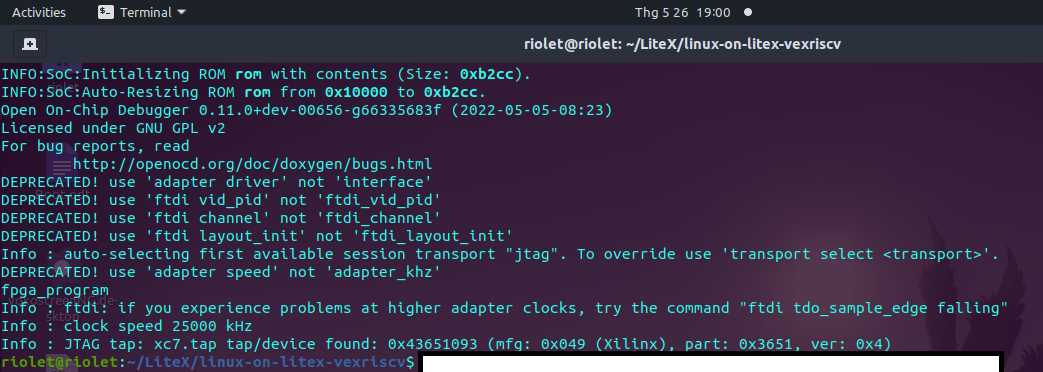
Trong đó:

* **cpu-count** là option khi build kiến trúc.
* **board** là option để chọn board FPGA nạp lên (cũng là option cần chỉ định khi build kiến trúc).
* **load** là option dùng để nạp bitstream lên board.



Hình 3.4 Nạp bitstream lên board KC705

Sau khi đã nạp thành công SoC lên FPGA, ta có thể thấy dãy led của board KC705 chạy vòng lặp phát sáng lần lượt từ trái sang phải. Lúc này phần cứng đã sẵn sàng để chạy các chương trình được đưa lên. Hoặc trên terminal sẽ không hiện báo lỗi mà hiển thị thông báo như sau:



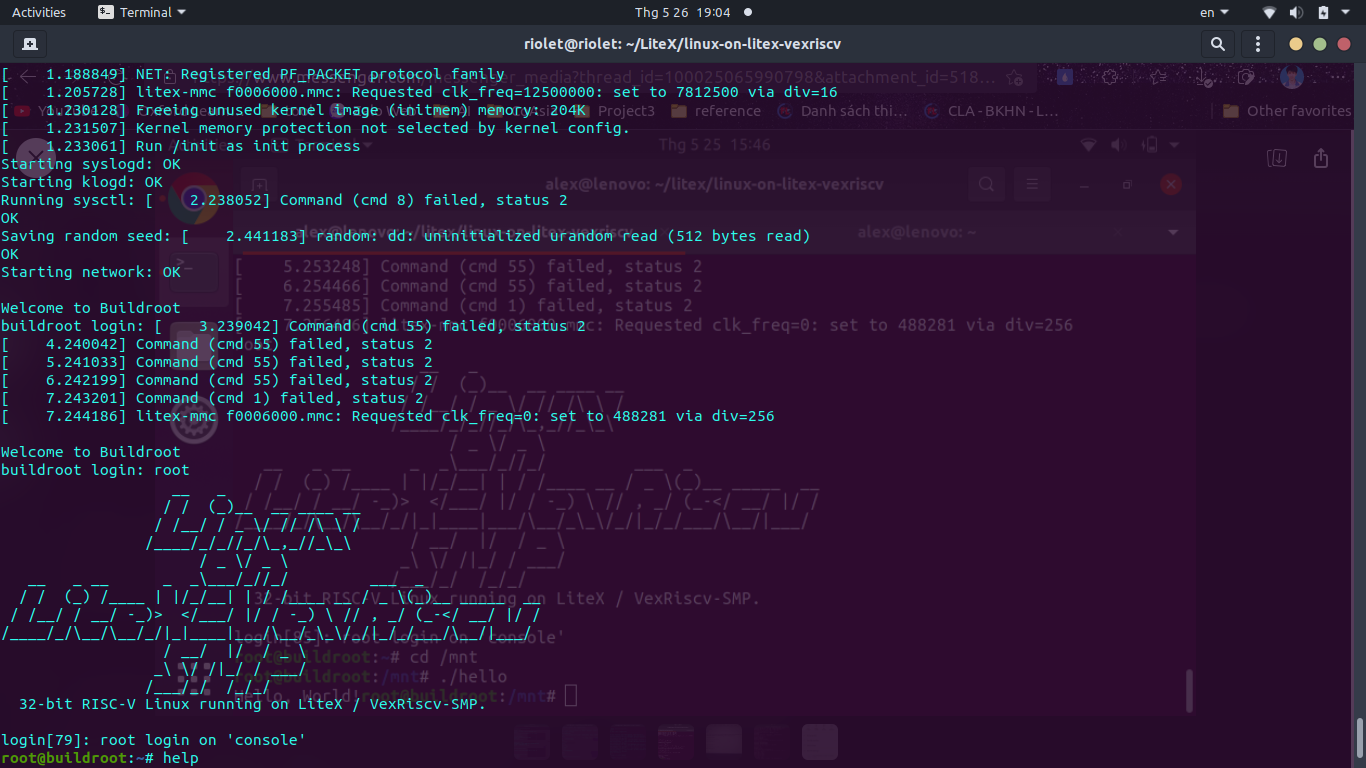
Hình 3.5 Nạp thành công SoC lên board thông qua cổng JTAG

Tiếp đến, để tiến hành boot linux, ta sẽ sử dụng công cụ “LiteX\_term” mà mã nguồn linux-on-LiteX-vexriscv cung cấp.

|  |
| --- |
| $ LiteX\_term /dev/ttyUSBx |

Chú ý rằng, ttyUSBx là tên device driver của giao tiếp UART giữa máy chủ với board. Ở đây ta không chỉ đường dẫn đến tập file image vì sử dụng phương thức boot sdcard. Nếu sử dụng phương thức serialboot thông qua cổng UART, lệnh LiteX\_term cần được thêm option “image=boot.json”.

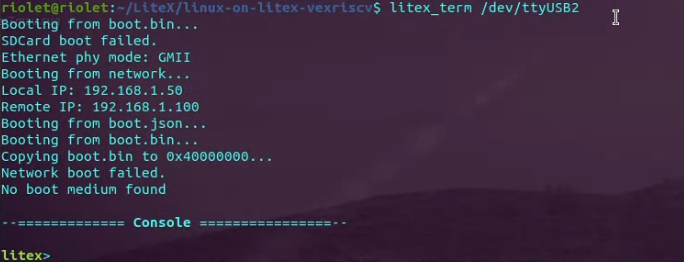
Sau khi chạy lệnh, tiến trình boot được thực hiện, LiteX sẽ vào boot sequence. Nếu không tìm thấy phương pháp boot nào thì giao diện người dùng sẽ được chuyển sang BIOS. Thẻ SD đã được tùy chỉnh cần được cắm vào board để LiteX xác nhận phương pháp boot là sdcardboot. Một khi bootloader sẵn sàng cho hệ thống nó sẽ đọc file boot.json và boot các chương trình chứa trong thẻ SD vào vị trí bộ nhớ thích hợp. Quá trình boot thành công sẽ yêu cầu người dùng đăng nhập. Lúc này nhập root rồi nhấn enter, terminal linux được chạy trên SoC sẽ hiện lên.



Hình 3.6 Linux chạy trên kiến trúc LiteX Vexriscv

### Chạy chương trình trên BIOS

Tương tự với boot linux, tuy nhiên khi chạy lệnh LiteX\_term, ta sẽ rút thẻ SD ra khỏi board. Lúc này LiteX không tìm được phương pháp boot nào vào sẽ tự động vào BIOS.



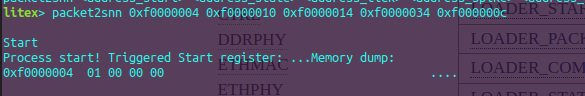
Hình 3.7 Terminal BIOS

Nhập “help” lên bios terminal rồi nhấn Enter để xem các lệnh có thể được thực thi ở bios. Một loạt các lệnh hiện ra sẽ chia cách nhau bởi từng nhóm lệnh đã được định nghĩa từ trước. Để có thể thực thi các lệnh này, nhập và chạy lệnh trên terminal sẽ đưa ra các gợi ý về tham số cần truyền vào (nếu có).



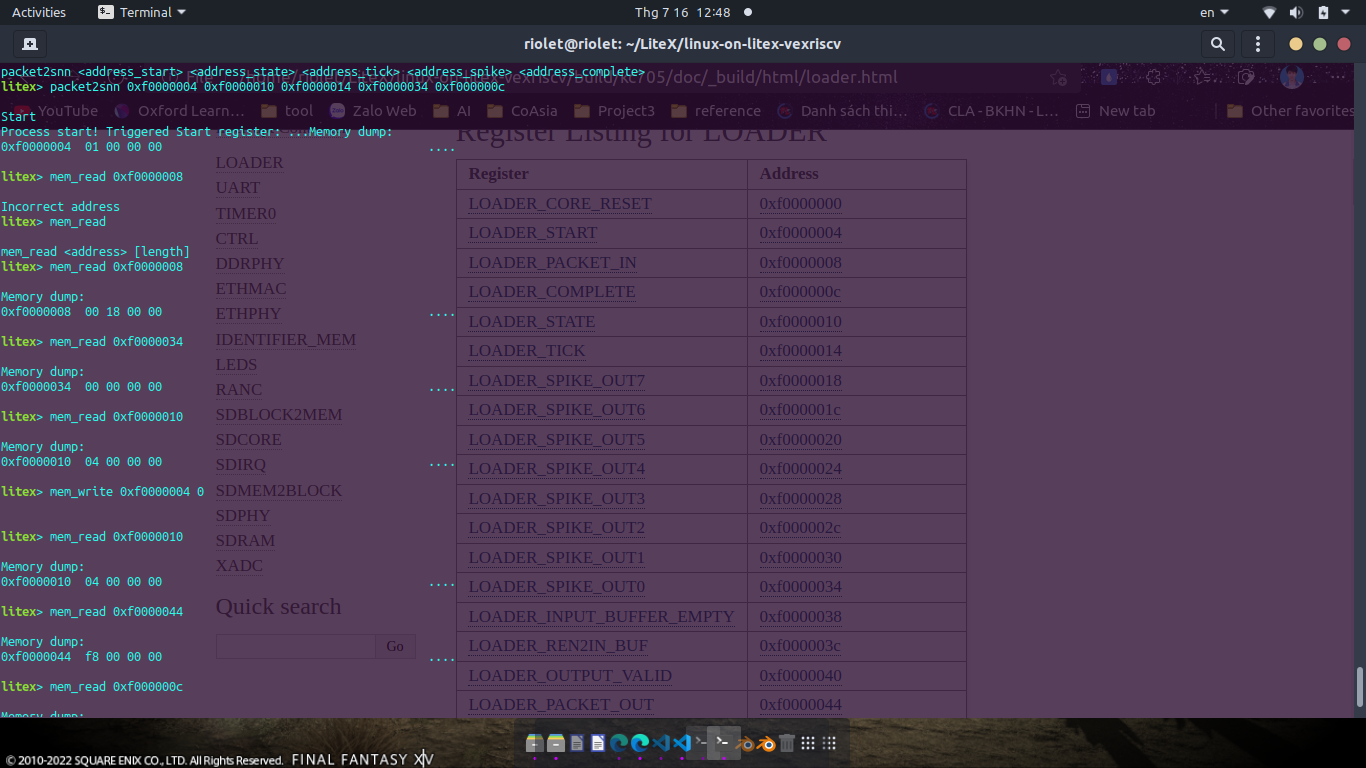
Hình 3.8 Gợi ý lệnh trên BIOS

Thực thi lệnh “packet2snn” đã được định nghĩa từ mục 3.2 . Tham số truyền vào sẽ là địa chỉ các thanh ghi CSR: start, state và complete.



Hình 3.9 Thực thi lệnh packet2snn trên BIOS

Kết quả thực thi lệnh có thể được kiểm tra bằng việc kiểm tra các tín hiệu đầu ra của khối chức năng phần cứng. Các tín hiệu này đã được gán để ghi vào thanh ghi CSR vì vậy ta có thể sử dụng lệnh “mem\_read” được cung cấp sẵn. Dữ liệu để so sánh là kết quả mô phỏng RTL của khối chức năng mà đồ án này sẽ không đề cập chi tiết.



Hình 3.10 Kiểm tra kết quả

Quá trình so sánh cho thấy rằng lệnh trên BIOS đã được thực thi và kết quả thực nghiệm trên board đã trùng khớp với kết quả mô phỏng.

# KẾT LUẬN

Dựa trên dự án Linux-on-litex-vexriscv, đồ án đã hoàn thành việc xây dựng một kiến trúc SoC và chạy thử nghiệm trên board FPGA KC705. Đồ án cũng đã mô tả cách điều chỉnh để có thể chạy thành công linux trên nền tảng SoC của LiteX. Ngoài ra, với mục đích xây dựng một hệ thống hoạt động độc lập trong tương lai mà đồ án có nói về cách sử dụng thẻ SD làm công cụ lưu trữ dữ liệu.

Việc phát triển và tùy chỉnh SoC có thể được thực hiện bằng Migen. Migen cho phép thêm các module viết bằng ngôn ngữ HDL hoặc một module mô tả bằng Python sử dụng thư viện migen. Sử dụng ngôn ngữ bậc cao như Python sẽ khiến cho việc tích hơp các module phần cứng đơn giản và dễ hiểu hơn. Để thử nghiệm, đồ án đã xây dựng một module viết bằng Verilog và tích hợp vào SoC. Module được giao tiếp với interface Wishbone bằng các thanh ghi CSR và các thanh ghi này sẽ được dùng để debug và xây dựng chương trình device driver trong tương lai. Kết quả so sánh cho thấy hệ thống hoạt động thực tế đã chính xác so với mô phỏng.

Đồ án ngoài ra cũng đã xây dựng cách thức để tùy chỉnh các chương trình trên BIOS giúp cho việc debug phần cứng gian đoạn đầu đơn giản hơn. Từ đó làm nền tảng để phát triển thành các chương trình nhúng chạy trên linux.

# TÀI LIỆU THAM KHẢO

1. Tuning the BlackParrot RISC-V Processor for the LiteX SoC Builder, Martin Troiber, 15/10/2021.
2. LiteX: an open-source SoC builder and library based on Migen Python DSL, Florent Kermarrec, Sebastien Bourdeauducq, Sebastien Bourdeauducq.
3. “Litex.” <https://github.com/enjoy-digital/litex>.
4. “Linux-on-litex-vexriscv”. [litex-hub/linux-on-litex-vexriscv: Linux on LiteX-VexRiscv (github.com)](https://github.com/litex-hub/linux-on-litex-vexriscv)
5. “Migen”. [Migen | M-Labs](https://m-labs.hk/gateware/migen/)