**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**

**KHOA KỸ THUẬT MÁY TÍNH**



**TÀI LIỆU HƯỚNG DẪN**

**Thiết kế nhận diện số viết tay hiện thực trên FPGA dùng Vitis High-Level Synthesis (HLS)**

**GVHD:** TS. Phạm Hoài Luân

**Sinh viên thực hiện:**

Nguyễn Minh Tuấn

Cao Đăng Tùng

🙡🙢 Tp. Hồ Chí Minh, 10/2025 🙠🙣

**MỤC LỤC**

[I. Train model 6](#_Toc212499875)

[1. Chuẩn bị code 6](#_Toc212499876)

[2. Dùng Google Colab 6](#_Toc212499877)

[II. Py2C 8](#_Toc212499878)

[1. Công cụ chuyển Python sang C 8](#_Toc212499879)

[2. Kết quả sau khi chạy Py2C 8](#_Toc212499880)

[3. Chuẩn bị file X.txt và Y.txt 8](#_Toc212499881)

[III. Bọc AXI: 9](#_Toc212499882)

[IV. C-to-RTL với Vitis HLS 2022.2: 10](#_Toc212499883)

[1. Tạo project 10](#_Toc212499884)

[2. Chạy C simulation 15](#_Toc212499885)

[3. C SYNTHESIS: 17](#_Toc212499886)

[4. C/RTL COSIMULATION: 18](#_Toc212499887)

[V. Tích hợp IP vào hệ thống SoC 21](#_Toc212499888)

[1. Tải Vivado 2022.2 21](#_Toc212499889)

[2. Tạo project 21](#_Toc212499890)

[3. Tạo hệ thống SoC 23](#_Toc212499891)

[VI. Code C điều khiển 35](#_Toc212499892)

[4. Chuẩn bị 35](#_Toc212499893)

[5. Viết code C điều khiển 36](#_Toc212499894)

[VII. Nạp Kit 40](#_Toc212499895)

**DANH MỤC HÌNH ẢNH**

[Hình 1. Chuyển sang GPU 7](#_Toc212499967)

[Hình 2. Chạy code Python 7](#_Toc212499968)

[Hình 3. Kết quả khi chạy xong 8](#_Toc212499969)

[Hình 4. Code xóa Dropout 8](#_Toc212499970)

[Hình 5. Tải file .h5 8](#_Toc212499971)

[Hình 6. File main.py 9](#_Toc212499972)

[Hình 7. Các file được tạo ra sau khi chạy xong main.py 9](#_Toc212499973)

[Hình 8. Tạo file X.txt và Y.txt cho file CNN\_tb.cpp 10](#_Toc212499974)

[Hình 9. File CNN.cpp sau khi thêm #pragma 10](#_Toc212499975)

[Hình 10. Tạo project 11](#_Toc212499976)

[Hình 11. Đặt tên project 12](#_Toc212499977)

[Hình 12. Add design files 12](#_Toc212499978)

[Hình 13. Chọn top function 13](#_Toc212499979)

[Hình 14. Chọn file testbench 13](#_Toc212499980)

[Hình 15. Chọn mã kit 14](#_Toc212499981)

[Hình 16. Nhập mã kit 14](#_Toc212499982)

[Hình 17. Finish 15](#_Toc212499983)

[Hình 18. Các file sau khi tạo thành công project 15](#_Toc212499984)

[Hình 19. Chọn số ảnh để test 16](#_Toc212499985)

[Hình 20. Run C simulation 17](#_Toc212499986)

[Hình 21. Chọn C simulation Dialog 17](#_Toc212499987)

[Hình 22. Kết quả sau khi chạy C simulation 17](#_Toc212499988)

[Hình 23. Run C Synthesis 18](#_Toc212499989)

[Hình 24. Set Clock 18](#_Toc212499990)

[Hình 25. Xem báo cáo tài nguyên 19](#_Toc212499991)

[Hình 26. Cấu hình của bọc AXI 19](#_Toc212499992)

[Hình 27. Run Cosimulation 20](#_Toc212499993)

[Hình 28. Cấu hình để xem waveform 20](#_Toc212499994)

[Hình 29. Giao diện chạy simulation Vivado 21](#_Toc212499995)

[Hình 30. Waveform 21](#_Toc212499996)

[Hình 31. Đọc dữ liệu đầu ra 22](#_Toc212499997)

[Hình 32. Hoàn thành chạy Cosimulation 22](#_Toc212499998)

[Hình 33. Export RTL 22](#_Toc212499999)

[Hình 34. Tạo tên, thư mục project 23](#_Toc212500000)

[Hình 35. Chọn RTL project 23](#_Toc212500001)

[Hình 36. Chọn board KV260 24](#_Toc212500002)

[Hình 37. Mở IP Repository 25](#_Toc212500003)

[Hình 38. Chọn đường dẫn CNN IP 25](#_Toc212500004)

[Hình 39. Tạo Block Design 26](#_Toc212500005)

[Hình 40. Thêm IP 26](#_Toc212500006)

[Hình 41. Thêm Zynq UltraScale+ MPSoC 27](#_Toc212500007)

[Hình 42. Thêm CNN IP 27](#_Toc212500008)

[Hình 43. Chọn Memory Type 28](#_Toc212500009)

[Hình 44. Đổi tên BRAM 28](#_Toc212500010)

[Hình 45. Sau khi thêm 2 BRAM 28](#_Toc212500011)

[Hình 46. Nối BRAM CTRL với BRAM 29](#_Toc212500012)

[Hình 47. Run Connection Automation 29](#_Toc212500013)

[Hình 48. Setting Run Connection Automation 30](#_Toc212500014)

[Hình 49. SoC hoàn chỉnh 30](#_Toc212500015)

[Hình 50. Lỗi khi chỉnh Range BRAM 31](#_Toc212500016)

[Hình 51. Sau khi chỉnh Range và Base Address hoàn tất 31](#_Toc212500017)

[Hình 52. Valid Design 31](#_Toc212500018)

[Hình 53. Generate Block Design 32](#_Toc212500019)

[Hình 54. Tạo HDL Wrapper 32](#_Toc212500020)

[Hình 55. Setting HDL Wrapper 33](#_Toc212500021)

[Hình 56. Sau khi tạo xong Wrapper 33](#_Toc212500022)

[Hình 57. Run Synthesis 33](#_Toc212500023)

[Hình 58. Run Implementation 34](#_Toc212500024)

[Hình 59. Tạo file Bitstream 34](#_Toc212500025)

[Hình 60. Tạo file .xsa 35](#_Toc212500026)

[Hình 61. Include bitstrean cho file .xsa 35](#_Toc212500027)

[Hình 62. Tạo file .bin 36](#_Toc212500028)

[Hình 63. Địa chỉ trong Fpga\_Driver.c 36](#_Toc212500029)

[Hình 64. Include các thư viện 37](#_Toc212500030)

[Hình 65. Các địa chỉ offset 38](#_Toc212500031)

[Hình 66. Lưu đồ code C điều khiển 38](#_Toc212500032)

[Hình 67. Hàm setup\_cnn\_pointer 39](#_Toc212500033)

[Hình 68. Hàm load\_weights\_to\_bram 39](#_Toc212500034)

[Hình 69. Hàm load\_image\_to\_bram 39](#_Toc212500035)

[Hình 70. Hàm start\_cnn 39](#_Toc212500036)

[Hình 71. Hàm wait\_cnn\_done 40](#_Toc212500037)

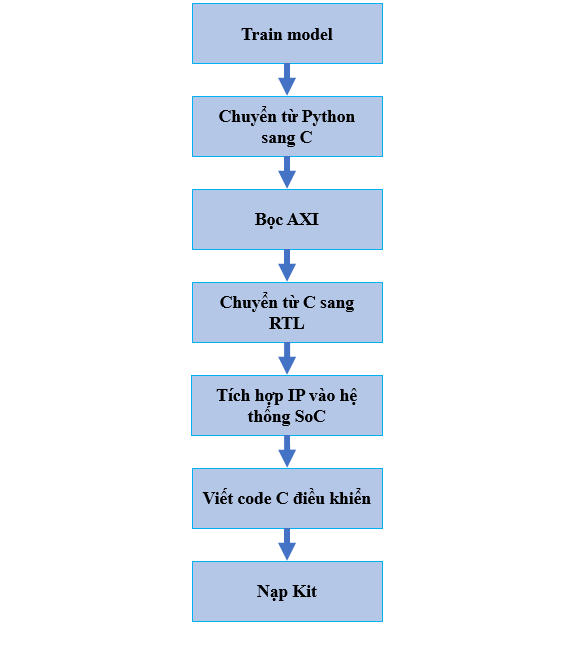
[Hình 72. Hàm read\_cnn\_output 40](#_Toc212500038)

[Hình 73. Hàm run\_inference 40](#_Toc212500039)

[Hình 74. Khởi tạo và thiết lập giao tiếp phần cứng 40](#_Toc212500040)

[Hình 75. Inference loop 41](#_Toc212500041)

**Quy trình thực hiện:**

****

**Link Github:** <https://github.com/TuanSuyTu/Lenet-5-HLS>

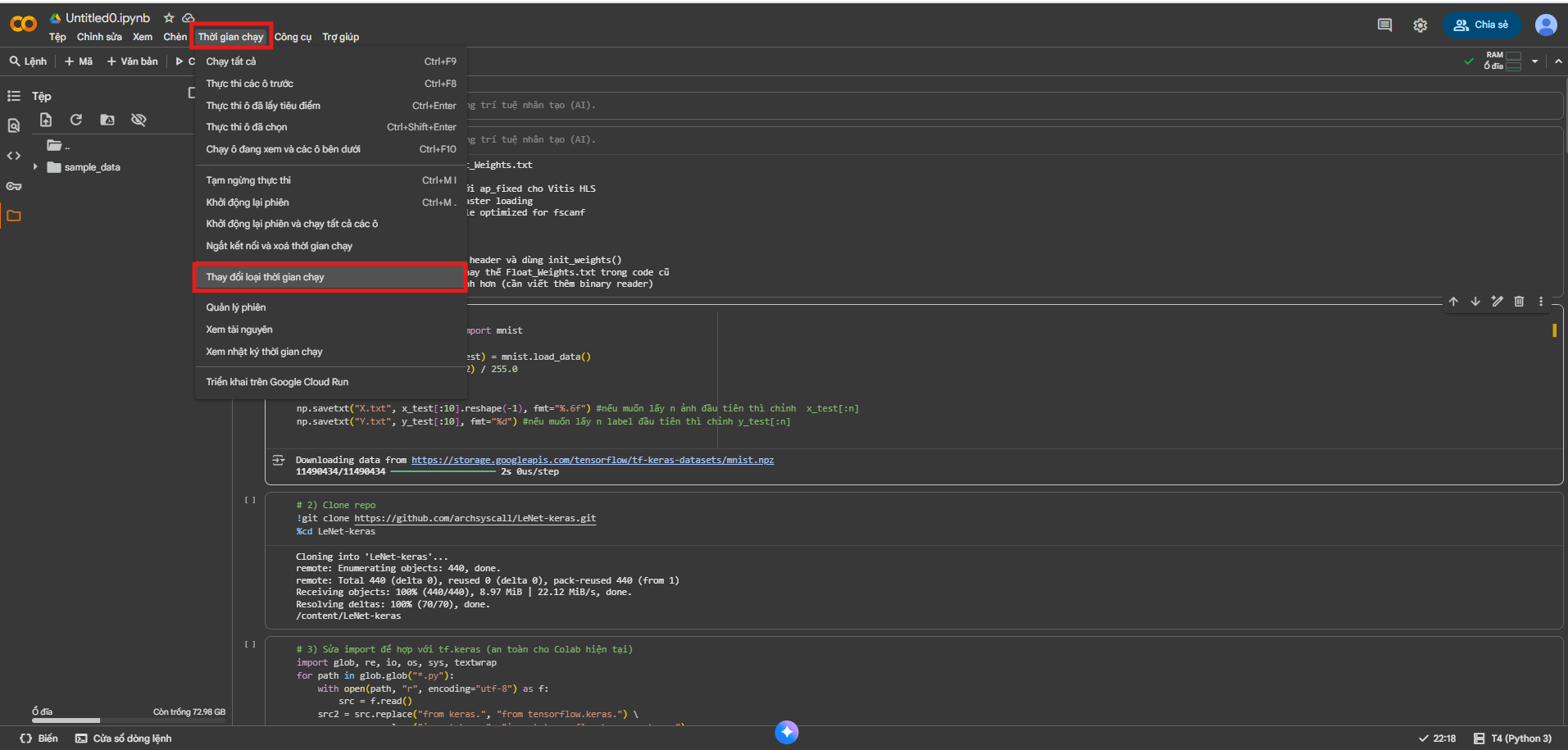
## Train model

### Chuẩn bị code

* Python (Keras/TensorFlow) để train mô hình (file lenet.py ở link Github)

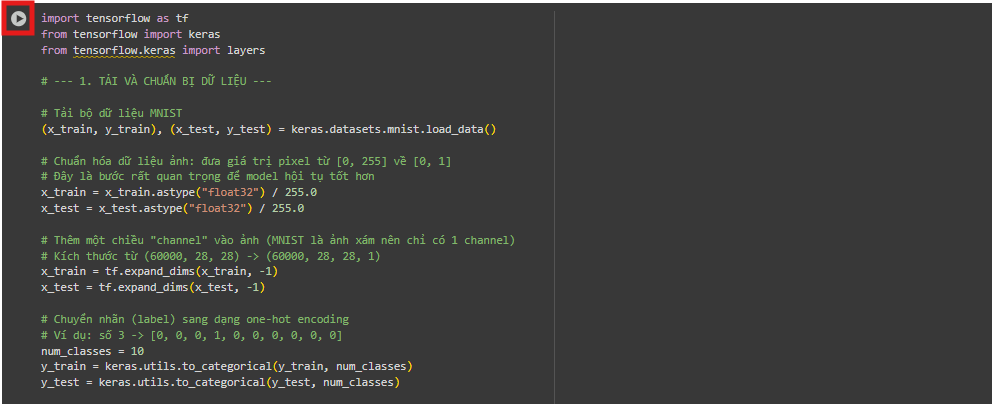
### Dùng Google Colab

* Chuyển sang GPU để train nhanh hơn, vào như trong hình để đổi sang GPU



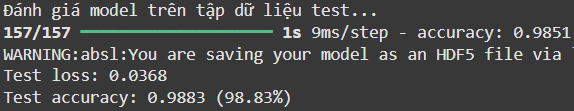
Hình 1. Chuyển sang GPU

* Chạy code Python để train model



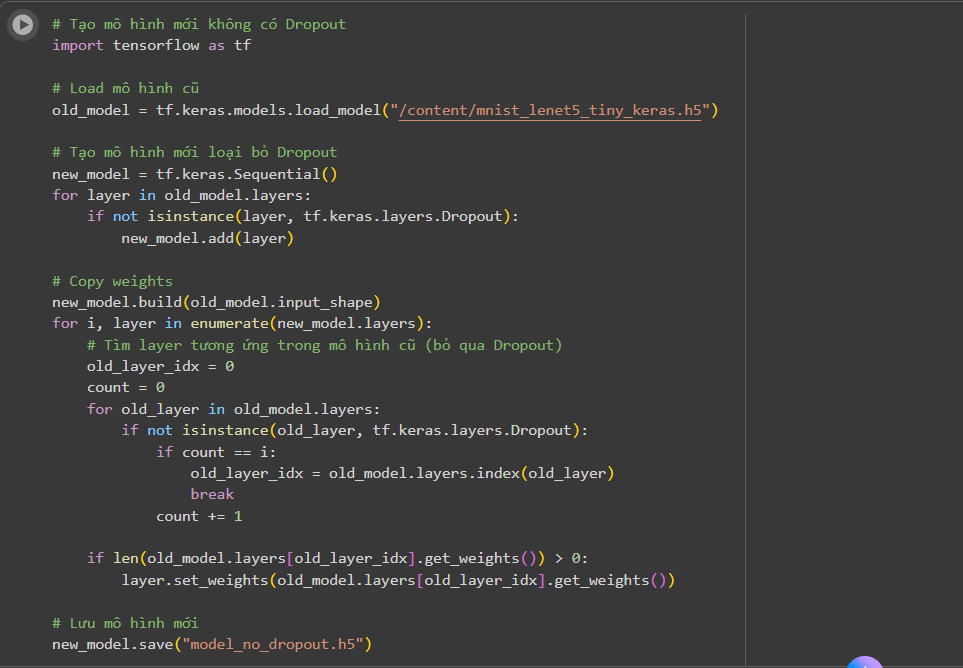
Hình 2. Chạy code Python

* Sau khi chạy xong



Hình 3. Kết quả khi chạy xong

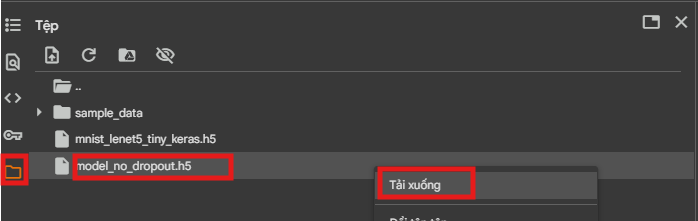
* **Lưu ý:** Nếu có sử dụng Dropout, chạy file remove\_dropout.py để tạo file .h5 không Dropout



Hình 4. Code xóa Dropout

* Tải file .h5 về

Vào biểu tượng Folder ở thanh bên trái nếu không thấy hãy nhấn refresh sau đó tải file model\_no\_dropout.h5 về máy

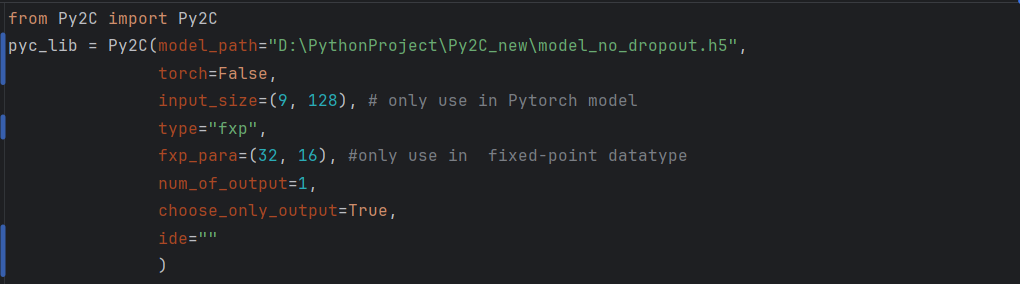


Hình 5. Tải file .h5

## Py2C

### Công cụ chuyển Python sang C

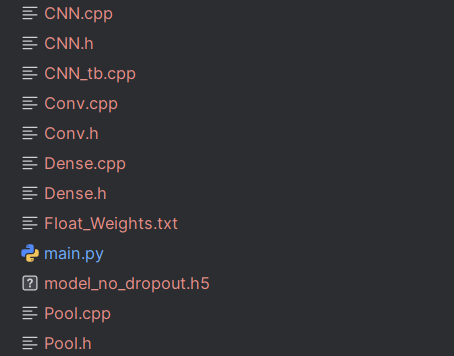
* Vào <https://github.com/WangNe2207/Py2C_new?tab=readme-ov-file> làm theo hướng dẫn để chuyển từ code Python sang code C
* File main.py cho Lenet-5 sẽ được chọn như sau
  + Thay đổi model\_path theo đường dẫn thực tế
  + Thay đổi type = “fxp”
  + Torch = False (nếu dùng Keras/TensorFlow)
  + Nếu dùng Visual Studio để ide = “vs”, nếu không thì để như ảnh



Hình 6. File main.py

### Kết quả sau khi chạy Py2C

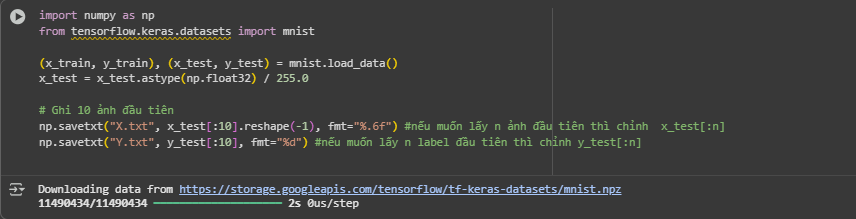
* Sau khi chạy xong sẽ có các file:



Hình 7. Các file được tạo ra sau khi chạy xong main.py

### Chuẩn bị file X.txt và Y.txt

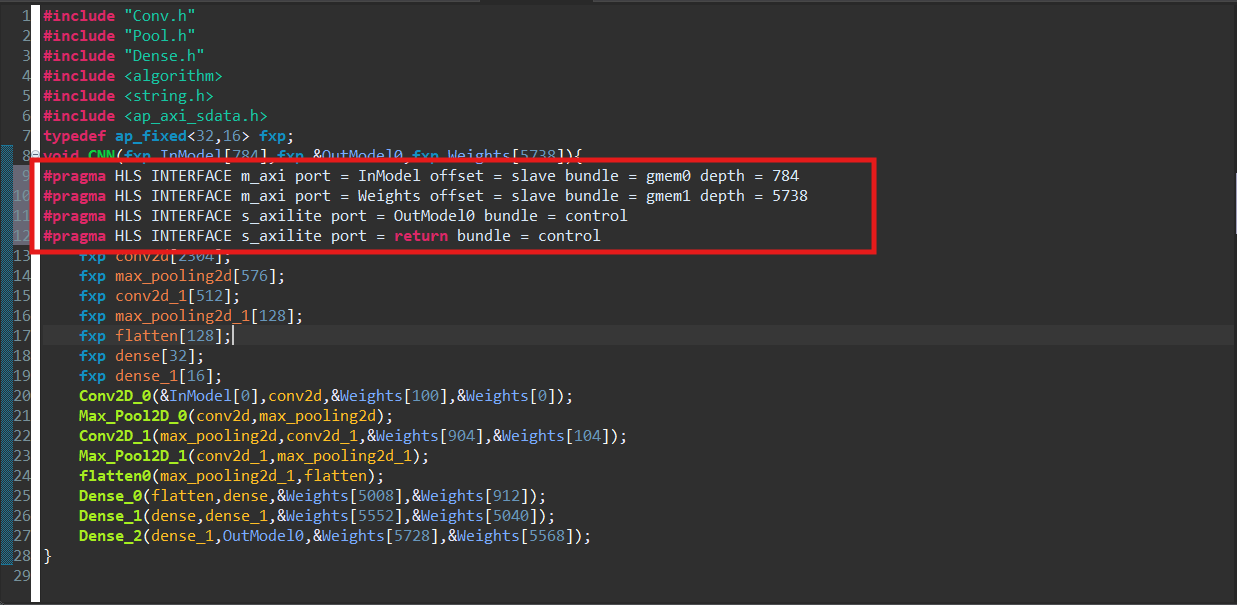
* File X.txt là file chứa các ảnh input để test
* File Y.txt là output mong muốn
* Dùng code Python như sau (file creat\_x\_y.py), chạy code trên Colab sau đó tải 2 file X.txt và Y.txt về.



Hình 8. Tạo file X.txt và Y.txt cho file CNN\_tb.cpp

## Bọc AXI:

Ở file CNN.cpp thêm các #pragma để tạo bọc AXI



Hình 9. File CNN.cpp sau khi thêm #pragma

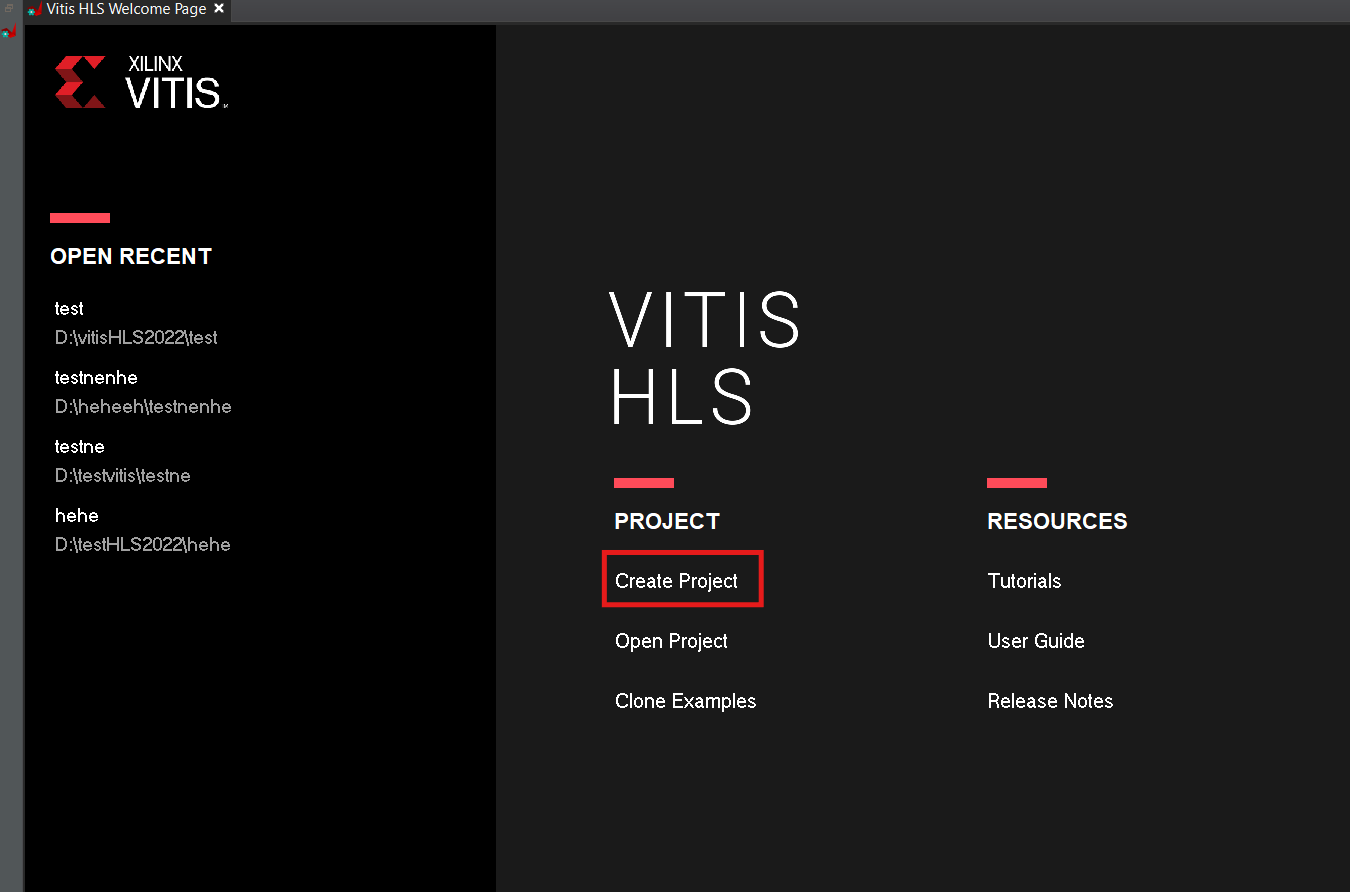
Trong đó:

* #pragma HLS INTERFACE m\_axi port = InModel offset = slave bundle=gmem0 depth = 784
  + m\_axi tạo 1 cổng AXI Master để IP có thể tự đọc từ BRAM (hoặc là DDR)
  + offset = slave thì HLS sẽ tạo ra 1 AXI-Lite lưu địa chỉ gốc của InModel CPU sẽ ghi vào địa chỉ này trước khi chạy IP, và IP sẽ thông qua địa chỉ này để giao tiếp trực tiếp với BRAM lưu InModel
  + bundle=gmem0 đặt tên bus cho cho cổng m\_axi là gmem0 để tách với Weights và cho phép đọc song song
  + depth = 784 thông báo kích thước tối đa của mảng vì đây là ảnh 28x28 = 784 phần tử
* #pragma HLS INTERFACE m\_axi port = Weights offset = slave bundle = gmem1 depth = 5738
  + Tương tự như trên
* #pragma HLS INTERFACE s\_axilite port = OutModel0 bundle = control
  + Tạo 1 AXI-Lite
  + Map OutModel0 thành thanh ghi AXI-Lite để CPU đọc kết quả.
* #pragma HLS INTERFACE s\_axilite port = return bundle = control
  + Tạo thanh ghi AP\_CTRL (ap\_start, ap\_done, …) cho việc điều khiển CNN IP.

## C-to-RTL với Vitis HLS 2022.2:

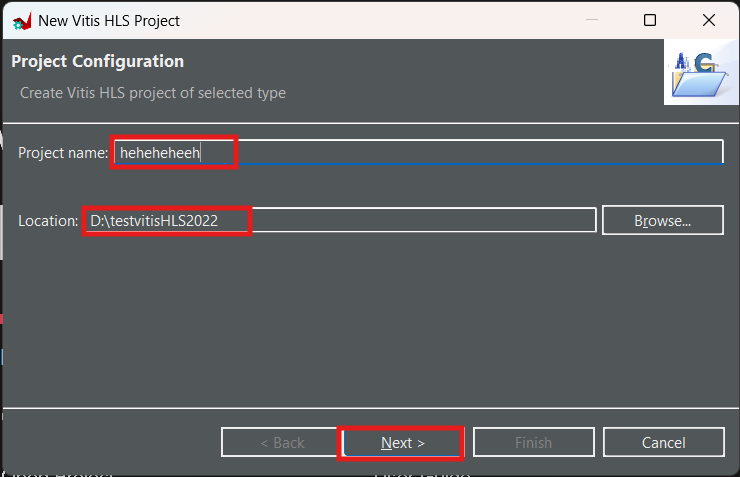
### Tạo project

* Chọn Create Project



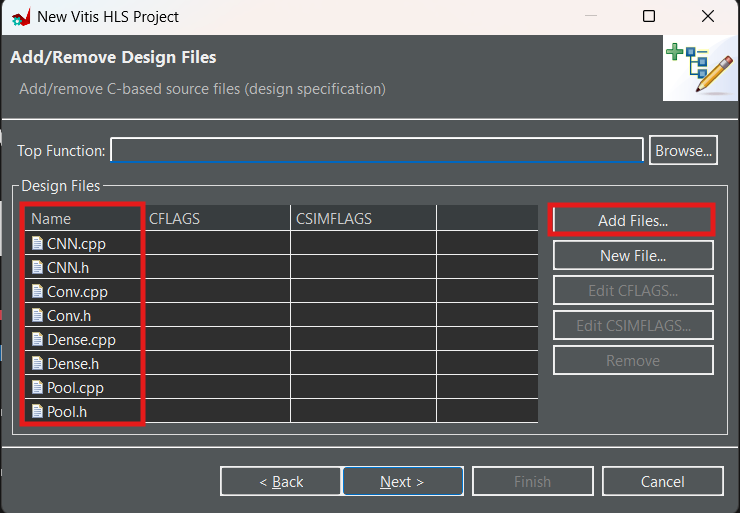
Hình 10. Tạo project

* Đặt tên, thư mục chứa project: nhấn Next



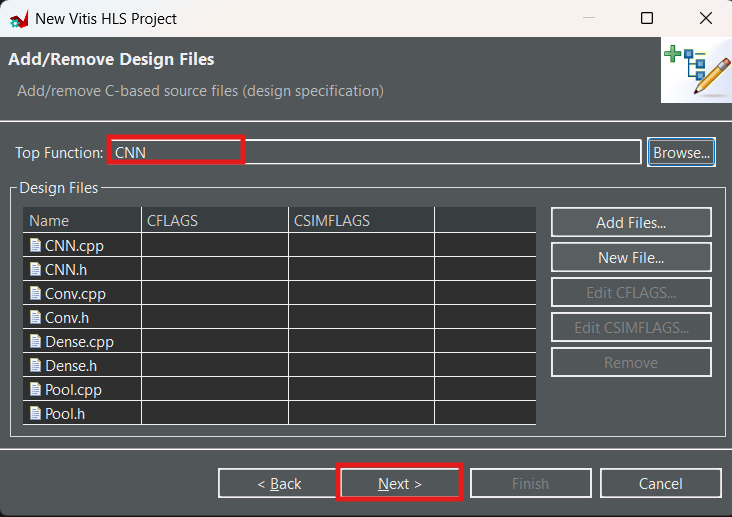
Hình 11. Đặt tên project

* Add/Remove Design file: add các file được tạo ra từ bước Py2C



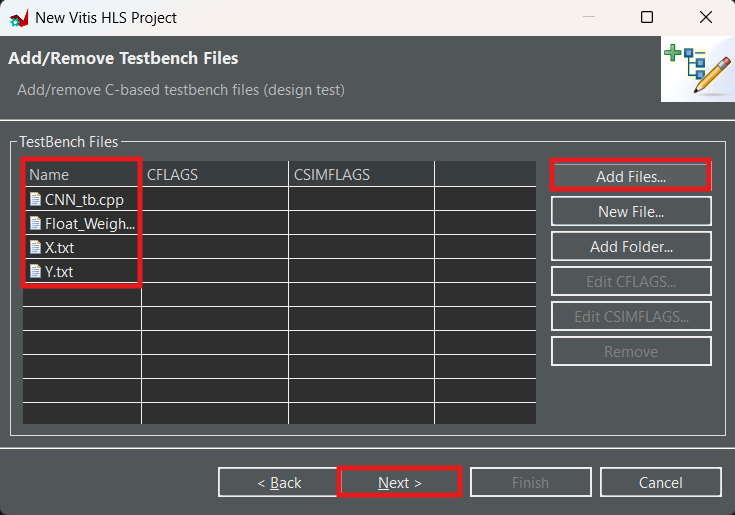
Hình 12. Add design files

* Chọn Top function là file CNN.cpp sau đó chọn next



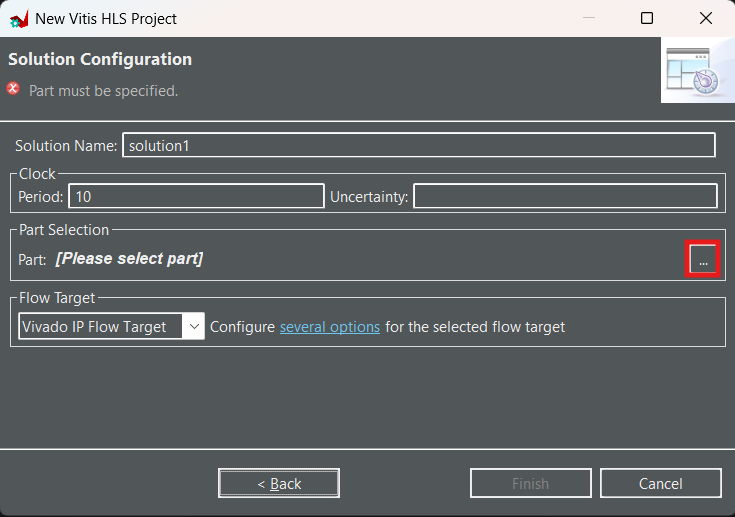
Hình 13. Chọn top function

* Add/Remove Testbench file: Add các file như hình sau đó chọn Next

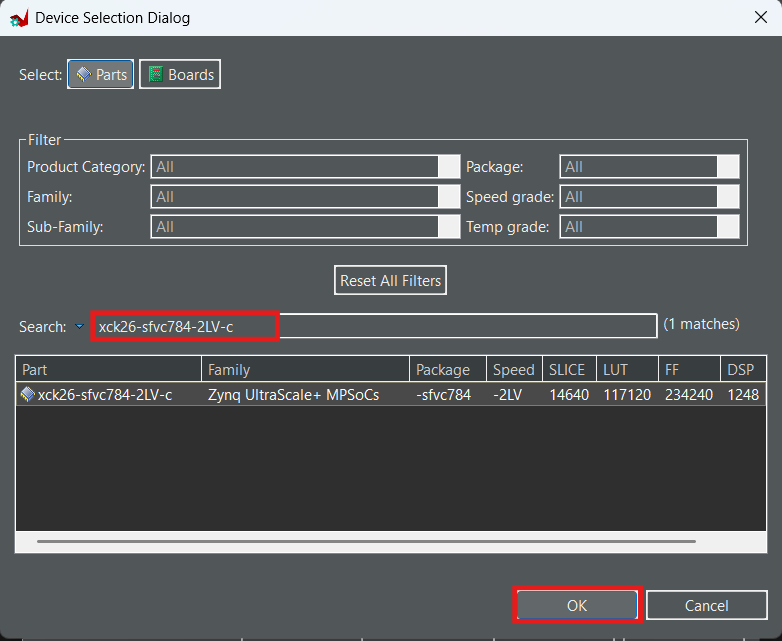


Hình 14. Chọn file testbench

* Solution Configuration: chọn dấu 3 chấm ở phần Part Selection chọn mã kit muốn nạp (nếu muốn sử dụng KV260 thì dùng mã xck26-sfvc784-2LV-c)

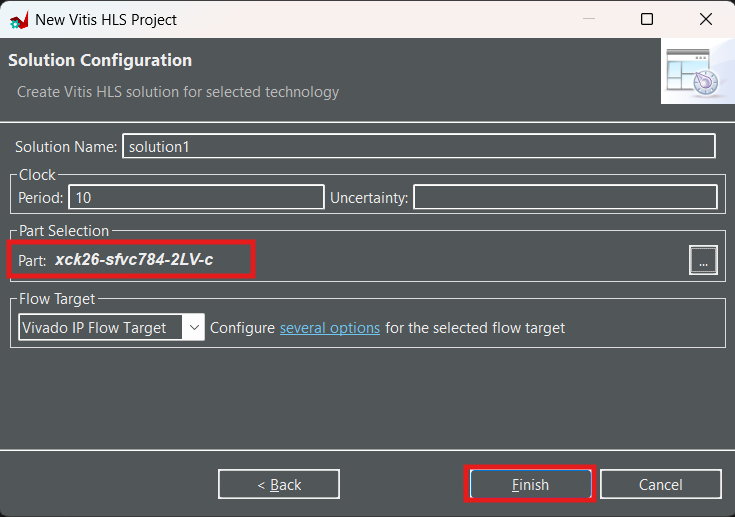


Hình 15. Chọn mã kit



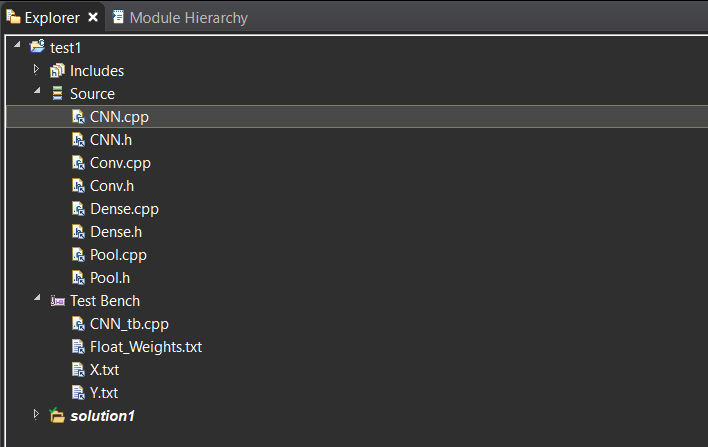
Hình 16. Nhập mã kit

* Finish



Hình 17. Finish

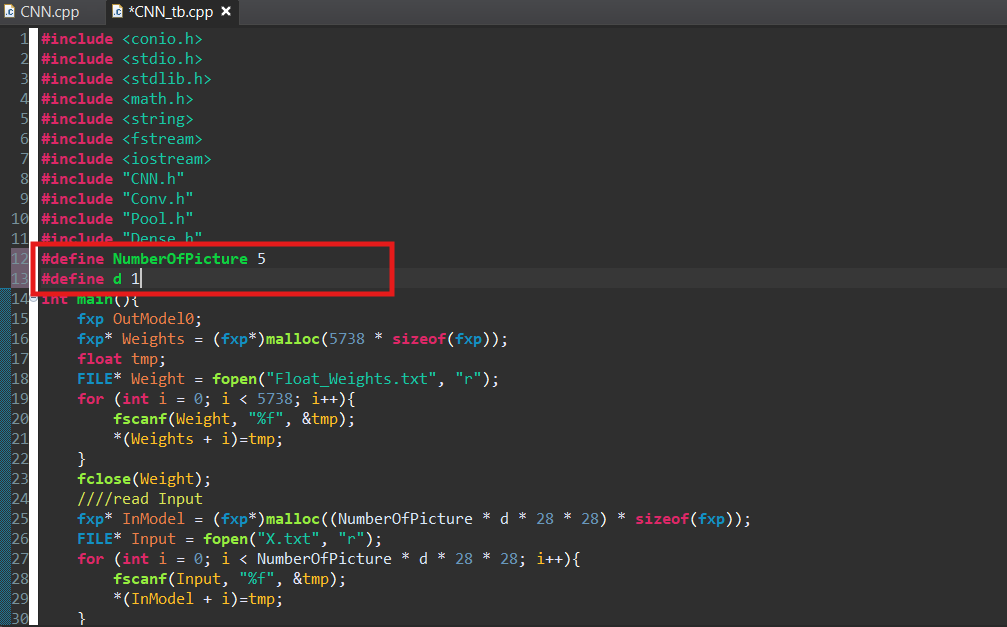
* Sau khi tạo thành công



Hình 18. Các file sau khi tạo thành công project

### Chạy C simulation

* Ở file CNN\_tb.cpp



Hình 19. Chọn số ảnh để test

* + Nếu muốn test trên n ảnh thì define NumberofPicture là n ví dụ ở đây là với 5 ảnh (nếu để quá lớn ở bước Cosimualtion sẽ phải đợi lâu và có thể bị timeout, chọn dump trace = all (ở hình 28)
  + Vì ở đây là ảnh trắng đen chỉ có 1 kênh đầu vào (28x28x1) nếu sử dụng ảnh màu thì có thể điền define d là 3

***Lưu ý:***

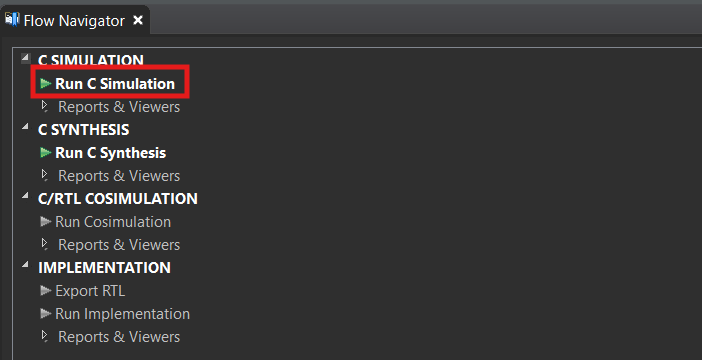
Có thể code ở dòng cuối cùng file Dense.cpp có thể thiếu dấu **\_** như vậy:

    output\_Dense0 = maxindex\_2;

Sửa lại như sau:

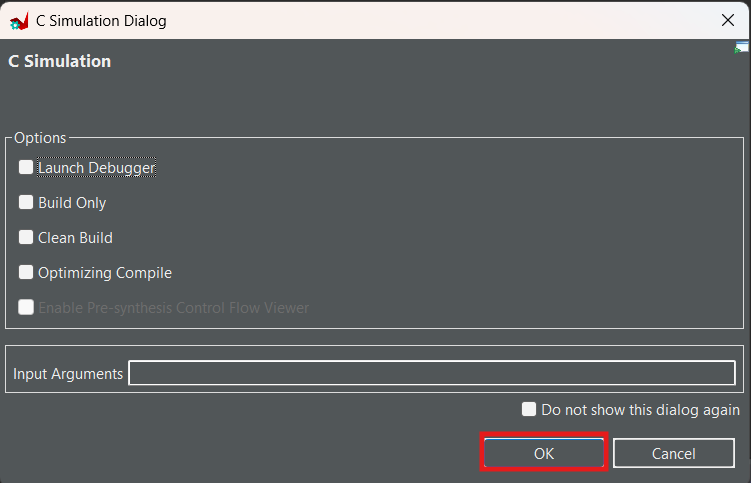
output\_Dense\_0 = maxindex\_2;

* Chọn Run C simulation



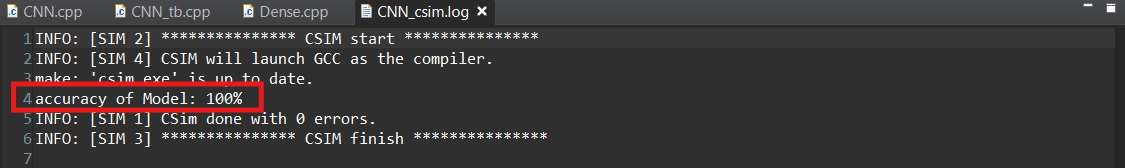
Hình 20. Run C simulation

* Tiếp theo nhấn OK



Hình 21. Chọn C simulation Dialog

* Sau khi chạy xong lưu ý phần accuracy of Model:

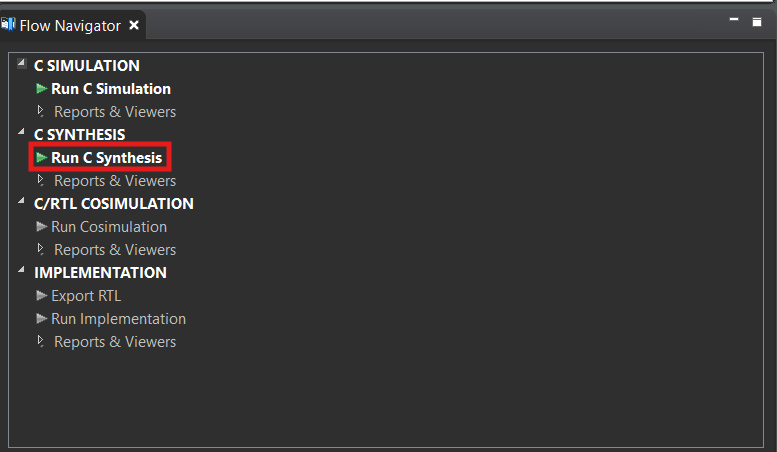


Hình 22. Kết quả sau khi chạy C simulation

Nếu accuracy quá thấp có thế code C đang bị sai cần kiểm tra và chỉnh sửa lại

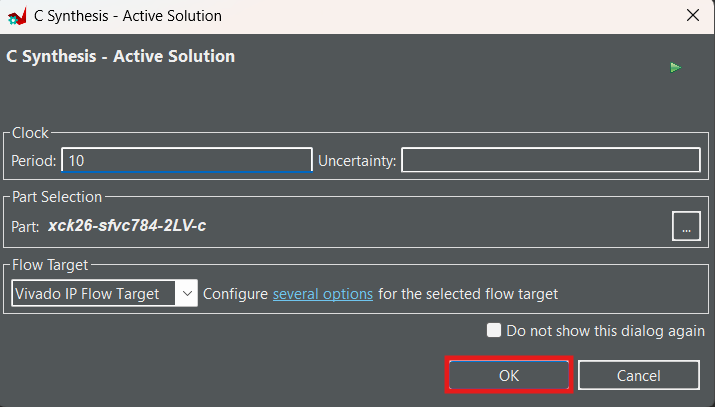
### C SYNTHESIS:

* Chọn Run C synthesis



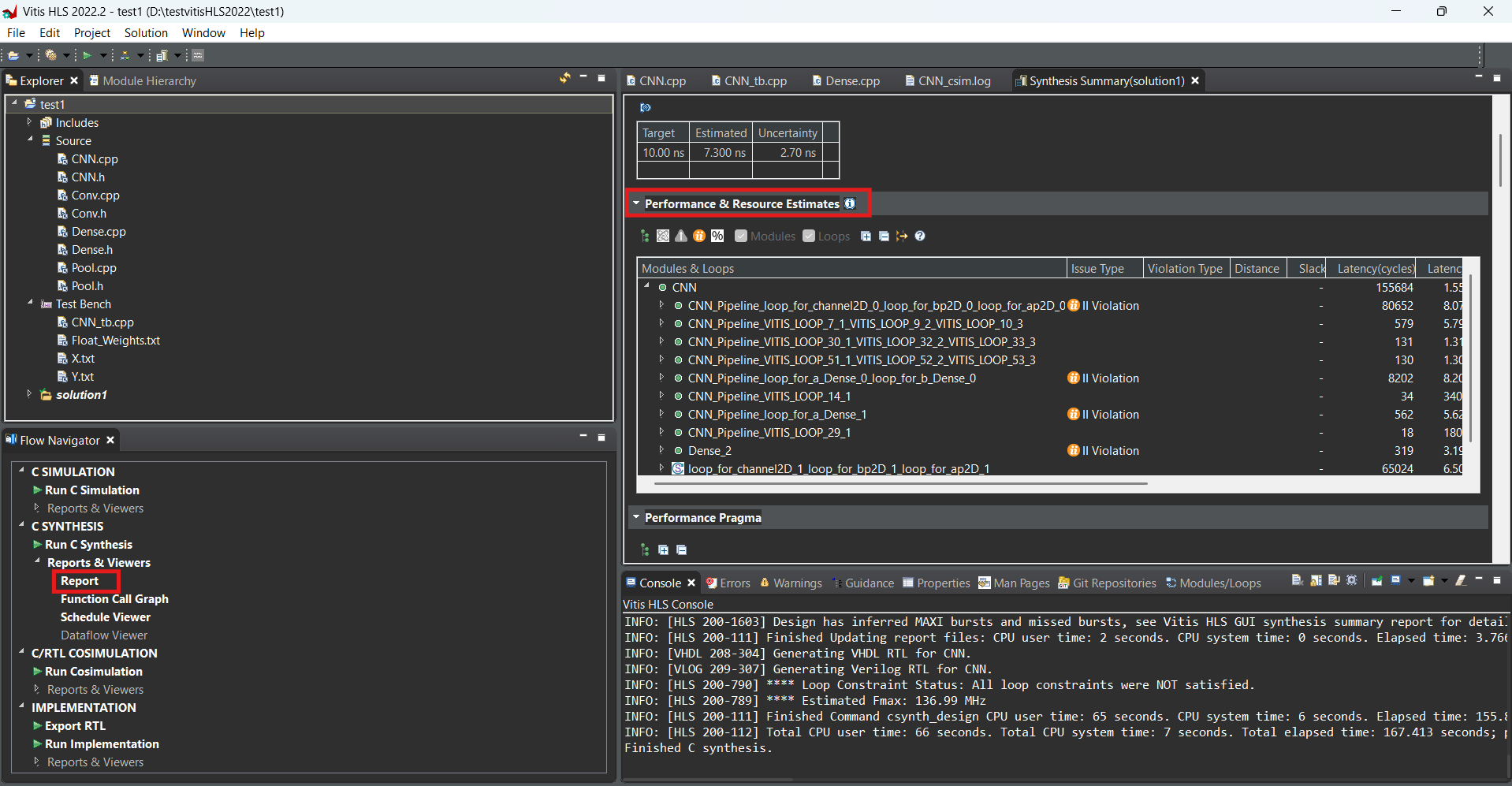
Hình 23. Run C Synthesis

* Tiếp theo chọn OK



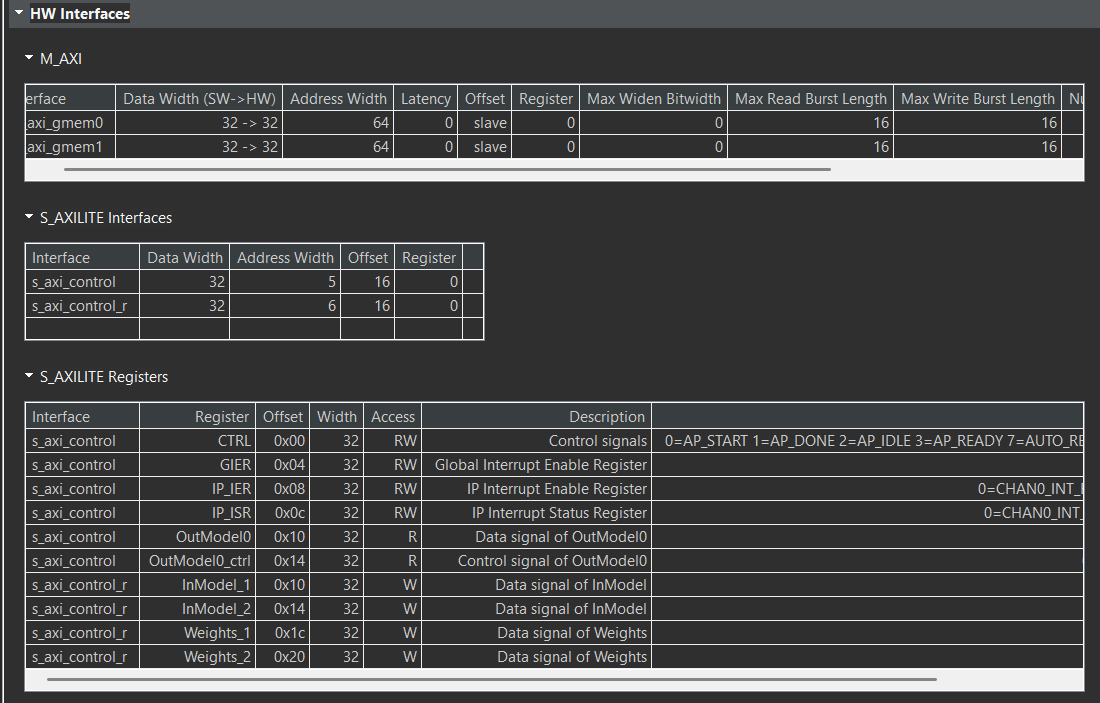
Hình 24. Set Clock

* Xem phần Performance & Resource Estimate để xem báo cáo tài nguyên của IP



Hình 25. Xem báo cáo tài nguyên

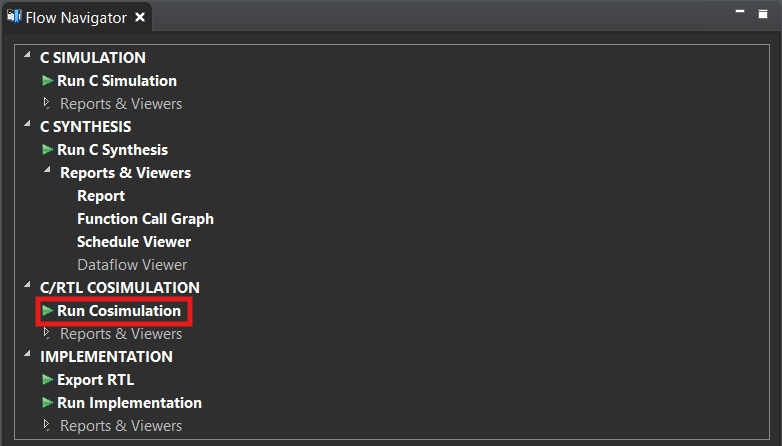
* Lưu ý ở phần HW Interfaces để có thể viết code C điều khiển ở phần sau



Hình 26. Cấu hình của bọc AXI

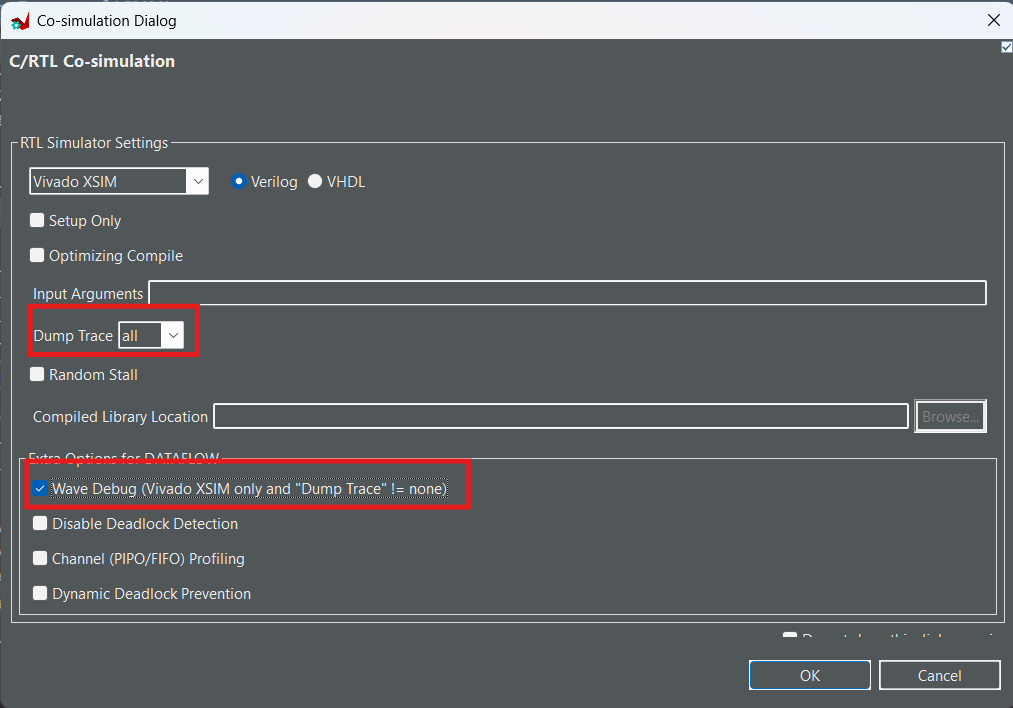
### C/RTL COSIMULATION:

* Đây là bước so sánh kết quả của RTL vừa được sinh ra ở phần Synthesis với code C để kiểm tra tính đúng đắn của RTL vừa được sinh ra
* Chọn Run Cosimulation



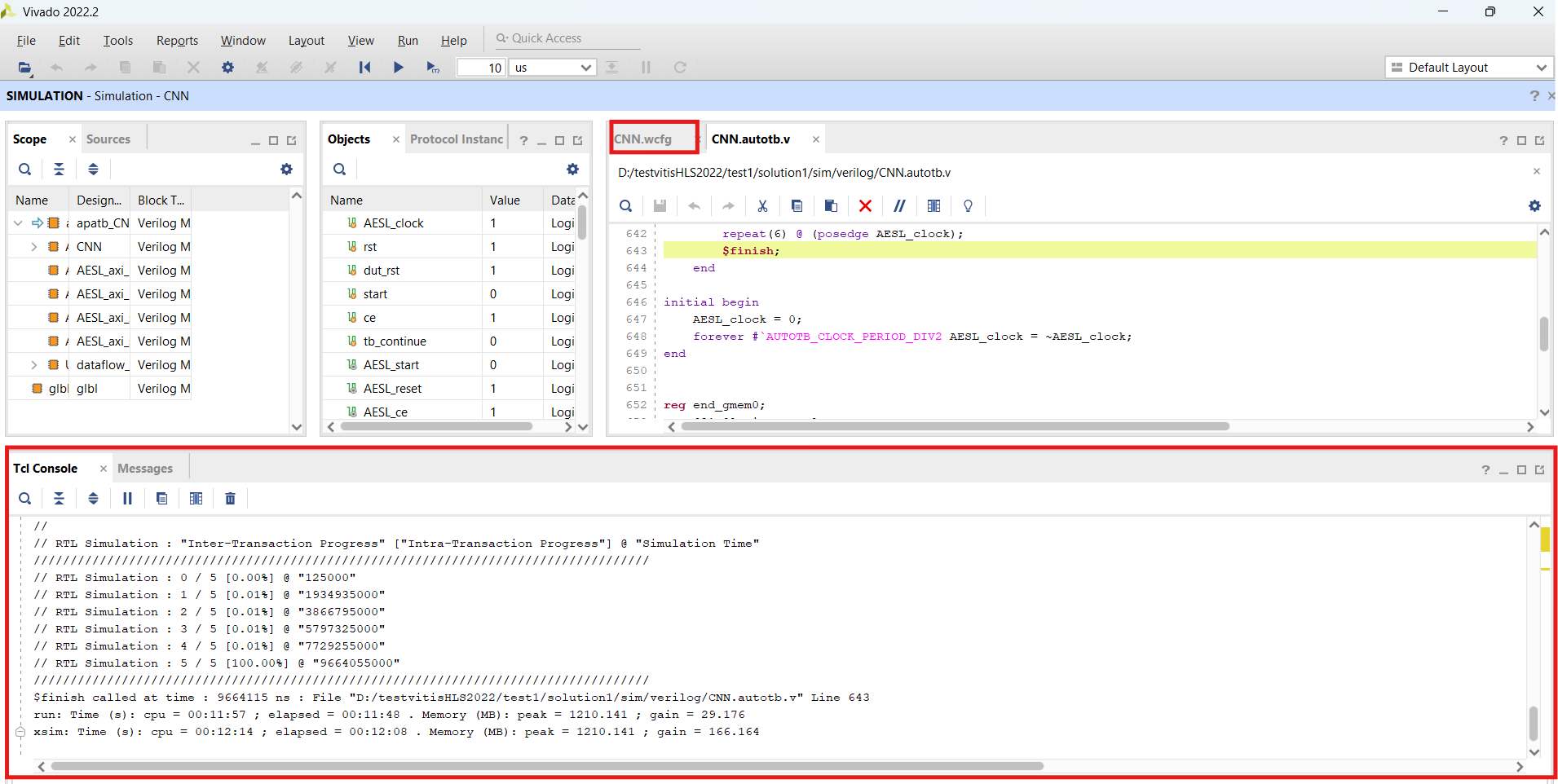
Hình 27. Run Cosimulation

* Chọn setting như hình để xem waveform (nếu không muốn xem waveform có thể tắt Wave Debug đi, và chọn Dump Trace = none để chạy bước này nhanh hơn)



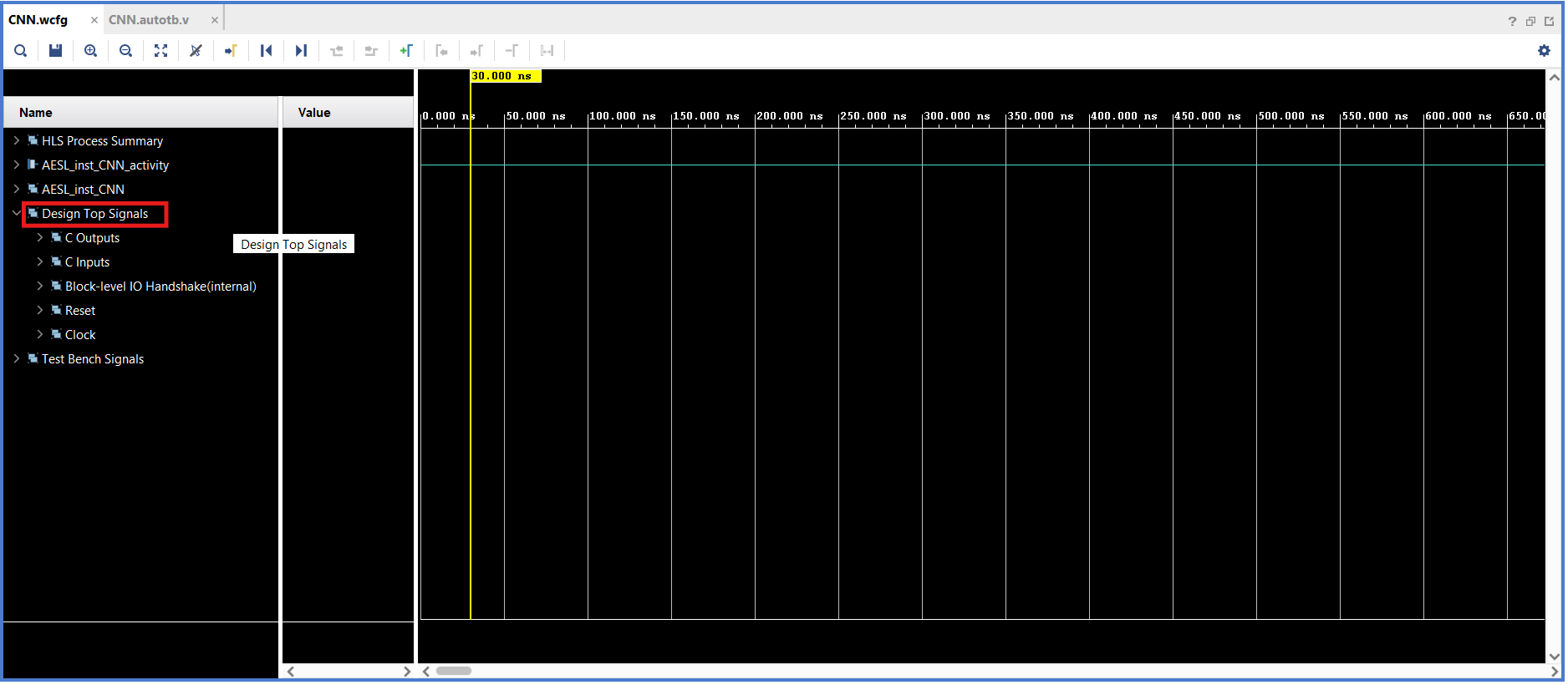
Hình 28. Cấu hình để xem waveform

* Lúc đang chạy, Vivado sẽ được mở đợi tới khi chạy xong chọn CNN.wcfg để xem waveform



Hình 29. Giao diện chạy simulation Vivado

* Trong waveform chọn Design Top Signals để xem các tín hiệu

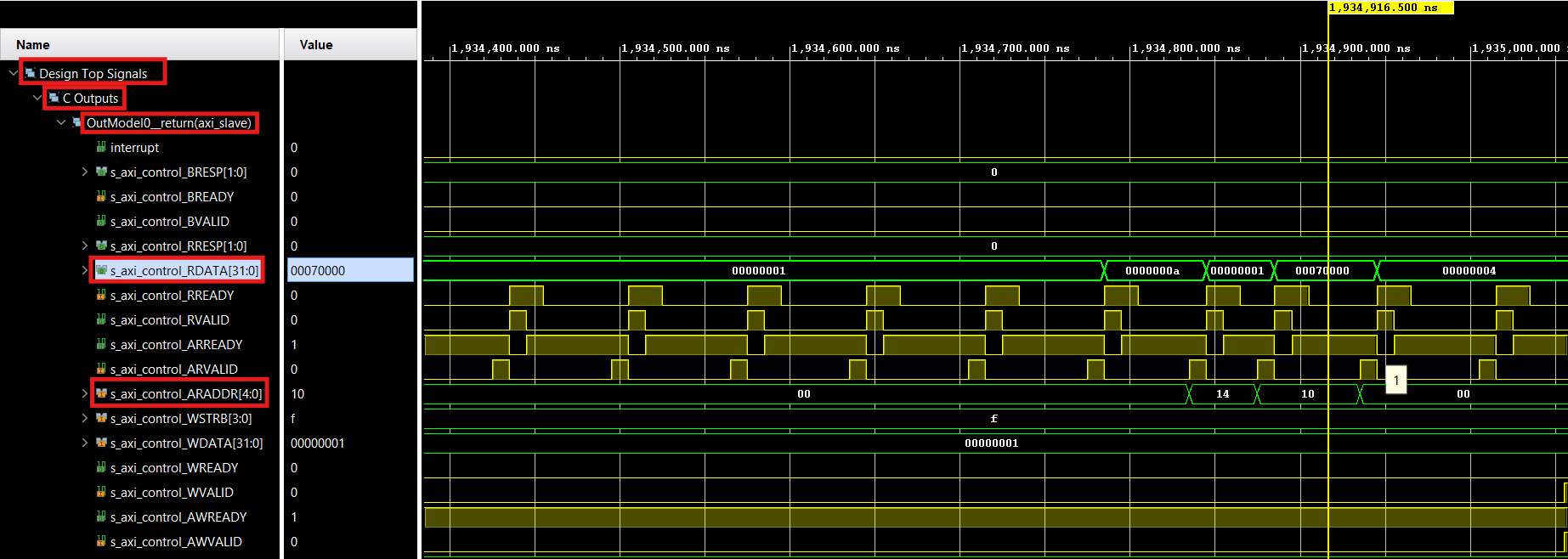


Hình 30. Waveform

Nếu muốn xem dữ liệu được xuất ra thì chọn C outputs 🡪 OutModel0 và chọn RDATA để xem output của Model

Lưu ý:

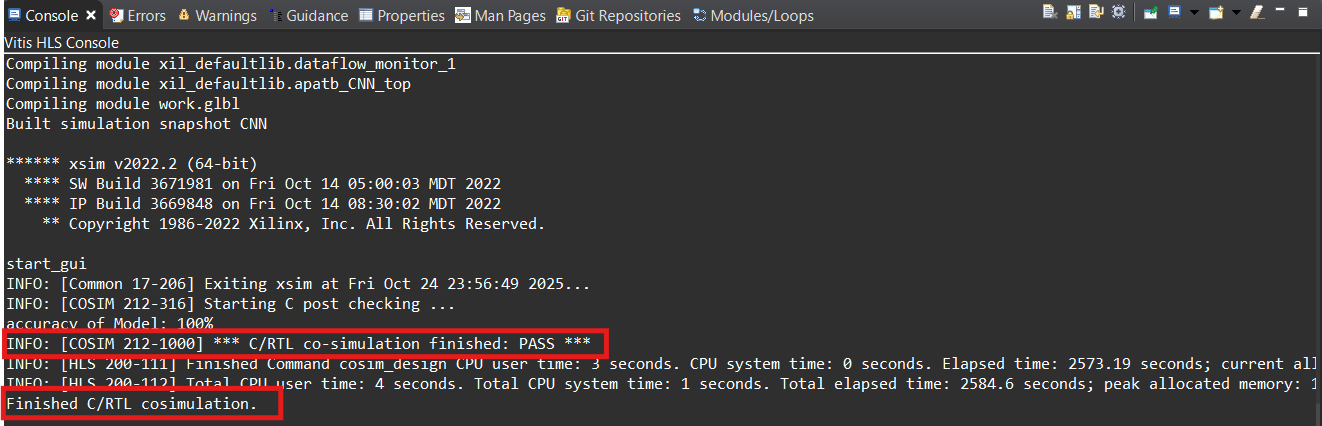
* + Address ở hình 25, Outmodel0 có offset là 0x10 nên khi nào ARADDR = 10 thì RDATA khi đó sẽ là output của mô hình sau khi chạy,tương tự như vậy với các giá trị ARADDR khác



Hình 31. Đọc dữ liệu đầu ra

Vì ở đây sử dụng fixed point Q16.16 nên số 7 sẽ được biểu diễn là 0x00070000

* Nếu đã xem xong thì thoát Vivado sau đó quay lại Vitis để hoàn thành chạy C cosimulation



Hình 32. Hoàn thành chạy Cosimulation

* Sau khi chạy xong nhấn vào Export RTL ở Implement để tạo IP



Hình 33. Export RTL

## Tích hợp IP vào hệ thống SoC

### Tải Vivado 2022.2

### Tạo project

* Đặt tên, chọn thư mục chứa project 🡪 Next

A screenshot of a computer

AI-generated content may be incorrect.

Hình 34. Tạo tên, thư mục project

* Project type chọn RTL 🡪 Next

A screenshot of a computer program

AI-generated content may be incorrect.

Hình 35. Chọn RTL project

* Add Source: Không add 🡪 Next
* Add Constraint: Không add 🡪 Next
* Default Part:
  + Chọn part hoặc board đang dùng, trong project này dùng KV260. Chọn sang Board, chọn KV260

A screenshot of a computer

AI-generated content may be incorrect.

Hình 36. Chọn board KV260

Lưu ý: Nếu không thấy board thì ấn Refresh để cập nhật.

* Finish

### Tạo hệ thống SoC

* Add IP CNN:

Tools

🡪 Settings

🡪 IP

🡪 Repository

A screenshot of a computer

AI-generated content may be incorrect.

Hình 37. Mở IP Repository

Trỏ đến project VitisHLS làm lúc đầu, Select .../impl/ip

A screenshot of a computer program

AI-generated content may be incorrect.

Hình 38. Chọn đường dẫn CNN IP

Đã tìm thấy IP 🡪 OK 🡪 OK

* Create Block Design 🡪 OK

A screenshot of a computer

AI-generated content may be incorrect.

Hình 39. Tạo Block Design

* Sau khi tạo xong Block Design, nhấn + để thêm IP

A screenshot of a computer

AI-generated content may be incorrect.

Hình 40. Thêm IP

* Add Zynq UltraScale+ MPSoC 🡪 Run Block Automation

A screenshot of a computer

AI-generated content may be incorrect.

Hình 41. Thêm Zynq UltraScale+ MPSoC

* Add CNN IP

A screenshot of a computer

AI-generated content may be incorrect.

Hình 42. Thêm CNN IP

* Add 2 Block Memory Generator. Trong Block Properties đổi tên thành inmodel\_bram và weights\_bram. Ấn vào từng IP để chỉnh Memory Type thành True Dual Port RAM
  + 2 Port A/B để hỗ trợ đọc/ghi ở 2 địa chỉ khác nhau cùng lúc, giúp tăng tốc độ xử lý. Có thể chia sẻ dữ liệu bằng một cổng cho CPU truy cập, cổng còn lại cho khối tăng tốc (CNN IP) sử dụng.

A screenshot of a computer

AI-generated content may be incorrect.

Hình 43. Chọn Memory Type

A screenshot of a computer

AI-generated content may be incorrect.

Hình 44. Đổi tên BRAM

A screenshot of a computer

AI-generated content may be incorrect.

Hình 45. Sau khi thêm 2 BRAM

* Add 2 IP AXI BRAM Controller. Đổi tên thành inmodel\_bram\_ctrl và weights\_bram\_ctrl. Nối dây Port A, B của 2 BRAM Controller sang Port A, B tương ứng của 2 Block Mem.

A diagram of a computer program

AI-generated content may be incorrect.

Hình 46. Nối BRAM CTRL với BRAM

* Run Connection Automation cho đến khi có hệ thống hoàn chỉnh, không còn hiện thông báo Run Connection Automation nữa.

A screenshot of a computer

AI-generated content may be incorrect.

Hình 47. Run Connection Automation

A screenshot of a computer

AI-generated content may be incorrect.

Hình 48. Setting Run Connection Automation

A screenshot of a computer

AI-generated content may be incorrect.

Hình 49. SoC hoàn chỉnh

* Chuyển qua tab Address Editor, chỉnh lại Base Address và Range của từng BRAM cho phù hợp. Khi chỉnh Range có thể gặp lỗi, phải chỉnh lại Base Address theo như thông báo
  + Ở đây Weights có 5738 phần tử, mỗi phần tử 1 word nên BRAM cần khoảng 22KB 🡪 chỉnh thành 32KB

A screenshot of a computer

AI-generated content may be incorrect.

Hình 50. Lỗi khi chỉnh Range BRAM

A screenshot of a computer

AI-generated content may be incorrect.

Hình 51. Sau khi chỉnh Range và Base Address hoàn tất

* Chuyển lại qua tab Diagram. Nhấn nút Valid Design(F6)

A screenshot of a computer

AI-generated content may be incorrect.

Hình 52. Valid Design

* Nhấn Generate Block Design 🡪 Generate

A screenshot of a computer

AI-generated content may be incorrect.

Hình 53. Generate Block Design

* Trong tab Source, right-click vào design.bd 🡪 Create HDL wrapper 🡪 OK

A screenshot of a computer

AI-generated content may be incorrect.

Hình 54. Tạo HDL Wrapper

A screenshot of a computer screen

AI-generated content may be incorrect.

Hình 55. Setting HDL Wrapper

A screenshot of a computer program

AI-generated content may be incorrect.

Hình 56. Sau khi tạo xong Wrapper

* Run Synthesis 🡪 OK

A screenshot of a computer

AI-generated content may be incorrect.

Hình 57. Run Synthesis

* Run Implementation 🡪 OK. Khi xong có thể kiểm tra report Utilization, Timing,... để kiểm tra các thông số

A screenshot of a computer

AI-generated content may be incorrect.

Hình 58. Run Implementation

* Generate Bitstream 🡪 OK để tạo file .bit

A screenshot of a computer

AI-generated content may be incorrect.

Hình 59. Tạo file Bitstream

* File -> Export -> Hardware để tạo file .xsa. Trong tab Output phải **include bitstream**.

A screenshot of a computer

AI-generated content may be incorrect.

Hình 60. Tạo file .xsa

A screenshot of a computer

AI-generated content may be incorrect.

Hình 61. Include bitstrean cho file .xsa

## Cài Petalinux

**\*\*Cách 1: Các bạn có thể tải file image mình build sẵn ở đường link bên dưới.**

<https://drive.google.com/file/d/1XiKV12dBukXk6_hllGrsGoc6sPSUN5GH/view?usp=sharing>

**\*\*Cách 2: Tự build lại file Linux cho KV260 FPGA như flow bên dưới:**

***Xem toàn bộ theo link youtube bên dưới và làm theo:***

<https://youtu.be/F1vxzkd7_DI>

**Các bước chi tiết như sau:**

* Tải bộ cài **PetaLinux 2022.2** từ trang chính thức Xilinx:  <https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools/archive.html>

**Cài đặt các gói phụ thuộc (Ubuntu/Debian)**

*sudo apt-get install tofrodos gawk xvfb git libncurses5-dev tftpd zlib1g-dev zlib1g-dev:i386 \*

*libssl-dev flex bison chrpath socat autoconf libtool texinfo gcc-multilib \*

*libsdl1.2-dev libglib2.0-dev screen pax libtinfo5 xterm build-essential net-tools*

**Cấp quyền thực thi cho file .run**

*chmod +x petalinux-v2022.2-\*.run*

**Chạy trình cài đặt**

./petalinux-v2022.2-\*.run

* Trong quá trình cài đặt, trình cài đặt sẽ hiển thị các thỏa thuận bản quyền:
  + Dùng PgUp / PgDn để đọc
  + Nhấn q để thoát khỏi phần hiển thị
  + Nhấn y để đồng ý và tiếp tục

**Thiết lập môi trường làm việc Petalinux**

**Source đến thư mục cài đặt Petalinux để sử dụng được các lệnh petalinux-\*:**

*source <đường\_dẫn\_cài\_petalinux>/2022.2/settings.sh*

**Tải bộ cài BSP cho KV260 FPGA từ trang chính thức Xilinx:**

<https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools/archive.html>

**Tạo project PetaLinux từ BSP**

*petalinux-create -t project -s <đường\_dẫn\_tới\_file\_BSP>.bsp --name KV260\_Linux*

*cd KV260\_Linux*

**Import phần cứng (.xsa) vào project Sau khi bạn export file .xsa từ Vivado (có chứa bitstream), hãy dùng lệnh sau để tích hợp phần cứng vào project:**

petalinux-config --get-hw-description=<path\_to\_the\_hw\_description\_file>

**Cấu hình kernel bootargs thủ công Sau khi chạy petalinux-config, hệ thống sẽ mở giao diện curses để bạn cấu hình sâu hơn. Điều chỉnh cấu hình kernel bootargs Trong cửa sổ cấu hình, thực hiện các bước sau:**

*Subsystem AUTO Hardware Settings --->*

*DTG Settings --->*

*Kernel Bootargs --->*

*[ ] generate boot args automatically*

*(user-defined) user set kernel bootargs*

Dán đoạn bootargs dưới đây vào phần user set kernel bootargs:

*earlycon console=ttyPS1,115200 root=/dev/mmcblk1p2 rw rootwait cpuidle.off=1 uio\_pdrv\_genirq.of\_id=generic-uio clk\_ignore\_unused init\_fatal\_sh=1 cma=256M*

Cấu hình này giúp khởi động đúng thiết bị, bật driver UIO, cấp vùng bộ nhớ CMA, và giữ clock cho các IP tự thiết kế trong PL.

**Chỉnh sửa Device Tree (system-user.dtsi)**

Để hệ điều hành Linux có thể sử dụng **IP tự thiết kế trong PL** thông qua driver uio, bạn cần chỉnh sửa file **Device Tree Overlay**. Trong file ở đường dẫn KV260\_Linux/project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi, chỉnh lại file thành:

*/include/ "system-conf.dtsi"*

*/ {*

*amba\_pl@0 {*

*MY\_IP@a0000000 {*

*compatible = "generic-uio";*

*};*

*};*

*};*

File system-user.dtsi mẫu được lưu trong thư mục KV260\_Linux ở github.

**Sau đó tiến hành build project**

petalinux-build

1. **G. Bước 7: Tạo image khởi động và rootfs cho Linux trên SoC FPGA**

Sau khi build project thành công, gõ lệnh này để đóng gói file khởi động BOOT.BIN cùng với U-Boot phù hợp cho hệ thống.

*petalinux-package --boot --force --u-boot*

Sau đó cắm SD card vào PC, tiến hàn phân vùng và định dạng thẻ nhớ SD. \*\*Bạn có thể làm theo hướng dẫn chi tiết trong Video hướng dẫn bên trên \*\* từ phút **53:40 đến 1:03:18** tại link bên dưới:

<https://youtu.be/F1vxzkd7_DI>

File rootfs này chứa hệ điều hành Debian đã được cấu hình sẵn cho kiến trúc ARM64, hỗ trợ giao diện XFCE và dễ dàng cài đặt thêm ứng dụng bằng apt.

<https://drive.google.com/file/d/1ZcJYuVHpn8ER11nLCjwCUjfc5ykqP0tM/view>

## Datatest Recognition (Code C)

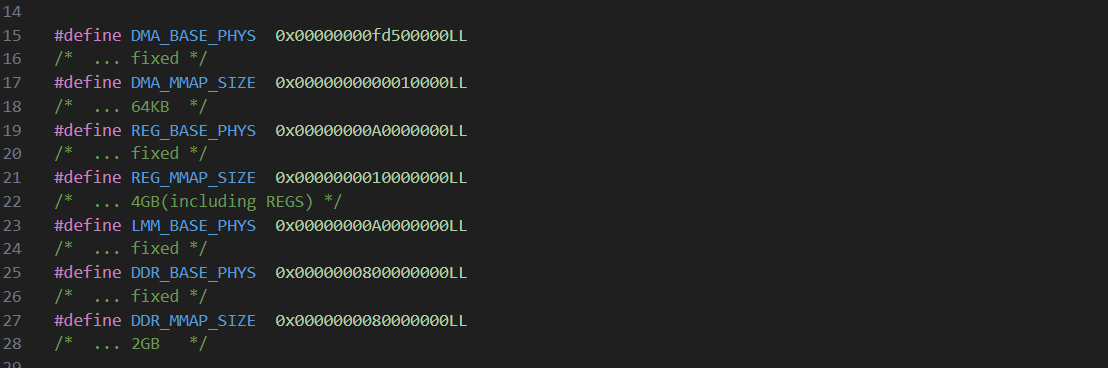
***Toàn bộ code nằm trong link bên dưới, copy lên trên account Linux trên KV260 FPGA:***

<https://github.com/AISeQLab/Real-time_Handwritten_Digit_Recognition_on_FPGA/tree/main/Embedded_Software/Datatest_Recognition>

### Chuẩn bị

* Tải file Fpga\_Driver.c

Lưu ý trong Fpga\_Driver.c:



Hình 63. Địa chỉ trong Fpga\_Driver.c

* Địa chỉ vật lý gốc của vùng thanh ghi điều khiển (REG\_BASE\_PHYS) được định nghĩa là 0x00000000A0000000 nên khi viết ở code C điều khiển không cần viết 0x00\_Axxx\_xxxx như các thanh ghi ở hình 50, mà chỉ cần viết các số sau A (vd: s\_axi\_control\_r 0x00\_A001\_0000 🡪 0x10000)
* Tương tự như trên địa chỉ vật lý gốc của vùng bộ nhớ nội (LLM\_BASE\_PHYS) được định nghĩa là 0x00000000A0000000 nên khi viết ở code C điều khiển không cần viết 0x00\_Axxx\_xxxx như các địa chỉ của memory ở hình 50 mà chỉ cần viết các số sau A (vd: InModel\_Bram\_ctrl/S\_AXI 0x00\_A002\_0000 🡪 0x20000)
* Xem ở phần sau để rõ hơn
* Tải file CRGA.h
* Tải hoặc tự tạo 2 files **weights\_array.c** và **test\_images\_1000** bằng cách chạy file **create\_files\_c** trong link Github ở đầu. Đoạn script này sẽ thực hiện:
  + Tải dữ liệu MNIST, chuẩn hóa và ghi ảnh chuẩn hóa, nhãn ra files X.txt, Y.txt
  + Đọc từ X.txt và Y.txt để sinh ra **test\_images\_1000.c**: mảng nhãn và ảnh fixed‑point
  + Đọc trọng số từ Float\_Weights.txt, chuyển từ float sang fixed‑point và ghi ra file **weight\_array.c**

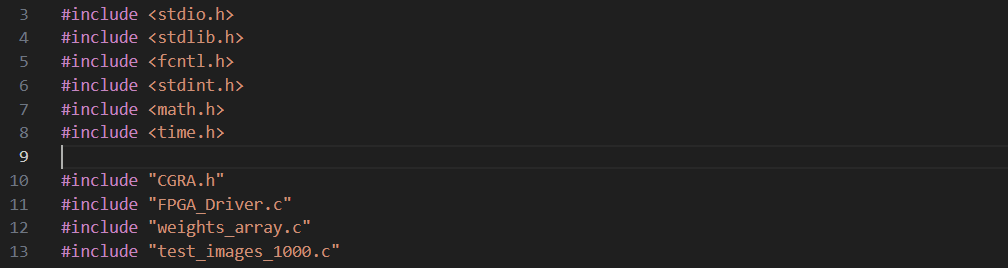
### Viết code C điều khiển

**Note:**

Xil\_Out(A, B) có nghĩa CPU là ghi giá trị B vào địa chỉ A

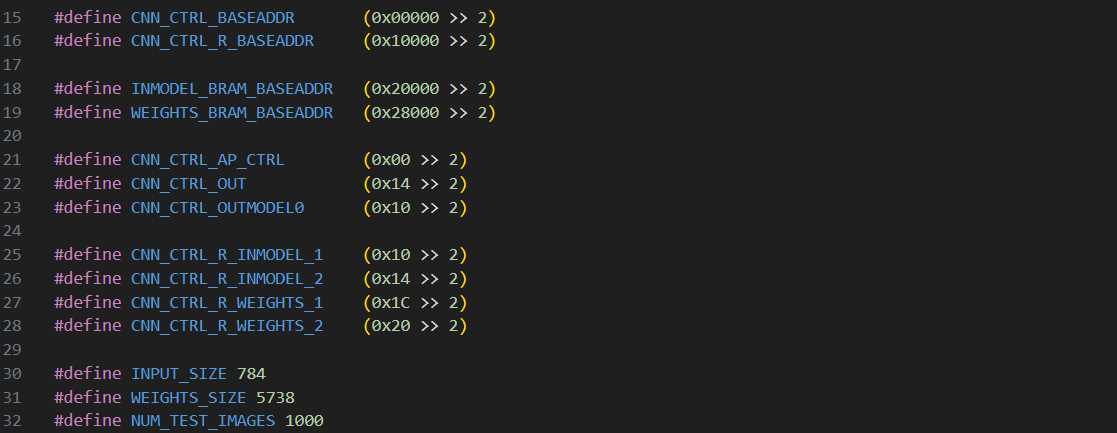
Xil\_In(A) có nghĩa là CPU sẽ đọc giá trị từ địa chỉ A

* Include thư viện



Hình 64. Include các thư viện

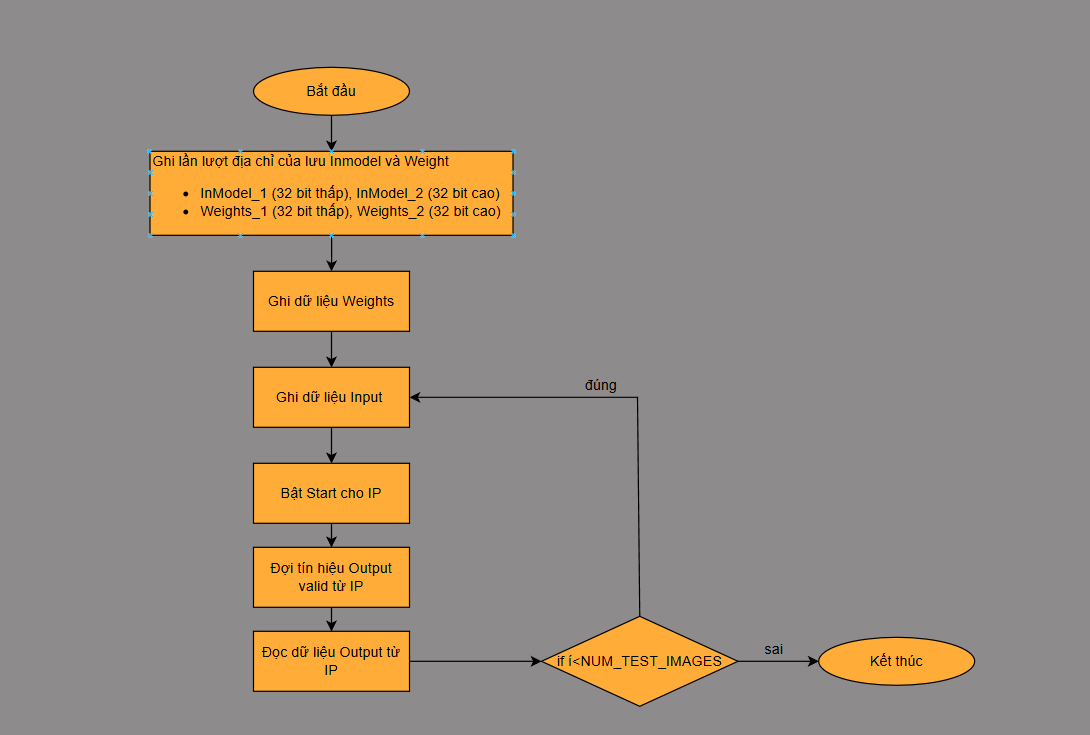
* Define các địa chỉ để thao tác (baseaddr xem ở hình 50 và các offset xem ở hình 25)
* Lưu đồ code C điều khiển



Hình 65. Các địa chỉ offset

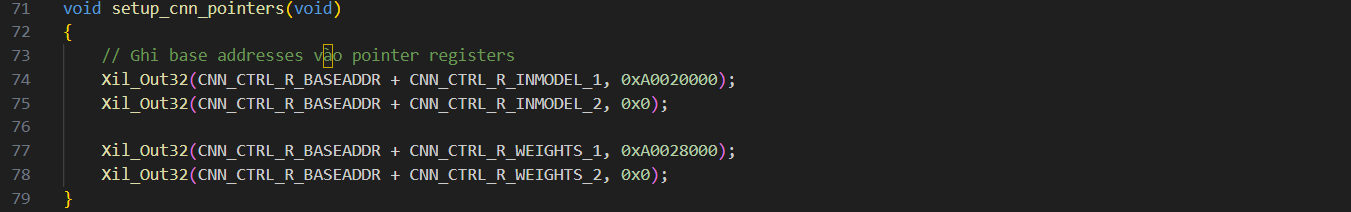
Chú thích:

* + Địa chỉ được định nghĩa theo byte nhưng hàm Xil\_In và Xil\_Out lại nhận giá trị theo word (được định nghĩa ở file FPGA\_Driver.c) nên cần dịch 2 bit để chuyển từ byte thành word
  + INPUT\_SIZE được define là 784 vì mỗi lần sẽ xử lý 1 ảnh là (28x28 = 784)
  + WEISGHTS\_SIZE sẽ là số lượng trọng số
  + NUM\_TEST\_IMAGES sẽ là số ảnh muốn chạy
* Lưu đồ cho code C điều khiển (cnn\_petalinux.c)



Hình 66. Lưu đồ code C điều khiển

* Hàm ghi địa chỉ InModel và Weight vào thanh ghi Ctrl\_r cho IP



Hình 67. Hàm setup\_cnn\_pointer

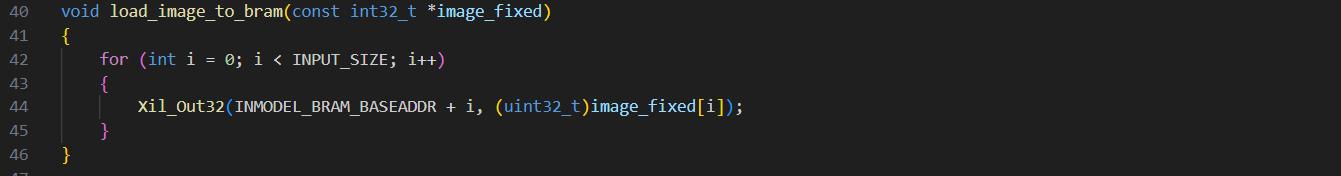
Giải thích:

* Vì địa chỉ chỉ có 32 bit nên chỉ cần ghi vào thanh ghi “\_1” và thanh hi “\_2” là 0
* Ghi địa chỉ của Block BRAM lưu INMODEL (0xA0020000) (xem ở hình 50) vào thanh ghi CTRL\_R\_INMODEL\_1 để IP có thế giao tiếp trực tiếp với BRAM
* Ghi địa chỉ của Block BRAM lưu Weights (0xA0028000) (xem ở hình 50) vào thanh ghi CTRL\_R\_WEIGHTS\_1
* Hàm ghi Weights vào BRAM



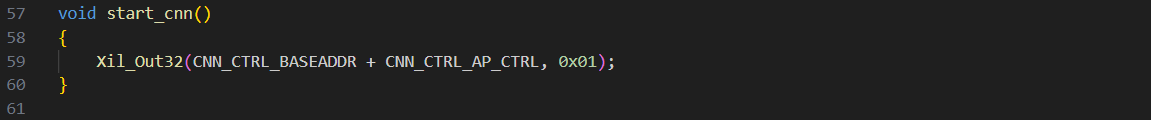
Hình 68. Hàm load\_weights\_to\_bram

* Ghi lần lượt dữ liệu Weights vào BRAM
* Hàm ghi Image vào Bram



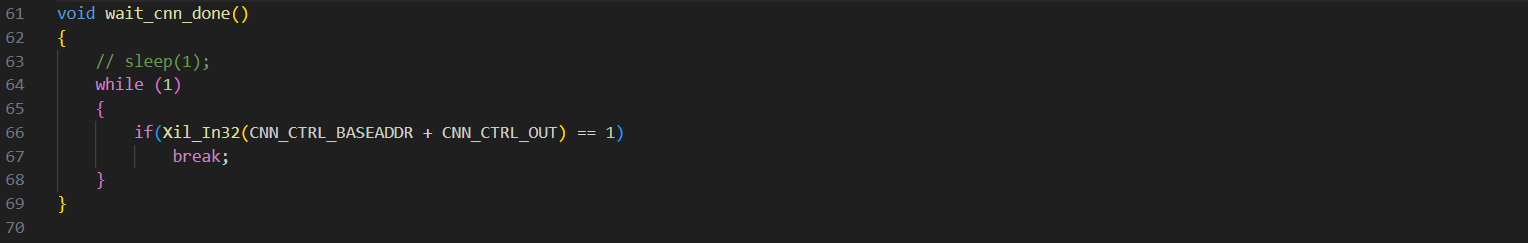
Hình 69. Hàm load\_image\_to\_bram

* Ghi lần lượt giá trị của từng pixel trong ảnh vào BRAM
* Hàm bật start cho IP



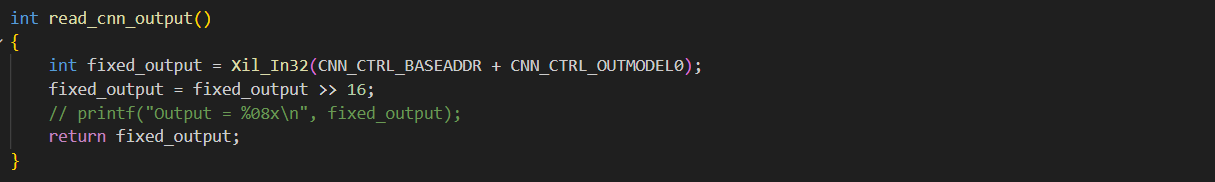
Hình 70. Hàm start\_cnn

* Ghi 0x01 vào thanh ghi AP\_CTRL vì bit 1 sẽ dành cho AP\_START (hình 25)
* Hàm đợi tín hiệu Outvalid từ IP



Hình 71. Hàm wait\_cnn\_done

* Đợi cho tới khi Output valid lên 1 CPU sẽ ngưng đợi và chuyển sang đọc Output của Model
* Hàm đọc Output của Model



Hình 72. Hàm read\_cnn\_output

* + Đọc dữ liệu Output từ thanh ghi OutModel0 (ở hình 25)
  + Vì dữ liệu xuất ra là dạng fixed point Q16.16 như ở hình 30 nên cần phải dịch 16 bit để đọc phần nguyên
* Hàm run\_inference



Hình 73. Hàm run\_inference

* Hàm main

A black background with red and blue text

AI-generated content may be incorrect.

Hình 74. Khởi tạo và thiết lập giao tiếp phần cứng



Hình 75. Inference loop

* Vòng for sẽ thực hiện nạp lần lượt từng ảnh vào BRAM để xử lý, sau đó sẽ kiếm tra kết quả được trả về từ IP so với test\_label (được lưu trong test\_images.c)

Để compile code C và chạy thì bạn gõ:

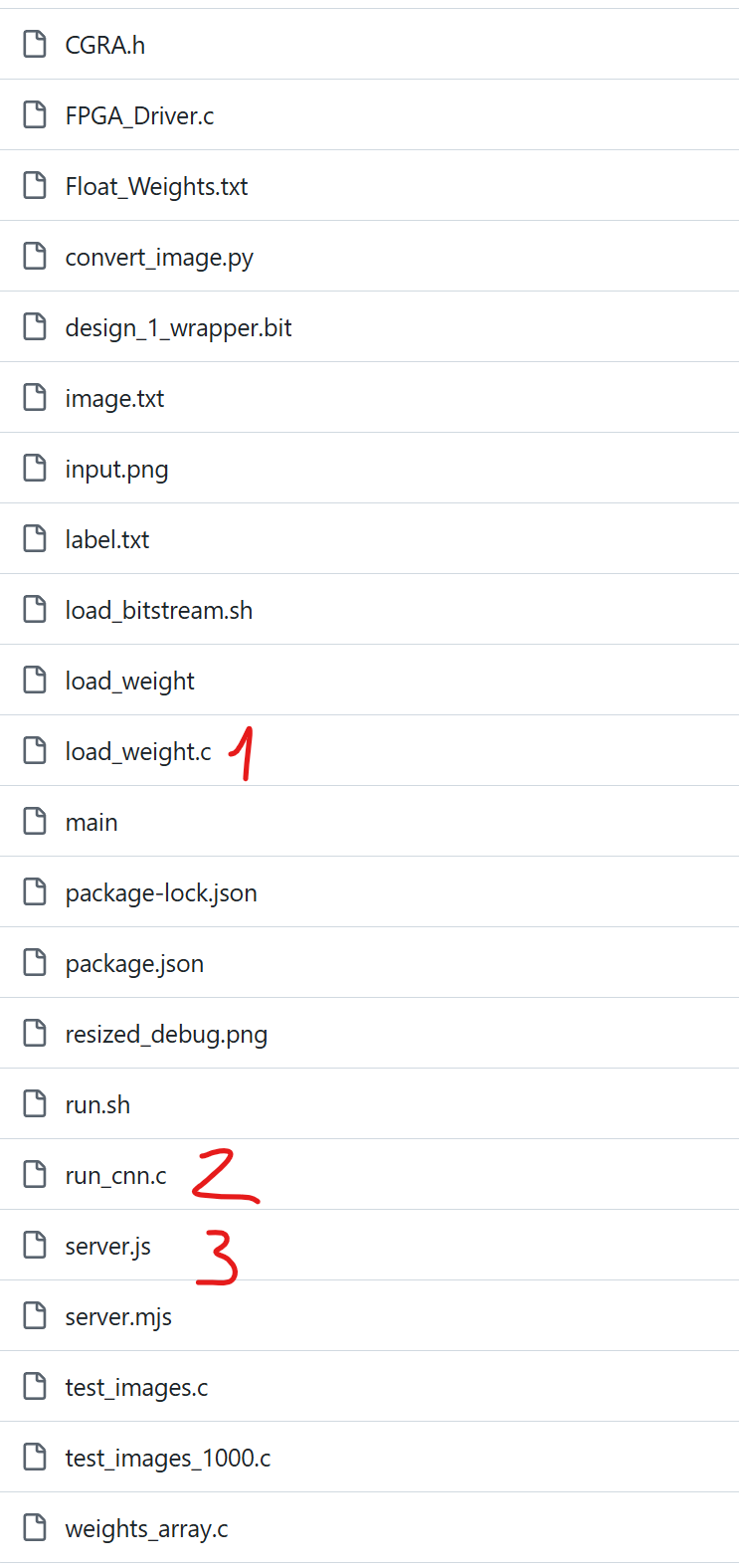
*gcc* [*cnn\_petalinux.c*](https://github.com/AISeQLab/Real-time_Handwritten_Digit_Recognition_on_FPGA/blob/main/Embedded_Software/Datatest_Recognition/cnn_petalinux.c) *– o main*

*./main*

## Real-time Recognition (Code C + Nodejs để host Websever)

***Toàn bộ code nằm trong link bên dưới, copy lên trên account Linux trên KV260 FPGA:***

<https://github.com/AISeQLab/Real-time_Handwritten_Digit_Recognition_on_FPGA/tree/main/Embedded_Software/Real-time_Recognition>



Trước hết chúng ta chia file [*cnn\_petalinux.c*](https://github.com/AISeQLab/Real-time_Handwritten_Digit_Recognition_on_FPGA/blob/main/Embedded_Software/Datatest_Recognition/cnn_petalinux.c)thành 2 file nhỏ là “load\_weight.c” và “run\_cnn.c” để tách riêng việc truyền các pre-trained parameter (weight) vào BRAM trước. Sau đó, “run\_cnn.c” sẽ chỉ chạy cho 1 input nhận từ cái số mà các bạn vẽ trên websever (chạy bằng file “server.js”).

Trước hết, các bạn compile 2 file “load\_weight.c” và “run\_cnn.c” như sau:

*gcc load\_weight.c -o load\_weight*

*gcc run\_cnn.c -o main*

Sau đó, chúng ta có thể chạy cái websever bằng lệnh bên dưới:

*sh run.sh*