CS10014 Computer Organization

Run C on Lab 4 CPU using FPGA

Tsung Tai Yeh

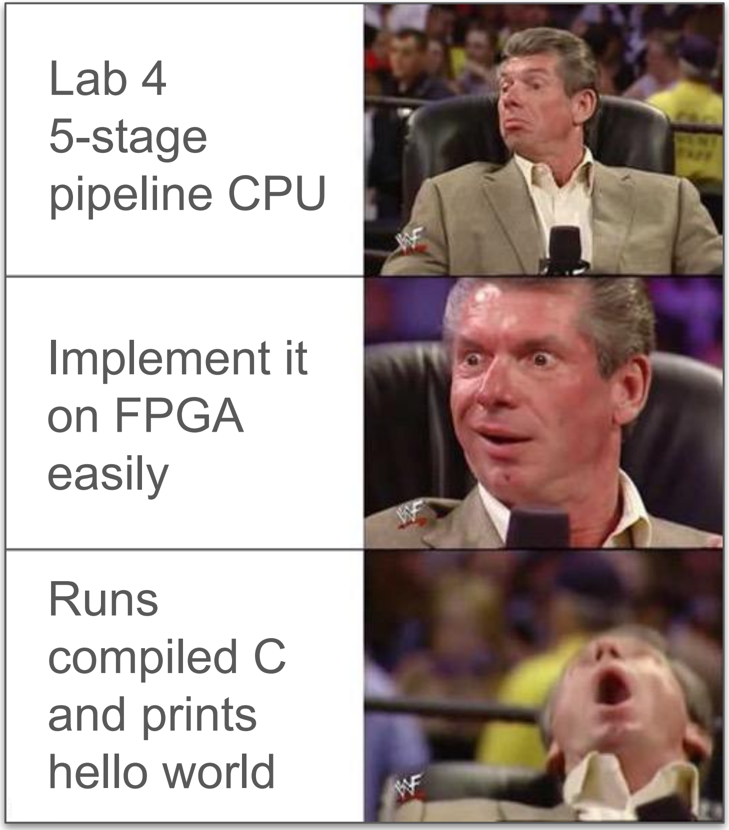
Department of Computer Science
National Yang Ming Chiao Tung University
Hsinchu, Taiwan

Don’t Panic

- This is not an assignment or lab, just a demonstration
- **You are not asked to do it**
- May be cool if you want to

Outline

- What is this?
- The FPGA We Are Using
- System Overview
- Software of the Lab CPU
- Details of the System
- Getting Hands-on with the Project:
    - Software Focus
    - FPGA & Simulation Focus

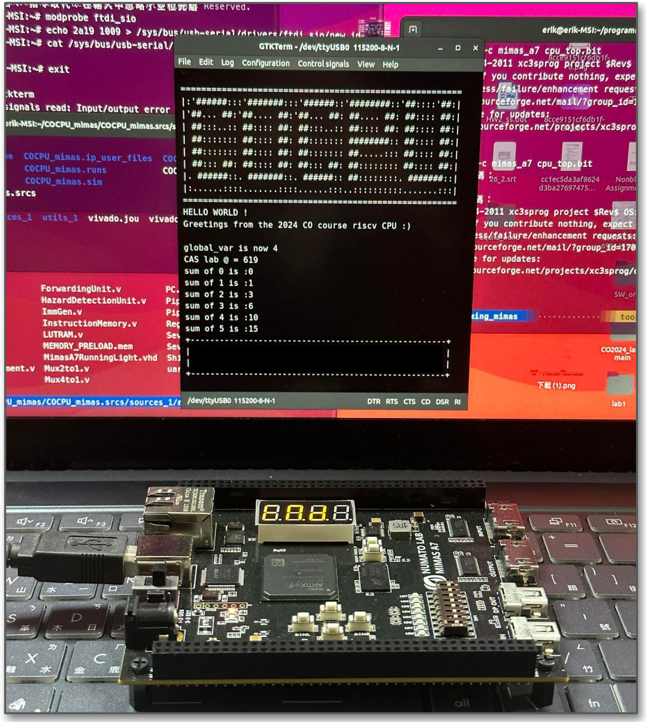


What is This?

A project for your CPU to run some very simple C on FPGA

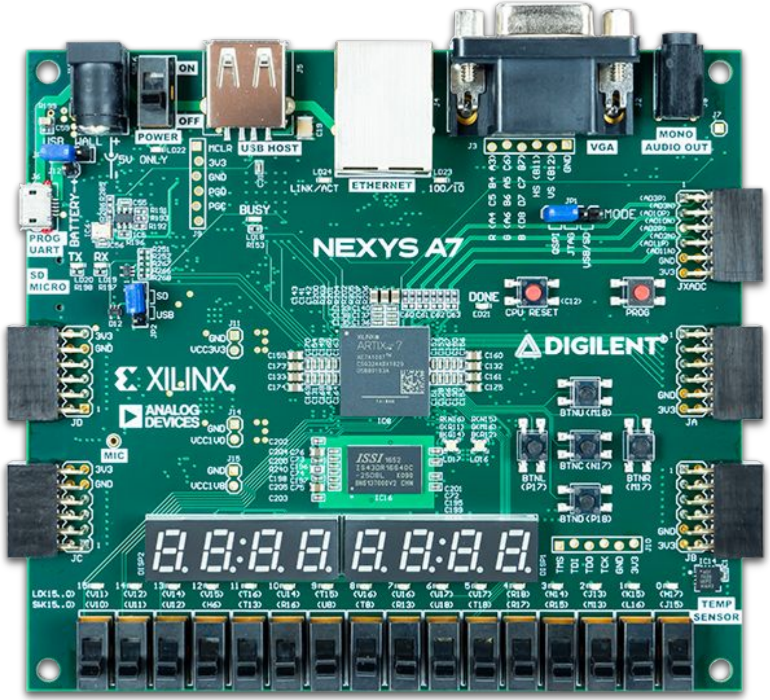
```
int global_var = 0;
int main() {
  printf("============================================== \n");
  printf("|:'#####:::'####:::'#####::'######::'##:::'##:|\n");
  printf("| ##:::..: #::: #: ##::..:: #::: ##: ##::: ##:|\n");
  printf("| ##:::::: #::: #: ##:::::: ######:: ##::: ##:|\n");
  printf("| ##:::::: #::: #: ##:::::: #....::: ##::: ##:|\n");
  printf("|. #####::. ####::. #####:: #:::::::. ######::|\n");
  printf("===============================================\n");
  printf("HELLO WORLD !\n");
  printf("Greetings from the 2024 CO course RISC V CPU :) \n");
  g_var += 4;
  printf(" global_var is now %d\n", g_var);
  printf(" CAS lab @ = %d\n", 600 + 10 + 9);
  printFib(7);          . . . and so on ...
```

Your lab CPU
On FPGA



< Just an example,
of course you
may write yours

The FPGA We Are Using — **Digilent Nexys A7-100T**

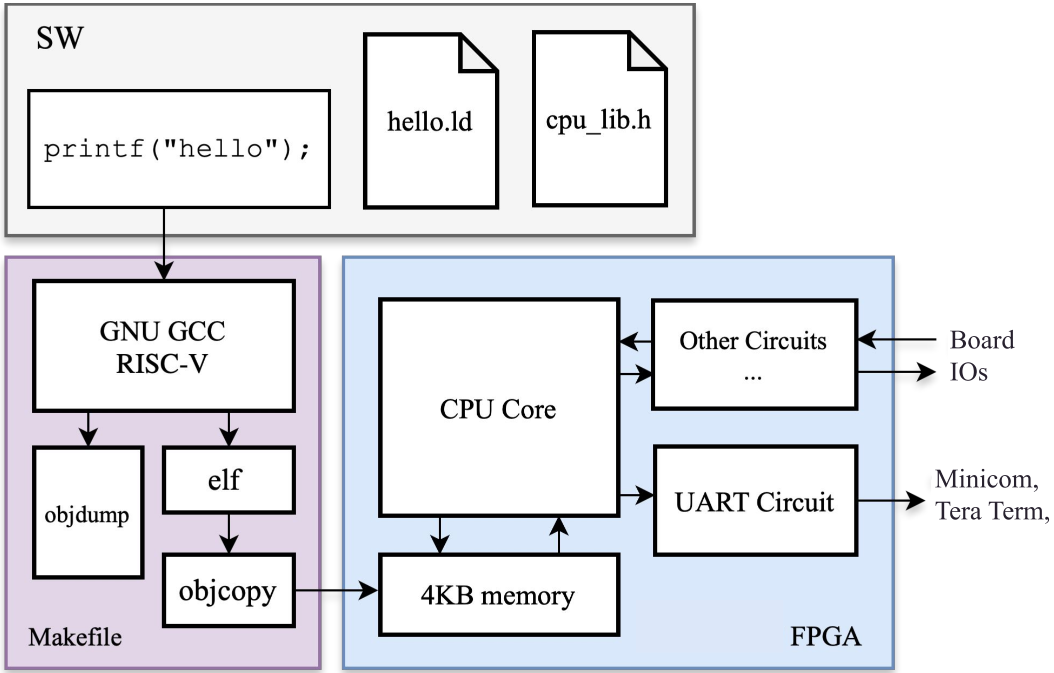


- **We use Xilinx Vivado to program the FPGA**
  - FPGA part number: `xc7a100tcsg324-1`
- The “`CPU_RESET`” button on the board (pin:C12) will be used as an asynchronous active low reset.
- Uses multiplexing to display the 4 7-seg digits
  - Our refresh rate for the 4 digits is 320 Hz.
- UART through the USB-RS232 Interface
  - Pin “C4” for UART RX and “D4” for TX.

System Overview

Contains following two parts:

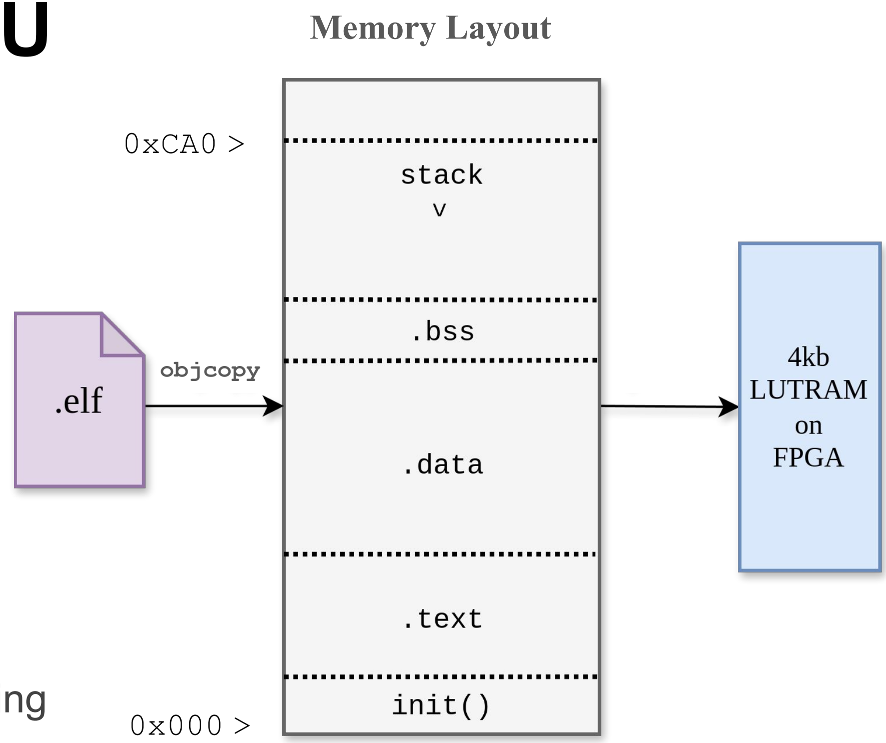
- ➢ Software
  - ○ minimal library for the CPU
  - ○ Linker script
  - ○ Makefile
- ➢ Hardware
  - ○ D-MEM / I-MEM using LUTRAM
  - ○ Vivado project for Nexys A7
  - ○ Top module w/ UART & stuffs
  - ○ Board constraints



Software of the Lab CPU

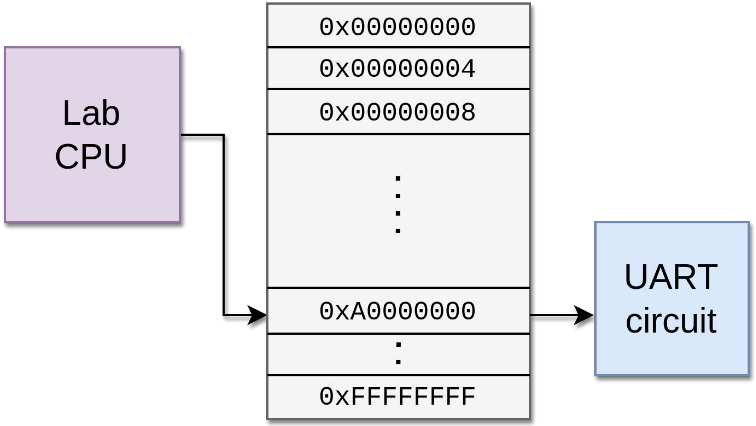
The software package contains:

- Minimal library for CPU
  - Half `printf()` for the system
  - `getchar()` to get char from UART
  - `init()` as init function
  - `exit()` after main
- Linker script
  - Memory layout
- Makefile
  - Integrates xxd, objdump, objcopy, compiling



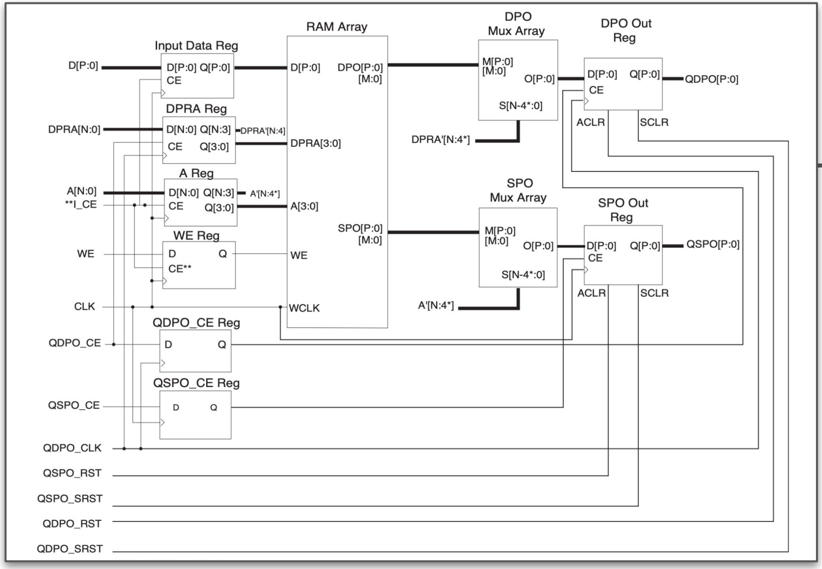
How `printf()` & `getchar()` in This System Works

- ASCIIs will be sent to UART circuit on top module by ‘sw’ instruction
  - Write some data to pointer that points to some specific address, then send to uart circuit
- Vice versa, CPU gets ASCIIs from register in UART circuit by ‘lw’

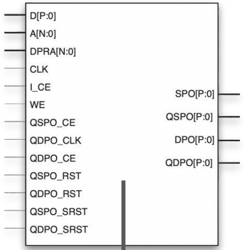


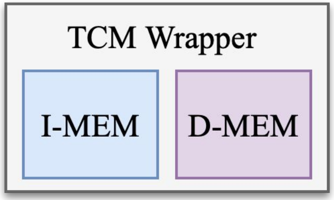
FPGA Resources for I-MEM & D-MEM

- ➢ To sync write & async read, we use the “Distributed Dual-Port RAM” IP by Xilinx, which is implemented by 2 distributed RAM resource of the LUTs (LUTRAM). (Ref: Xilinx Manual)
- ➢ 2 LUTs shares common write logic, but different address buses for reading (DPO, SPO).



Distributed Dual-Port RAM

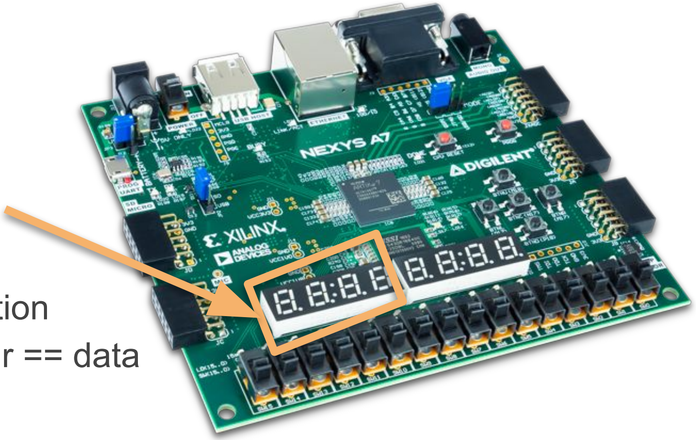




Figures are from Xilinx Manual.

For further discussions on LUTRAMS, check out this website, it contains details for SLICEM and stuffs.

7-Segment Self-Check Codes



- The 7-seg display provides some information about the system
  - We test some basic “sw” instructions before entering the main function
  - CPU writes some data to the self-check circuit, and check if sw addr == data

`t01`

No data written to the self-check circuit, “sw” or others may fail.

`E01`

Data written to the self-check circuit is wrong (addr != data), check the datapath of writing data memory.

`run`

Simple sw test passed, trigger “run” before entering main.

`End`

Returned from main, execution ends.

Getting Hands-On with the Project

Software Focus

Building the Software Toolchain

We need a cross compile environment to generate RISC-V asm.
Skip this slide if you build it already in lab 0!!

1. Clone the RISC-V GNU compiler from github.

```
$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
```

*The whole repo is around **6.5GB**, so cloning it may takes plenty of time!*

2. Compile the riscv32-unknown-elf-gcc toolchain with parameter rv32i, since our lab CPU supports integer and some basic instructions only.

```
$ ./configure --prefix=/opt/riscv --disable-linux --with-arch=rv32i
$ sudo make -j $(nproc)
```

Also takes plenty of time.

Writing Some C for Your Core

Under the SW_dev directory, there are the following files:

```
SW_dev/
 ├── cpu_lib.h
 ├── cpu_lib.c
 ├── hello.c
 ├── hello.ld
 └── Makefile
```

`cpu_lib.c`

Some minimal functions and parameters for the system to work.

`hello.c`

Write your hello world here.

`hello.ld`

Linker script to perform the specific layout for our CPU.

`Makefile:`

This makefile will export the PATH and env variables automatically.
Modify the makefile if your prefix at toolchain compiling is not default.

➢ Compile your C with:

```
$ make
```

Writing Some C for Your Core (Cont.)

After ‘make’, some files are generated…

- `hello.objdump`

```
000003e4 <main>:
 3e4:    fe010113            add   sp,sp,-32
 3e8:    01312623            sw    s3,12(sp)
 3ec:    57800513            li    a0,1400
 3f0:    00112e23            sw    ra,28(sp)
                                       And so on
```

You may check if there are unsupported instructions generated.

Beware that some of them are pseudo-instructions.
- Here’s a useful tool to check.

- `mem_preload.mem`

```
00000011001001110000000001010100
00000011001001101100000001010011
10000011001001101000000001010011
00010011000000010000000111111110      And so on
```

Contains 32-bit binaries to load in memory, just like the previous lab.

Getting Hands-On with the Project

Hardware & FPGA Focus

Additional Requirements for Your Core

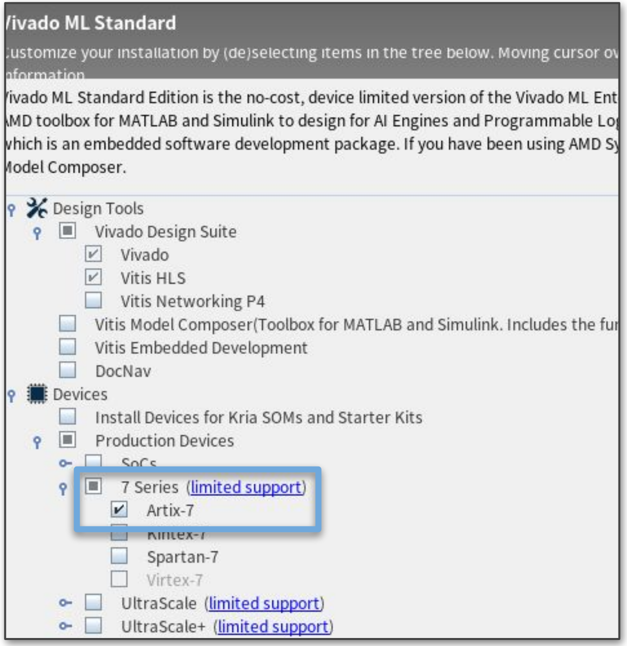
➢ In addition to the lab 4 instruction requirements, there are still some other instructions that are required to implement for running the demo C program.

- `LBU [Load Byte Unsigned]`
  - `putc()` in `printf()` sends one char at a time to the UART circuit.
- `LUI [Load Upper Immediate]`
  - For CPU to load some numbers larger than 4096.
    Ex. `for(int i = 0; i < 4000; i++)` , compiler sometimes ‘lui’ loads 4096 then -96 to get 4000.
    (riscv32-unknown-elf-gcc -O2)
- `SLL [Shift Left Logical]`
  - 1 << n is same as 2^n but faster.

➢ **Finish them to run the demo C program!**

Installing Vivado

1. Install Vivado w/ package for Artix-7



2. Download the board files for Nexys from Digilent github repo

```
board_files/
├── ...
└── nexys-a7-100t/D.0/ << Download this
```

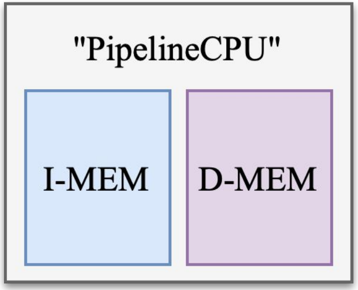
3. Place it under:

/path/to/Vivado/{version}/data/xhub/boards/XilinxBoardStore/boards

```
boards/
├── Xilinx/
├── Digilent/
│    └── nexys-at-100t/ << put it like this
```

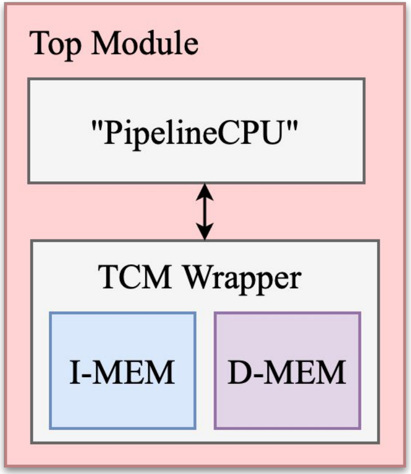
Connecting to the Top Module

Delete the I-MEM / D-MEM instances in your core, and connect those wires to outside of the core.



The Lab 4 Hierarchy

Modify it!



Hierarchy of this Project

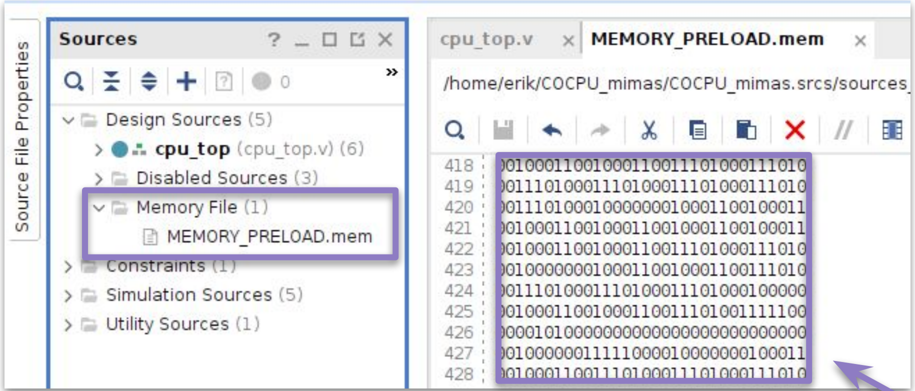
(IO definition from previous labs)

```
module PipelineCPU
(
    input clk,
    input start,
    output [31:0] r [0:31]
);
```

```
module PipelineCPU (
    input          clk
    input          start,
    output         byteMask,
    output         memWrite,
    output         memRead,
    output  [31:0] address,
    output  [31:0] writeData,
    input   [31:0] readData,
    output  [31:0] readAddr,
    input   [31:0] inst
);
```

Self-check code "t01" & "E01" will be triggered if you don't connect them correctly!

Preload the Compiled Binaries to Memory



Load the compiled binary file to the MEMORY_PRELOAD.mem in the vivado project, just like loading the instructions in previous lab.

Now they will be preloaded to the distributed RAM which we mentioned previously.

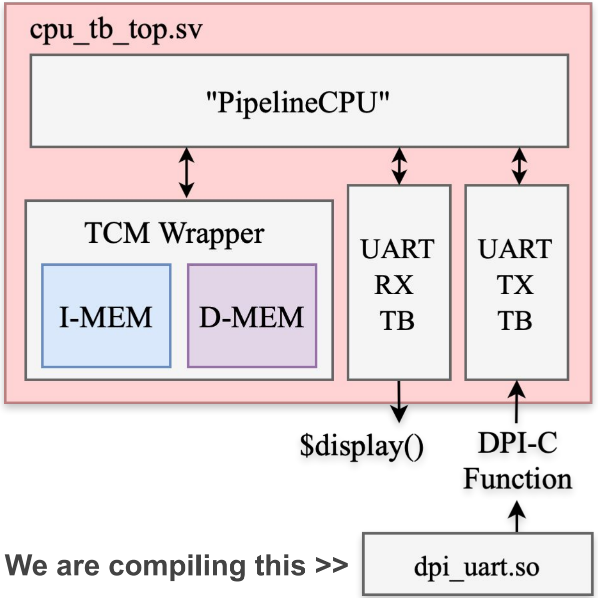
mem_preload.mem

```
00000011001001110000000001010100
00000011001001101100000001010011
10000011001001101000000001010011
00010011000000010000000111111110
```

Then generate bitstream

Simulation First! – DPI-C Functions

After porting your core on the vivado project, you may want to do some simulation first.



We are compiling this >>

To simulate the UART TX is easy, $display() or $write() is all we need.

But when it comes to RX, there are no corresponding interactive stdio function in verilog, so we are using the DPI-C function in verilog to get data from stdio.

Go to:

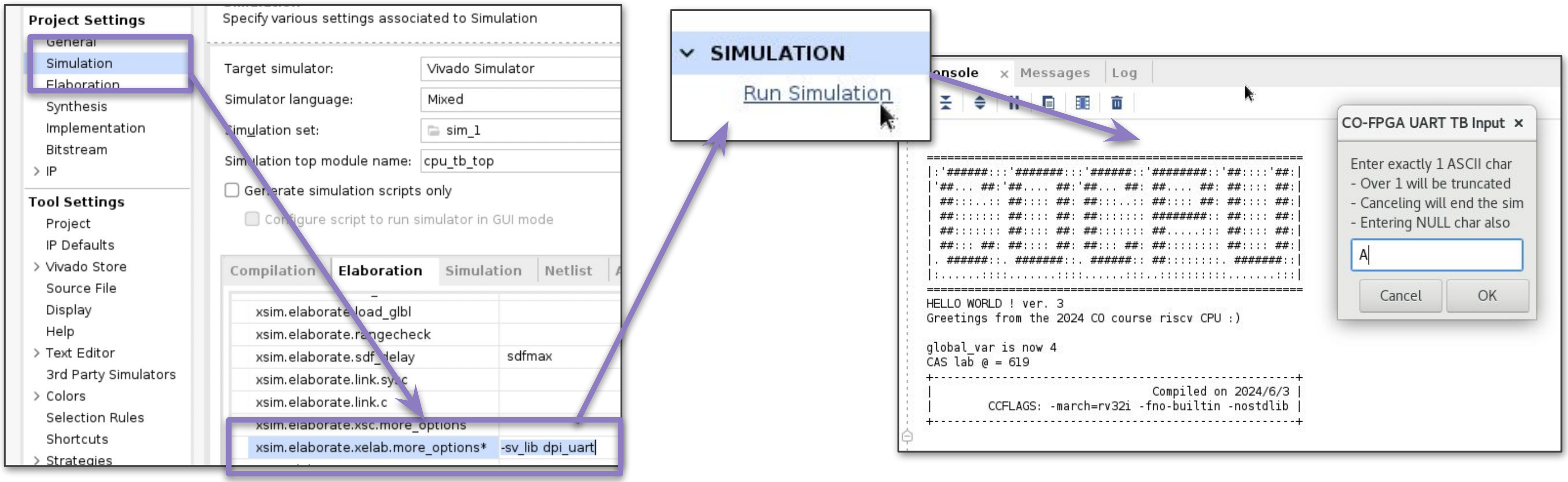
/path/to/CO-FPGA/CO-FPGA.sim/sim_1/behav/xsim

and

```
$ make
```

Simulation First! – DPI-C Functions

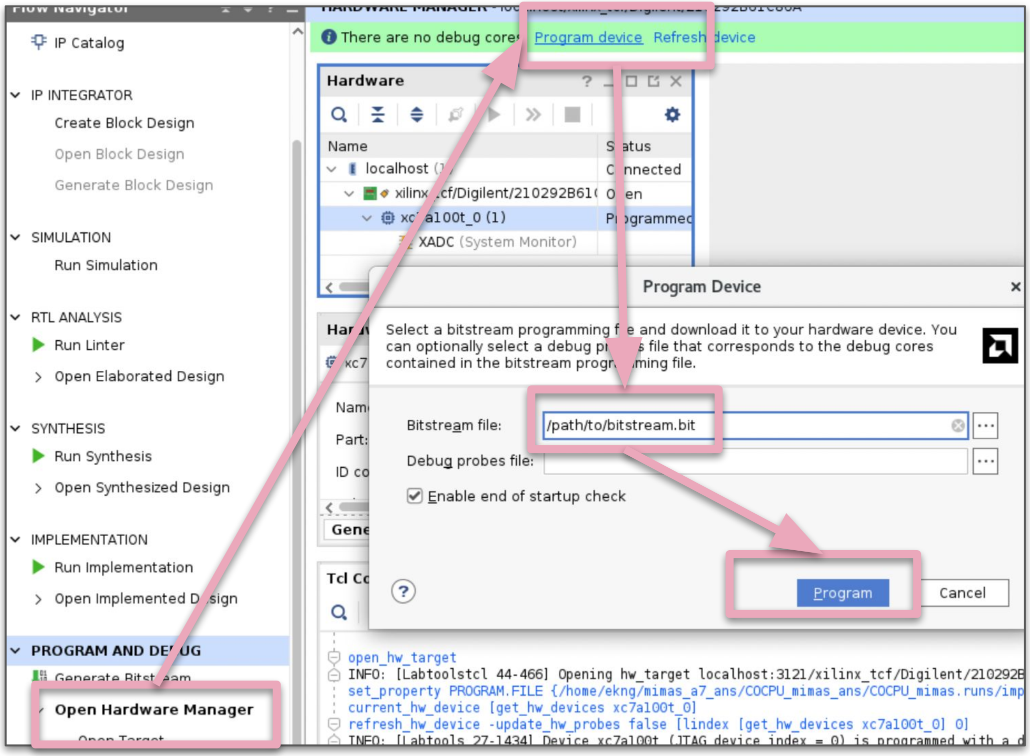
After porting your core on the vivado project, you may want to do some simulation first.



Make sure the xelab options contains the dpi shared library

You can now enter ASCIIs in this window, and observe output from the console!

Programming the Bitstream to FPGA



For the last step, open hardware manager and program the bitstream to the FPGA.

Then open the Minicom we has just configured, now you may see the output now!

If you are using `getchar()` in your program, simply type on Minicom and it will transmit the ASCII through UART to the RX circuit on FPGA.

Configure Minicom to use UART

Minicom is a light and useful tool for doing the receiving and transmitting works through UART.

1. Install Minicom and plug in the board.

```
$ sudo dnf install minicom-2.7.1-9.el8.x86_64
```

or

```
$ sudo apt-get install minicom
```

2. Run Minicom and select “Serial Port Setup”.

```
$ sudo minicom -s
```

3. Make sure config matches (115200 8N1), then select the correct serial port.

```
+-----------------------------------------+
| A -    Serial Device      : /dev/{board}  |
|                                         |
| C -   Callin Program      :             |
| D -  Callout Program      :             |
| E -     Bps/Par/Bits      : 115200 8N1  |
| F - Hardware Flow Control : Yes         |
| G - Software Flow Control : No          |
|                                         |
|   Change which setting?                 |
+-----------------------------------------+
```