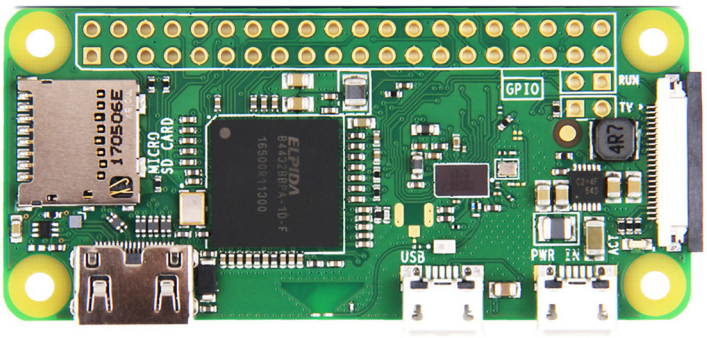
University of Cape Town



EEE3096S/EEE3095S

Embedded Systems II

Work Packages 2021

---

28 October 2021

Abstract

Welcome to the practicals for EEE3096S. These instruction are applicable to all practicals so please take note! It is critical to do the pre-practical/tutorial work. Tutors will not help you with questions with answers that would have been known had you done the pre-practical work. The UCT EE Wiki (wiki.ee.uct.ac.za) is a very useful point to find any additional learning resources you might need. Use the search functionality on the top right of the page.

Practical Instructions

- **Naming**
  All files submitted to Vula need to be in the format. Marks will be deducted if not.

```
pracnum_studnum1_studnum2.fileformat
```

- **Submission**
  - Work packages consist of a tutorial and a practical. Be sure you submit to the correct submission on Vula!
  - All text assignments must be submitted as pdf, and pdf only.
  - Code within PDFs should not be as a screenshot, but as text.
  - Where appropriate, each pdf should contain a link to your GitHub repository.
- **Groups**
  All practicals are to be completed in groups of 2. If a group of 2 can't be formed, permission will be needed to form a group of 3. You are to collaborate online with your partners. See more here.
- **Mark Deductions**
  Occasionally, marks will be deducted from practicals. These will be conveyed to you in the practical, but many will be consistent across practicals, such as incorrectly names submissions or including code as a screenshot instead of formatted text.
- **Late Penalties**
  Late penalties will be applied for all assignments submitted after the due date without a valid reason. They will work as 20% deduction per day, up to a maximum of 60%. After this, you will receive 0% and have the submission marked as incomplete.

Contents

1 Work Package 6 **3**

1.1 Practical: A simple CPU 3

1.1.1 CPU in Verilog0 3

1.1.2 IoT application 7

Chapter 1

Work Package 6

An important note about this work package and EEE3095S students. Computer Science students are required to carry out additional work in this course due to a higher credit rating. This tutorial and practical include these additional tasks and are therefore required for all students registered for EEE3095S. EEE3096S students are strongly encouraged to at least look through and possibly also perform these additional tasks for their educational value, but are not required to do so, demonstrate such, or submit handins for these additional tasks. All EEE3095S additional tasks are clearly indicated as such, and students are to complete all tasks unless otherwise indicated.

1.1 Practical: A simple CPU

As in the tutorial there are 2 sections to the practical, the 2nd portion is only required by EEE3095S students.

In this practical you will:

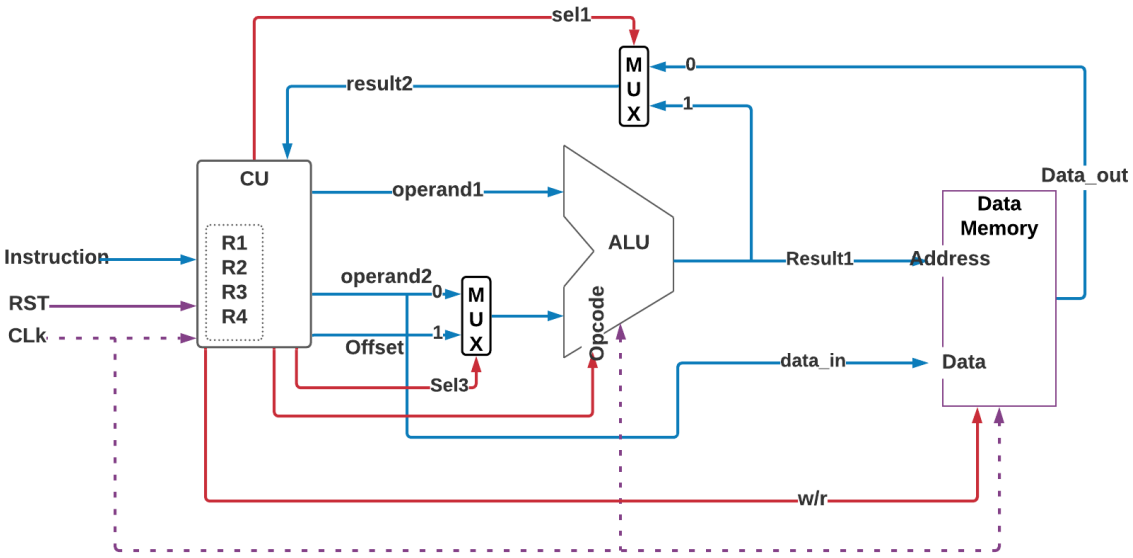
- Complete a control unit
- Edit a test-bench for the full CPU
- EEE3095S students only will also implement an algorithm in Python and deploy it via Balena

1.1.1 CPU in Verilog0

Building on the Tut and WP5, the git repo now contains:

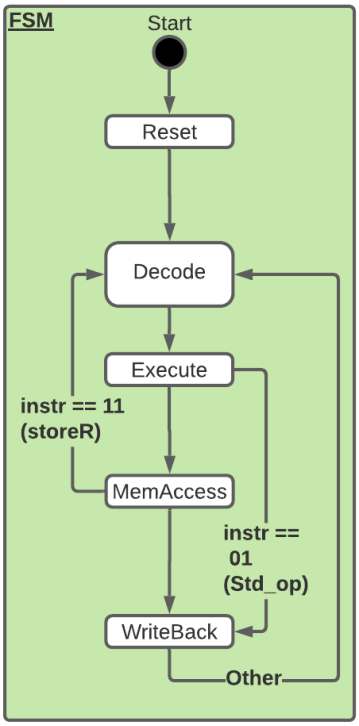
1. alu.v // A very cut back ALU

2. CU.v // A control unit for the simple CPU shown bellow with the code for storeR removed which you will need to complete
3. RegMem.v //The same as from the tut, this is unchanged and functioning
4. top.v //A file that connects all the modules into a simple CPU
5. RegMem_tb.v //The same functioning test bench for the RegMem module
6. tb_simple_CPU.v //A functioning test-bench for the simple CPU in top



Thing to note and understand before beginning

This is a custom simple CPU so you need to understand both the pipeline it uses and the instruction structure it assumes. Review Week 10 of lectures (instruction and ALU essentials in E1_CPU_Architecture1.pdf, pipelining aspects in E2, not that we are going to a complex pipeline design) to remind yourself of these aspects. Below are figures showing both:


FSM
Start
Reset
Decode
Execute
instr == 11
(storeR)
MemAccess
instr ==
01
(Std_op)
WriteBack
Other

Instruction Structure

Instr (2)	X1 (2)	X2 (2)	X3 (2)	Offset Max(DATA_WIDTH, ADDR_BITS)	Opcode (4)
19:18	17:16	15:14	13:12	11:4	3:0

Instr codes:
00: reset
01: std_op opcode: 0000 (+) or 0001 (-)
10: loadR opcode: 0000
11: storeR opcode: 0000

std_ops
  ADD X1, X2, X3 X1 = X2 + X3
  SUB X1, X2, X3 X1 = X2 - X3

StoreR
  STR X1, [X2,20] Store X1 contents in address X2+20

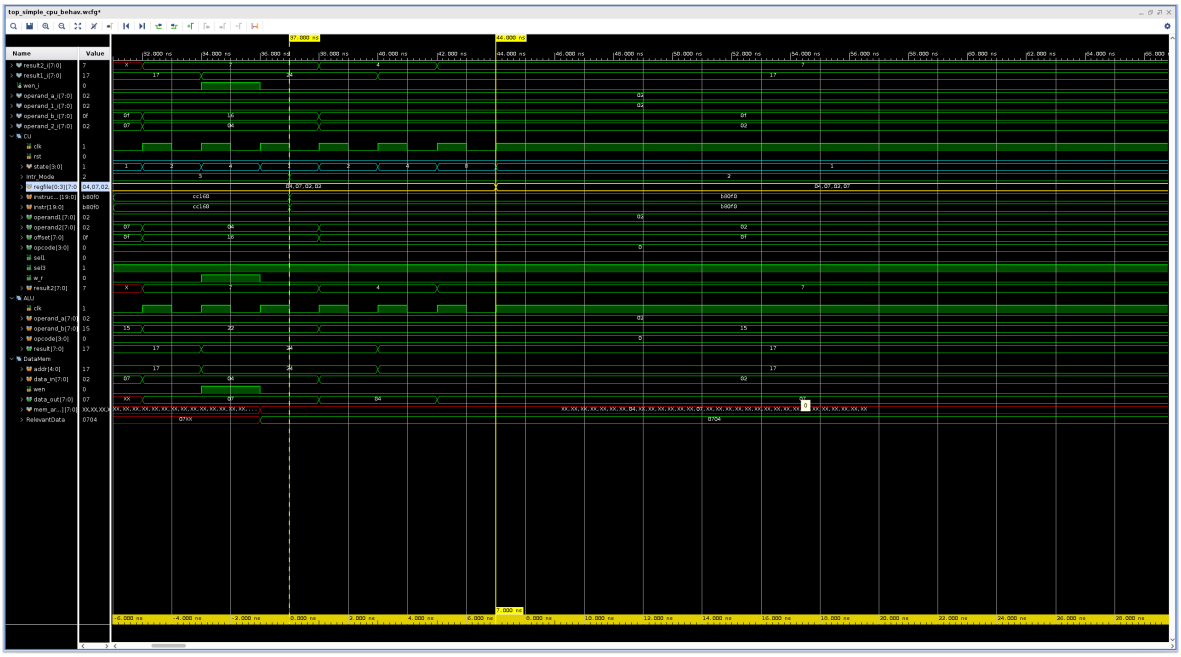
LoadR
  LDR X1, [X2,20] Load contents of mem address X2+20 into X1

The above are available at the top level in WP6 folder, but are also available again in a Vivado project. If you use EDAPlayground, some hints:

- If you drag and drop the source into the EDA window it will copy all modules into a single file. Alternatively, you can upload each file, and in top.v use 'include ¡filename¿ for each other file
- Use dumpvars() to get all available ports in the wave viewer not just those in top module. Note: You aren't able to see the internal signals in EDA so will need to create a new port in the CU module to pass the regfile up to top.v so you can see it in the simulator wave viewer. This will involve editing the CU module in CU.v and its instantiation in top.v. In Vivado you can see subsignals directly (see bellow, or for a better view, see

the original in github). In the screenshot the regfile contents are shown in the yellow signal.

- Note that the different operations take different amounts of time (number of cycles).



Prac Deliverables

A pdf containing:

- A link to a **public** git repo containing a CU with the storeR functionality restored. When this is working, the provided testbench will result in the final load instruction fetching 7 back out of the data memory and loading it into reg3 such that the regfile will contain: 4,7,2,7
- The repo must also contain an extension to the provided testbench that adds a second loadR instruction
- Simulation screenshot(s) showing the functioning of both of the above. If using EDA you will need to follow the instruction above to expose the regfile to the top module so it's visible in the waveform.
- Make sure your code is appropriately commented

1.1.2 IoT application

Again, the following section is only required of EEE3095S students.

Now that you have a webserver running on your Pis and a development pipeline that uses the BalenaEcosystem. In your teams you're going to configure 1 pi as the IoT sensor node, and use the other to represent a server. In a real world deployment you wouldn't use a Pi as a server but the toolchain and development process are appropriately representative.

Sensor Node: Pi1 In WP4 you configured your Pi to sampling a temperature sensor and Light dependent resistor (LDR) connected to the ADC every 10s. Your 2 tasks are:

First: Using your code and circuit from WP4, configure your Pi to do the same as it was doing before but using a Balena container to deploy the code. Note, the same as you needed to enable SPI and I2C in Raspberian, you need to expose the GPIO pins to the Balena container that your code is running in. See documentation on doing so here.

Second: Once you have your old code working and deployed again in the Balena container. Modify the code using Python Sockets to send the values over TCP. here's a good starting point example. You're going to be sending the data to the other Pi, **Pi2**, which is going to be still hosting the webserver but with improvements. You'll need to take note of device IPs. You can either hard code the IP of the remote Pi2 webserver, or if you find you have time you can learn to use Python Config.

It is strongly recommended that you complete everything else 1st and only look at using config if you still have time/want to. It is **not** required. If you do choose to use a config file, the design is that you can store this file in the git repo where your src is stored and then you can update it with new IP addresses directly using git. Once updated with the latest IP for the Pi2webserver, you can push a rebuild image.

Debugging the Balena container using ssh: In the tut deployment should have been quite easy. However in both of the above tasks you're likely going to encounter times when it becomes hard to work out what's gone wrong and why your container isn't starting. Two suggestions for debugging success:

- Make sure you're codes working before containerising and deploying. This is a good strategy in general. Potentially this might require you go back to Raspberian. Perhaps configure 1 Pi with Raspberian 1st temporarily until the first task is complete.
- You might have seen that you can ssh into your device from both the webportal and the commandline. Read the documentation here to follow examples. Note that you can ssh into both the host OS (similar to ssh'ing into your Pi when it was running Raspberian), and the container itself which is being hosted by the OS. Be aware of which you have ssh'd into so you know what to expect to find available and running.

'Server': Pi2 The Pi2 platform will operate as both a network cloud server, a sink (i.e. gathering data) for recording data being sent from Pi1. It will also provide a web interface

so that a user can view the status of the server, check that samples are being received, and request a simple report.

Considerations for TCP comms between Pis

As per the tutorial, the send operation will send a block of data (you don't need to add additional flags unless there are needs to do so for specialized comms purposes you want to use). You could make the communication between client (P1) and server (P2) text-based, that can help with debugging but it can be inefficient for sending sensor data as text. If you do want to send text messages to test your comms, you need to suitable encode Python strings using encode, e.g. s = "text"; clientSocket.send(s.encode()); (further details on sending types of data can be see on https://pythontic.com/modules/socket/send). But the simple tutorial should be sufficient.

The Pi1 sensor needs to send a message each 10s to the Pi2 server. Remember the send command just sends a block of data. The Pi1 can start sending samples by default when turned on, but if time permits adding a function to turn on/off sampling (the SEND command below) is indicated. You need to structure the data sent to carry information about what message it is and what data is carried for that message. In effect, you need to develop a simple message protocol, as in a few commands, with will be send as messages over the TCP socket.

You could construct your packets transmitted with send to have the following structure:

Command	Length	Data
1 byte	1 byte	Length bytes

Table I: Suggested message structure

As suggested above you could send a message in the form indicated above. You could start the message with just a character to say what message it is. Then, if the message is going to contain data (which a sensor data message would need to do) then you could add a length value to indicate how many bytes will follow the length (if you need to send more than 255 bytes in a packet you might want to change the packet structure so that Length is longer, e.g. has low byte and a high byte).

Sensor Data Message:

You need at lease these messages:

Server to Client Messages (P2 to P1)

- SEND : Allow for the server being able to turn on or turn off Pi1's sending of sensor data, see *sending* flag below. (You can have two separate messages, e.g. SENDON,

SENDOFF or you can use a parameter to indicate the stage, make sure you document your approach).

- CHECK : Check that the client is active (the P1 should returns a STATUS message if it is on, see below).

Client to Server Messages (P2 to P1)

- SENDACK : (optional) You could return an acknowledgement from receiving a SEND so that the server knows the request got through (this is a little redundant at the CHECK can be used for this).
- STATUS : The client replies to the server indicating if its sampling of data is active, and the time (in HH:MM:SS) at which the last sample was sent.
- SENSORS : this sends back sampled sensor data, the sensed temperature value, the sensed LDR value, and the time, i.e. a timestamp, in HH:MM:SS. You can decide how to package this into the data block of the TCP message (e.g. you could send two 16-bit values, the temp an the LDR).

sending flag: Remember that the client, Pi1, needs a flag, which you can call sending, to say if it is sending SENSORS data to the server. On system startup this should be on (i.e. set to 1).

Data stored on the server

The Pi1 will read temperature and LDR sensed light level. If you have a MCP3008 this should be able to give you a 10-bit reading. You can store this in an int variable, although you can save memory using a short int if you like. You need to then send this every 10s (when sending is active).

On startup the server Pi2 needs to create a *sensorlog* file to save logged sensor readings. You are suggested to simple name the file in the format *date-time*.csv, where date is the current date e.g. DDMMYY (day, month, year) and time is HHMM (hour and minute). Alternatively you can just use the same file, like sensorlog.csv to save time.

When the server Pi2 receive the SENSOR messages, it needs to add an entry to the *sensorlog* file. Depending on your time availability, you could just put the sensor readings in an array and save that array to the csv on exit or when the array gets full, i.e using the array like a buffer (but this is not perfectly reliable, it is preferably to append an entry to the file each time a sensor reading is obtained).

Web Interface

The P2 server needs to provide a simple web interface. This can be an entirely text-based interface. It should provide the following facilities, these can be provided either by hyperlinks

that activate functions or buttons on the web page:

- **Sensor On**: Turn on the sensors (i.e. send a SEND message to tell Pi1 to start sampling).
- **Sensor Off**: Turn off the sensors (i.e. send a SEND message to tell Pi1 to stop sampling).
- **Status**: use the CHECK message to see what the status of the Pi1 is (e.g. is it sampling, when did it last send a sample).
- **Log Check**: This function needs to print out the last 10 samples from the current run (or however many there are if less than 10 samples) to the screen.
- **Log download**: Allow user to download the current *sensorlog* file.
- **Exit**: This just exits the server program. It could show a screen exited.

The following illustration shows the overall system design for this development, showing the separation and communication between the P2 Sever and the P1 Client.

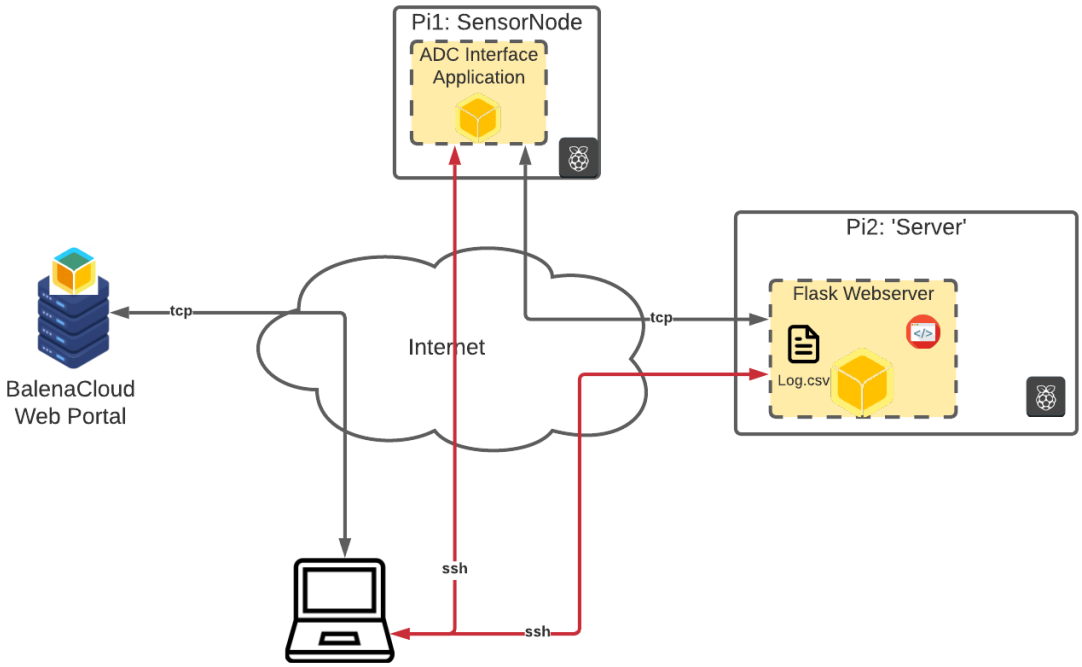


Figure 1.1.1: Illustration of client-server IoT system

Report Deliverables on Pi1

1. Cover page with team names
2. Introduction indicating structure of report
3. Link to the git repo(s) containing the Dockerfile and source respectively for Pi1 collecting sensor data and transmitting it to Pi2
4. Link to the git repo(s) containing Pi2's webserver Dockerfile and source.
5. A screenshot of the Balena webportal showing both Pis up and running
6. Ensure both repos have good READMEs and commented code
7. A screenshot of your webpage providing the 6 functions required.
8. A short (<3min) video clip showing the webpage with each of the 6 actions in operation. Post the clip in one of the git repos.