Lab Session 1

Introduction and Bare-metal ARM programming

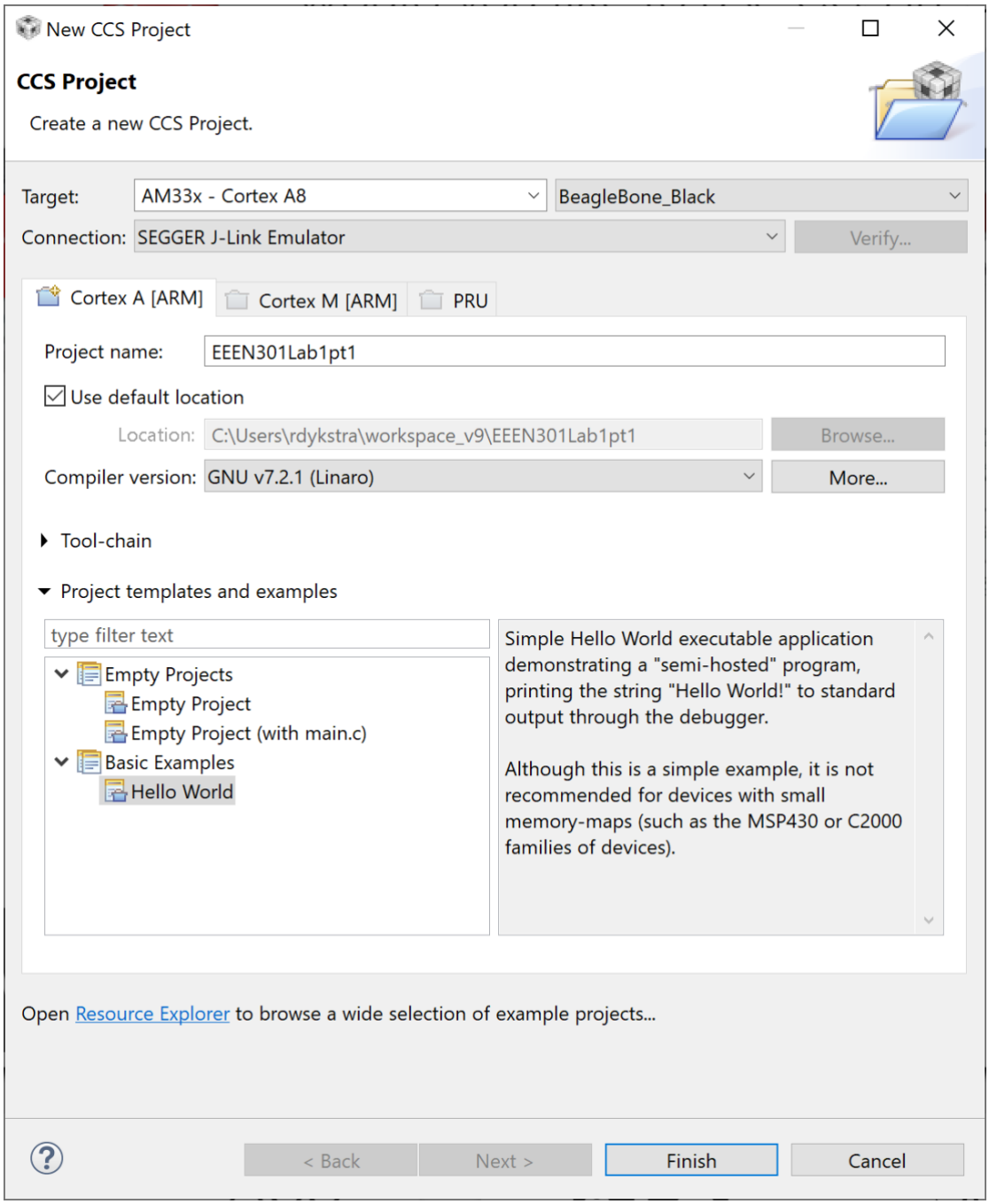
Up to now we have been focusing on the use of a relatively simple 8bit 8051 series microprocessor. While this may be suitable for some low performance applications, it is obvious that for more demanding applications such as a mobile phone we need something more powerful. This lab will be rather easy and is intended to introduce the Texas Instruments Sitara AM3358 ARM Cortex-A8 32bit microprocessor and development environment as well as the Beaglebone Black development board.

For this lab your development board is connected to two of your PC's USB ports. One USB port is used to host the JTAG debugger and the second USB port will be configured as a RS232 serial port. The JTAG debugger tool will connect into the Texas Instruments Code Composer Studio software development tool and we will use it to download and execute code to the Beaglebone board. The RS232/USB module will connect a UART on the Beaglebone board to a terminal program such as Putty.

For this first lab we will be writing and compiling C code for the ARM processor but without using an operating system. This is known as “baremetal” programming as the only code within the system is the code that you write and compile. This is exactly like what we were doing with the 8bit, 8051 processor in the previous labs. The advantage of this method is that you have full control over the resources and you often end up needing less memory. The disadvantage is that you have to write everything.

For this lab you will need the following bits:
1. Beaglebone black development board with JTAG connector attached.
2. 5V power adapter.
3. Segger J-Link USB-JTAG module + JTAG cable and USB cable.
4. TTL-232R-3V3, UART to USB adaptor cable.

Step 1: Launch Code Composer Studio on your PC. It has a Rubik’s cube icon. Select the “project” menu and then select “New CCS project”. A dialog box will appear as below.

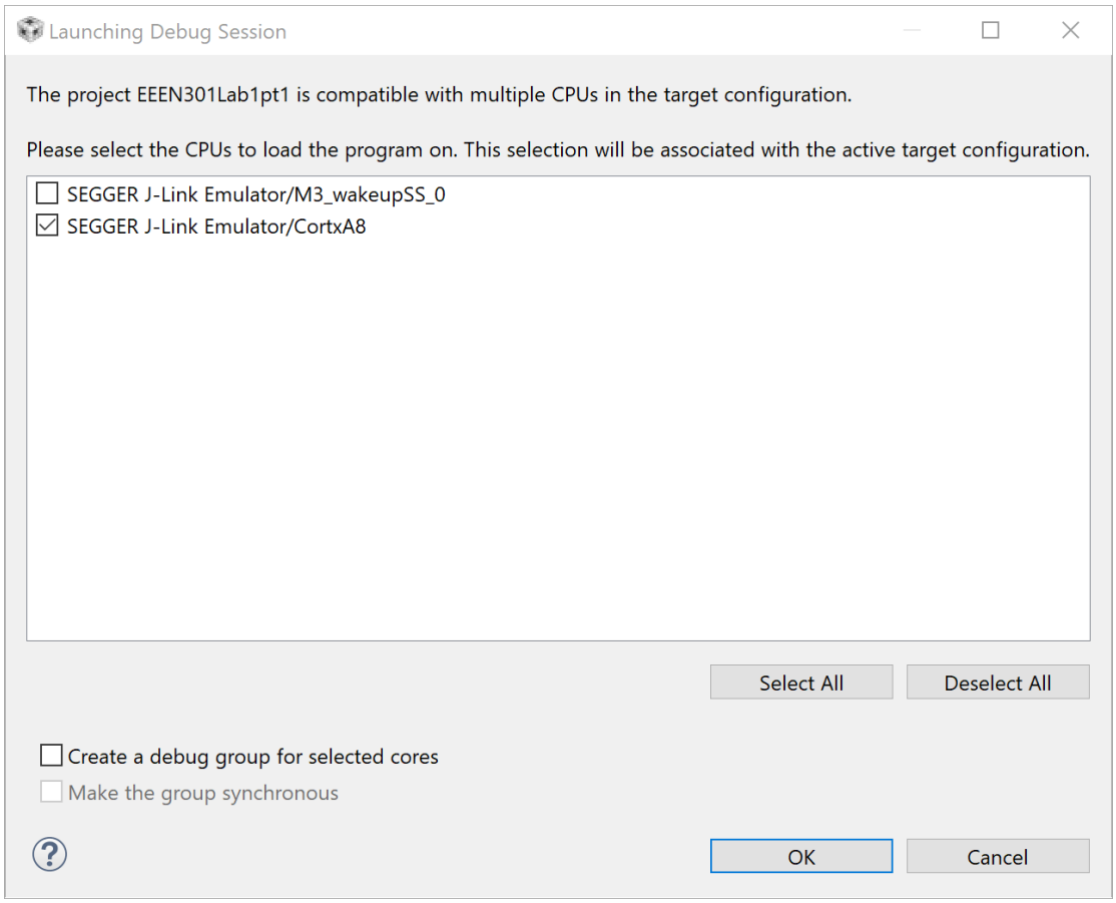


Choose the processor type, debugger connection, project name, project location, compiler version and project template as per the picture above. In your case you will choose a location on your H drive. Click finish when you are ready. Your project will now be automatically generated. Expand your project tab within the “Project Explorer” window and examine the project contents. The “hello.c” file contents should be visible in a text editor window.

Step 2: Locate the button "S2" on the Beagle bone black board. It is the button at the opposite end of the board from the "S1" RESET and "S3" Power buttons. Press this button and at the same time connect the 5V power supply. Continue pressing this button for at least 30 seconds. The Beaglebone black has a preloaded Linux operating system stored in Flash memory and this normally boots up when power is applied. By pressing "S2" when applying power we are forcing the Beaglebone black to try and boot from a different source. In our case there is nothing to boot so the processor halts.

Step 3: Connect the Segger JTAG debugger cable to the JTAG port of the Beaglebone black board. You will notice that the connector is keyed. Next, connect the USB cable of the JTAG debugger to a USB port of your PC.

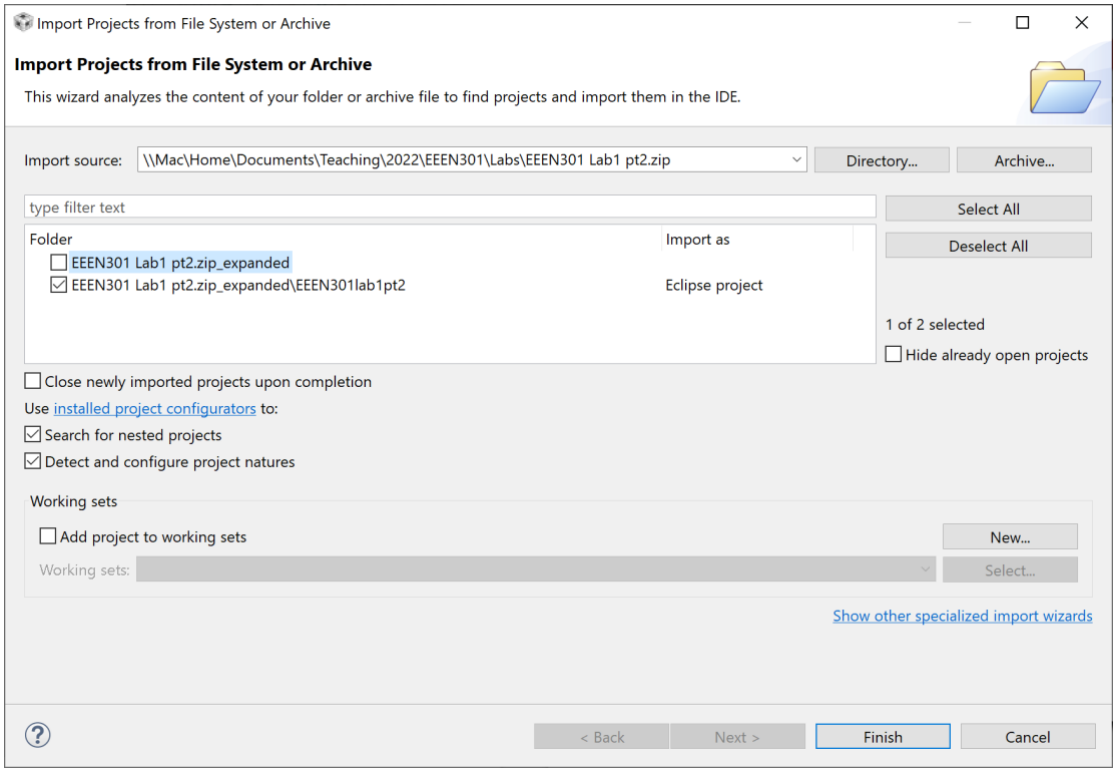
Step 4: In Code Composer Studio, click on the little bug symbol to launch the debugger. Click "accept" when the "terms of use" window pops up. The project will be compiled again, but for debug purposes and the interface will change to a "debug" view environment. The first time the project is "debugged" the window below will appear. Deselect the first box and click OK.



Step 5: Under the "Window" tab select "Show View" and then select "Disassembly". A new window should appear showing the disassembled code. The program counter will be set to the start of the program.

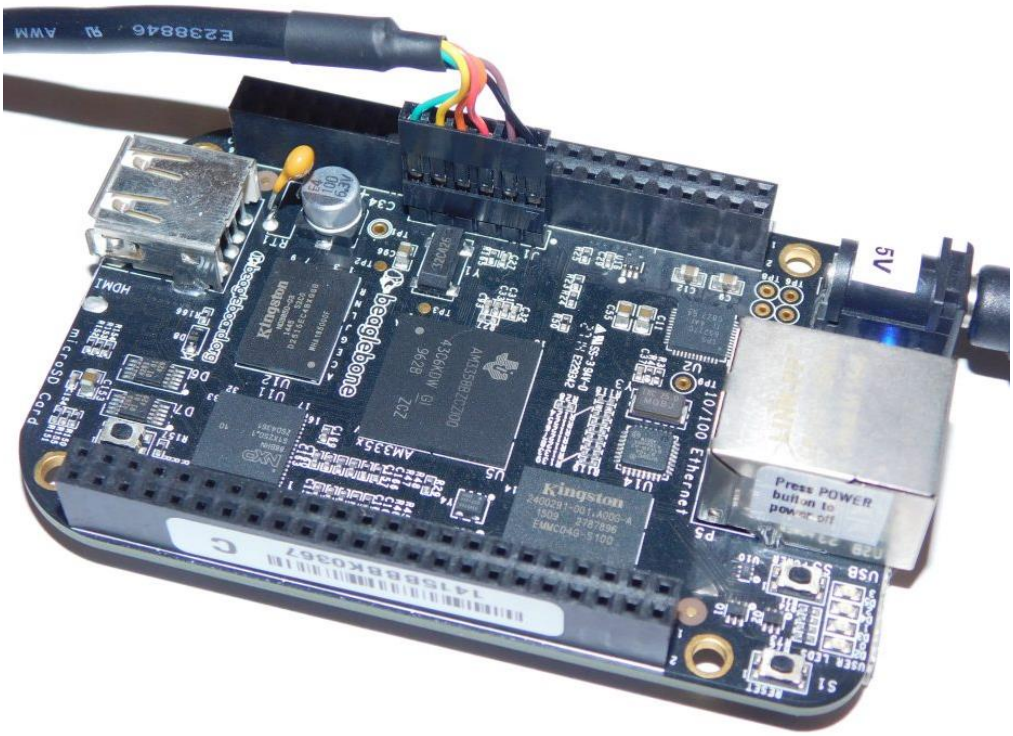
Step 6: Click the green arrow (towards the left of the tool bar). The code will then execute and you will then see the message "Hello World!" in the debug console window. Click the red box to terminate the debug session. Try changing the message and then run the code via debug. When finished, right click on the project within the project explorer and then select "close project".

Step 7: Download the zip file: EEEN301 lab1 pt2.zip. In Code Composer Studio, select file menu and then "open projects from file system". Click the "Archive" button and then select the zip file. Finally, unselect the first option as per the figure below and then select finish.



A new project will then be loaded. Double click on the project name to make it active and then launch the program via debug. When it runs you will see the 4 blue LEDs flash. Experiment with the "LEDs.c" code, change the rate of the flashing and make some new patterns.

Step 8: Download the zip file: EEEN301 lab1 pt3.zip and load into Code Composer Studio.



Connect a USB/Serial cable as per the figure above. Open a terminal program (Tera-term or Putty) and set the BAUD rate to 115200. Run the code via debug and you should see the message "UART0 Initialized..." appear in the window of the terminal program. Type characters on the keyboard and you should see them being echoed to the screen. This code demonstrates the use of the UART. Modify this program so that it displays the lower 4 bits of the ASCII character onto the 4 blue LEDs.

Lab Session 2

JTAG Debugging

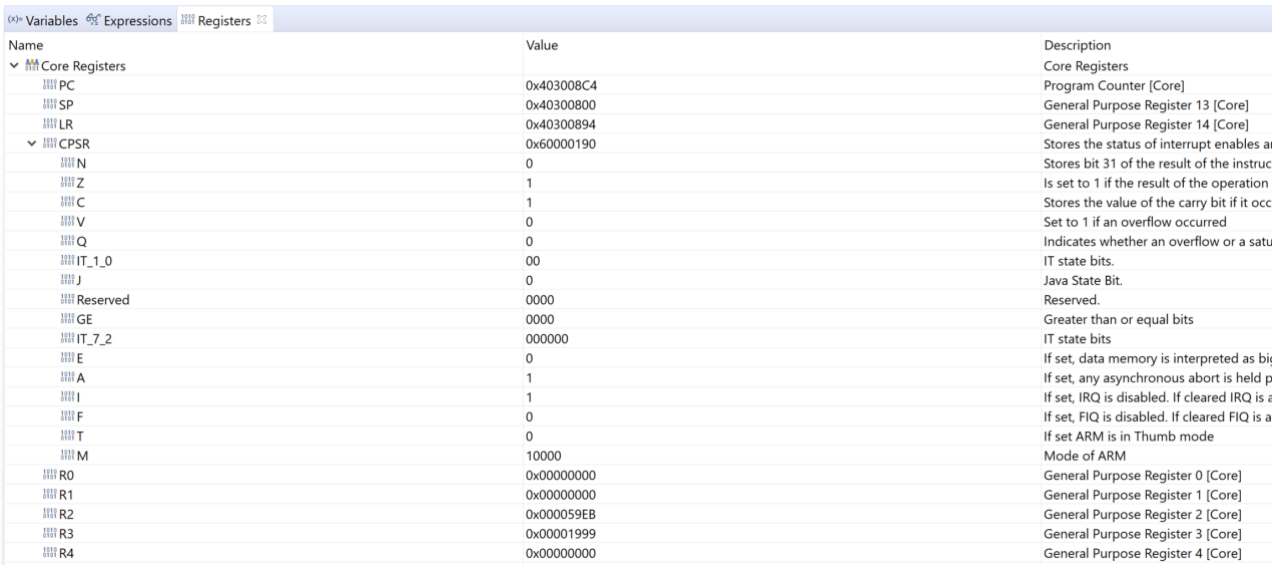
In the previous lab we investigated the execution of Bare-metal C code on the Beaglebone ARM processor. To download and run the program we used the JTAG in system programming features. For this lab we will investigate the Debug capabilities of JTAG.

Step 1: Power up the Beaglebone board in Bare-metal mode and then connect the SEGGER JTAG debugger unit.

Step 2: Download the zip file: EEEN301 lab1 pt2.zip and load into Code Composer Studio.

Step 3: Click on the Debug button to launch the debug session. Bring up the disassembler window to observe the loaded code.

Step 4: Expand the "Registers" window as per below.



Name	Value	Description
Core Registers		Core Registers
PC	0x403008C4	Program Counter [Core]
SP	0x40300800	General Purpose Register 13 [Core]
LR	0x40300894	General Purpose Register 14 [Core]
CPSR	0x60000190	Stores the status of interrupt enables a
N	0	Stores bit 31 of the result of the instruc
Z	1	Is set to 1 if the result of the operation
C	1	Stores the value of the carry bit if it occ
V	0	Set to 1 if an overflow occurred
Q	0	Indicates whether an overflow or a satu
IT_1_0	00	IT state bits.
J	0	Java State Bit.
Reserved	0000	Reserved.
GE	0000	Greater than or equal bits
IT_7_2	000000	IT state bits
E	0	If set, data memory is interpreted as bi
A	1	If set, any asynchronous abort is held p
I	1	If set, IRQ is disabled. If cleared IRQ is a
F	0	If set, FIQ is disabled. If cleared FIQ is a
T	0	If set ARM is in Thumb mode
M	10000	Mode of ARM
R0	0x00000000	General Purpose Register 0 [Core]
R1	0x00000000	General Purpose Register 1 [Core]
R2	0x000059EB	General Purpose Register 2 [Core]
R3	0x00001999	General Purpose Register 3 [Core]
R4	0x00000000	General Purpose Register 4 [Core]

You will notice the PC (program counter register) has the value equal to the start address of your loaded program. It is possible to change the value of the program counter or any other register by clicking on the value inside the "Value" column. Try changing the value of R0.

This feature allows you to alter the contents of the registers while stepping through a program.

Look at the start of the program and you will see the lines:

```
80000494:   E52DE004       str      lr, [sp, #-4]!
80000498:   E24DD00C       sub      sp, sp, #0xc
```

Question 1: What do you think this code does?

Step 6: Click on the "Window" and then the "Show View" menus and select "Memory Browser". A new window should appear on the right of your screen. In the address entry region, enter the value of the SP (0x8001F328). The contents of the stack region should appear.

Step 6: Click on the small green arrow  . This tells the debugger to execute one single instruction. You will notice that in the disassembler window the green bar has moved down one step. You should also notice the PC and SP registers and memory have changed. This is indicated by the yellow highlighting.

Step 7: Click on a memory location in the memory view window and change the value. The debugger then alters the memory on the Beaglebone board. This feature allows you to alter the memory as you step through and debug programs.

Step 8: Stop the debug session by clicking on the red box.

Step 9: Download the zip file: EEEN301 lab2 pt2.zip and load into Code Composer Studio. Examine the contents of the file "ASMCODE.c". This file demonstrates how you can insert assembly code instructions into a C project. Launch the debug, expand the register window and then single step through the program. In this example you should notice that the "Z" flag in the "CPSR" register changes. Try changing the ADDS instruction to just ADD and see what happens to the flags during execution.

Question 2. What is the difference in behaviour between the ADDS and ADD instructions?

Step 10: Alter the assembly language code so that while single stepping through it you set the “N” status bit.

Step 11: Relaunch your debug session and then in the C code or the disassembly window, double click on the left-hand side of the line number or address. You should see a small symbol appear. This is a breakpoint. Click the green run (arrow) button to run the program without single stepping and you will see that the program stops executing at the breakpoint. This is a very useful feature to see the state of the processor and memory when you have reached a specific point in the code. Breakpoints can be set, removed and disabled via the run menu.

Step 12: Reactivate the project EEEN301 lab1 pt2 and launch debug. Set a breakpoint at line 28 of the C code. Click the green run (arrow) button to run the program and you will see that the four LEDs will light up and the program will halt execution at the breakpoint. Click on the step over arrow (F6) a few times and observe the progression through the set of GPIO_clrPin function calls. You will also notice the LEDs turning off.

Question 3. List the debug capabilities of the JTAG port.