ECE471: Embedded Systems – Homework 3
Linux Assembly and Code Density

Due: Friday, 26 September 2025, 5:00pm EDT

1. **Use your Raspberry Pi for this assignment**

   - Download the code from:
     `https://web.eece.maine.edu/~vweaver/classes/ece471/ece471_hw3_code.tar.gz`
     and copy it to the Raspberry Pi.
   - Uncompress/unpack it with the command `tar -xzvf ece471_hw3_code.tar.gz`
   - Change into the ece471_hw3_code directory `cd ece471_hw3_code`
   - Put all answers to questions into the included text `README` file. This will automatically be bundled up with your submission.

2. **Modify the hello_world assembly program to return the value 25. (2 points total)**

   (a) Note the goal of this question is to **return** the value 25 via the exit syscall as described in lecture. The code also prints "Hello world" to the screen but you should not in any way change the printing code.

   (b) Also Note! How you do this depends on if you have a 64-bit or 32-bit version of Linux installed! To find this out run the command `uname -m`
   - If the result is `aarch64` you are on 64-bit
   - If it is `armv7l`, `armv6`, `armv8`, or anything starting with `arm` you are 32-bit.

   (c) For 32-bit, `cd` into the `32bit` directory and edit the file `hello_world_arm32.s`
   - Brief 32-bit assembly reminder, to move the number 5 into a register you would do `mov r0,#5`
   - For the 32-bit the Linux the kernel ARM syscall ABI:
     Arguments go in `r0` to `r6`
     System Call Number goes in `r7`
     Use `swi 0x0` to trigger a system call.

   (d) For 64-bit, `cd` into the `64bit` directory and edit the file `hello_world_arm64.s`
   - Brief 64-bit assembly reminder, to move the number 5 into a register you would do `mov x0,#5`
   - For 64-bit Linux the kernel ARM syscall ABI:
     Arguments go in `x0` to `x6`
     System Call Number goes in `x8`
     Use `svc 0x0` to trigger a system call.

   (e) **Be sure you comment the code you are modifying!**

   (f) Run `make` to generate an updated version
   If you get a weird error about operand must be an integer register from the assembler, double check you are following directions for the correct (32-bit or 64-bit) system (also be sure you are compiling on a Pi).

(g) To test, run `./hello_world_arm32` (or `./hello_world_arm64` on a 64-bit system) followed by `echo $?` which will show you the previous program's exit status, which should be 25 if you did everything right.

(h) Some reminders about Linux GNU assembler (`as`) syntax:
`.equ IDENTIFIER,value` sets a macro replacement, like
`#define IDENTIFIER value` would in C
There are a few different ways to comment your code but I recommend using C++ style comments such as `// this is a comment`

3. **Investigate the code density of `integer_print` (3 points total)**

(a) **NOTE** because most people are using 64-bit systems these days, but I want you to investigate 32-bit code density as well, I now provide pre-compiled executables to investigate.

(b) We will investigate the code in the `integer_print` subdirectory, so use `cd integer_print` to get into that directory.

(c) To see the algorithm we use for printing an integer, look at the source code (the same file is used for all of the C versions): `integer_print.c`. The algorithm used will be described in class.

(d) **Find the size of the `print_integer()` function in the ARM32 executable (see below) and record it in the README**

i. Find the start of the `print_integer()` function in the disassembly listing. Look at the the `integer_print_arm32.dis` file I provide that was generated with the `objdump` tool. If you use the `less` text file viewer you can use the '/' slash character to start a search and then search for `print_integer`. The first location you find is the call from main() to the function, press '/' again to find the actual function.

ii. (Note, to quit `less`, just type `q`)

iii. You should find something like the following:

```
0001043c <print_integer>:
   1043c:       e52de004        push    {lr}            @ (str lr, [sp, #-4]!)
   10440:       e30ccccd        movw    ip, #52429      @ 0xcccd
   ...
```

The first column is the address in memory where this code lives, the next is the raw machine code for the instruction, the next is the decoded assembly language, and the last is the disassembler giving "helpful" hints about what's going on.

iv. You can see in this first case all the instructions are 32-bit hex values.

v. Calculate the length of this function and note it in the README. You can calculate length by scrolling down to where the function ends, (there will be a break in the disassembly, in the provided code it's after the return (pop pc) instruction and the start of the .fini section)

vi. **Please report the length in decimal bytes**, you can do this by taking the address *after* the last instruction in the function and subtracting the start address from the function. Use a hexadecimal calculator to do the subtraction (you can find those online) and then convert the result to decimal.

(e) Now go back and do the same for the THUMB2 `integer_print_thumb2` executable too. The disassembly is in `integer_print_thumb2.dis` **Record the size of the function in the README.**

(f) **Also Answer in README**: Does the machine language generated look different for Thumb2 compared to ARM32? How?

4. **C vs Assembly code density: (2 points total)**
   Put the answer to these in the README.

   (a) Compare the size of the ARM32 and Thumb2 `integer_print` executables. To make the comparison more fair I've provided "stripped" versions of the executables, the Linux `strip` command has been used to strip off extraneous things (like debug info) from the executables to make them smaller.

   So `cd` into the `stripped` direction and you can find filesizes with `ls -l` (that's a lowercase L) command.

   (b) The `integer_print_static` file is also generated. This has the C library "statically linked" (included in the executable instead of dynamically loading the system copy of the C library at runtime). How does the size of the static version compare to the others?

   (c) I also provide a 64-bit version of the integer print code. What is the size of that file?

   (d) Finally, I provide a pure assembly language version of the integer print code: `integer_print_asm`. What is the size of that file?

   (e) Given these results, Which language (C vs assembly) might you use in a space constrained embedded system? Why?

   (f) Which code to you think is easier to write, the C or assembly one?

5. **Use `gdb` to track down the source of a segfault. (2 points total)**

   (a) Change into the "crash" directory and run `make`

   (b) This will build a program called `crash`

   (c) Run that program. It should crash with a `Segmentation fault` error.

   (d) Use the `gdb` debugger to find the source of the error.

   (e) Run `gdb ./crash`

   (f) When it comes up to a prompt, type `run` and press enter to run it until it crashes.

   (g) It should tell you it crashed, then tell you what line of code caused the crash. Put the line that caused the crash in the `README`

   (h) You can do various other things here, such as run `bt` to get a backtrace, which shows you which functions were called to get you to this error. You can run `info regis` to see the current register values.

   (i) Run `disassem` to get a disassembly of the function causing problems. There should be an arrow pointing to the problem code. Cut and paste this line into the `README`

   (j) In the end, what was the cause of the error in this program? (again, put this in the `README`)

6. **Linux Command Line Exploration (1 point total)**

   You can use your Linux machine to find the time/date, just type `date` at the command prompt.

   For years and years there was a program called `cal` bundled with Linux that would make a helpful ASCII art calendar. Sadly they decided to remove this, I guess the 70k of disk space it takes up was deemed too much. You can install it back yourself by doing `sudo apt-get install ncal` if you want to try it out.

   By the default it prints the current month:

```
$ cal
   September 2025
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30
```

   You can also `cal 2025` or `cal 12 2025`. Beware not to do `cal 25` as that will give you year 25, not 2025.

   (a) Run `cal 9 1752` and you will get this odd output.

```
$ cal 9 1752
   September 1752
Su Mo Tu We Th Fr Sa
        1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

   Is there a bug here? Can you explain what is happening?

7. **Submitting your work**

   - Run `make submit` which will create `hw3_submit.tar.gz` containing the various files. You can verify the contents with `tar -tzvf hw3_submit.tar.gz`
   - e-mail the `hw3_submit.tar.gz` file to me by the homework deadline. Be sure to send the proper file!