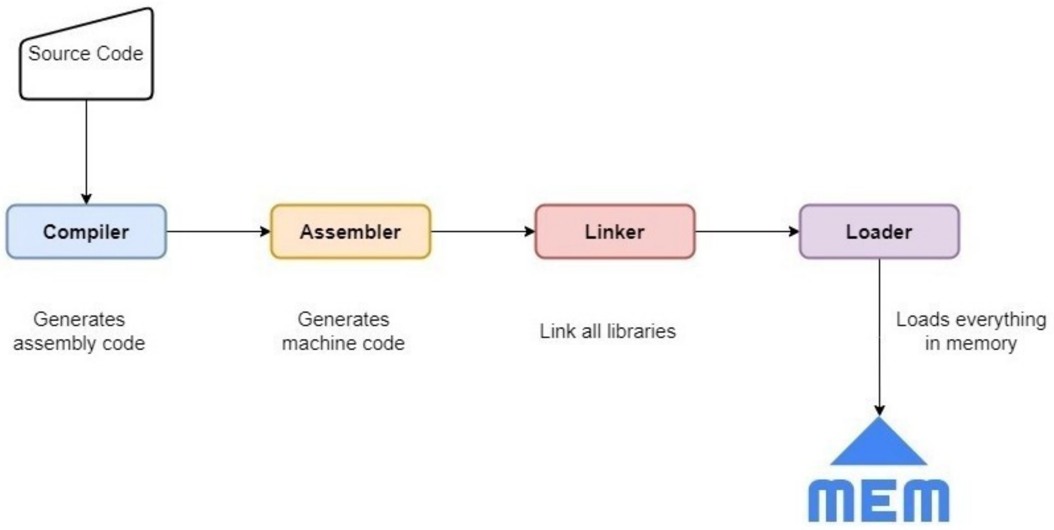
CSCI210 Computer Architecture and Organization
ARM Intro Lab

1. Download ***hello_world.s*** and ***hello_world_2.s*** from Brightspace.
2. Connect to your PI
3. Either copy the files over to the PI or write them directly using VS Code or Nano
4. Examine the code ARM assembly code in ***hello_world.s*** It uses a Linux system call to print and can be assembled and linked directly
5. From the terminal assemble ***hello_world.s*** using the assembler tool (**as**) and link and create the binary using the linker tool (**ld**)

    a. as -o hello.o hello_world.s
    b. ld -o hello hello.o
    c. Now you can execute the compiled binary ***hello***

The following image depicts the role of the assembler and linker in the build process



The assembler translates our assembly code to the machine code and then stores the result in an object file. This file contains the binary representation of our program. The assembler also gives a memory location to each object and instruction in our code. The memory location can be physical as well as virtual.

A virtual location is an offset that is relative to the base address of the first instruction.

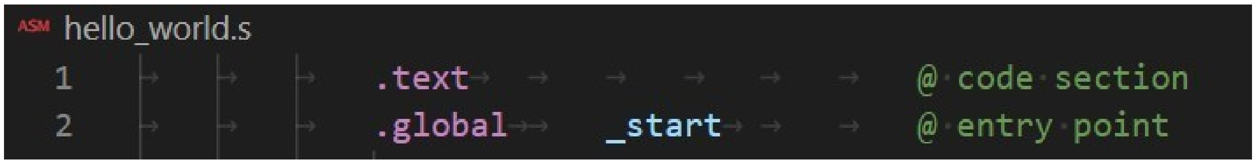
The linker combines all external programs (such as libraries and other source files) with our program to create a final executable. At the end of the linking step, we get the executable for our program. So, the linker takes all object files as input, resolves all memory references, and finally merges these object files to make an executable file.

6. ***hello_world_2.s*** uses a call to the C function ***printf*** and must be compiled with gcc so that the linker can pull in the code for ***printf***

7. Compile ***hello_world_2.s***
    a. gcc hello_world_2.s
    b. Now you can execute ***a.out***
    c. Or you can specify the name of your binary with
        i. gcc -o my_binary hello_world_2.s

8. Take screen shots of the build and run process to submit

Description of ***hello_world.s***



```
hello_world.s
            .text                               @ code section
            .global     _start                  @ entry point
```

- **_start:** We need to define this as a ***global symbol***, so that the linker (the ld command) can see it. The Assembler marks the statement containing **_start** as the program entry point, and then the linker can find it because it is a global variable. All our programs will contain this somewhere. It is analogous to ***main*** in Java and C/C++. Our program can consist of multiple **.s** files, but only one can contain **_start**.

- **.text:** This is an assembler directive that identifies the ***code segment***. This is important because the assembler and linker need to generate the proper segments when creating the ELF binary

- We only use three different Assembly language statements in this example:

    - **MOV, which** moves data into a register. In this case, we use an immediate operand, which starts with the "#" sign. Therefore, **mov R1, #4** means move the number 4 into **R1**. In this case, the 4 is encoded into the instruction and not stored somewhere else in memory. In the source file, the operands can be upper- or lowercase; I tend to prefer lowercase in my program listings, this is a style preference.

    - **ldr R1, =helloworld** statement which loads register 1 with the ***address of*** the string we want to print. More specifically it is the ***offset*** of the value (where it exists in relation to the beginning of the program segment)

    - **svc 0** command that executes software interrupt number 0. This sends control to the interrupt handler in the Linux kernel, which interprets the parameters we have set in various registers and does the actual work. The code is telling Linux to print the value at the address contained in **R1**

- Next, we have the **.data directive** that indicates the following data definitions are in the data section of the program. This is a different segment of the ELF binary from the **.text** segment

    - In this, we have a label "helloworld" followed by an **.ascii** directive and then the string we want to print.

    - The **.ascii** statement tells the Assembler just to put our string in the data section and then we can access it via the label as we do in the LDR statement.

    - The last "\n" character is how we represent a new line. If we do not include this, you must press return to see the text in the terminal window. Linux knows how many characters are in the string due to the instruction on line 7

This program makes two ***Linux system calls*** to do its work. The first is the Linux ***write to file*** routine (syscall #4). Normally, we would have to open a file first before using this command, but when Linux runs a program, it opens three files for it:

1. stdin (input from the keyboard)
2. stdout (output to the screen)
3. stderr (also output to the screen)

The Linux shell will redirect these when you ask it to use >, <, and | in your commands. For any Linux system call, you put the parameters in registers **R0–R4** depending on the quantity of parameters. Then place a return code in **R0** (which we are bad and not checking). Specify a system call by putting its function number in **R7**.

The reason we do a software interrupt rather than a branch or subroutine call is so we can request Linux routines without needing to know where the routine is in memory. This is rather clever and means we do not need to change any addresses in our program as Linux updates and its routines move around in memory. The software interrupt has another benefit of providing a standard mechanism to switch privilege levels.

Reverse Engineer with objdump

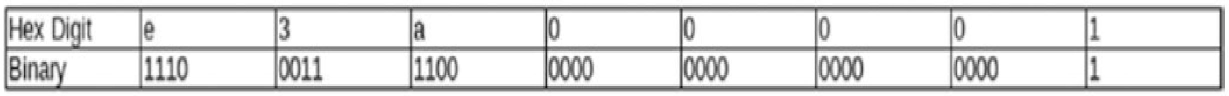
Objdump: Use the objdump command in Linux to provide thorough information on object files. The programmers who work on compilers mainly use this command, but still it is a handy tool for normal programmers when it comes to debugging. Experiment with the following options.

objdump -s -d hello

The top part of the output shows the raw data in the file including our eight instructions, then our string to print in the .data section. The second part is a disassembly of the executable .text section.

Look at the first **MOV** instruction that assembled to **0xe3a00001**

Binary representation of the first MOV instruction



Hex Digit	e	3	a	0	0	0	0	1
Binary	1110	0011	1100	0000	0000	0000	0000	1

- Each instruction starts with the hex digit "e" (14 decimal or 1110 binary). This is the condition code, which allows conditional execution of an instruction, and now we know "e" means execute the instruction unconditionally.

- The next 3 bits specify 001, which indicates the operand type, which in this case is a register and an immediate operand.

- The next 4 bits are 1110, which is the opcode for the MOV instruction.

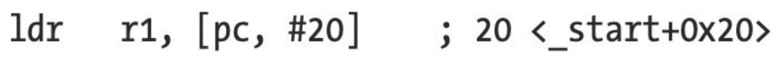
- The next bit is 0 which indicates the type of immediate mode parameter, which in this simple case doesn't matter.

- The next 4 bits are the register number, which is 0.

- If you look at the other **MOV** instructions, you can see the register number at this location.

- The remaining bits make up our immediate mode number, which is 1.

  Look at the LDR instruction, it changed from: **ldr R1, =helloworld**



```
ldr   r1, [pc, #20]    ; 20 <_start+0x20>
```

This is the Assembler helping you with the ARM processor's obscure mechanism of addressing memory. It lets you specify a ***symbolic address***, namely, "helloworld", and translate that into an offset from the program counter. I am certainly happy to have a tool do that bit of nastiness for me.

You might notice that the raw instructions in the top part of the output have their bytes reversed, compared to those listed in the disassembly listing. Little Endian. Experiment with the example

OBJDUMP options

1. **Show the file header:**	objdump -f hello
2. **Show the sections:**	objdump -h hello
3. **Show all sections:**	objdump -x hello
4. **Disassemble:**	objdump -d hello

Intro to Debugging with GDB:

1. GDB is the ***GNU Project Debugger*** and gives programmers the ability to perform debugging tasks, such as

    1. Viewing the contents of registers
    2. Setting break points
    3. Stepping through code a single line at a time
    4. GDB Cheat Sheet: http://www.yolinux.com/TUTORIALS/GDB-Commands.html

2. In preparation for using GDB you must pass an additional option to the assembler. Do this with your ***hello_world.s*** source

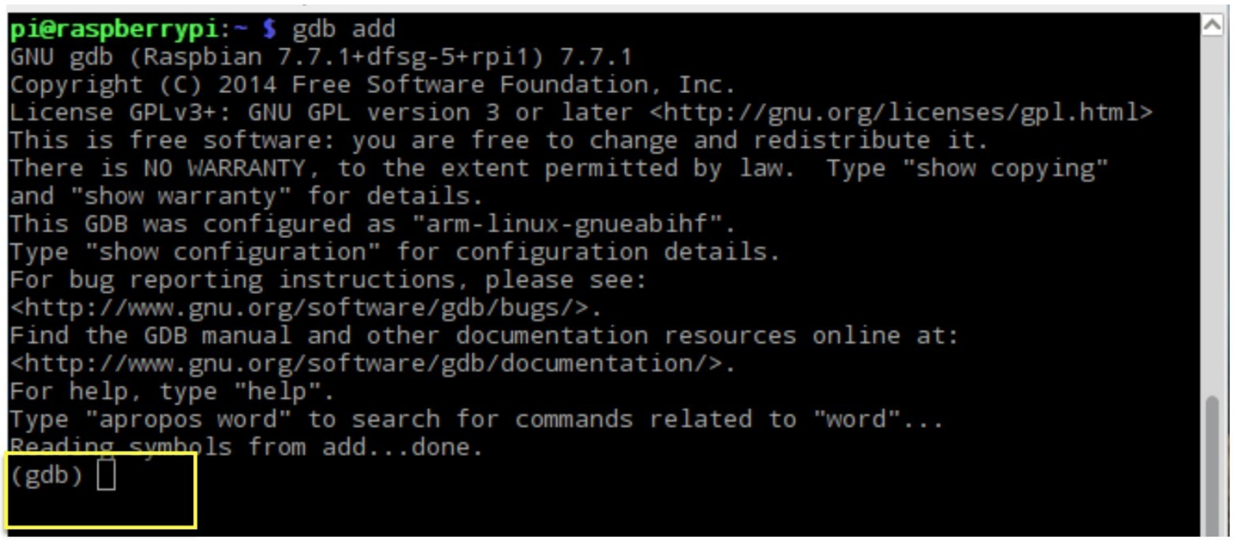
    1. **as -g -o hello.o hello_world.s**

    2. The "g" option generates debugging symbols that GDB needs to function properly. This includes generating symbol tables so that breakpoints can be created

    3. Link the files as you normally would

    4. **ld -o hello hello.o**

3. Start gdb by executing gdb <filename> (where filename is the name of the compiled binary you are attempting to debug). This will place you in an interactive gdb session where you enter commands to begin debugging. Notice the (gdb) prompt. Here you will execute gdb commands and run debugging tasks



```
pi@raspberrypi:~ $ gdb add
GNU gdb (Raspbian 7.7.1+dfsg-5+rpi1) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from add...done.
(gdb) 
```

1. Type list to get a program listing. This will show you the source code that you are currently debugging

2. To quit gdb use quit.

1. You can set a break point in gdb using the b command. A break point can be set at a label or a line number

1. ***b _start*** will create a break point at the start label
2. ***b 10*** will create a break point at line number 10
3. You should see visual confirmation of this at the terminal
4. Typing ***info b*** will list the current break points
5. You can delete a break point by executing ***delete N*** where the number relates to a break point from the info listing

- Set a breakpoint at the label _start by typing **b _start**
- To show breakpoint information type **info b**
- Run the program for debugging by typing **run**. Your program will run normally up to the first break point where it will halt (break) and allow you to examine what is occurring at that line. Notice the => arrow pointing to the current line. This is the instruction that will execute when you step
- Type **disassemble** to view the assembly. Abbreviate this to **disas**. Notice the hex address and the offsets **<+0>, <+4>, <+8>** etc
- Type **info r** to list the contents of the registers at that given moment. If you have not executed any code during the debugging process, there will not be anything to look at here so be sure to run your code first. You can show the contents of a single register by typing: **print/x $r0** (where ‘x’ means “show in hex”) continue executing up to the next break point by typing continue

- If you want to step through your code line by line use (**s**)tep instead of **continue**. Try stepping through your code line by line. Display the contents of the registers after each step. Make sure you understand the changes that occur. Try predicting what the changes will be. You can continue stepping by hitting enter. This will execute the previously executed gdb command
- You can perform a memory dump using the x (examine) command

    1. Type **x/22w 0x<memory address>** where memory address is the address of a program segment in memory. Get this value from the disassemble output above
    2. For example: **x/22w 0x0000874**
    3. This command is saying: “examine the contents of memory beginning at address 0x0000874. Show me this in 22 units of size word”

Build your ***hello_world*** source files for debugging with GDB. Execute GDB on your source file. Experiment with the basic debugging options above. HAVE FUN with it, mess around you will not break anything and the learning possibilities are endless. You are required to use this to debug your programs. Debugging assembly is tiresome and error prone when using just your eyes and it is superfluous overhead to include copious print statements (we will get there)

Using GDB with the Text User Interface (TUI)

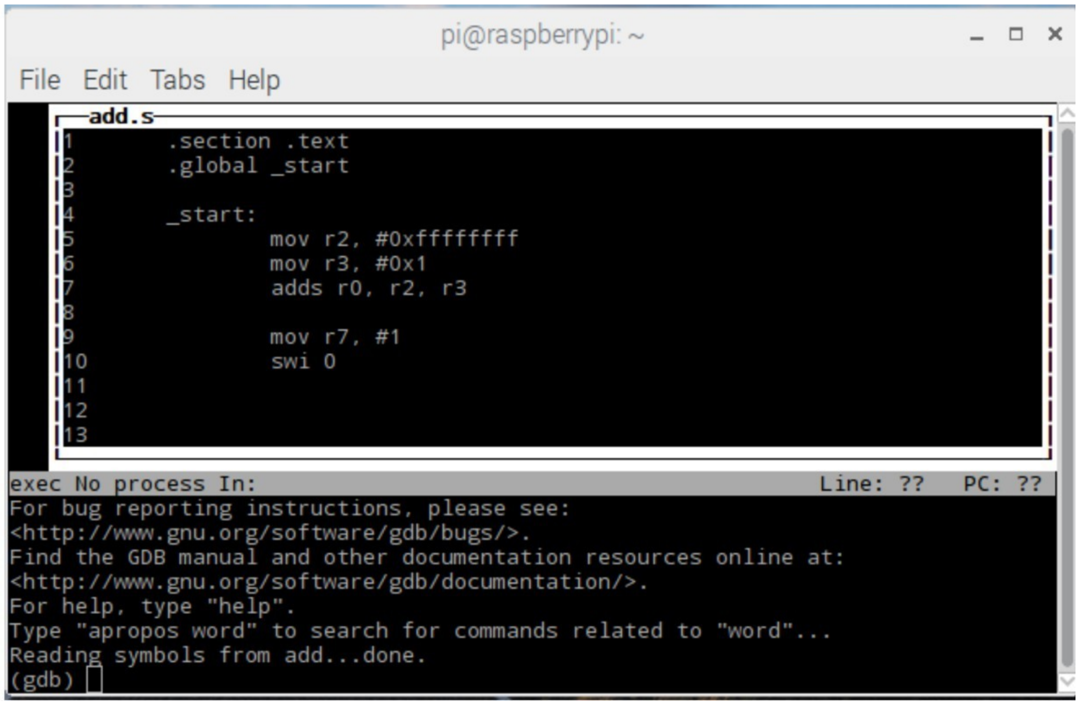
The TUI tool will give you multiple windows so that you can view different layouts in a single terminal session. Consult the nice cheat sheet for gdb and tui:

https://beej.us/guide/bggdb/#regasm

Start the gdb graphical mode by executing gdb with the -tui argument:

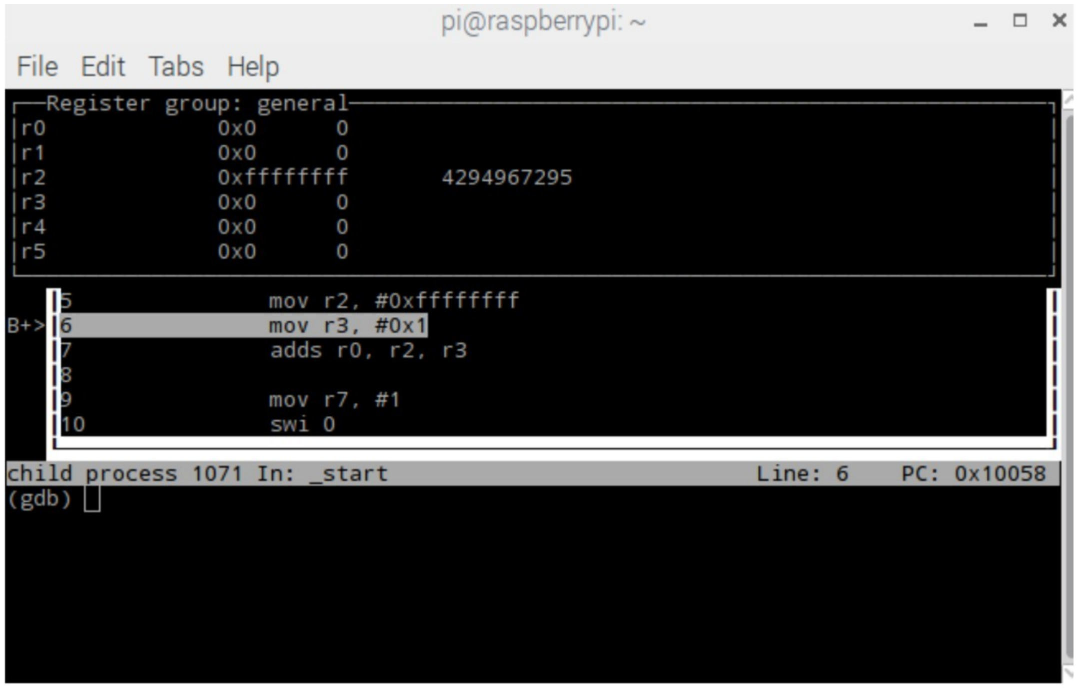
gdb -tui file_to_debug

This will take you to the source layout. If it does not automatically show the source, you can type layout source



```
add.s
        .section .text
        .global _start

        _start:
                mov r2, #0xffffffff
                mov r3, #0x1
                adds r0, r2, r3

                mov r7, #1
                swi 0



exec No process In:                                          Line: ??   PC: ??
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from add...done.
(gdb)
```

Notice that you still have the gdb prompt at the bottom of the TUI display. You can set breakpoints, step and do all normal debugging tasks here while viewing the contents of the registers and your source all in the same terminal session. Type **layout reg** to have gdb give you a pane to view the registers



```
Register group: general
r0             0x0      0
r1             0x0      0
r2             0xffffffff       4294967295
r3             0x0      0
r4             0x0      0
r5             0x0      0

                mov r2, #0xffffffff
B+>             mov r3, #0x1
                adds r0, r2, r3

                mov r7, #1
                swi 0
child process 1071 In: _start                         Line: 6     PC: 0x10058
(gdb)
```

EXPERIMENT!! Now you can step through your code line by line and see the state changes of the CPU in real-time.

TASK ONE: Create a new source file. Implement the following allocations. Don’t worry about making things ***static.*** Experiment with objdump, gdb and gdb -tui

```
        .data
i:      .word   0
j:      .word   1
fmt:    .asciz  "Hello\n"
ch:     .byte   'A','B',0
ary:    .word   0,1,2,3,4
```

(A) **Declarations in assembly**

```
static int i = 0;
static int j = 1;
static char fmt[] = "Hello\n";
static char ch[] = {'A','B',0};
static int ary[] = {0,1,2,3,4};
```

(B) **Declarations in C**

Write assembly code to declare variables equivalent to the following C code:

```
/* these variables are declared outside of any function */
static int foo[3];   /* visible anywhere in the current file */
static char bar[4];  /* visible anywhere in the current file */
char barfoo;         /* visible anywhere in the program */
int foobar;          /* visible anywhere in the program */
```

Filling and Aligning

On the ARM CPU, you can move data to and from memory one byte at a time, two bytes at a time (half-word), or four bytes at a time (word). Moving a word between the CPU and memory takes significantly more time if the address of the word is misaligned on a four-byte boundary (one where the least significant two bits are zero). Similarly, moving a half-word between the CPU and memory takes significantly more time if the address of the half-word is misaligned on a two-byte boundary (one where the least significant bit is zero). Therefore, when declaring storage, it is important that words and half-words are stored on appropriate boundaries. The following directives allow the programmer to insert as much space as necessary to align the next item on any boundary desired.

.align abs-expr, abs-expr, abs-expr

Pad the location counter (in the current subsection) to a particular storage boundary. For the ARM processor, the first expression specifies the number of low-order zero bits the location counter must have after advancement. The second expression gives the fill value to be stored in the padding bytes. You may omit that and the comma. Omitting this will result in a value of 0. The third expression is also optional. If it is present, it is the maximum number of bytes to skip by this alignment directive.

.balign[lw] abs-expr, abs-expr, abs-expr

These directives adjust the location counter to a particular storage boundary. The first expression is the byte-multiple for the alignment request. For example, ***.balign 16*** will insert fill bytes until the location counter is an even multiple of 16. If the location counter is already a multiple of 16, then no fill bytes are created.

The second expression gives the fill value to be stored in the fill bytes. You may omit that and the comma. Omitting this will result in a value of 0. The third expression is also optional If it is present, it is the maximum number of bytes to skip by this alignment directive.

The ***.balignw and .balignl*** directives are variants of the .balign directive. The .balignw directive treats the fill pattern as a 2-byte word value, and .balignl treats the fill pattern as a 4-byte long word value.

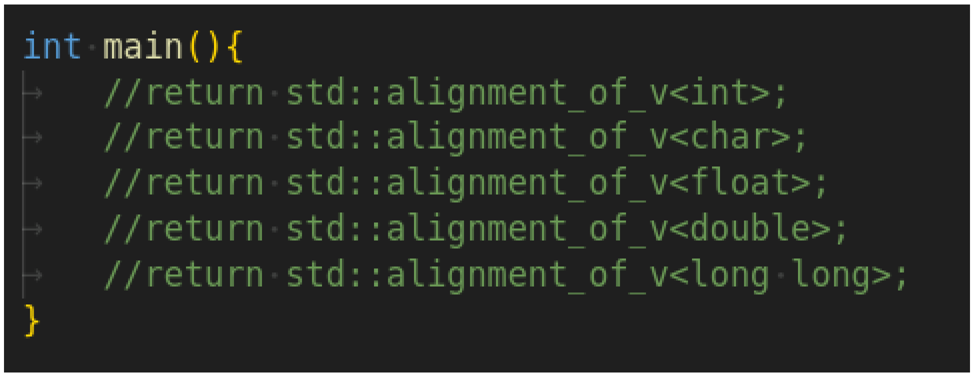
For example, ".balignw 4,0x368d" will align to a multiple of four bytes. If it skips two bytes, the value 0x368d is inserted (the exact placement of the bytes depends upon the endianness of the processor).

.skip size, fill .space size, fill

Sometimes it is desirable to allocate a large area of memory and initialize it all to the same value. Accomplish this by using these directives. These directives emit size bytes, each of value fill. Both size and fill are absolute expressions. If the comma and fill are omitted, fill is 0. For the ARM processor, the .space and .skip directives are equivalent. This directive is very useful for declaring large arrays in the .bss section.

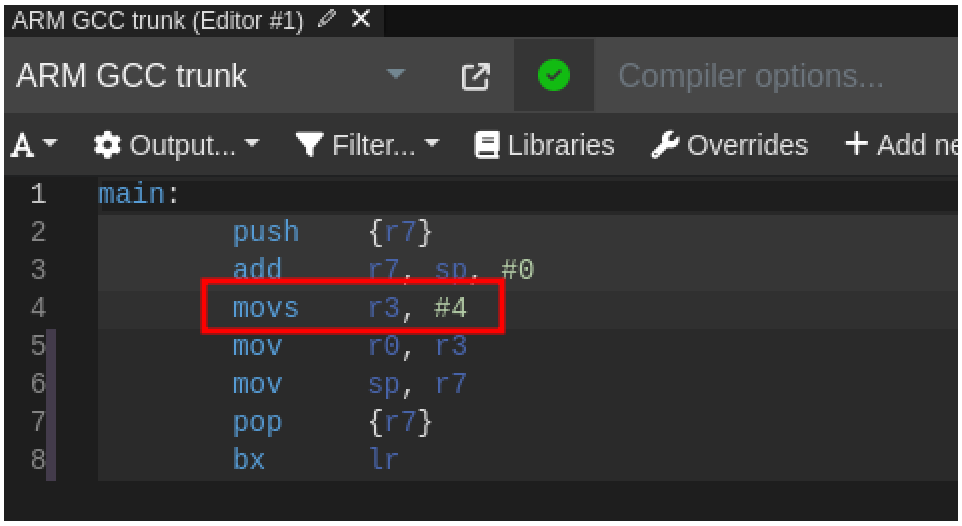
TASK TWO: Exploring alignment and padding with GodBolt

1. Open GodBolt in your browser
2. Select **ARM GCC (trunk)** as the compiler
3. Do the following
    1. Include the **<type_traits>** header. This header in C++ provides a set of ***compile time*** utilities that help analyze and modify types.
    2. Write the following main function. Use single return statements for each line to simplify the ARM assembly that the compiler generates. Simply un-comment line by line to analyze the output



```
int main(){
    //return std::alignment_of_v<int>;
    //return std::alignment_of_v<char>;
    //return std::alignment_of_v<float>;
    //return std::alignment_of_v<double>;
    //return std::alignment_of_v<long long>;
}
```

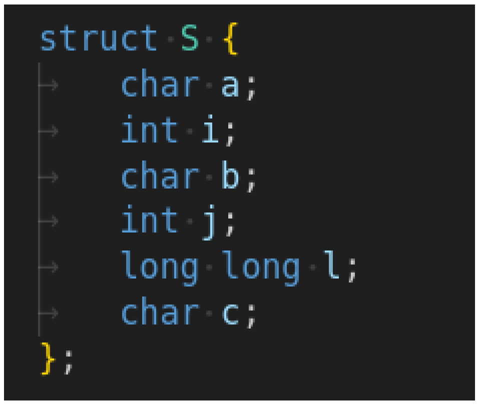
With the first line un-commented you should see this compiler generated assembly code. In this example ***Line 4*** is what we are interested in. Register 0 holds the return value and the compiler is moving 4 into this location. This tells us that the ***int type*** has an alignment of 4 (4 byte types are placed at memory addresses that are divisible by 4)



```
main:
        push    {r7}
        add     r7, sp, #0
        movs    r3, #4
        mov     r0, r3
        mov     sp, r7
        pop     {r7}
        bx      lr
```

Analyze the remaining type alignment and note these answers for inclusion as comments in the upcoming source file.

In the same GodBolt window, define the following struct.



```
struct S {
    char a;
    int i;
    char b;
    int j;
    long long l;
    char c;
};
```

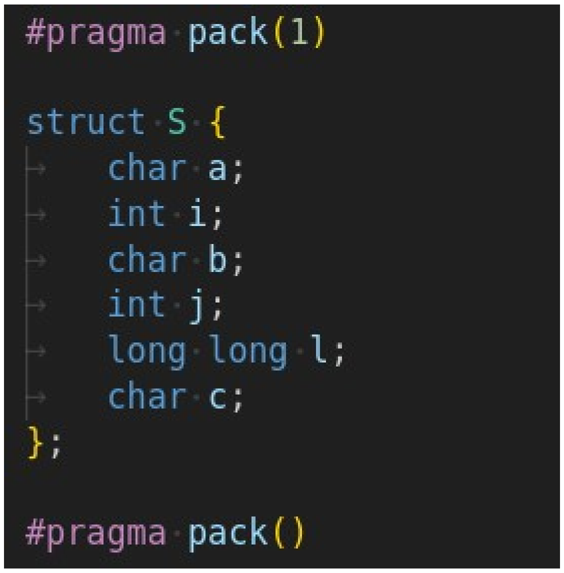
Count the number of bytes that are allocated in this struct. Use the ***sizeof*** operator if you are unsure of the size of each type. What did you come up with?

In your main function analyze the result of ***return sizeof(S);***

Is there a difference between your calculated size and the size that the compiler generates?

What gives? Reflect on this and be prepared to write your thoughts in comments.

Add the following and analyze the size of the struct in GodBolt



```c
#pragma pack(1)

struct S {
    char a;
    int i;
    char b;
    int j;
    long long l;
    char c;
};

#pragma pack()
```

What are the results? What is happening here?

Reorganize: Modify the struct to ***group data by size*** going from largest → smallest

Did this save any space? WHY

TASK THREE: Create a new assembly source code file and in comments, answer the padding/alignment questions from above.

ARM assembly alignment 1) Type the following code into your new source file.. Insert the minimum number of .align directives necessary in the following code so that all word variables are aligned on word boundaries and all half-word variables are aligned on half word boundaries, while minimizing the amount of wasted space.

```
        .data
        .align  2
a:      .byte   0
b:      .word   32
c:      .byte   3
d:      .hword  45
e:      .hword  0
f:      .byte   0
g:      .word   128
```

Reorganization 2) Re-order the directives in the original problem so that no .align directives are necessary to ensure proper alignment. How many bytes of storage did the original ordering of directives, compared to the new one, waste?

Submit the file with comments on your observations