Learning Objectives

1. Shift instructions in ARM
2. Explore branch instructions in ARM assembly
3. Arrays and pointers in assembly
4. Memory inspection for debugging
5. Function calling conventions

1 Lab Setup

1. Use illinoisNet_Guest to connect Raspberry Pi to the internet. Make sure you sign in for illinoisNet_Guest by opening the browser and visiting illinois.edu.
2. Clone your repository CS233 repository.
3. Run (all in one line) to add our repository as a remote

   ```
   git remote add honors_release \\n   https://github-dev.cs.illinois.edu/cs233-fa20/_honors_release.git
   ```

Run the following (separate commands) to get the files for this lab

   ```
   git fetch honors_release
   git merge --allow-unrelated-histories honors_release/RaspberryPiLab1
   ```

You only need to do this step once. Once you make a commit with merged lab, it will be available in your repository and simply cloning it would be enough.

4. Commit and push your work frequently. If it gives you a missing git configuration error then configure git without the `--global` option. This is important because the hardware will be shared. We don’t want you to modify any global settings.

5. Once you complete your lab, commit and push your final version. Then delete the files from Raspberry Pi. This is important because the hardware will be shared.

6. You have the option of either working in pairs or individually but we strongly encourage pairs. If you work in pairs, please make sure to fill the partners.txt file.

7. You will have to do this lab during TA Office Hours but please read the full lab manual before coming to lab.
2  ARM Instructions for Lab 1

2.1  Shift Instructions

ARM offers shift instructions for Shift Right (Logical and Arithmetic), Shift Left and Rotating a register. Some examples with explanation are given below. You can also look them up in the reference card.

- **LSR r1, r2, #3**  
  // Logical Shift Right: r1 = r2 >> 3. The 3 most significant bits are 0.

- **ASR r1, r2, #3**  
  // Arithmetic Shift Right: r1 = r2 >> 3. The 3 most significant bits are copies of most significant bit.

- **LSL r1, r2, #3**  
  // Logical Shift Left: r1 = r2 << 3. The 3 least significant bits are 0.

- **LSL r1, r2, r3**  
  // Logical Shift Left: r1 = r2 << r3. The shift amount is specified in r3.

In ARM, shift can also be done as part of the arithmetic or logical instructions. So you can optimize your code by combining shift and arithmetic operations. For this course we recommend that you work without optimizations and keep your shift and operations separate.

2.2  Conditional Branch Instructions

The conditional branch instructions in ARM depend on a special register called “Current Program Status Register” CPSR. This register contains status flags for indicating different facts about the results of the instructions that request status flag update. The flags that are important for branch instructions are:

- **Negative flag**: Commonly written as N. This flag is set when the result of the last computation is negative. This flag is CPSR[31] i.e. most significant bit of the status register.

- **Zero flag**: Commonly written as Z. This flag is set when the result of the last computation is 0. This flag is CPSR[30].

All arithmetic instruction can request status flag updates but you have to add an S at the end of the instruction mnemonic i.e. use ADDS instead of ADD.

The most important arithmetic instruction for branching is the **CMP**. A few example are shown below:

- **CMP r1, r2**  
  // Updates status flags based on computation r1 - r2

- **CMP r1, #3**  
  // Updates status flags based on computation r1 - 3

Once the status flags are set, we are ready for the conditional branch instructions. The syntax of the conditional branch instructions is **B<Condition> <label>** where the condition can be EQ, NE, LT, LE, GT or GE which stand for equal, not equal, less than, less than or equal, greater than and greater than or equal respectively. If no condition is specified, then it is an unconditional branch. Consider the examples below for more details.
• BEQ end_of_loop
  // Jumps to end_of_loop if Z flag is set. If flag is not set, go to next instruction.

• BLE end_of_loop
  // Jumps to end_of_loop if N flag is set or Z flag is set. Go to next instruction if conditions are not satisfied.

• B end_of_loop
  // Jumps to end_of_loop.

We can now translate a small C function to ARM assembly as follows:

```c
int abs(int x){
  int y;
  if (x < 0)
    y = -x;
  else
    y = x;
  return y
}
```

```asm
.text
.global abs
abs:       // x is in r0
           // y = r1
 CMP r0, #0
 BGE else_case
 MOV r2, #0
 Sub r1, r2, r0    // y = 0 - x = -x
 B end_of_function
else_case:
 MOV r1, r0      // y = x
end_of_function:
 MOV r0, r1      // return value must be in r0
 bx lr
}
```

If you want to explore more about conditional branch instructions, then refer to the assembler documentation. The branch instructions described above are pseudo-instructions. i.e. They are not part of the ARM instruction set. Assembler translates these instruction to instructions supported by the hardware.

### 2.3 Loads and Stores

It is generally not possible to store all data manipulated by a program in registers because we only have a few registers and a program may have to manipulate a large amount of data (hundreds of B, kB, MB, GB or even TB. That all depends on your program). Therefore data must be stored in memory. Load instruction brings data from memory to registers so that it can be processed
and Store instructions put processed data back in memory. The general syntax of load and store instructions is similar.

\[
\text{LDR/STR\{<size>\} <register> , [base register]{, <offset>}}
\]

\text{size} is an optional field. If size is not specified then word is considered the default size. Options for size are B, SB, H and SH which stand for byte, signed byte, halfword, signed halfword. For load instruction, <register> is the register which gets the value retrieved from memory. For store instruction, <register> is the register whose value will be written to the memory. Finally, the base register and \text{offset} define the memory location which will be read by a load instruction or written by store instruction. \text{offset} is an optional field. Some examples of load and store instructions are given below:

- LDR r1, [r2]
  // Fetch the word stored at address \[r2\] in memory and place it in r1.

- LDR r1, [r2, #4]
  // Fetch the word stored at address \[r2 + 4\] in memory and place it in r1.

- LDR r1, [r2, r3]
  // Fetch the word stored at address \[r2 + r3\] in memory and place it in r1.

- LDRH r1, [r2]
  // Fetch the half word stored at address \[r2\] in memory and place it in lower 16 bits of r1. The top 16 bits will be 0.

- LDRSB r1, [r2, r3]
  // Fetch the byte stored at address \[r2 + r3\] in memory and place it in lowest 8 bits of r1. The top 24 bits will be copies of most significant bit of the byte. This is called sign extension.

- STR r1, [r2, r3]
  // Write the word in register r1 at the memory address \[r2 + r3\].

- STRH r1, [r2, #4]
  // Write the half word in the lower 16-bits of register r1 at the memory address \[r2 + 4\].

A example code which increments all elements of an array is shown below. This example is also included in the lab files.

```c
/*
int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int main()
{
    for (int i = 0; i < 10; i++)
    {
        array[i]++;
    }
    return 0;
}
*/
```
.data
array: .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

.text
.global main

arrayadr: .word array

main:
  mov r0, #0
  ldr r1, arrayadr // r1 = array
loop:
  cmp r0, #10
  bge end_loop
  lsl r2, r0, #2 // r2 = r0 * 4
  ldr r3, [r1, r2]
  add r3, r3, #1
  str r3, [r1, r2]
  add r0, r0, #1
  b loop
end_loop:
  mov r0, #0
  bx lr

2.4 Memory Inspection for Debugging

The file array.inc.s is included in the lab files. You can compile it and run it. We will use this file to explore how we can inspect memory using GDB. Table 2.4 is repeated from last lab with some additional commands to help you debug the programs.

Follow the steps described below for array.inc.s:

1. gcc -g -o array array.inc.s
   Compiles array.inc.s with debugging information.

2. gdb array
   This will start GDB and change the prompt to (gdb).

3. b main
   Set a breakpoint in array.inc.s at label main. Note: this will not work if compilation is done without -g flag.

4. run
   This will run the program and stop at the start of main.

5. x/1xw arrayadr
   This will display the address of the array. Note down this address.

6. nexti 2
   Run 2 statements of array.inc.s
<table>
<thead>
<tr>
<th>GDB Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td>List available commands</td>
</tr>
<tr>
<td>break [address]</td>
<td>Set breakpoint at the specified address. The address can be given in many forms. Examples: break first.s:13 // sets a breakpoint at line 13 of file first.s break first.s:main // sets a breakpoint at label main in file first.s</td>
</tr>
<tr>
<td>run</td>
<td>Runs the program to be debugged. It will stop at the breakpoint.</td>
</tr>
<tr>
<td>s</td>
<td>Single step. When the step in the source file is a function call, gdb will step into the function and stop.</td>
</tr>
<tr>
<td>stepi [N]</td>
<td>Single step N times.</td>
</tr>
<tr>
<td>n</td>
<td>Next statement. When the statement in the source file is a function call, gdb will execute the whole function and stop at the next line in source file.</td>
</tr>
<tr>
<td>nexti [N]</td>
<td>Next statement N times.</td>
</tr>
<tr>
<td>c</td>
<td>Continue execution until the next break point.</td>
</tr>
<tr>
<td>info break</td>
<td>Shows a numbered list of breakpoints in the program</td>
</tr>
<tr>
<td>delete break [number]</td>
<td>Delete breakpoint. The <strong>number</strong> is the number of the breakpoint in the list of breakpoints shown by the info break command.</td>
</tr>
<tr>
<td>info registers</td>
<td>Displays register values.</td>
</tr>
<tr>
<td>x [address] or x /[nfu] [address]</td>
<td>Examine the memory starting at the given address. n is the repeat count - it specifies how much memory to display. f is a specification of the display format (use d for decimal, x for hexadecimal, and i for machine instructions). u us the unit size (use w for word, b for byte, etc). For more information and examples, check out this link</td>
</tr>
</tbody>
</table>

Table 1. GDB Reference

7. i r
   This will display all registers. Notice the value of register r1. It should be the same as array address after load.

8. x/10dw <address of array noted earlier> e.g x/10dw 0x21024
   This will show you the array.

9. nexti 8
   Run 8 statements of array_inc.s

10. x/10dw <address of array noted earlier> e.g x/10dw 0x21024 This will show you the array. Note how the values in memory have changed.

11. Repeat the last 2 steps and notice how the memory values change as the program executes.
    Keep repeating the steps till you reach the end of the program.

   TODO: Follow the steps listed above for array_inc.s

2.5 Function calling conventions

In the last lab handout, we saw that some registers have multiple names. These names indicate special roles for these registers. The table is repeated in Figure 1 for reference.
The roles of the registers as as follows:

- The first four registers `r0-r3 (a1-a4)` are used to pass argument values into a subroutine and to return a result value from a function. They may also be used to hold intermediate values within a routine. That mean values of these register may change if you call a function. Therefore these registers should not be used to store any variables that will be needed after the function call.

- The registers `r4-r11 (v1-v8)` are used to hold the values of a routine’s local variables. You can expect the functions you call to preserve these registers. It also means that if some function calls your routine, then you must make sure that these registers are not modified. This severely limits the number of available registers but we can get around it. A possible solution is to store the values of the registers in memory and use the registers in our function. When our function is complete we can restore the values from the memory. We will do this using the stack memory segment.

- In Registers `r12-r15` have special roles. In these roles they are labeled IP, SP, LR and PC.

- PC is the program counter register and contains the address of the next instruction to execute. We will not modify this register directly. Instead we will use instructions like BX and BLX to change this register and jump to functions.
- We have seen LR register in the programs we have written so far. LR stores the address of the location where our function must return after completing its job. Therefore the value of this register must not change if we want to properly return to our calling function. But we also need to use this register while calling other functions to tell the other functions where to return. This means that we need to store this register before we call any functions and restore the value after all function calls are done.

- SP is the stack pointer register. It contains address of a memory segment known as the stack. In a correct implementation this register should always point to the top of the stack segment. We will use PUSH and POP instructions to modify this register instead of directly modifying this register. Stack memory segment is the location where we will store the values of the v registers and lr register to preserve them. At the end of our function we will restore the values of the register from this location. We will see how this is done with an example later.

The program below shows how we use the stack to free up registers for our own use. TODO: Compile and debug calling_conventions.s file. Observe how the variable registers maintain their values in the main function, even when the values are changed inside calling_convention_test function. Go back the the code you competed in Lab 0. How does call to printf and scanf change the registers? Which registers are preserved and which registers loose their value?

```c
/* This is a dummy program for demonstrating calling conventions */
 .text
 .global main

main:
 // Saving registers
    push    {v1, v2, v3, v4, v5, v6, v7, v8, lr}
 // Setting up variable register values
    mov     v1, #1
    mov     v2, #2
    mov     v3, #3
    mov     v4, #4
    mov     v5, #5
    mov     v6, #6
    mov     v7, #7
    mov     v8, #8
 // Setting up function arguments
    mov     a1, #11
    mov     a2, #12
    mov     a3, #13
    mov     a4, #14

 // Calling function
    bl     calling_convention_test

 // Stop here and check register values
 // Values of a registers are not preserved
 // but values of v registers are preserved
```
add  a1, a1, v1

// Restoring function values
pop   {v1, v2, v3, v4, v5, v6, v7, v8, lr}
bx    lr

calling_convention_test:
// Saving registers
push  {v1, v2, v3, v4, v5, v6, v7, v8, lr}

// Setting up variable values as required in this function
mov   v1,  #100
mov   v2,  #200
mov   v3,  #300
mov   v4,  #400
mov   v5,  #500
mov   v6,  #600
mov   v7,  #700
mov   v8,  #800

mov   a1,  #110
mov   a2,  #120
mov   a3,  #130
mov   a4,  #140
// Stop an observe register values
add   a1, a1, v1

pop   {v1, v2, v3, v4, v5, v6, v7, v8, lr}
bx    lr

3 Assignment

You and a partner are chefs in a kitchen, cooking different meals for delivery. Together, you are cooking with a total of 11 ingredients. However, the other chef is missing crucial ingredients needed to complete their orders. In order to communicate this need, the other chef sends you a message containing the amounts of each ingredient they require.

Due to limited memory, they could not send you an array of ints, and as a result the message they sent is a long int. How much the other chef is requesting of each ingredient is 5 bits of the long int. The least significant 5 bits of the long int indicate how much of the first ingredient the other chef is requesting, the next least significant 5 bits of the long int indicate how much of the second ingredients is being requested, and so on. It's up to you to decode their request so you know what the other chef is requesting. The bits are read from least significant to most significant. The most significant 9 bits are "padding" and can be ignored.

For example, if you received 0xBDB90EF893860385 as your request, the message would be broken down as follows:
Thus, 5 of ingredient 1, 28 of ingredient 2, 0 of ingredient 3, 12 of ingredient 4, 24 of ingredient 5, 9 of ingredient 6, 2 of ingredient 7, 31 of ingredient 8, 14 of ingredient 9, 8 of ingredient 10, and 14 of ingredient 11 are being requested. You ignore the most significant 9 bits.

Since we are using 32-bit ARM, our registers can only contain 4 bytes. However, a long int is 8 bytes large. In order to properly store a long int, you need 2 registers to hold it, one of which holds its most significant 4 bytes and the other of which holds its 4 least significant bytes. These are referred to as the "hi" and "lo" registers for the long. The endianness of this long int is also important and determines which is the "hi" register and which is the "lo" register. You can use the command `lscpu` to check the endianness. It should be little endian.

In a little endian system, the long int \texttt{0xBDB90EF893860385} would be stored in memory as follows:

![Memory layout diagram]

In this case, the "lo" register would be \texttt{a1} and the "hi" register would be \texttt{a2}.

When implementing your decoder, you must keep in mind how long ints are stored in memory and how you can access the values of each of its bits. After storing the value of the long int, you can determine the amount requested of each ingredient using bit shifting and bit masking. A decoder implementation in C is included in your github directory in the \texttt{decode.s} file and in a file called \texttt{decode.c}.

### 3.1 Problem 1: decode_request

This function takes an unsigned (64-bit) long integer and an integer array as arguments and changes the values of the array to correspond to the amount of each ingredient requested. The function does not return anything.

```c
// Sets the values of the array to the corresponding values in the request
void decode_request(unsigned long int request, int* array) {   
  // The hi and lo values are already given to you, so you don’t have to  
  // perform these shifting operations. They are included so that this  
  // code functions in C.
  unsigned lo = (unsigned)((request << 32) >> 32);  
  // Sets the values of the array to the corresponding values in the request
  array[1] = (int)(request >> 28);  
  array[2] = (int)((request >> 20) & 0x000000FF);  
  array[3] = (int)((request >> 12) & 0x000000FF);  
  array[4] = (int)(request & 0x000000FF);  
  array[5] = (int)((request >> 8) & 0x000000FF);  
  array[6] = (int)((request >> 0) & 0x000000FF);  
}
```
unsigned hi = (unsigned)(request >> 32);

for (int i = 0; i < 6; ++i) {
    array[i] = lo & 0x0000001f;
    lo = lo >> 5;
}
unsigned upper_three_bits = (hi << 2) & 0x0000001f;
array[6] = upper_three_bits | lo;
hi = hi >> 3;
for (int i = 7; i < 11; ++i) {
    array[i] = hi & 0x0000001f;
    hi = hi >> 5;
}

The decode_request function requires you to implement a loop. Write your code in the file decode.s. Make sure you follow the calling conventions if you use the variable registers. If you need to debug then do not forget to add -g flag. Compile and test using the following command:

gcc -mcpu=cortex-a53 -o decode decode_main.c decode.s

The next problem involves doubly-nested loops, dealing with structs, and needing to save/restore registers and call other functions.

3.2 Problem 2: bubble_sort

Now that you can decode your partner’s requests, you can help them gather the ingredients they need. However, some ingredient requests are bound to be more important than others, and choosing what order to fulfill them in is important. In order to do this, you need to be able to sort the ingredient requests somehow.

For this part of the lab, you will implement a bubble sort function. A bubble sort works by iteratively going through each element of an array and swapping it with adjacent elements if they are in the wrong order. What the correct order is can be defined by the user, so it doesn’t necessarily have to be sorting by magnitude.

Because you need to remember what amount corresponds to which ingredient, we use a specialized struct to hold this information:

```c
struct Ingredient {
    unsigned ing_type;
    unsigned amount;
};
```

In addition, there is another struct to represent an entire request:

```c
struct Request {
    unsigned length;
    Ingredient ingredients [11];
};
```

With this information, we can implement a bubble sort as follows:
// Performs a bubble sort on the given request using the given comparison function
void bubble_sort(Request* request, int (*cmp_func)(Ingredient*, Ingredient*)) {
    for (int i = 0; i < request->length; ++i) {
        for (int j = 0; j < request->length - i - 1; ++j) {
            if (cmp_func(request->ingredients[j], request->ingredients[j + 1]) > 0) {
                Ingredient temp = request->ingredients[j];
                request->ingredients[j] = request->ingredients[j + 1];
                request->ingredients[j + 1] = temp;
            }
        }
    }
}

This bubble sort implementation uses a function pointer to a comparator function. When you implement this code, you can treat that argument the same as an address, just stored in the a2 register. In addition to the BL instruction which allows to jump and link to a defined location in your code, BLX allows you to jump to the address within a register, similar to BX.

For this problem, we will be sorting by magnitude. Thus, given two ingredients, you want to return 1 if more of the first ingredient is requested than the second ingredient.

// A comparison function for use with bubble_sort
int compare_ingredients(Ingredient* ingredient1, Ingredient* ingredient2) {
    if (ingredient1->amount > ingredient2->amount) {
        return 1;
    } else {
        return 0;
    }
}
4 Additional Resources

1. GNU ARM Assembler Quick Reference

2. The gnu Assembler
   (https://web.eecs.umich.edu/prabal/teaching/resources/eecs373/Assembler.pdf)

3. Debugging Assembly Code with gdb
   (http://web.cecs.pdx.edu/apt/cs491/gdb.pdf)

4. Introduction to ARM Assembly Basics
   (https://azeria-labs.com/writing-arm-assembly-part-1/)

5. ARM Quick Reference Sheet
   (http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf)

6. Procedure Call Standard for the ARM® Architecture

   (https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf).