Branches & Subroutines

Prof. John McLeod

ECE9047/9407, Winter 2021

This lesson continues the discussion of assembly language, with specific emphasis to the code base for the ARM®Cortex-A9 processor. The basics of branching, the stack, and subroutines are provided.

Assembly Language and Microcontrollers

As previously mentioned, the basic categories of assembly language instructions are:

1. Data movement instructions.
2. Data manipulation instructions.
3. Conditionals and test instructions.
4. Branch instructions.
5. Subroutine instructions.
6. Interrupt instructions.

The first three were discussed in last week's lesson. This lesson will examine the next two.

Branching

We already discussed how the **program counter** (register `pc`) keeps track of the next instruction in memory. Under normal circumstances the program counter automatically increments by `4` after each instruction is completed, so the program is read out sequentially from memory.

- Note that for **conditional exectution**, the instruction is still *read* sequentially, it just may not be *executed* depending on the present condition.

For a fully functional programming language, sequential execution is not sufficient: there must be a way to **branch** to non-sequential regions of code.

- As `pc` acts as a normal register, programmers who are fans of anarchy already know how to create a branch in their programs: just conditionally add or subtract values from the `pc`.
- This is a pretty extreme solution, and as you might have guessed, their are better options.

The preferred way to create branches in your program is to use the equivalent of a `GOTO` command to jump to a label somewhere else in your code.

Mnemonic: To perform a **direct branch** to a known location in your program, use the `b` mnemonic. This requires a single label as an operand.

Mnemonic: To perform an **indirect branch** to a variable location in your program, use the `bx` mnemonic. This requires a single register as an operand, the register should hold the memory address of appropriate instruction.

Like every mnemonic, both `b` and `bx` can have appended conditional codes. Consider the following examples:

```
mov r0, #8       @initialize r0
cmp r0, #5       @is r0 < 5?
strlo r0, [r1]   @store to memory if so
```

This piece of code stores the value in register `r0` to the memory address in `r1` if the value in `r0` is less than 5. Pretty clutch piece of code, eh? We can implement the same thing using a branch:

```
mov r0, #8       @initialize r0
cmp r0, #5       @is r0 > 5?
bhi more_code    @skip next line if not
str r0, [r1]
more_code:
```

This piece of code puts the conditional on the branch mnemonic `blo`, so the `str` instruction be completely skipped if $8 > 5$. [1] In this example, using a branch is probably a worse idea than just using a conditional execution of `str`. However if there were multiple lines of code that were conditional, using one branch mnemonic to skip over then entire block rather than write each instruction as a conditional execution may be better.
Finally, just for fun, and to to confuse your enemies, you could implement the same code using indirect branches:

[1] As opposed to the first example where it would be read but not executed if $8 > 5$.

```
mov r0, #8
mov r2, pc
add r2, #12
cmp r0, #5
bxhi r2
str r0, [r1]
more_code:
```

In this example, the program counter is stored in r2 and then increased by 12, as the line after the more_code label is four lines ahead of the point where pc is stored in r2. [2]
The above examples of branching are a obviously contrived, and also more confusing than just using conditional execution. Some more relevant examples will follow.

[2] Here pc is incremented by 4 before it is written to r2, and $16 = 12 + 4$.

If... Then... Else

Most programming languages have an *if...then...else* conditional structure.

- This can be implemented without branches, just by having the appropriate conditionals and counter-conditionals on the *then* and *else* statements.

```
cmp  r0, #5     @if r0 <= 5
addls r1, r2    @then
subls r0, #1    @then
addhi r1, r3    @else
addhi r0, #1    @else
```

Adding conditional codes makes the mnemonics longer and arguably harder to read. If there is only one or lines of code for each condition, then it is probably appropriate. However your conditional requires several lines of code for the *then* and/or *else* block, using a branch to avoid one or the other can make the code simpler to read.

- Without using branches, you are also unable to update the **status flags** inside an *if...then...else* block — so you can't have any nested conditionals. Using branches fixes this problem.

The following example uses the same basic code as the above example, but uses branches to allow the subtraction step to include a check in the event that `r0` becomes negative.

```
if:     cmp r0, #5      @check r0=5
        bhi else
then:   add r1, r2
        subs r0, #1     @decrement r0
        bmi neg         @is r0 negative?
        b endif
else:   add r1, r3
        add r0, #1
        b endif
neg:    mov r0, #10   @roll r0 back to 10
endif:  str r1, [r4]
```

In additional to the extra functionality and flexibility of branches, the explicit use of labels can help make your code a bit more readable.

Creating Loops

Creating loops is an important functionality for a programming language, and this can be done neatly using conditional branching. There is no explicit *for...next* or *do...while* loops in assembly as there might be in other programming languages, but the same functionality can be obtained by using conditional branches in the correct place. For example, the following code implements a *for* loop that iterates 10 times.

```
          mov r4, #10   @initialize counter
loop:     cmp r4, #0    @exit loop?
          beq endloop
          /*a whole bunch of code here*/
          sub r4, #1    @decrement counter
          b loop        @go back to start
endloop: /*rest of code here*/
```

To make an un-iterated loop is even simpler, just omit the counter.

```
loop:     cmp r4, #17
          beq endloop
          /*More code here.
            I hope somehow r4 will equal 17
            or this loop will never end*/
          b loop
endloop: /*rest of code here*/
```

It is also worth noting that many microcontrollers are designed to run in perpetuity: unless shut down manually by the user they

should repeat their programming.

- It is typically good practice to have the entire main program in an endless loop. Once the end of the code is reached, the last line should be `b _start` or something similar.
- It may also be necessary to loop almost endlessly while waiting for user input.

The following example demonstrates a loop that waits unit one of the switches on the DE1-SoC development board is thrown. When using the simulator and stepping through the code line by line, there is plenty of time to toggle switches. However a real ARM®Cortex-A9 operates at clock speeds of at least 800 MHz — millions of processor cycles will have passed before your fat fingers manage to touch one of the switches.

```
ldr r4, =0xff200040 @memory address for switches
ldr r0, [r4]        @get initial state
/*some code here*/
pause_for_input:
   ldr r1, [r4]     @read switches
   cmp r1, r0       @check if switches changed
   beq pause_for_input
mov r0, r1          @new "old state" of switches
/*rest of code*/
```

The Link Register

All of this branching and looping is undeniably cool, but it is a bit rigid. What if we want to write a subroutine that can be called, and then return to the original spot in code? This is where indirect branching comes in. Consider the following example:

```
/*some code here*/
mov r3, pc
add r3, #8
b pause_for_input
/* more code */
mov r3, pc
add r3, #8
b pause_for_input
/* even more code*/
pause_for_input:
   ldr r1, [r4]      @read switches
   cmp r1, r0        @check if switches changed
   beq pause_for_input
   bx r3         @return to original code
```

In this example, `pause_for_input` is now a subroutine.

- By storing the state of `pc` and incrementing it by 8 before branching to the subroutine, `r3` now contains the address of the instruction that comes immediately after the branch instruction.
- By exiting the `pause_for_input` block using an indirect branch,

the program returns to the appropriate place in the code.

This procedure is so common that as special register and some mnemonics are available to streamline the process, and avoid the cludgy "`add r3, #8`"-type code to increment the program counter.

Definition: The **link register** `lr` is a special register used to store the address of an instruction immediately *after* the current one (i.e., the address `pc + 4`).

Mnemonic: To save the next instruction before branching to a direct label, use the **direct branch with link** mnemonic `bl`.

Mnemonic: To save the next instruction before branching to a indirect location, use the **indirect branch with link** mnemonic `blx`.

This cleans up the above example, as shown below. Note that there is now no need to explicitly save the program counter, or to appropriately increment it.

```
/*some code here*/
bl pause_for_input
/* more code */
bl pause_for_input
/* even more code*/
pause_for_input:
   ldr r1, [r4]      @read switches
   cmp r1, r0        @check if switches changed
   beq pause_for_input
   bx lr         @return to original code
```

The Stack

The above discussion of the link register demonstrate how simple subroutines can be implemented in assembly. However there is still a lot of functionality missing:

- There is only one link register, so subroutines cannot be nested. [3]
- There is a finite set of registers, many of which are probably needed in the main program. But the subroutine needs registers too! Your subroutines have to know which registers it can overwrite and which it must preserve.

A more flexible way of approaching subroutines is to store the **state** of the program prior to the subroutine call to memory, than restore it just before the subroutine returns. This method is common enough that a special area of memory and a special set of mnemonics are available to simplify and standardize writing subroutines in assembly.

Definition: The **stack** is a special area of memory used to store "snapshots" of the registers. It is a *last-in-first-out* data structure.

Definition: The **stack pointer** `sp` is a special pointer that holds the most recently-used memory address in the stack.

Definition: When data is written to the stack, we say it is **pushed** onto the stack.

Definition: When data is read from the stack, we say it is **popped** off the stack.

[3] Well, technically these kind of subroutines *can* be nested, if you use a different register to hold the previous `pc+4` address for each level of nesting. But this is clumsy and requires advance knowledge of how many levels of nesting are possible.

There are a few variations in how a stack is constructed. First, there are two options for how data is written to the stack.

Definition: A **descending stack** starts at a high memory address, and *decrements* the stack pointer as data is written to the stack. The stack is described as *growing downwards*.

Definition: An **ascending stack** starts at a low memory address, and *increments* the stack pointer as data is written to the stack. The stack is described as *growing upwards*.

Second, there are two options for what the stack pointer addresses.

Definition: In a **full stack**, the stack pointer addresses the last memory location which contains data that was written to the stack.

Definition: In an **empty stack**, the stack pointer addresses the first empty memory location in the direction of the stack's growth.

An ARMv7 system uses a **full descending stack**. Similar to other concepts we've discussed (like endian-ness), the architecture of the stack is only important if you try to access it directly. Consider the following examples:

```
str r1, [sp, #-4]! @push to stack
```

The above is the correct way to manually push data onto a full descending stack.

- This uses pre-index addressing to change the value of the stack pointer *before* writing the data, so the data is written to the next empty memory cell.

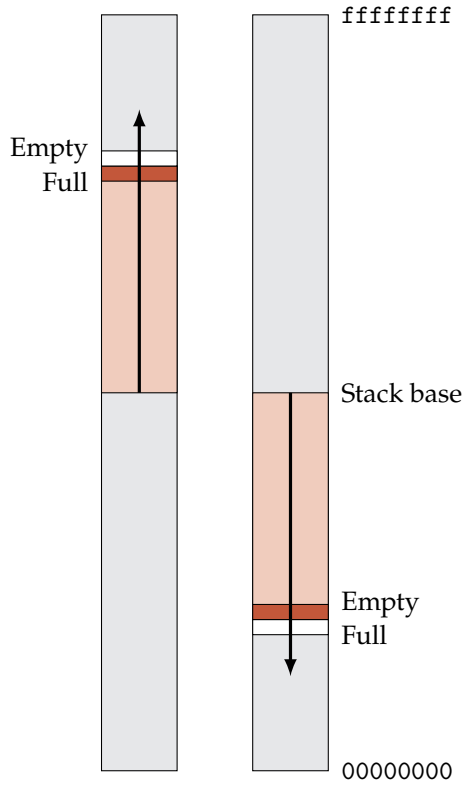

Figure 1: Schematics of different types of stacks. Left: an ascending stack, right: a descending stack. The stack pointer would address the dark red cell for a full stack, or the white cell for an empty stack.

- The pre-index is -4 bytes, since the stack is descending. A pre-index is necessary because a full stack is used (`sp` addresses a full memory element).

```
str r2, [sp], #-4 @push to stack - whoops!
```

The above is an incorrect way to manually push data onto a full descending stack.

- This uses post-index addressing. With a full stack, that will overwrite the last element in the stack before incrementing the stack pointer.

- However, this is the correct way to manually push data onto an empty descending stack.

Please note that this code will still compile and run without error. The CPU doesn't really "know" what is special about a stack — it is just an ordinary place in memory.

```
str r2, [sp, #4]! @push to stack - whoops!
```

Similarly, the above is also an incorrect way to manually push data onto a full descending stack — rather this is for a full ascending stack.

```
ldr r3, [sp], #4  @pop from stack
```

The above is the correct way to manually pop data out of a full, descending stack.

- This uses a post-index of 4 bytes which is appropriate for a full, descending stack.

- Post-indexing is necessary because the stack pointer for a full stack always points to the last data element.

```
lrd r4, [sp, #4]!  @pop from stack - whoops!
```

The above is an incorrect way to manually pop data out of a full, descending stack.

- By pre-indexing a shift of 4 bytes this will actually skip over the last element in the stack, and return the *second-last* element.

You can avoid all the hassle of trying to remember how the stack architecture is defined by using special mnemonics designed to streamline using the stack.

Mnemonic: To **push** data onto the stack, use the `push` mnemonic, followed by a list of registers within curly braces.

Mnemonic: To **pop** data out the stack, use the `pop` mnemonic, followed by a list of registers within curly braces.

With this in mind, you can push and pop data without worrying about the correct way to update the stack pointer — that will happen automatically. [4] However when pushing/popping multiple registers at once, it is important to recognize the order this occurs.

- If you `push {r0,r1,r2}`, then `r2` is pushed on the stack *first*, and `r0` is pushed to the stack *last*.
- Consequently, the first item removed when you pop data off the stack will be whatever was formerly in `r0`.

[4] *Warning!* In the simulator, you can always `pop` from the stack even if there is nothing in it! This will cause the stack pointer to "wrap around" and start reading the opcodes in the program itself.

- Similarly, if you `pop {r0,r1,r2}`, then the stack is popped into `r0` *first*, and into `r2` *last*.
- This way, `push {r0, r1, r2}` followed by `pop {r0, r1, r2}` does not change the contents of the registers.

As an explicit example:

```
mov r0, #3
mov r1, #15
mov r2, #255
push {r0, r1, r2}
pop {r2}
pop {r0, r1}
```

Here the registers are initialized with some arbitrary numbers, then pushed to the stack.

- In an ARMv7 system, the first (word) address in the stack is `0xfffffffc`. As the stack is a full descending stack, *before* anything is on the stack the stack pointer is `0x00000000`.
- When the registers are pushed to the stack, `r2` goes in first, to address `0xfffffffc`. Then `r1` goes to address `0xfffffff8`, and finally `r0` goes to address `0xfffffff4`.
- The contents of the registers is unchanged after pushing them to the stack.
- When the stack is popped to `r2`, the last stack value (the number pushed from `r0`) is moved to `r2`, and the stack pointer is incremented up to `0xfffffff8`.

- When the stack is popped twice, registers `r0` and `r1` are overwritten with the remaining two values.

This process is shown schematically in Figure 2. It is worth stressing, again, that the "stack architecture" is just a *programming convention*, not a *hardware implementation*. If you are coding in assembly language, you don't need to use the stack as defined by the system: You can set aside any arbitrary block of memory and use `str` and `ldr` with the appropriate pre- or post-index addressing to act as a custom stack. Furthermore `sp` can act as a general purpose register, so as long as you *don't* call `push` or `pop` you can use `sp` as the pointer for your custom stack.

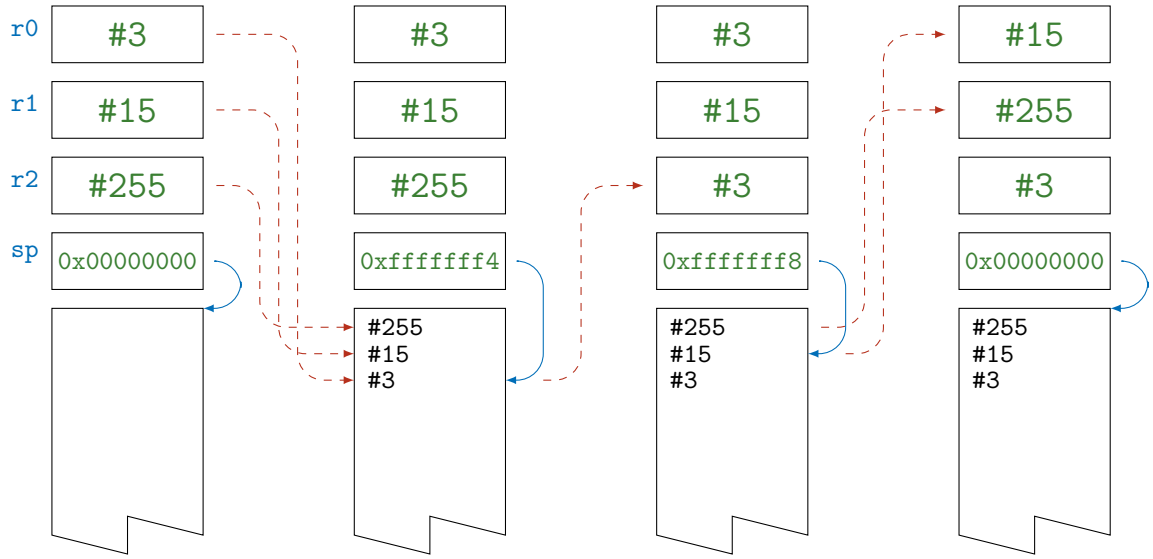

Figure 2: Representation of the code example showing how the stack contents, stack pointer, and the register contents change as data is pushed and popped from the stack. Note that in an ARMv7 system, the first word in the stack is address `0xfffffffc` (the first byte is address `0xffffffff`).

Subroutines

Good, reusable, and portable code uses the concepts of *structured programming*, and often relies on **subroutines**. [5]

Definition: A **subroutine** is a block of code that is called from the main code, performs some task, then returns to the same point in the main code. Subroutines often take one or more data values as inputs, and often return one or more data values after completion.

For simple programs, implementing a subroutine is as easy as using a **direct branch with link**, then after completing the subroutine, returning to the main code with a **indirect branch** using the link register, as was discussed above. Simplicity is often a good idea in programming, but this method doesn't always produce a *reusable* or *portable* subroutine. Some issues are:

- The CPU has a limited set of registers. How many registers were holding critical data from the main program, and how many are needed by the subroutine?
- If the subroutine is returning one or more values, which registers or memory addresses will be used for these data?

Note that the first issue is something that is foreign to programmers only familiar with high-level programming: in those languages, the set of possible variables is basically infinite, and subroutines often exist within their own **scope**, so new variables for manipulating

[5] In various languages these are also called **functions**, **methods**, **procedures**, **routines**,... and who knows what else.

data can be defined within the subroutine as necessary. But in assembly, where everything has to go through a limited of registers, data organization can be a important issue.
With this in mind, a "good" subroutine is one that *preserves the main program's status*, and *follows a standard protocol for input and output parameters*.

1. The first is accomplished by the subroutine code immediately **pushing** all relevant registers to the stack as soon as it is called, then **popping** these registers off the stack just before it exists. This ensures the data used by the main program is preserved, while still allowing those registers to be used temporarily by the subroutine.

2. The second is accomplished by just making sure all the subroutines you write follow the same protocol.

For writing subroutines, is *probably* a good idea to follow the same protocol everyone else uses:

- The first four registers (`r0` to `r3`) are used to pass parameters into the subroutine, and return values from the subroutine: when the subroutine is called from the main program any data in those registers can be assumed to be an input parameter, and when the subroutine returns to the main program any any data in those registers is assumed to be an output parameter. The main program's state is *not* preserved with these registers, they are *not* pushed or popped to the stack.

- The remaining registers (`r4` and up, as well as `sp` and `lr`) should be preserved after the subroutine returns to the main program: if any of these registers are needed by the subroutine, the original value should be pushed to the stack, and restored before the subroutine exits.

This is especially important if you are writing assembly code that may interact with code written using higher-level programming (C code, in particular) — the higher-level code will automatically follow this protocol after it is compiled. Following these standards also makes it easy to have arbitrarily nested subroutines.
Consider the example below. This is a subroutine that does the valuable chore of reading from an analog-to-digital converter (A/D converter), then adds some constant to it. Except — I am a bad student, and don't know how to use the A/D converter yet! So that part is hidden in another subroutine that I will write later.

- In my programs, I only use registers `r4` to `r8` for local variables — I don't trust registers higher than `r8` as they are shifty and of poor personal character [6] — so I only need to push those registers, and the link register, to the stack.

- I am preserving the state of the stack pointer by ensuring that I always pop for every push made during this subroutine.

- I only use `r4` in this example as a local variable, so technically I don't need to push `r5` to `r8` to the stack — this subroutine doesn't touch them, so their value will be preserved — but I want to keep the same standard "state preservation" for all of

[6] Your opinion of these registers may vary.

my subroutines. If I *really* cared about making my code as fast and efficient as possible, I would only push the registers that are needed by the subroutine.

- This subroutine expects the input parameter to be in `r0`, and the return value is also stored in `r0`.
- This subroutine calls two nested subroutines: `read_AD` to read the A/D converter, and `fix_big_number` that somehow "fixes" when an unsigned addition generates a carry-out. I don't know how either of these two subroutines actually work, but that is not important: as long as they preserve the state and put their output in `r0`, the actual details of their code is irrelevant to this program.

```
/* my super subroutine
   this reads input from an A/D converter
   and adds a constant to it.
   the constant should be in r0
   the output is returned in r0*/
read_AD_add_X:
   @ preserve state
   push {r4, r5, r6, r7, r8, lr}
   @ pass constant to r4
   mov r4, r0
   @ get A/D input
   @ I don't know how to do this!
   @ I will put it in a separate subroutine
   bl read_AD
```

```
    @ ok, somehow the A/D input is now in r0
    @ add constant to A/D input, save in r0
    adds r0, r4
    @ is result valid?
    @ a carry-out means hs condition is met
    blhs fix_big_number
    @ restore state
    pop {r4, r5, r6, r7, r8, lr}
    @ return to main code
    bx lr
```