ECE9047: Reference Material for ARMv7

John McLeod

Winter 2021

Overview of Online Simulator

The simulator is available at https://cpulator.01xz.net/?sys=arm-de1soc.

- Write assembly language code in the *Editor* window (the main window).
- Compile your code by pressing F5. Check the *Messages* window (at the bottom) for compile errors.
- If the code successfully compiles, the main window will change from the *Editor* to the *Disassembly* window.
- Press F2 to step through your code line-by-line, press F3 to continuously execute the code (and press F4 to stop execution).
- The CPU registers are shown in the *Registers* window on the left side. The contents of these registers should change as your program runs.
  - The *Program Counter* (`pc`) register should record the line number of the current piece of code.
  - The *Link Register* (`lr`) should record the line number of any code that is linked to after a branch.
  - The *Current Program Status Register* (`cpsr`) should update if compare or arithmetic operations are performed, since this register holds the Z, N, C, and V flags (but the register is displayed as a hexadecimal number, so it can be tricky to figure out which flag is being shown).
- Peripheral *Devices* are shown to the right. These can be operated by the appropriate code. The *memory-mapped address* of each device is shown at the upper right of each device window.

Language Reference

The following is a brief description of the parts of assembly that may be most useful for this lab. More complete references can be found online. Some good ones are:

- I found the Azeria Labs tutorial very helpful. https://azeria-labs.com/writing-arm-assembly-part-1/.
- Coronac has a more detailed tutorial that compares ARMv7 to other assembly languages and to GCC. Probably more detail than is necessary for this lab. https://www.coranac.com/tonc/text/asm.htm.
- The official documentation is available on the ARM website. http://infocenter.arm.com/help/index.jsp.

ARMv7 code has a lot in common with other computer codes, particularly that it is composed of *labels*, *keywords* (called *mnemonics*) and *comments*.

Labels

Labels are user- or compiler-defined text that is not directly related to instructions for the microprocessor.

- Labels for specific lines of code, variable names, or subroutines should end in a colon when defined in the code (`mylabel:`). This colon should be omitted when using that label in the code (`b mylabel`).
- Labels that help the compiler start with a period. These kind of labels are called *directives* (like `.data`, `.global`, or `.text`).

Comments

Comments can be added to code as: `@ here are some comments` or as: `/* here are some more comments */`. Comments are not necessary, of course, but since assembly language can often be hard to read, adding comments to your code is probably a good idea.

Mnemonics

Some of the mnemonics that you might need to use in this lab are:

`mov` Moves values from one register to another, or moves a hard-coded number into a register.

Example: `mov r0, #5` places the number 5 (a decimal number) in register `r0`.

`ldr` Loads a value from memory into a register. Also used to load an address (even a hard-coded address) into a register.

Example: `ldr r0, [r1]` places the number found at the memory address stored in register `r1` into register `r0`.

`str` Stores the value in a register into memory.

Example: `str r0, [r1]` places the number in register `r0` into memory at the address stored in register `r1`.

`cmp` Compares two values by subtracting them, updating the arithmetic flags, and discarding the answer.

Example: `cmp r0, #5` subtracts 5 (in decimal) from the value stored in register `r0`. The flags in the `cpsr` register are updated, but the actual number "`r0`-5" is thrown out.

`b` Branch (or "goto") to the label given.

Example: `b mylabel` will directly jump to the line of code with the label `mylabel:`.

`bl` Branch to the label given, but store the current place in code in the *link register* (`lr`). This is a simple way to implement a subroutine, as you can branch to this code from multiple places in your main code, and get back to the original point using the link register.

Example: `bl mylabel` will directly jump to the line of code with the label `mylabel:`, but will store the line number of the next instruction prior to branching in `lr`.

`bx` Branch to the address in the register given. Usually this is the link register (`lr`), that was set up using a previous `bl` call. This is a simple way of getting out of your subroutine back to the original place in your main code.

Example: `bx lr` will directly jump to the line of code stored in the link register `lr`.

`add` Adds two numbers together. This mnemonic does *not* update the arithmetic flags. Use `adds` to add and update.

Example: `adds r0, r1, r2` will add the contents of register `r1` to `r2` and store the result in `r0`, and also update the C, V, Z, N flags.

Second example: `add r0, r1` will add the contents of register `r0` to `r1` and store the result back in `r0`. This code is equivalent to `add r0, r0, r1`. This will *not* update the C, V, Z, N flags, since we used `add` instead of `adds`.

`sub` Subtracts two numbers. This mnemonic does *not* update the arithmetic flags. Use `subs` to add and update.

Example: `subs r0, r1, r2` will subtract the contents of register `r2` from `r1` and store the result in `r0`, and also update the C, V, Z, N flags.

Second example: `sub r0, r1` will subtract the contents of register `r1` from `r0` and store the result back in `r0`. This code is equivalent to `sub r0, r0, r1`. This will *not* update the C, V, Z, N flags, since we used `sub` instead of `subs`.

`push` Stores a number to the stack.

Example: `push {r0}` will store the value in `r0` to the stack.

`pop` Loads a number from the stack.

Example: `pop {r0}` will remove the top value from the stack and store it `r0`.

Generally speaking, the first argument for a mnemonic must be a register.

- The code `add r0, #4` will work, but the code `add #4, r0` will not.
- This is the reason why ARMv7 includes a *reverse subtract* mnemonic, so `sub r0, #4` implements "`r0`-4", and `rsb r0, #4` implements "$4-$`r0`".

Comparisons

The ARMv7 language allows two-letter conditions codes to be added to almost any mnemonic. This is how the `cmp` mnemonic is used, after a comparison these condition codes can be used to control decision-making and program flow. The condition codes are:

Condition Code	Type	Description	Status Flags Checked
`eq`	Any	Equal ($x = y$)	`Z=1`
`ne`	Any	Not equal ($x \neq y$)	`Z=0`
`mi`	Any	Negative, or "minus" ($x - y < 0$)	`N=1`
`pl`	Any	Positive, or "plus" ($x - y > 0$)	`N=0`
`vs`	Any	Overflow flag is set	`V=1`
`vc`	Any	Overflow flag is clear	`V=0`
`gt`	Signed	Greater than ($x > y$)	`Z=0` and `N = V`
`ge`	Signed	Greater than or equal ($x \geq y$)	`N=V`
`lt`	Signed	Less than ($x < y$)	`Z=1` or `N =` $\overline{\texttt{V}}$
`le`	Signed	Less than or equal ($x \leq y$)	`N`$\neq$`V`
`hi`	Unsigned	Greater than ($x > y$)	`C=1` and `Z=0`
`hs`	Unsigned	Greater than or equal ($x \geq y$)	`C=1`
`lo`	Unsigned	Less than ($x < y$) `C=0`	
`ls`	Unsigned	Less than or equal ($x \leq y$)	`C=0` or `Z=1`

An example of using these condition codes after a comparison is:

```
cmp r0, r1
addle r0, #5
bne mylabel
```

This code compares the values in registers `r0` and `r1`. If `r0` $<$ `r1`, the decimal value 5 is added to `r0`. If the two registers do not have equal value, the program branches to the label `_mylabel`. Note that because we used `addle` instead of `addsle`, the flags are not updated after the addition so the `bne` condition still applies to the original `cmp`.

Bitwise Operations

The ARMv7 language allows simple bitwise operations to be performed during a regular mnemonic code. These operations can be performed to the *second* operand in a mnemonic. Some of these bitwise operations are:

Operation	Description
`lsl`	Logical shift left
`lsr`	Logical shift right
`asl`	Arithmetic shift left
`asr`	Arithmetic shift right
`ror`	Rotate right
`rrx`	Rotate right with extend

Each of these operations is followed by a number, indicating the number of bits for the operation. Logical and arithmetic shifts fill in the missing bits with zeros, while rotate wraps around. The only difference between logical and arithmetic shifts is that arithmetic shifts preserve the *most significant bit* of the number, in case that is a sign flag.

For example, if an 8-bit register r0 holds the binary value "0110 0111", then `mov r1, r0, lsl #3` will put the value "0011 1000" into register r1. (From the number 0110 0111, add three zeros on the right to get 011 0011 1000, then discard the first 3 bits on the left to return to an 8-bit number.)
These bitwise operations are sometimes used to simplify math — as shifting all the bits left or right is a much quicker way to multiply or divide by 2 than actually using the ALU to calculate it.
Another, possibly more common use for these operations is when the 32-bit ARM registers need to communicate with peripherals that have larger or smaller memory cell sizes. For example, if a 32-bit number is recorded as two 16-bit numbers a and b (where a has the least significant bits) in some peripheral, these values can be combined into a single 32-bit register on the CPU by:

```
ldr r0, a
ldr r1, b
ldr r2, [r0]
ldr r3, [r1]
add r2, r3, lsl #16
```

Addressing Modes

Te ARMv7 language allows several methods of addressing memory. The simplest is to just use an address stored in a register, as we have seen: `ldr r1, [r0]`. We can also use the address stored in r0 as a *base address* an add literal shifts.
To illustrate, assume r0 stores the address 0x4000 0000.

- *Addressing with an offset* means we add a shift to the base address, without changing any stored addresses. The code `ldr r1, [r0, #4]` loads the value stored 4 bytes (32 bits) ahead of the address r0 (so address 0x4000 0004) into r1, without changing the value of r0.
- *Pre-indexed addressing* means we change the actual address stored in the register, then look up the value. The code `ldr r1, [r0, \#4]!` updates the address in r0 by 4 bytes (so now r0 stores the address 0x4000 0004), then loads the value at that address into r1.
- *Post-indexed addressing* means we look up the value according to the current address, then change the address by the offset. The code `ldr r1, [r0], \#4` stores the value at r0 =0x4000 0000 into r1, then changes r0 to 0x4000 0004.

These forms of addressing are very useful when reading/writing *arrays* of data. They are also useful when dealing with *memory mapped peripherals*. A complicated peripheral may have several control or interface addresses. In that case we can store the base address of the peripheral in one register and use offsets to access the different controls (instead of defining a long list of almost-identical addresses to access each part explicitly).

A Complete Example

Here is a complete example of an ARMv7 program, adapted from an example in the Azeria Labs tutorial.

```
.data
    var1: .word 17
    var2: .word 4

.text
.global _start

_start:
    ldr r0, adr_var1
    ldr r1, adr_var2
    ldr r2, adr_var3
    ldr r3, =0xff200000
    ldr r4, [r0]
    str r5, [r1]
```

```
    mov r5, #1
    str r5, [r3]
    bkpt

adr_var1: .word var1
adr_var2: .word var2
adr_var3: .word 0xff200020
```

The first part of the code (`.data`) defines two variables stored in memory (`var1` and `var2`). These are both "words", or 32-bit values (we can also define half-words and bytes, but there is little point in worrying about that for this lab).
The second part of the code (`.text`) contains the main routine (`_start`) and the actual code.
At the very end (technically still in the `.text` part), labels (`adr_var1`, etc.) are defined to help reference memory addresses of some variables (`var1`) and peripherials (`0xff200020`).
In the actual code, we load the addresses and values into various registers, and store some registers back into memory. Copy this code into the online simulator, compile (F5), and step through the program line-by-line (F2). Observe how the values in the registers (`r0` through `r5`, and the program counter `pc`) change. Once you see where the variable `var1` is stored in memory, you can check the *Memory* window to see if it is there.

- Notice that something is special about memory addresses `0xff200000` and `0xff200020`. What happens when we store the value `#1` to address `0xff200000`?

Branches and Subroutines

This lab requires you to write assembly language to write the first 10 Fibonacci numbers to the seven segment display. As such, it is not actually necessary to write reusable, general-purpose code to complete this lab.

- You can probably complete this lab just using the mnemonics `mov`, `ldr`, `str`, `cmp`, `b` and some condition codes.
- If you want to write more portable code, you can write simple, unnested subroutines using `bl myroutine` to branch to the labelled subroutine and `bx lr` to get back to the main code.
- If you want to write truly portable code that can be arbitrarily nested, you will need to `push` the current state of the system (at least the `pc` register, possibly several other registers to preserve system data) to the stack, and then `pop` that data off the stack after completing the subroutine.

Writing data to, and reading data from the stack isn't particularly difficult, but I don't think it is necessary for this lab.

The Seven-Segment Display

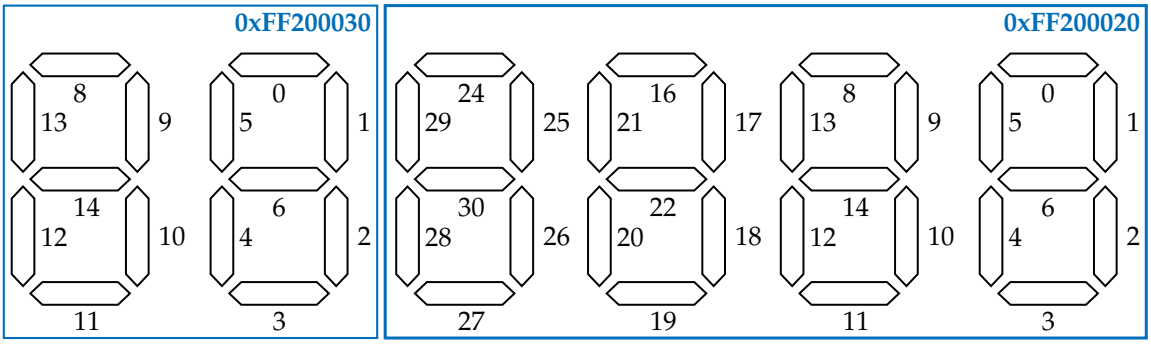
The ARM DE1-SoC has a six-digit seven-segment display. The lowest significant digit (right-most) of this display is memory mapped to `0xff200020`. Writing a number to this memory address will activate the display. For example:

```
ldr r0, =0xff200020
mov r1, #6
str r1, [r0]
```

Will cause the display to *represent* the decimal number 6. The only tricky thing about the seven-segment display is that *each segment is mapped to a single bit and is triggered individually*. Therefore, as you will see if you try the code above, writing the decimal number 6 to the display does not actually cause the display to show the number "6".

- For the 32-bit word starting at address `0xff200020`, bits 0 – 6 control the first digit of the display, bits 8 – 14 control the second digit, bits 16 – 22 control the third digit, and bits 24 – 30 control the fourth digit.
- For the 32-bit word starting at address `0xff200030`, bits 0 – 6 control the fifth digit of the display, and bits 8 – 14 control the second digit.

Therefore, to actually write the number "6" on a blank display, we would need to toggle bits 0, 2, 3, 4, 5, and 6: the binary number 0111 1101, equivalent to the decimal number 125.



- To reset, or blank out the display, it is necessary to write zeros to `0xff200020` (and `0xff200030` if you used those digits as well).
- Note that bits 7, 15, 23, and 31 are unmapped — it doesn't matter what value is written to those bits as it will have no effect on the display.

The Timer

The ARM DE1-SoC has several timers that can be used to time the program. The timer has a clock cycle of 100 MHz, and "ticks" once per cycle. A suitable timer is memory mapped to `0xff202000`. Working with this timer is tricky because it has several different memory addresses for different functionalities, and it is also tricky because this timer uses only 16-bit numbers, even though they are stored in 32-bit segments.

Name	Address Offset	Description
Status	[base]	Bit 0 is timeout flag, bit 1 is running flag.
Control	[base]+4	Bit 0 is enable interrupts, bit 1 is continue, bit 2 is start, and bit 3 is stop.
Low Period	[base]+8	Least significant 16 bits of the timer period, in clock cycles.
High Period	[base]+12	Most significant 16 bits of the timer period, in clock cycles.
Low Counter	[base]+16	Least significant 16 bits of the current timer counts.
High Counter	[base]+20	Most significant 16 bits of the current timer counts.

- Notice that setting the timer period requires splitting a 32-bit number into two 16-bit numbers.
- Likewise, reading the current timer counts requires combining two 16-bit numbers into a single 32-bit number.
- The section above, *Bitwise Operations* can help you do this.

For the control word, only 4 bits of the available 32 bits are used. Setting the *interrupt bit* allows interrupts — something too advanced to discuss here. Setting the *continue bit* tells the timer to immediately start counting down again (and reset the *timeout flag* in the status word) after completing a count down. Setting the *start bit* to be one (equivalent to writing decimal number 4, assuming we don't want interrupts or continuous count downs) will start the timer (notice the use of an *offset* from the timer base address):

```
ldr r0, =0xff202000
mov r1, #4
str r1, [r0, #4]
```

Similarly, writing decimal number 8 to the control word will stop the timer.

- After starting the timer, it will normally continuously count down. However the *timeout bit* in the status word will be set to one as soon as the timer completes the given *period*.
- It is important two write the period word first, before starting the timer!

For various (probably complicated and technical reasons), we should not try to read the period words to determine the current state of the timer. Instead, we should write *anything* to a counter word (the value we are writing doesn't matter, it is ignored) which tells the timer to update the current timer count into the counter words. Then we can read from the counter words like a normal memory address.
A suitable algorithm for using the timer is:

1. Write the countdown period, in terms of $100\,\mathrm{MHz}$ clock cycles, to the *period words*.
2. Start the timer by setting the *control word*.
3. Loop while loading the *status word* into a register.
4. In the loop, keep checking whether the *timeout bit* is set.
5. Once the timeout bit is set, exit the loop and stop the timer by writing the appropriate *control word*.

There are a few things to note about using the timer for this lab:

- There are other ways to obtain an approximate $1\,\mathrm{s}$ delay without using a timer.
- For step-by-step debugging (cycling through your code using F2), you should probably disable the $1\,\mathrm{s}$ delay timer — since it will involve millions (or even billions) of loops of code to complete.