COMP15111 notes

Todd Davies

December 22, 2013

Note, extra space has been allocated for the right hand margin to allow for more extensive margin notes. Also, it gives you space to make your own annotations.

Contents

1 Introduction **3**
1.1 A Computational Model . . . 3
1.2 Simple View Of A Computer . . . 3
1.2.1 Memory . . . 3
1.2.2 Bus . . . 3
1.2.3 Processor . . . 3
1.3 Three-address instructions . . . 3
1.3.1 Three address example . . . 4
1.3.2 Memory bottleneck . . . 4
1.4 Registers . . . 5
1.5 Instruction Styles . . . 5
1.5.1 One address . . . 5
1.5.2 Load-store . . . 5

2 ARM **6**
2.1 Assembly Language . . . 6
2.2 ARM instructions . . . 6
2.3 Transferring data between registers and memory . . . 6
2.4 ARM processing instructions . . . 6
2.5 ARM control instructions . . . 7
2.6 Stored programs and the Program Counter . . . 7
2.7 Fetch-Execute Cycle . . . 7
2.8 Decision Making . . . 7
2.8.1 An example . . . 8
2.9 Allocating memory . . . 8

3 Storing values **9**

4 ARM assembly programming **9**
4.1 Different types of values . . . 9
4.2 Loading and storing values in memory . . . 9
4.3 Endianness . . . 10
4.4 Addressing memory . . . 10
4.5 Instruction encoding . . . 10
4.6 Literals . . . 11

4.6.1 Negative literals . . . 11
4.7 Supervisor calls . . . 11
4.8 Pseudo instructions . . . 11
4.9 Loading an address into a register . . . 12
4.10 Directives . . . 12
4.10.1 DEF commands . . . 12
4.10.2 Align . . . 12
4.10.3 Entry . . . 12
4.10.4 EQU . . . 12

5 Arithmetic **13**
5.1 Making good use of registers . . . 13
5.2 Using literals in expressions . . . 14
5.3 Status flags . . . 14
5.4 Other useful arithmetic commands . . . 14

6 If and While **15**
6.1 If statements in ARM . . . 15
6.2 If else statements . . . 15
6.3 While statements . . . 16

7 Addresses and Addressing **16**
7.1 Direct Addressing . . . 16
7.1.1 Problems with direct addressing . . . 17
7.2 Register Indirect Addressing . . . 17
7.2.1 Address Arithmetic . . . 17
7.3 Offset Addressing . . . 18

8 Strings, bit shifts, rotations and tables **18**
8.1 Working with Strings . . . 18
8.1.1 Loading a String . . . 18
8.1.2 Finding the length of a String . . . 18
8.1.3 Getting the index of a character in a String . . . 19
8.2 Bit shifting and rotations . . . 19
8.3 Accessing a row in a table . . . 20

9 Stacks **20**
9.1 Pushing to a stack . . . 20
9.2 Poping from the stack . . . 22
9.3 Other ways of accessing the stack . . . 22
9.4 Stacks and method calls . . . 23

10 Methods **23**
10.1 Saving the value of registers . . . 24
10.2 Passing parameters . . . 24
10.3 Stack frames . . . 24

11 Switch statements **25**

12 Types of values **26**
12.1 Signed and unsigned integers . . . 26

1 Introduction

1.1 A Computational Model

The simplest, earliest, commonest, most important computational model is the **Von-Neumann Imperative Procedural Computer Model**

According to this model, a computer can:

1. Store information
2. Manipulate the stored information
3. Make decisions depending on the stored information

1.2 Simple View Of A Computer

The simplest model of a computer can be represented as:

$$Memory \Leftrightarrow Bus \Leftrightarrow Processor$$

1.2.1 Memory

Memory is a set of locations which can hold information, such as numbers(or programs). Each memory location has a unique (numerical) address, and there are typically thousands of millions of different locations. There are various ways of depicting memory; a common one is a 'hex dump' that often looks something like this:

Run the command *hexdump* to generate hexdumps.

Address	**Values (8 bit numbers)**	**Characters**
00000000	48 65 6C 6C 6F 0A	Hello.

Each item that is in the memory has a unique address.

1.2.2 Bus

A bus is a bidirectional communication path. It is able to transmit addresses and numbers between components inside the computer.

1.2.3 Processor

The processor obeys a sequence of instructions, commonly referred to as a program. Historically the processor was often referred to as a CPU, however, this is inappropriate nowadays since typical processors consist of several processing cores.

1.3 Three-address instructions

Every kind of processor has a different set of instructions, real world examples include: Pentium, ARM and others

Each three-address instruction:

1. Copies the values from any two memory locations and sends them to the processor (source operands)
2. Copies some operation e.g. adds the copied numbers together
3. Copies the result back from the processor into a third memory location (destination operand)

For example, if we wanted to convert the Java code $sum = a + b$; into a three-address instruction we would:

1. Identify the two *source operands*: a holds 2, b holds 3
2. Perform the *operation*: $2 + 3 = 5$
3. Let the variable *sum* equal the answer 5. This is the *destination operand*

1.3.1 Three address example

Question: Convert the Java code *product = c * d;* into the three- address style and draw a two box view of it.

First we need to re-write the Java code in the three-address style:

$$product \leftarrow c * d$$

Now we can draw the box view of it:

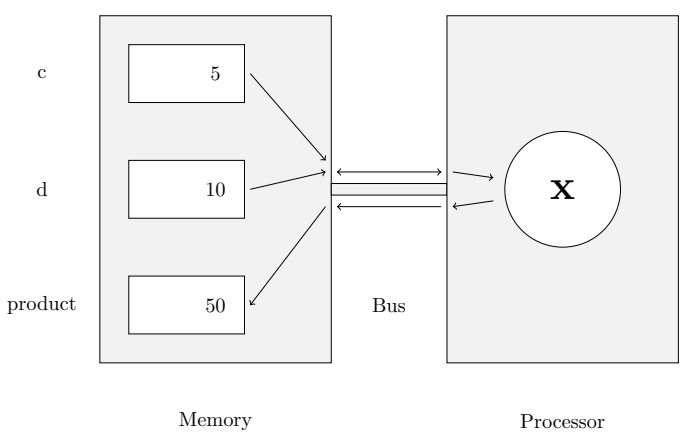


Figure 1: An example of the two box model

1.3.2 Memory bottleneck

Most processors can process instructions faster than they can be fed by memory. Each instruction in the three-address cycle requires four memory cycles:

1. Fetch the instruction
2. Read the first operand
3. Read the second operand
4. Write the result to memory

Each of these memory cycles could take hundreds of processor clock cycles to complete, and so in this time the processor would be doing nothing. However, most modern processors employ a *cache* to temporarily store commonly accessed memory locations, and so avoid some of the memory cycles.

1.4 Registers

Registers are very small amounts of storage build into a processor. Since they are inside the processor data doesn't need to be transferred over the bus, and so they are very fast. Registers are used instead of the main memory which speeds up program execution.

Each register can only hold one value and each processor will only generally have a few dozen registers (e.g. ARM has sixteen).

1.5 Instruction Styles

1.5.1 One address

The one address style can only use up to one memory location in each instruction, all other operands must be registers. An example may be:

$$R1 \leftarrow R0 + \textit{memory location}$$

1.5.2 Load-store

The load-store style cannot perform operations on memory locations at all. Instead, values from memory must be loaded into a registers before the operation takes place and then the operation can be performed on the registers. Following the operation, the result must be stored back into memory again.

$$\begin{aligned} R1 &\leftarrow \textit{memory location} \\ R1 &\leftarrow R0 + R1 \\ \textit{memory location} &\leftarrow R1 \end{aligned}$$

This means that we need extra instructions to do stuff with memory locations:

1. **Load** the value from memory into a register before the operation.
2. **Store** the value in the register back to memory after the operation.

For example, the Java code *Sum = a + b + c;* would be run as:

R1	←	a	(i.e. load from a)
R2	←	b	(i.e. load from b)
R3	←	R1 + R2	(i.e. a+b)
R4	←	c	(i.e. load from c)
R5	←	R3 + R4	(i.e. (a+b)+c)
Sum	←	R5	(i.e. store to sum)

You can see that the load-store style favours lots of very simple, very fast instructions.

2 ARM

Computers obey programs which are sequences of instructions. Instructions are coded as values in memory. The sequences are held in memory adjacent memory locations. Values in memory can be interpreted as you please, from numbers to text, images or anything really!

Any given set of binary digits can be read as a decimal number, but not always as text, so values in memory are often represented as numbers for convenience.

2.1 Assembly Language

Assembly language is a means of representing machine instructions in a human readable form.

Each type of processor has its own assembly language (since each language is specific to a partial architecture) but they typically have a lot in common:

- A mnemonic, that specifies the type of operation
- A destination, such as a register or memory location
- And one or more sources that may be registers or memory locations.
- Possibly with a comment too which will help programmers understand what's happening and aren't interpreted by the assembler.

When a program has been written in assembler, it must be *assembled* by an *assembler* to run it.

2.2 ARM instructions

ARM has many instructions but we only need three categories:

- Memory operations that move data between the memory and the registers.
- Processing operations that perform calculations using value already in registers.
- Control flow instructions are used to make decisions, repeat operations etc.

2.3 Transferring data between registers and memory

Memory operations load a register from the memory or store a register value to the memory.

For example, a into register 1 ($R1 \leftarrow a$) we would write: `LDR R1, a`

Or to store the value in register 5 into sum ($sum \leftarrow R5$): `STR R5, sum`

In these examples, a and sum are aliases for the addresses of memory locations.

2.4 ARM processing instructions

ARM has many different instructions to perform operations such as addition, subtraction and multiplication.

The syntax for such operations is usually:

```
[operand] [destination register] [register 1] [register 2]
```

For example, to add two numbers together, we might write:

```
ADD R2, R0, R1
```

This will add the value of R0 to the value of R1 and store it in R2.

2.5 ARM control instructions

The most common control instruction is the branch. Similar to `GOTO` in other languages, a branch will change the PC register (see section 2.6) to another value so the order of execution of the program is changed.

Branches can be made to be conditional by appending a conditional operator (coming up later) on to the command.

The syntax is something like:

```
B[conditional operator] [branch name]
```

Some examples of different conditional operators are:

Command	Function
`B`	Branches to a different location in the code.
`BNE`	Branches, but only if the previous condition was false.
`BEQ`	Branches, but only if the previous condition was true.

2.6 Stored programs and the Program Counter

A computer can make decisions, and choose which instructions to obey next depending upon the results of those decisions. A **Program Counter** (PC) register is used to hold the memory address of the next instruction to be executed. ARM uses register 15 as its PC.

2.7 Fetch-Execute Cycle

The processor must first fetch instructions from memory before it can execute them. This is called the fetch-execute cycle, and it involves:

1. **Fetch**: copy the instruction, pointed to by the PC, from memory and set PC to point to the next instruction
2. **Execute**: obey the instruction (exactly as before)
3. Repeat.

In ARM, the PC starts with a values of `0x00000000` when the program is initially run. On each cycle of the Fetch-Execute cycle, the PC is incremented by 4, since instructions each occupy 4 memory locations.

2.8 Decision Making

In order to make decisions, the computer mustn't just execute instructions one after the other in a linear manner. Instead, branches must be used to change the sequence of instructions to be executed.

In order to perform a conditional branch, we must first perform a compare command to perform the comparison before we do the branch.

2.8.1 An example

If we wanted to do a 1 discount on a shopping list if the price was over 20, we would do the following:

```
                  LDR     R0, total       ; Load the total price
                                                  ; into R0

                  CMP     R0, #20         ; Compare R0 and 20
                                                  ; (the literal)
                  BLT     nodiscount      ; If the price is too low,
                                                  ; then don't discount
                  SUB     R0, #1          ; Deduct 1
                  STR     R0, total       ; Store the result back
                                                  ; into memory

  nodiscount      SVC     2               ; Finish

  total           DEFW    25              ; Lets say the total is \$25
```

2.9 Allocating memory

The `DEFW` (define word) operation puts a value in memory before the program is run. Any define operation is executed before the program is run.

The actual memory location that is used to store the value isn't known to the running program, however, an *alias* is attached to the memory location by the programmer and the memory location can be referenced through that.

The syntax for the `DEFW` command is as follows:

```
myage DEFW 18
```

Where `myage` is the alias and `18` is the value.

`DEFW` can also be used to define a number of words:

The label is associated with the lowest address (i.e. 0)

```
squares DEFW 0, 1, 4, 9, 16, 25
```

`DEFB` stores a single byte in memory. It is useful for strings such as "hello":

```
hi DEFB "hello"
```

`DEFS` sets a block of bytes to a set value:

```
reserved_space DEFS 10, 5
```

The above will set 10 bytes to the value '5'.

3 Storing values

There are many ways to store data. For example, we could store what lights are one in a traffic light in many different ways. First, we must decide how many different states the traffic light can be in:

`Red, Red Amber, Green, Amber`

You can see that we have four states. This could be represented in binary as two bits:

`00`	Red
`01`	Red Amber
`10`	Green
`11`	Amber

We could also store the states as their binary representations of their names:

R	`01010010`
RA	`0101001001000001`
G	`01000111`
A	`01000001`

You can see though, that this isn't as efficient as storing just two binary digits.

4 ARM assembly programming

4.1 Different types of values

ARM has the capacity to work with many different types and sizes of values. Each type has a different use case. The main ones are described below:

Name	**length**	**Use**
Byte	8 bits	Used for characters
Word	32 bits	Used for integer es, addresses and instructions

There are other types too (such as the halfword and doubleword) but they aren't needed for this module.

ARM processors require that memory locations are aligned. This means that values stored in memory start at specific places. For example, a word address must be a multiple of four.

This means that after a `DEFW` statement, the `ALIGN` command must be called (See 4.10.2 for more on the `ALIGN` command.).

4.2 Loading and storing values in memory

The commands `LDR` and `STR` are used to move values between memory and registers. The commands are detailed in full below:

Command	**Function**
`STR`	Copies the whole (32 bit) register into memory.
`LDR`	Loads a 32 bit word from memory into a register.
`STRB`	Stores a single 8 bit byte into memory from a register.
`LDRB`	Loads a byte from memory into a register. The upper 24 bits of the register are zeroed.

4.3 Endianness

Endianness is a property of a memory location that defines the order of the bits. There are two types of endianness, **little endian** and **big endian**.

In the word `0x12345678` there are four bytes:

- `0x12`
- `0x34`
- `0x56`
- `0x78`

In little endian, the first byte would be `0x12` since bits are read from left to right in little endian.

In big endian, the first byte would by `0x78` since bits are read from right to left in big endian.

This is important when we deciding what the most and least significant bits in a word are. For example, in this instance the *lsb* is `0x12` in little endian, but `0x78` in big endian.

N.b. The least significant bit is the smallest address.

In this course, little endian is used, though ARM can use either.

4.4 Addressing memory

ARM uses 32 bit addresses, so there are 2^{32} different bytes that can be addressed in memory (or $\frac{2^{32}}{4}$ different words). However, there is no guarantee that the system on which the program is running will have that much memory available.

4.5 Instruction encoding

Each ARM instruction is encoded into a four byte word. The exact meaning of each of the bits varies per instruction.

For example, in the branch instruction, the first four bits specify the condition, the second four bits represent the actual operation to perform (i.e. branch) and the remaining twenty four bits define the memory location of the next instruction to branch to.

However, this presents a problem. We only have twenty four bits with which to define the next location to branch to, which allows us to define 2^24 different locations. However, there are 2^32 possible addresses that we could use!

This problem is overcome by treating the 24 bits as an offset to the address of the current instruction. This works since most of the time, the address that is being branched to is fairly close to the current instruction.

In order to be able to branch to addresses before and after the current instruction, we must use two's complement to allow signed integers to be used to specify the offset. This means we can branch to any instruction at an address $\pm 2^{23}$ from the current instruction.

4.6 Literals

ARM is able to encode literal values into instructions. This saves time having to access registers or memory in order to perform operations such as arithmetic.

An example is to increment a register:

```
ADD R1, R1, #1
```

However, ARM only assigns up to 12 bits for a literal value, so we can only have 2^{12} values. However, ARM employs a strange method of encoding these values so that more useful values are available (for example, #512 is allowed, but #257 isn't).

4.6.1 Negative literals

Technically, ARM doesn't support negative literals, however, the assembler will usually be able to find a way to implement them. Some examples are given below:

`CMN` is compare negative.
`MVN` is move not.

```
ADD R1, #-1  →  SUB R1, #1
CMP R2, #-2  →  CMN R2, #2
MOV R3, #-3  →  MVN R3, #3
```

4.7 Supervisor calls

Supervisor calls are functions implemented by the operating system, not ARM itself. The parameter of an `SVC` call defines it's exact operation.

SWI is another name for SVC - they do the same thing.

In this module, the SVC call does the following for each parameter:

`SVC 0`	Output a character
`SVC 1`	Input a character
`SVC 2`	Stop execution
`SVC 3`	Output a string
`SVC 4`	Output an integer

4.8 Pseudo instructions

The ARM assembler provides some instructions that are translated into sequences of more complicated instructions at the time of assembly for our convenience.

One such instruction is loading a literal into a register. This is done using the `LDR` command as usual, however a literal is used with the '=' character instead of a '#'. E.g. to load the value 100 into register one, we do:

```
LDR R1,=100
```

However, this is a pseudo instruction and will be converted by the assembler to:

```
MOV R1,#100
```

However, if the number is very large, it becomes:

```
constant DEFB 100
LDR R1, constant
```

4.9 Loading an address into a register

The `ADR` command loads an address into a register, for example:

```
constant        DEFB    100
                ADR     R1, constant
```

Will load the memory address of `constant` into register one.

4.10 Directives

Directives are evaluated at the time of assembly.

4.10.1 DEF commands

The `DEF{W,B,S}` command reserves an amount of memory dependent on the operation used (see the table) and puts an initial value in it.

Command	Function
`DEFW num`	Reserves a *word* of memory and puts the initial value `num` in it.
`DEFB value`	Reserves *byte*(s) of memory and puts the initial value `value` in it. Note that the value can be a string literal, in which case the number of bytes reserved will be equal to the length of the string.
`DEFS size, fill`	Reserves a *block* of memory of `size` bytes and initialises them with the value `fill`.

4.10.2 Align

The align command leaves as many blank bytes as needed so that the next item in memory will start at a word boundary (a multiple of 4).

4.10.3 Entry

Sets the PC at the start of the program (i.e. where the program should start from)

4.10.4 EQU

Allows you to name a literal, which can go a long way to making the code more maintainable. We could define the literal 18 as *drinking_age*:

```
drinking_age    EQU     #18
                ; Check the person is over 18
                CMP     R1, #drinking_age
                BLT     too_young
```

5 Arithmetic

5.1 Making good use of registers

Registers are a precious resource when programming on ARM. When writing software to evaluate expressions, it's often tempting to load all the variables into registers first, and then perform the arithmetic in separate registers like so:

```
; a = b + c + d
LDR     R1, b
LDR     R2, c
LDR     R3, d
ADD     R4, R1, R2
ADD     R5, R4, R3
STR     R5, a
```

However, this is pointless - the values in the registers `R4` and `R5` aren't going to be needed again, so we may as well do:

```
; a = b + c + d
LDR     R1, b
LDR     R2, c
LDR     R3, d
ADD     R1, R1, R2
ADD     R1, R1, R3
STR     R1, a
```

But, we can optimise even further here. Instead of loading all the variables into registers before we do arithmetic, we can save a register and load only the ones we need before each `ADD` instruction:

```
; a = b + c + d
LDR     R1, b
LDR     R2, c
ADD     R1, R1, R2
LDR     R2, d
ADD     R1, R1, R2
STR     R1, a
```

Sometimes, it's useful to re-order an arithmetic expression so it can be implemented using less registers. This can often be achieved by increasing the nesting of brackets in an expression such as this:

$$(b - e) + (c * d) = ((c * d) + b - e)$$

Though these expressions are both equal, the right hand side will use a register less in ARM code, since each instruction can be executed sequentially, however the left hand side requires two expressions to be evaluated (and stored in a total of three registers) and then both expressions added together.

5.2 Using literals in expressions

If there is a literal in the expression you want to evaluate, then it's possible to use a literal instead of a register. For example, these two programs will end up with the same answer in `R0` but the one on the right uses less registers:

```
        ; (a + 5) * b
        LDR     R0, 5
        LDR     R1, a
        ADD     R0, R0, R1
        LDR     R1, b
        MUL     R0, R0, R1

five    DEFW    5
```

```
        ; (a + 5) * b
        LDR     R0, a
        ADD     R0, R0, #5
        LDR     R1, b
        MUL     R0, R0, R1
```

Literals can be any expression that the assembler can evaluate, for example, the following are all valid:

```
ADD R0, R0, #(1 + 2)

ADD R0, R0, #-2

ADD R0, R0, #(2 - 1)
```

Note that `MUL` cannot use literals. To get around this, first `MOV` the literal into a register and then multiply.

5.3 Status flags

ARM has some 1-bit status flags that are set after a `CMP` instruction as shown below:

Flag	Meaning
Negative	Previous result was negative
Zero	Previous result was zero
Carry	Previous add or subtract generated a carry
Overflow	The previous add or subtract overflowed and went out of range

You can get any data operation to alter the flags by appending `S` to the instruction. For example:

```
SUBS R0, R1, R2
```

The above command will subtract `R2` from `R1` and store the result in `R0`, but it will also set the flags according to the value of `R0`. For example, if the value in `R0` was negative after the operation, the `Negative` flag would be set.

5.4 Other useful arithmetic commands

The `RSB` command is reverse subtract, and is useful for negating a literal:

```
RSB R1, R0, #0; R1 = 0 - R0 = -R0
```

The `MLA` command is multiply and add. It can only use registers as operands.

```
MLA R1, R2, R3, R4 ; R1 = (R2 * R3) + R4
```

6 If and While

6.1 If statements in ARM

Given an if statement written in Java such as this:

```
if(condition)
{
    actionStatement;
}
```

How would we convert that into ARM assembly? The best way is to do the following:

1. Check to see if the condition is else
2. If it is, then branch to the end of the if statement
3. Otherwise, perform the action statement

An example implementation may be:

```
        CMP     R1, R2  ; Compare two registers
        BGE     endif   ; The condition
        MOV     R1, #0  ; Whatever the action
                                        ; statement may be
endif
```

Note that it is often easier to implement the inverse of some comparisons. For example, if we were evaluating `(a >= b)` we could do either of these:

- 1. Check whether `a` is greater than `b`
  2. Check whether `a` is equal to `b`
  3. Work out how to branch away if both are false
- 1. Inverse the comparison and check if `a` is less than `b`
  2. Branch away if it is

6.2 If else statements

If else statements are much like if statements, except that if the condition is false then you branch to the else action, and if the condition is true, then you branch to after the else statement once you've executed the action. Here's an example:

Lets implement `if(a==b)  a += 1; else  b += 1;` :

```
        LDR     R0, a
        LDR     R1, b
        CMP     R0, R1  ; Compare
        BNE     else    ;If they aren't equal
        ADD     R0, R0, #1
        B       end
else    ADD     R1, R1, #1
end
```

6.3 While statements

Implementing a while statement in ARM assembler is very similar to implementing an if statement, except after you have executed the action statement, you branch back to the initial condition again, like so:

Note that if the code in your while loop is being executed repeatedly, then it's probably worth optimising it!

```
        ; Compare two registers
        start   CMP     R1, R2
        ; The condition
        BGE     endif
        ; Whatever the action statement may be
        ADD     R1, R1, #1
        ; Branch back to the start
        B       start
endif
```

In ARM, we can make *any* instruction have a conditional code on it. Consequently, we can get rid of a branch instruction, like so:

```
        ; Compare two registers
        start   CMP     R1, R2
        ; Add only if the flag is less than
        ADDLT   R1, R1, #1
        ; Branch back to the start
        BLT     start
endif
```

You can compare the result of an arithmetic instruction with zero by appending `S` to the instruction like so:

```
start   ADDS    R0, #1
        BNE     start
endif
```

This will keep adding `1` to `R0` until it reaches 0. This is often useful when translating for loops that count up to a number. Instead of counting up, you can count down using the `SUBS` instruction and only iterate if the result isn't zero.

7 Addresses and Addressing

When a processor references memory it needs to produce an address.

The address needs the same number of bits as the memory address. i.e. 32 in ARM

addressing modes - mechanisms for generating addresses

7.1 Direct Addressing

Direct addressing is a mode where the address is simply contained within the instruction.

This requires an instruction longer than the address size which is a problem because ARMs maximum bit length is 32.

So far, we assumed that direct addressing uses LDR/STR instructions, for example:

```
LDR   R0,   b
LDR   R1,   c
ADD   R0,   R0, R2
STR   R0,   a
```

This looks like direct addressing but on ARM it's 'faked' by the assembler as a pseudo-instruction

7.1.1 Problems with direct addressing

ARM: both instructions and addresses are 32 bits, but the instruction also specifies operation so it can't contain every possible address.

Solution: allow a register to contain an address, use the address in the register to do loads and stores.

This is **Register Indirect Addressing**

7.2 Register Indirect Addressing

The address is held in the register

It takes only a few bits to select a register (4 bits in the case of ARM R0-R15)

A register can (typically) hold an arbitrary address (32 bits in the case of ARM)

ARM has register indirect addressing

Example loading a register from a memory location: LDR R0, b

Could be done using register indirect addressing:

```
ADR   R2,   b      Move the address of b into R2
LDR   R0,   [R2]   Use address in R2 to fetch the value of b
```

This is still a bit limited - addresses are:

- range limited (within ADR 'instruction')
- fixed

but:

- ADRL pseudo-op allows larger range (at a price)
- having addresses a variable once it is often used again
- variable are usually 'near' each other

7.2.1 Address Arithmetic

We can operate on registers, so we can:

- store/load/move addresses
- do arithmetic to calculate addresses

Rather than use e.g. extra ADD instructions, we often use **Base + Offset Addressing** - address addition done within the operand.

We have actually been using this all along: Base = PC register.

7.3 Offset Addressing

In offset addressing the address is calculated from a register value and a number.

The register specifier is just a few bits, The offset can be 'fairly small'.

With one register 'pointer' any of several variables in nearby addresses may be addressed.

ARM allows offsets of 12 bits in LDR/STR

These bits can be added or subtracted, for example:

```
LDR   R0,   [R1, #8]
STR   R3,   [R6, #-0x240]
LDR   R7,   [R2, #short-constant]
```

This provides a range of ± 4 kilobytes around a 'base' register

In practice this method is adequate for most purposes.

8 Strings, bit shifts, rotations and tables

8.1 Working with Strings

A String is just a list of bytes where each byte represents a character. The last byte in the list is the null byte which contains the value `0`.

8.1.1 Loading a String

In order to load the String into a register, the `ADRL` command must be used, for example:

```
msg     DEFB    "My message", 0
        ALIGN

        ADRL    R0, msg
        SVC     3
```

8.1.2 Finding the length of a String

In order to find the length of a String, we need only to loop over each byte in the String until we get to the null byte, and add up how many iterations we've done as we go along:

```
        ADRL    R1, message
        MOV     R2, #0
count   LDRB    R0, [R1, R2]
        CMP     R0, #0
        ADDNE   R2, R2, #1
        BNE     count
```

Here, we use `R1` to store the message, `R0` to read each character, and `R2` to keep the count.

```
        STR     R2, length
```

8.1.3 Getting the index of a character in a String

In order to do the equivalent of `String.indexOf(character)`, we can do something similar to finding the length of the String, but we only need to loop until we get to the character we're looking for.

```
        ADRL    R1, message
        LDRB    R2, character
count   LDRB    R0, [R1] #1
        CMP     R0, #0
        BEQ     end
        CMP     R0, R2
        BNE     count
end     ADRL    R0, message
        SUB     R1, R1, R0
        SUB     R1, R1, #1
        STR     R0, index
```

Note how this snippet has an optimisation to use one less register during the looping process than the string length snippet does. It only uses one register to loop over the string and then works out the number of times it's looped after the character has been found.

8.2 Bit shifting and rotations

Shift operations move all the bits in a word in one direction or another. For example:

Shift direction	Resultant word
No shift	`10011011`
Left shift	`00110110`
Right shift	`01001101`

Shifting to the left is like multiplying by two, and shifting to the right is like dividing by two.

You might have noticed that a zero is appended to whatever side the bits are moving away from in order to ensure that the word is still the same number of bits as before.

Rotation operations are very similar to shift operations, except instead of padding the word with zeroes, the bit that was lost is appended to the other side of the word:

Rotation direction	Resultant word
No rotation	`10011011`
Left rotation	`00110111`
Right rotation	`11001101`

ARM has four different operations for shifting and rotations:

Mnemonic	Meaning	Function
`LSL n`	Logical shift left	Shifts left by n bits. Any bits that are lost are zeroed.
`LSR n`	Logical shift right	Shifts right by n bits. Any bits that are lost are zeroed.
`ASR n`	Arithmetic shift right	Shifts right by n bits. Any bits that are lost are set as the sign bit (to preserve the signed bit).
`ROR n`	Rotate Right	Rotates right by n bits.

8.3 Accessing a row in a table

We can use bit shifts to access a row in a table:

```
          ADRL    R1, table
          LDR     R2, [R1, R0, LSL #2]

table     DEFW    0
          DEFW    3
          DEFW    6
          DEFW    9
          DEFW    12
          DEFW    15
          DEFW    18
```

This snippet will allow us to access the row with an index stored in `R0`, so if `R0` was `2` then the output in `R2` would be `6`.

9 Stacks

A stack is a data structure that is used by ARM assembly (an in a variety of other computing applications) to help maintain the state of a program.

The only operations that can be performed on a Stack are `push` and `pop`. The former adds an item onto the stack and the latter removes the last item from the stack. Because these are the only two operations, a Stack is a type of *Last In First Out* data structure.

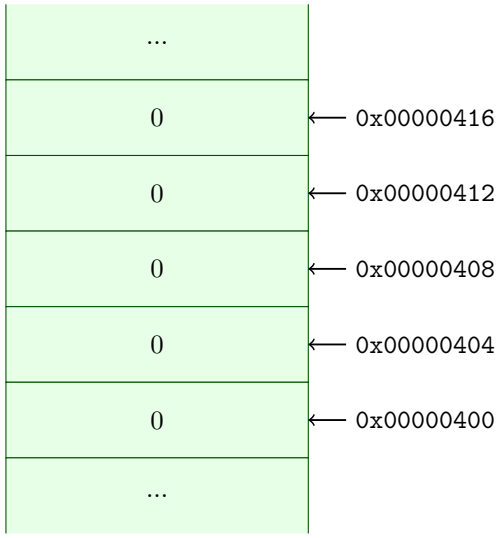
is usually initialised using the `DEFS` command to reserve a block of space in memory. This means that the stack will be a fixed size.

In ARM assembly, one register is reserved as the Stack Pointer (SP). This keeps track of the memory location that addresses the last item in the stack.

Stacks can be implemented where the stack pointer starts at the highest memory address in the range allocated to the stack, or where it starts at the bottom. In ARM assembly, it is usual to start the stack pointer at the highest address and work down.

9.1 Pushing to a stack

Lets initialise a stack of five memory locations:



The stack pointer will currently be pointed at `0x00000420`, so the next stackable memory location will be `0x00000416`.

If we want to *push* an item onto the stack, we need to do two things:

- Write the value to the current memory location addressed by the stack pointer.
- Decrement the stack pointer by `4`

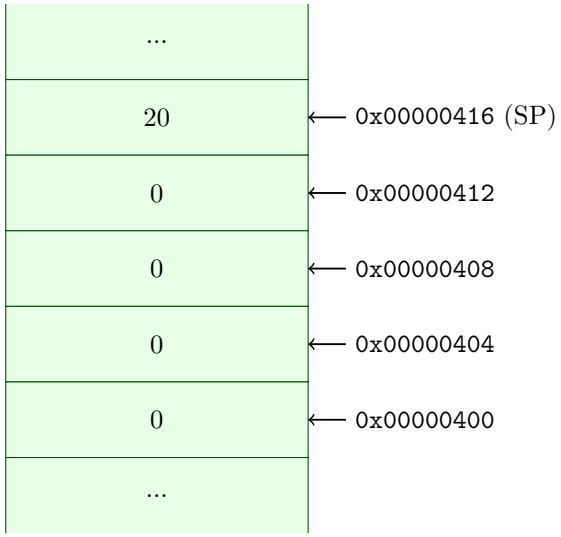
The ARM command for a stack push is `PUSH`. An example might be:

```
PUSH  \#20
```

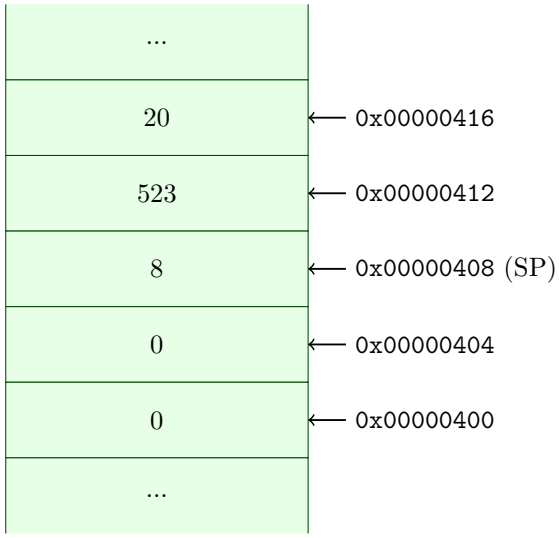
Or if you wanted to push one or more registers:

```
PUSH  R1, R2, R3
```

If we executed the first of these commands (`PUSH #20`), the stack would look like this:



Executing `PUSH R2, R3` when the values of `R2` and `R3` were `523` and `8` respectively would result in a stack such as:

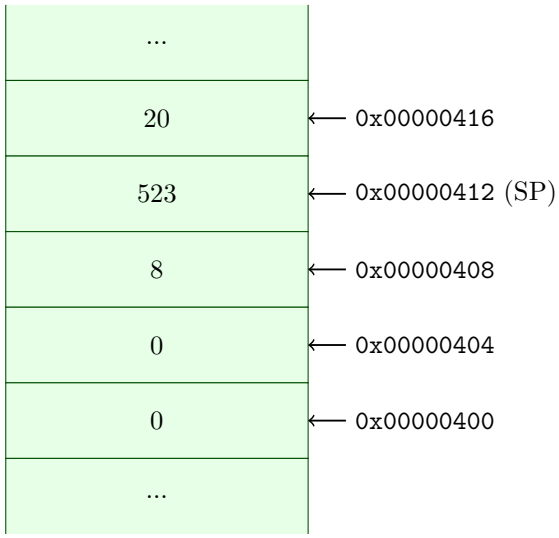


9.2 Poping from the stack

The ARM syntax to pop an item from the stack is very similar. To pop the last item from the stack into `R1` the command would be:

```
POP  R1
```

The stack would now look like:



And the value of `R1` would be `8`. This works for multiple registers too just like the `PUSH` command does.

Note that we don't actually have to reset the memory location `0x00000408` since it will be overwritten on any subsequent push and since the stack pointer is behind it, it will never be read from.

9.3 Other ways of accessing the stack

It's important to remember that the stack is just a set of memory locations, and the stack pointer is just a normal register. This means that other commands for accessing memory can also be used to access and manipulate the stack. For example, take a look at this block:

```
        LDR     R1, SP
        ADD     SP, SP, #4
```

Of course, we could use post indexed addressing in order to do the same thing: `LDR R1, [SP], #4`

This will load the value of the memory location addressed by the value inside the stack pointer into `R1` and then increment the stack pointer by 4. Essentially, this is doing the same thing as `POP R1` will do.

In order to do the same thing for pushing to the stack we can do:

```
        SUB     SP, SP, #4
        STR     R1, SP
```

Which is equal to `STR R1, [SP, #-4]!`

9.4 Stacks and method calls

It is important that when a program branches from one method to another, the values stored in the registers are not overwritten for when the program branches back to the original method.

This problem is resolved by pushing the values stored in the registers onto the stack before a branch is made, and then poping them off the stack once the method has finished executing. Alternately, the method that is being called can preserve the values in the registers by pushing and poping them to and from the stack at the start and the end of the method. Obviously, only one such method needs to be employed, so it's best to stick to one convention for the whole program.

10 Methods

The most basic way of calling a method in ARM assembly is by using the `BL` command. This will do two things:

1. Move the current value of the program counter into the link register.
2. Branch to the label defined in the instruction.

The method being called will use move the value of the link register back into the program counter once it has finished execution. Here's an example:

```

        start   B main

 add R1R2       ADD     R1, R1, R2
        ; Move the value of the Link
        ; Register back into the PC
        MOV     PC, LR

 main   MOV     R1, #4
        MOV     R2, #2
        BL      addR1R2

```

This program will move the values `4` and `2` into `R1` and `R2` respectively, and then branch to a method that will add the two registers together and store the result in `R1`.

10.1 Saving the value of registers

What if a method used lots of registers internally. We should make sure that the state of the registers is the same at the end of the method as they are at the start of the method, since otherwise there could be issues with corrupted data.

The easiest way to do this is to stack the parameters, like so:

```
method
        ; First lets store the registers
        ; we're going to use into memory
        PUSH    {R0-R2}
        ; ==========================
        ; Do stuff with the registers
        ; ==========================
        POP     {R0-R2}
        ; Branch back to the calling instruction
        MOV     PC, LR
```

If a method calls another method, then the link register will need to be stacked too!

10.2 Passing parameters

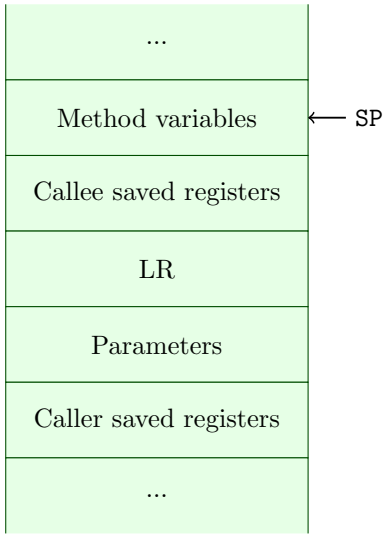
Passing parameters to a method is easy; just add them to the stack! To do this, just load the variable into a register, then add it to the stack:

```
        LDR     R0, myvar
        STR     R0, [SP,#-4]!
```

When the method wants to access a parameter, it can just load the value in memory addressed by the stack pointer, adding an offset of `#4` if the second parameter needs to be accessed, `#8` for the third and so on. If the fourth parameter was to be loaded, then the instruction would be `LDR R0 [SP, #12]`.

10.3 Stack frames

A stack frame is a set of stacked values that are related to a particular method call. They usually have a format that is specific to the method being called. For example:



11 Switch statements

Obviously, a switch statement could be compiled down to a series of `if else` statements and then converted into ARM assembly. However, if we had a long list of conditions, and our condition was either near the bottom of the list, or maybe even the `default` choice, then we'd need to test the switch expression against all of the conditions before being able to decide what to do.

This has the potential to be very inefficient, so thankfully, there is another way to do it. We can use the value of the switch variable to branch directly to the piece of code that we want to execute for that case.

This can be achieved using a table of values where the offset in the table corresponds to the next address to load into the PC (i.e. the branch location). For example:

```
; Define a table of variables with
; the locations to branch to
swtable DEFW    case0
        DEFW    case1
        DEFW    default
        DEFW    case2
; Load the base address of the table
; into R1
        ADR     R1, swtable
; Load the value of R1 into the PC.
; Offset the value in R1 by (R0*4)
        LDR     PC, [R1, R0, LSL, #2]

case0   ; Do stuff
        B end

case1   ; Do stuff
        B end

```

The `LSL` command is used to shift the bits in `R1` left. The number of places shifted is equal to the value of `R0`. This has the effect of adding `(4*R0)` onto `R1`.

```
case2   ; Do stuff
        B end

default ; Do stuff
        B end
```

It's important to catch cases where the expression isn't in the table too, for example, if (following the previous example) the values we wanted to implement specific logic for were `(0, 1, 3)`, then we would need to catch cases where the value was less than `0` and greater than `3`.

```
        CMP     R0, #0
        BLT     default
        CMP     R0, #3
        BGT     default
```

The above would go before the table lookup would occur.

12 Types of values

12.1 Signed and unsigned integers

Lets assume that integers only have 8 bits, and so can hold 2^8 (256) different values. If we use a signed integer, the most significant bit would indicate whether the number was negative or not. This means that we loose a bit's worth of data and therefore can only represent the numbers $0 - 128$, though we do gain the ability to distinguish negative numbers from positive numbers.

As you should already know, using a signed integer is also known as 2's complement.

Note that the numbers $0-127$ are represented as the same as both signed and unsigned integers, however, when a number reaches the value 128, then as a signed integer it would mean -1 and as an unsigned integer it would mean 128.

It is important to note, that ARM has different types of compare instructions for signed and unsigned integers. We can use this to our advantage in the switch example outlined in the previous section (see page 25).

If we treat the register `R0` as a signed integer, we will have to check whether it is below `0` and whether it's above `4`. However, if we use unsigned comparisons, we only need to check whether the register is above `4` since any negative value will be represented as a positive value much larger than `4`!

```
        CMP     R0, #4
        BHI     default
```