ARM11 – Assembler and Emulator

120.3, Programming III, Assessed C Laboratory

24th May – 23rd June, 2017

Aims

- To ensure you can program proficiently in C.
- To learn about and use a wide range of C language features, including bit manipulation operators.
- To design and implement two sizeable and inter-related C programs.
- To gain experience of development in a group.
- To undertake independent learning.
- To reinforce your understanding of instruction set architectures and low-level aspects of machine operation.
- To learn how to read and write binary files.
- To design, document and implement your own extension to a given specification.

Introduction

In this exercise you will be working with a family of *Reduced Instruction Set Computer* (RISC) based computer processors. You already have experience of writing x86 assembler programs; here you will be working with a subset of the ARM[1] instruction set architecture and with a Raspberry Pi[2]. A Raspberry Pi contains a Broadcom BCM2835 system on a chip (SoC), which has an ARM1176JZF-S 700 MHz processor. Over the course of four weeks, you will have four things to do:

Part I
Implement an ARM emulator, i.e. a program that simulates the execution of an ARM *binary file* on a Raspberry Pi.

Part II
Implement an ARM assembler, i.e. a program that translates an ARM assembly *source file* into a binary file that can subsequently be executed by the emulator.

Part III
Write an ARM assembly program that flashes an LED on a provided Raspberry Pi.

Part IV
Design, implement, and document an extension of your own choice. You might add new instructions to the ARM assembler or emulator to take advantage of features found on the Raspberry Pi, or construct a debugger or high-level compiler. The documentation will take the form of a LaTeX report.

[1] See http://infocenter.arm.com/help/index.jsp.
[2] See http://www.raspberrypi.org/

The programs that you will write in Parts I and II can be used together to execute ARM assembly programs. It is recommended that you write the emulator first, as it will make it easier for you to test the output from your assembler. You will work in groups of three or four.

All members of your group should contribute as equally as possible. You might want to divide the work up among the group members or you might each wish to work independently on a solution and then merge the solutions to form a polished deliverable. The primary objective is to ensure that you can *all* program proficiently in C by the end of the course.

This is also an exercise in learning to develop software in a team. As part of that you will reflect upon your contribution to your group, and provide feedback to your colleagues on how you think they're doing. There will be two online WebPA assessments that will ask you to consider how your group is doing in terms of response, performance, communication and management. We will make available your feedback after each assessment so that you may improve as the project progresses. You will also reflect upon your own feedback in a final report at the end of the project.

You will be provided with a test suite to help test your emulator and assembler. In the test suite, assembler files (e.g. `add01.s`, `bne02.s` and `ldr08.s`), some ARM binaries (`add01`, `bne02` and `ldr08`) and some result files (`add01.out`, `bne02.out` and `ldr08.out`) are provided, so you can thoroughly test your programs to see if they produce the same outputs. You may also want to create your own test cases. Bear in mind that you should also design the code so that it can be written and tested incrementally.

Part I – The Emulator

The emulator main program, `emulate.c`, should begin by reading in ARM binary *object code* (i.e. compiled assembly code) from a *binary file* whose filename is specified as the sole argument on the command line, e.g. `add01`:

```
% ./emulate add01.bin
```

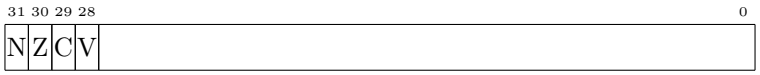
The object code consists of a number of 32-bit 'words', each of which represents either a machine instruction or data.[3] These words should be loaded into a C data structure (an array?) representing the memory of the emulated Raspberry Pi.

Your emulator may assume that ARM machine memory has a capacity of 64KB (i.e. $2^{16} = 65536$ bytes) which means that all addresses can be accommodated in 16 bits. The ARM memory is byte-addressable. However, all instructions are 32 bits (4 bytes) long and aligned on a 4-byte boundary, i.e. all instruction addresses are multiples of 4. All memory locations and registers should be initialised to 0, reflecting the state of an ARM machine when it is first switched on. Once the binary file has been loaded, the emulator then runs the program contained within it by simulating the execution of each instruction in turn, starting with the instruction at location 0, which is the initial value of the *program counter* (PC) register.

You may assume that an ARM system has 17, 32-bit registers. Registers 0 - 12 are general purpose, registers 13 and 14 (called SP and LR respectively) can be ignored for this exercise. The PC is register 15, and the CPSR (see below) is register 16. The execution of a branch instruction may change the PC to a specified target address. Instructions may modify values in both the ARM memory and general-purpose registers. In addition to the memory array you will also need a data structure (another array?) to store the values of the registers. A good idea would be to define a C `struct` to capture the state of an ARM machine.

The ARM instruction set does not have a `halt` instruction, i.e. the processor will run forever.[4] However to enable easier testing of your emulator we will deviate slightly from ARM and interpret an all-0 instruction (which, as you will see when making your assembler, corresponds to `andeq r0, r0, r0`) as the signal that your emulator should terminate. Upon terminating, you should print to standard output the value of each register and the contents of any non-zero memory location.

The CPSR register (register 16) is used to configure the operating mode of the ARM processor, and to check conditions for conditional ARM instructions. For this exercise you should initialise the CPSR to zero. You only need to be concerned with its top four bits, which carry the status flags from computations run on the arithmetic logic unit (ALU).[5]



The flag bits represent:

N: the last result was negative

Z: the last result was zero

C: the last result caused a bit to be carried out

V: the last result overflowed

Precisely when and how to read and update the CPSR will be covered in the section on interpreting instructions, below.

[3] Of course, at the machine level there is no difference between programs and data!

[4] Please remember this later when you create programs to run on your Raspberry Pi!

[5] Note that throughout this document the words are presented in *big-endian* order, so the most significant bit is numbered *31* and the least significant bit is numbered *0* (right to left)

Three Stage Pipeline

An ARM processor executes an instruction in three phases: first the 4-byte instruction must be fetched from memory; second it must be decoded (working out what kind of instruction it is, which parts of the instruction refer to registers, memory or constants and resolving these), and finally the decoded instruction is executed, making the effect of the instruction visible to the registers or memory.

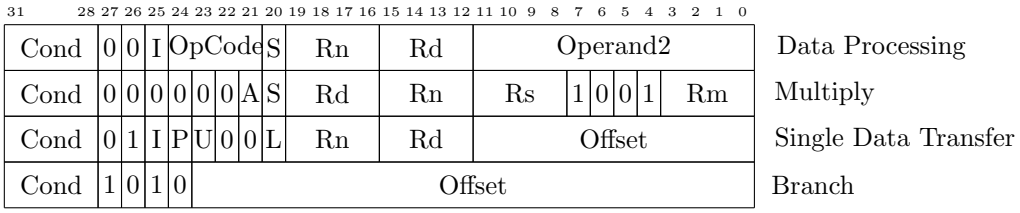
In order that all the processing and memory parts of the ARM architecture can operate continuously, a pipeline of instructions to decode and then execute is built up. So, simultaneously with an instruction being fetched from memory, the previously fetched instruction is decoded, and its ancestor instruction is executed.

This use of a pipeline has an interesting side-effect. As an instruction is being executed at the top of the pipeline, the instruction being fetched into the bottom of the pipeline is two instructions further on in memory. This has the consequence that the PC is 8 bytes greater than the address of the instruction being executed. Your emulator will need to simulate the effect of the pipeline, as branch instructions in ARM binaries take into account this 8-byte offset.

Figure 1 presents a step-by-step example of the execution of a test program with branching it it, showing the state of the pipeline after each execute, decode and fetch cycle.

ARM instruction set

Your emulator will need to support a subset of the ARM instruction set.[6] The full ARM instruction set consists of 15 instruction types; these instructions allow the ARM processor to process data, transfer data from/to registers, load/store to/from memory and to execute software interrupts. You will need to support four of these: the *Data Processing*, *Multiply*, *Single Data Transfer* and *Branch* instructions.



All ARM instructions have a condition code (*Cond*) which identifies under which conditions the operation should be executed. These conditions refer to the state of the top four bits of the CPSR register. The following table illustrates the minimal set of conditions that the emulator should support.

Code	*Suffix*	*CPSR flags*	*Interpretation*
0000	`eq`	Z set	equal
0001	`ne`	Z clear	not equal
1010	`ge`	N equals V	greater or equal
1011	`lt`	N not equal to V	less than
1100	`gt`	Z clear AND (N equals V)	greater than
1101	`le`	Z set OR (N not equal to V)	less than or equal
1110	`al`	(ignored)	always

Just before executing an instruction the condition is checked. If it succeeds (or is `al`) then the instruction is executed as normal. If it doesn't, the instruction is ignored.

[6] For the full ARM instruction set, and further details on all the instructions presented here, please see Chapter 4 of the ARM7TDMI Data Sheet (available from the given files of this exercise).

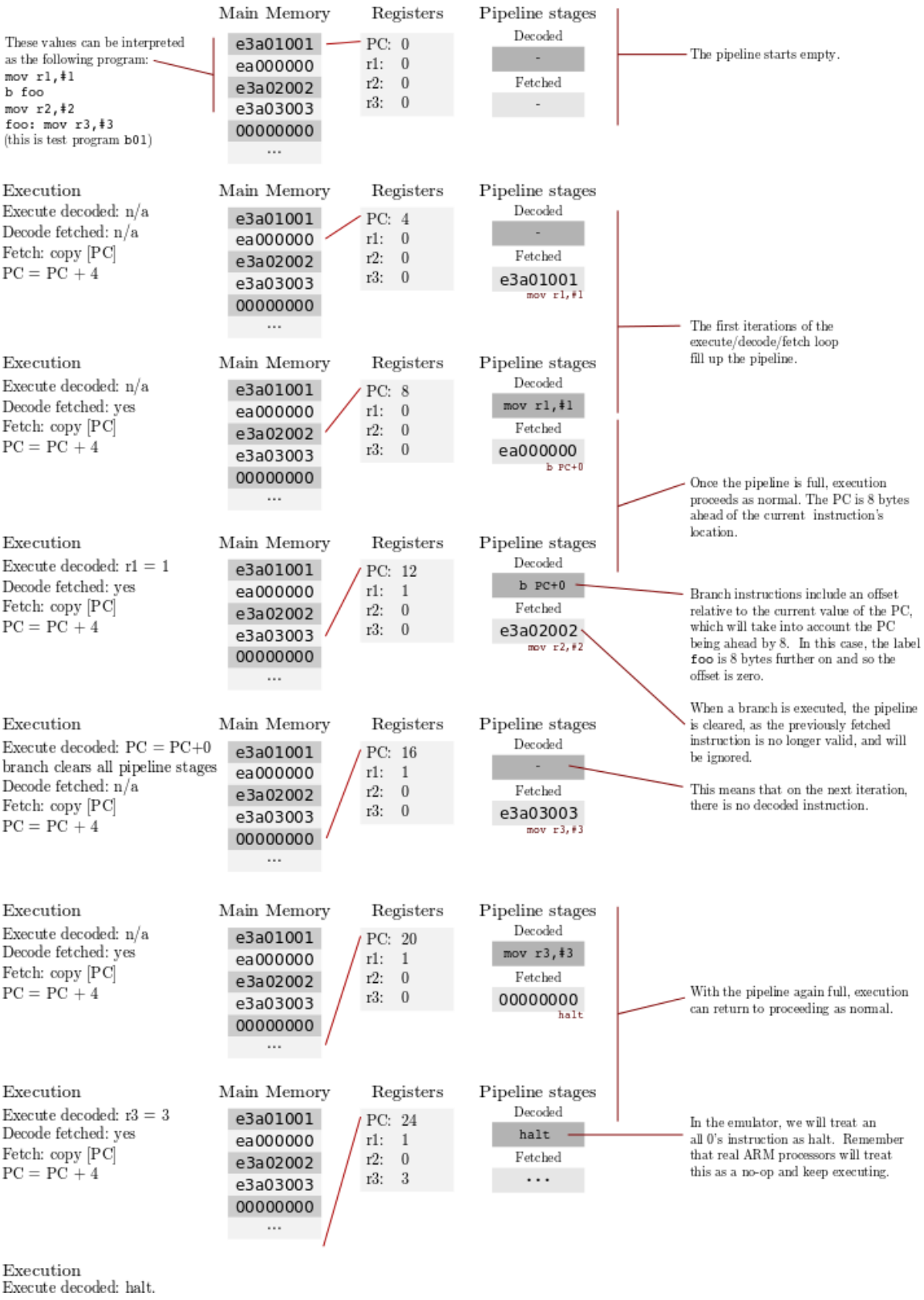
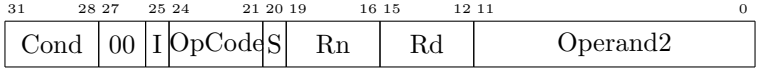


Figure 1: Sample execution of a branching program, showing simulated pipeline and register state at each step.

Data Processing Instructions

A Data Processing Instruction takes the following form:



where the different parts mean:

- Cond: *Condition field*. See above.
- I: *Immediate Operand*. If the bit is set, it means *Operand2* is an immediate constant, otherwise it is a shifted register.
- OpCode: *Operation Code*. The minimal set of opcodes that your ARM emulator and assembler should support are:

Opcode	*Mnemonic*	*Interpretation*
0000	`and`	Rn AND operand2
0001	`eor`	Rn EOR operand2
0010	`sub`	Rn - operand2
0011	`rsb`	operand2 - Rn
0100	`add`	Rn + operand2
1000	`tst`	as `and`, but result not written
1001	`teq`	as `eor`, but result is not written
1010	`cmp`	as `sub`, but result is not written
1100	`orr`	Rn OR operand2
1101	`mov`	operand2 (Rn is ignored)

- S: *Set condition codes*. If the bit is set then the CPSR flags should be updated during execution of this instruction.
- Rn: *First operand register*. The first operand is always the content of register Rn.
- Rd: *Destination register*. The operations write their results to register Rd.
- Operand2: The second operand (12 bits available) can be a rotated 8 bit immediate constant or a shifted register (Rm), depending on the state of the I flag. The details of how these 12 bits are interpreted is given below.

The data processing instruction is only executed if the Cond field is satisfied by the CPSR register. Each instruction performs an arithmetic or logical operation using one or two operands and may write the results to the register specified by Rd. The first operand is always register Rn and the second operand is sent to the ALU via the *barrel shifter*. ARM does not have shift instructions; instead it provides a barrel shifter to perform shifts, but always as part of an other instruction. The operations that the barrel shifter supports are described below with the register-based operands.

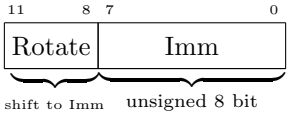
If the S bit is 0, the CPSR register is unaffected. If the S bit is set then the CPSR flags should be set as follows:

- The V bit will be unaffected.
- The C bit in logical operations (`and`, `eor`, `orr`, `teq`, `tst` and `mov`) will be set to the carry out from any shift operation (i.e.the result from the barrel shifter). In arithmetic operations (`add`, `sub`, `rsb` and `cmp`) the C bit will be set to the carry out of the bit 31 of the ALU. For addition, C is set to 1 if the addition produced a carry (unsigned overflow), it is set to 0 otherwise. For subtraction (including comparison), the bit C is set to 0 if the subtraction produced a borrow, otherwise is set to 1.
- The Z bit will be set only if the result is all zeros.
- The N bit will be set to the logical value of bit 31 of the result.

For this exercise, you may assume that the PC will not feature as any register in a Data Processing instruction.

Operand2 is an immediate value

If Operand2 is an immediate value (the I flag is = 1), then it has the following form:

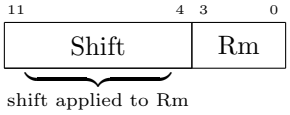


The unsigned 8 bit immediate value (Imm) is zero-extended to 32 bits, and then rotated to the right an even number of positions as specified in the rotate field (i.e. rotate right by 0,2,4...30). Any rotation amount is twice the value in the 4 bit rotate field.

Notice that there is no single data processing instruction which will load a 32-bit immediate value into a register; this has to be done by performing a load instruction from memory.

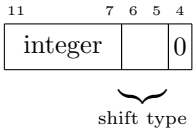
Operand2 is a register

If Operand2 is a register (the I flag = 0), then it has the following form:

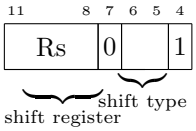


Shift operations are applied to the value held in register Rm. The shift value is specified by a 5-bit unsigned integer or by another register (Rs) depending on the value of bit 4. Note that the case when bit 4 is a 1 is optional.

Shift by a constant amount:



Shift specified by a register (optional):



The bottom byte of Rs (i.e. the last byte of the Rs register's content) specifies the amount to be shifted. So, for example, if the bottom byte of Rs is zero, the unchanged contents of Rm will be used as the second operand. You may assume that Rs can be any general purpose register except the PC.

The different types of shift operation that can be applied to the constant value or to the register Rs are:

Code	*Shift Type*	*Interpretation*
00	logical left (lsl)	Moves each bit to the left by the specified amount.
01	logical right (lsr)	Moves each bit to the right by the specified amount.
10	arithmetic right (asr)	As lsr but preserves the sign bit; used for 2's complement operations.
11	rotate right (ror)	Rotate cyclically with bit 0 shifting into bit 31.

The value of operand2 is computed with the contents of Rm. The contents of Rm are not modified as a side effect of the instruction. If a logical shift left operation is applied, then operand2's least significant bits are filled with the specified number of zeros, while the high bits of Rm are discarded. However, the least significant discarded bit of Rm becomes the shifter carry output which is latched into the C bit of the CPSR if the instruction has to set CPSR flags. The logical shift right operation is similar, but the contents of Rm are moved to the least significant positions of operand2's value. The arithmetic shift right operation is similar to logical shift right operation, however the high bits of operand2's value are filled with bit 31 of Rm instead of zeros. The rotate right operation shifts the bits from left to right in a cyclic fashion. Figure 2 illustrates the results in operand2's value after applying different shift operations.

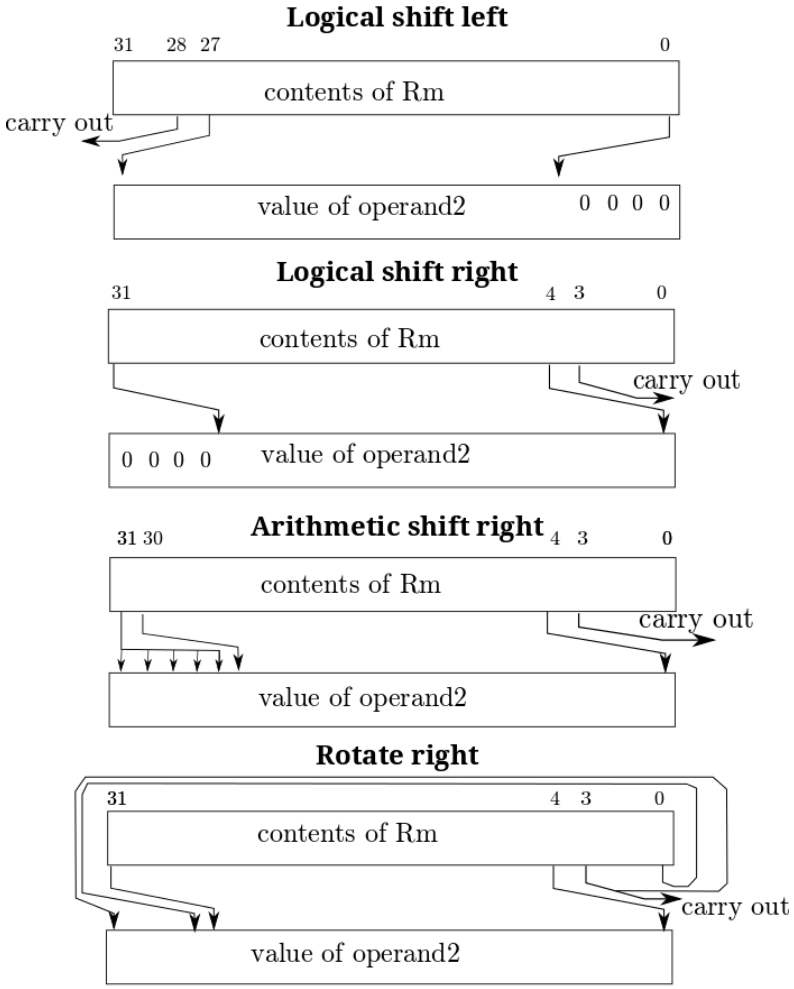
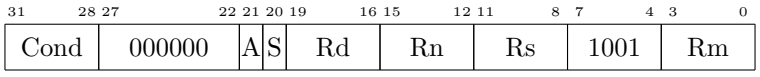


Figure 2: `lsl`,#4, `lsr`,#4, `asr`,#4 and `ror`,#4.

Multiply Instructions

A Multiply Instruction takes the following form:



where the different parts mean:

- Cond: *Condition field*, as for Data Processing Instructions.
- A: *Accumulate*. If the bit is set, the instruction performs a multiply and accumulate; otherwise it performs multiply only.
- S: *Set condition codes*. If the bit is set then the CPSR flags are be updated during execution of this instruction.
- Rd: *Destination register*. The operations write their results into register Rd.
- Rn, Rs and Rm: *Operand registers*.

These instructions can only be executed if any condition specified by the Condition field is satisfied. The multiply instruction performs Rd:=Rm×Rs. Rn is ignored. The multiply-accumulate instruction performs Rd:=Rm×Rs+Rn.[7]

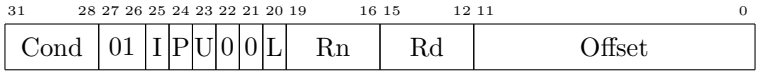
Both forms of this instruction can work with signed (2's complement) or unsigned integers, as the result will be the same. The difference between a signed or unsigned multiply relies on the upper 32 bits of the result. However the 32 lower bits (which the result will be truncated to) remains the same in both cases. For example, consider the multiplication of two operands: operand A (0xFFFFFFFD) and operand B (0x0000000F). The 32-bit result is (0xFFFFFFD3). If the operands are interpreted as unsigned, operand A has the value 4294967293 and operand B has the value 15. The result is then 64424509395, which is represented as 0xEFFFFFFD3. If we truncate to only the lower 32 bits, the result is 0xFFFFFFD3. If the operands are interpreted as signed, operand A has the value -3 and operand B the value 15, then the result is -45, which it is represented as 0xFFFFFFD3.

You may assume that the PC will not be used as an operand or destination register. In addition register Rd will not be the same as the operand register Rm. If the S bit has been set, then the N and Z flags of the CPSR should be updated. N is set to bit 31 of the result and Z is set if and only if the result is zero.

Single Data Transfer instructions

Single data transfer instructions are used to load/store data from/to memory. ARM does not support memory to memory data processing operations, so data from memory must be loaded into registers before use. The memory address to access is computed by adding or subtracting an offset from the base register.

A Single Data Transfer Instruction takes the form:



where the different parts mean:

- Cond: *Condition field.* As for Data Processing Instructions.
- I: *Immediate offset.* If the flag is set then Offset is interpreted as a shifted register. If the flag is not set, the offset is interpreted as an unsigned 12 bit immediate offset[8]. To compute the value of Offset a shifted register, see the description when Operand2 is a register in the description of Data Processing Instructions.
- P: *Pre/Post indexing bit.* If the flag is set (pre-indexing) the offset is added/subtracted to the base register before transferring the data. If the flag is not set (post-indexing), the offset is added/subtracted to the base register after transferring. For the purposes of this exercise, pre-indexing will not change the value of the base register, however post-indexing will change the contents of the base register by the offset. Figure 3 has examples of the different types of addressing.
- U: *Up bit.* If the bit is set, the offset is added to the base register. Otherwise the offset is subtracted from the base register.
- L: *Load/Store bit.* If the bit is set, the word is loaded from memory, otherwise the word is stored into memory.
- Rn: *Base register.*
- Rd: *Source/Destination register.*
- Offset: The offset may be either an unsigned 12 bit immediate offset or a register (possibly shifted), see the description of the I component, above.

[7] Notice that this instruction can save an ADD instruction in some cases

[8] Note that the opposite is true for Data Processing instructions.

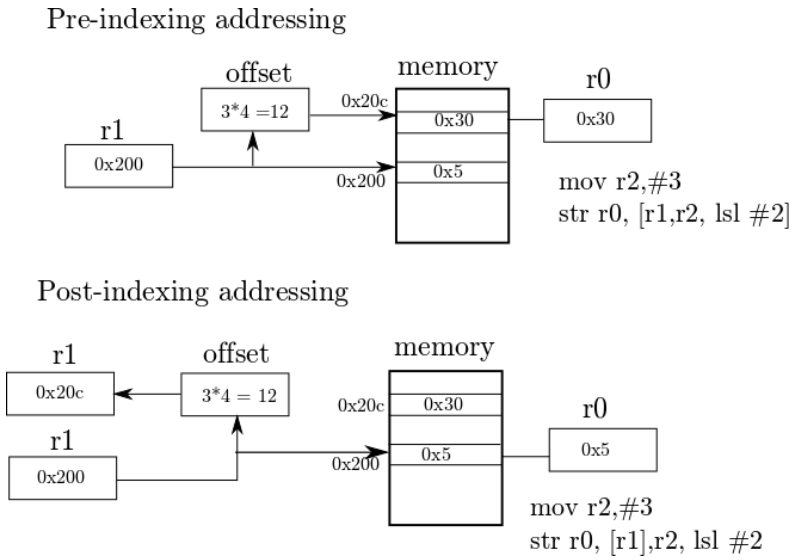
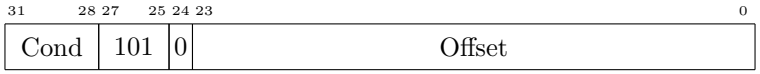


Figure 3: Difference between pre-indexing and post-indexing addressing.

These instructions are only executed if the Cond field is satisfied by the CPSR register. You may assume a load (`ldr`) loads a word and a store (`str`) stores a word. You may assume the PC cannot be specified as the register offset (Rm), or as the source register (Rd). A post-indexing `ldr` or `str` in which Rm is the same register as Rn is not allowed. Remember, if the PC is used as the base register (Rn), then your emulator must ensure it contains the instruction's address plus 8 bytes due to the effects of the three stage pipeline.

Branch Instructions

A Branch Instruction takes the form:



where the different parts mean:

- Cond: *Condition field.* As for Data Processing Instructions.
- Offset: a signed 24 bit offset.

Branch instructions are only executed if the Condition field is satisfied by the CPSR register. Branch instructions contain a signed (2's complement) 24 bit offset. This offset is shifted left 2 bits, sign extended to 32 bits and then added to the PC. Therefore, the branch instruction can specify a branch of +/- 32 Mbytes. The offset will take into account the effect of the pipeline (i.e. PC is 8 bytes ahead of the instruction that is being executed).

Object code file format

ARM binary files store each word in a *little-endian* byte order, an illustration of which may be seen in the example object code program shown in Part II. You should make sure that your memory layout also follows this convention, otherwise loading from and storing to memory will not work correctly.

What to do

Your program should be called `emulate` and should be broken down into a set of functions that collectively load and execute the object code in a specified binary file. The emulator should check for and report errors that occurred during program execution; examples include an attempt to execute an undefined with address greater than 65536. Writing the emulator involves the following key tasks:

- Building a binary file loader.
- Writing the emulator loop, comprising:
  - Simulation of the ARM pipeline, with an execute, decode, fetch cycle.
  - Simulated execution of the four types of instruction (data processing, multiply, single data transfer, branch) given above. Note that there are no opcodes for the different instruction types, so you will have to find a pattern to distinguish between instructions.
  - The emulator should terminate when it executes an all-0 instruction.
  - Upon termination, output the state of the registers.

To help you test your emulator we have provided a test suite, see the section on the test platform at the end of this document.

Part II – The Assembler

The assembler's main program `assemble.c` should read in source code from an ARM source file whose filename is given as the *first* command line argument, and output ARM-binary code to a file whose filename is given as the *second* command line argument. For example, to assemble the ARM source file `add01.s` to the ARM binary file `add01`:

```
% ./assemble add01.s add01.bin
```

Two-pass assembly

There are several ways of performing the assembly. Arguably the simplest way is to perform *two* passes over the source code. The first pass creates a *symbol table* which is used for associating *labels* (strings) with memory addresses (integers). In the second pass the assembler reads in the opcode *mnemonic* and operand field(s) for each instruction and generates the corresponding binary encoding of that instruction. As part of this process it replaces label references in operand fields with their corresponding addresses, as defined in the symbol table computed during the first pass.[9]

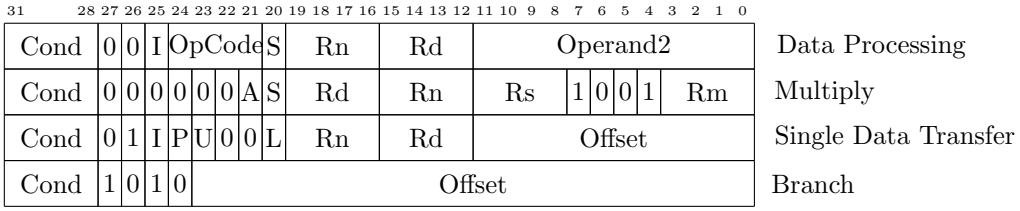
An alternative is to perform the assembly in one pass. The problem now is that you cannot immediately resolve forward references, where one instruction refers to a label that has yet to be defined. To get this you need to maintain a `list` (`set...`) of addresses (within instructions) that represent forward references to unresolved label, L, say. Once the address for L is known, this is written to each location in the list and L's address is added to a *symbol table* in case there is a backward reference to L later on in the program.

Assembler file format

Each (non-empty) line of an assembler file contains either an assembly *instruction* or an assembler *directive*, optionally preceded by a *label*. Labels are strings that begin with an alphabetical character (`a`–`z` or `A`–`Z`) and end with a `:`, as in "`label:`". The value of the label is the address of the machine word corresponding to the position of the label. You may assume no line is longer than 511 characters in length.

Each instruction is mapped to (exactly) one 32-bit word during the assembly process (second pass). For the purposes of this exercise, an instruction comprises an operation mnemonic (e.g. `add`, `ldr`, ...), and one, two, three or four operand fields, depending on the instruction type. Registers are referred to by the strings r0, r1, ..., r16.

Below is repeated the instruction type summary from the emulator, followed by a summary of all the operations your assembler should support. The following sections will discuss each type in more detail.



[9] Note that the symbol table encodes a mapping from strings to integers (c.f. `[(String,Int)]` or `String` → `Int` in Haskell).

Mnemonic	*Instruction*	*Action in Emulator*	*Type*
`add`	Add	Rd := Rn + Op2	Data Processing
`sub`	Subtract	Rd := Rn - Op2	Data Processing
`rsb`	Reverse Subtract	Rd := Op2 - Rn	Data Processing
`and`	AND	Rd := Rn AND Op2	Data Processing
`eor`	Exclusive OR	Rd := Rn XOR Op2	Data Processing
`orr`	OR	Rd := Rn OR Op2	Data Processing
`mov`	Move Assignment	Rd := Op2	Data Processing
`tst`	Test bits	CPSR flags := Rn AND Op2	Data Processing
`teq`	Test bitwise equality	CPSR flags := Rn XOR Op2	Data Processing
`cmp`	Compare	CPSR flags := Rn - Op2	Data Processing
`mul`	Multiply	Rd := Rm × Rs	Multiply
`mla`	Multiply accumulate	Rd := (Rm × Rs) + Rn	Multiply
`ldr`	Load register	Rd := (address)	Single Data Transfer
`str`	Store register	(address) := Rd	Single Data Transfer
`beq`	Branch if equal	PC := address	Branch
`bne`	Branch is not equal	PC := address	Branch
`bge`	Branch if greater than or equal	PC := address	Branch
`blt`	Branch if less than	PC := address	Branch
`bgt`	Branch if greater than	PC := address	Branch
`ble`	Branch if less than or equal	PC := address	Branch
`b`	Unconditional branch	PC := address	Branch
`lsl`	Left shift	Rd := shifted Rd	Special
`andeq`	`andeq r0,r0,r0`	Halt	Special

Data Processing Instructions

The data processing instructions can be broken down into three main types:

- Instructions that compute results: `and`, `eor`, `sub`, `rsb`, `add`, `orr`.

  Their syntax is `<opcode> Rd, Rn, <Operand2>`

- Single operand assignment: `mov`.

  Its syntax is: `mov Rd, <Operand2>`

- Instructions that do not compute results, but do set the CPSR flags: `tst`, `teq`, `cmp`.

  Their syntax is: `<opcode> Rn, <Operand2>`

The different parts of the syntax can be interpreted as follows:

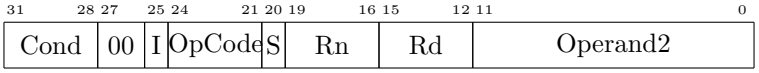
- Rd, Rn and Rm represent registers. These are usually written (e.g.) `r0`, `r1`.
- `Operand2` represents an operand. It can take several forms, an expression, `<#expression>`, or a shifted register, Rm{`,<shift>`}. Note that for this exercise supporting the shifted register case is optional.
- `<#expression>` is a numeric constant. The assembler will attempt to generate an 8 bit immediate value, see the description of when a Data Processing Instruction's Operand2 is an immediate value in the emulator for how to represent this in the binary. If the numeric constant cannot be represented, your assembler should give an error.[10] The numeric constant may be specified in decimal or hexadecimal. In the case of the latter, values will be preceded by the string "`0x`".
- `<shift>` describes a shift. It can take the form `<shiftname> <register>` or `<shiftname> <#expression>`. See the description of when a Data Processing Instruction's Operand2 is a register in the emulator for the binary formats of different shift operations. Supporting this case is optional.

[10] Note that any numeric constant is generated using the 8-bit immediate value which is zero-extended to 32 bits, and then rotated right by the amount needed.

- `<shiftname>` can be one of `asr` (algebraic shift right), `lsl` (logical shift left)[11], `lsr` (logical shift right) or `ror` (rotate right).

The binary format of a Data Processing Instruction that your assembler should target is described in the emulator section. The key details are:

Opcode	*Mnemonic*
0000	`and`
0001	`eor`
0010	`sub`
0011	`rsb`
0100	`add`
1100	`orr`
1101	`mov`
1000	`tst`
1001	`teq`
1010	`cmp`



When assembling the testing instructions (`tst`, `teq` and `cmp`) you should set the S bit to 1. For the other instructions, you should set the S bit to 0.[12] All Data Processing Instructions should populate the Cond field with `1110` (which corresponds to the `al`ways condition).[13]

Examples

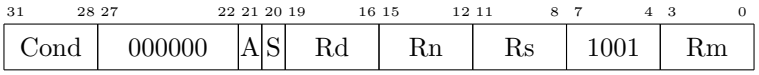
- `add r2, r4, r3` will put into register 2 the sum of registers 4 and 3.
- `mov r1, #56` will load the numeric constant 56 into register r1.
- (Optional) `sub r1, r2, r3, lsl r4` will apply a logical left shift to register 3 by the amount specified by the bottom byte of register 4 to compute a new value. This value will be subtracted from register 2, and the final result stored in register 1.
- `mov r1, #0x0F` will move 15 (represented in hexadecimal as F) into register 1.

Multiply Instructions

There are two multiply instructions:

- Multiply, with syntax: `mul Rd, Rm, Rs`.
- Multiply with accumulate, with syntax: `mla Rd, Rm, Rs, Rn`.

Recall the binary format for a Multiply Instruction:



The `mul` instruction will set the A bit to 0, `mla` will set it to 1. As with the Data Processing Instructions, you should assemble with the S bit to 0 and the Cond field should be populated with `1110`.

Examples

- `mla r1, r2, r3, r4` will store in register 1 the sum of register 4 with register 2 × register 3.
- `mul r1, r2, r3` will store in register 1 the result of computing register 2 × register 3.

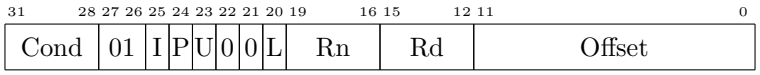
[11] Notice that `asl` (algebraic shift left) is equivalent to `lsl`.

[12] In a full ARM assembler, many instructions can be given an `s` suffix to indicate the S bit should be set. The test instructions always have this bit set regardless. You may wish to implement this later as an extension.

[13] Again, in a full ARM assembler, most instructions can feature a condition suffix (which defaults to `al` if not present).

Single Data Transfer Instructions

There are two single data transfer instructions, `ldr` and `str`, with syntax `<ldr/str> Rd, <address>`. Recall the binary format for a single data instruction:



`ldr` loads from memory into a register which will require the L bit to be set. `str` stores a register into memory, which will require the L bit to be clear.

`<address>` can take one of several forms:

- A numeric constant of the form: `<=expression>` (`ldr` only).

  In the general case, the assembler should put the value of `<expression>` in four bytes at the end of the assembled program, and use the address of this value, with the PC as the base register and a calculated offset, in the assembled `ldr` instruction. It will be always a pre-indexed address.

  In this way, `ldr` is used as a `mov` instruction for constants that will not fit in a traditional `mov` instruction, and a way to direct the assembler to reserve memory with a constant in it.

  However sometimes the value of `expression` will fit in the argument of a `mov`, and thus the assembler could compile the instruction as a `mov` instead of an `ldr`. For this exercise, if the argument is less than or equal to `0xFF` you should use a `mov` instruction instead.

- A pre-indexed address specification, one of:

  - `[Rn]`, using base register Rn, with an offset of zero.
  - `[Rn,<#expression>]`, using base register Rn, offset of `<#expression>` bytes.
  - (Optional) `[Rn,{+/-}Rm{,<shift>}]`, using base register Rn, offset by +/- the contents of index register Rm which has been shifted by `<shift>`. Supporting this syntax is optional.

- A post-indexing addressing specification, one of:

  - `[Rn],<#expression>`, offset base register Rn by `<#expression>` bytes.
  - (Optional) `[Rn],{+/-}Rm{,<shift>}`, offset base register Rn by +/- the contents of index register Rm which has been shifted by `<shift>`. Supporting this syntax is optional.

Examples

- `ldr r0, =0x42`, the assembler should treat this as `mov r0, #0x42`, as the constant 0x42 will fit inside a mov instruction.

- `ldr r0, =0x555555`; the assembler should place the constant 0x555555 in four bytes at the end of the assembly file, and then compute the offset between the current location and the newly created one (taking into account the off-by-8 bytes effect of the pipeline), and generate `ldr r0, [PC, offset]`.

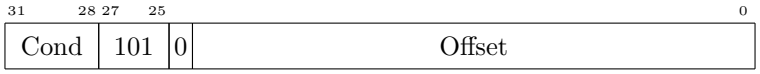
- (Optional) `str r0, [r3, r2]`; this instruction will direct the emulator to store register 0 at the memory address obtained by adding the contents of registers 3 and 2.

- (Optional) `ldr r0,[r1,r2,lsl #2]`; this instruction will load into register 0 the contents from a memory location, which is computed by adding the contents of register 1 and register 2 multiplied by 4.

Branch Instructions

All branch instructions are of the form `b<cond> <expression>`. However, if it makes things simpler, you may choose to interpret the branch conditions separately, i.e. treat each of `beq`, `bne`, `bge`, `blt`, `bgt`, `ble` and `b` as separate instructions.

Recall the structure of the branch instruction from the emulator:



The Cond field should use the values corresponding to the type of branch, repeated below.

Code	*Suffix*	*Interpretation*
0000	`eq`	equal
0001	`ne`	not equal
1010	`ge`	greater or equal
1011	`lt`	less than
1100	`gt`	greater than
1101	`le`	less than or equal
1110	`al` (or no suffix)	always

The `<expression>` is the target address, which may be a label. The assembler should compute the offset between the current address and the label, taking into account the off-by-8 bytes effect that will occur due to the ARM pipeline. This signed offset should be 26 bits in length, before being shifted right two bits and having the lower 24 bits stored in the Offset field.

Example

The following program uses a label and a branch to create a loop. The value of register 2 loops from 23 down to 0, decrementing by one on each iteration.

```
mov r2,#23
wait:
sub r2,r2,#1
cmp r2,#0
bne wait
```

Special Instructions

There are two instructions you should support in order to enable testing, and to make implementing Part III of this exercise easier. These are `andeq` and `lsl`.

`andeq`

Your assembler should compile the instruction `andeq r0, r0, r0` to the binary value 0x00000000. You may implement this either as a special case, or as part of the Data Processing Instructions (it is really an `and` instruction that sets the Cond field to the `eq` condition).

Your emulator should be interpreting the binary value 0x000000 as a signal to halt, this is the ARM instruction that will produce all 0's.

`lsl`

Your assembler should compile the instruction `lsl Rn, <#expression>` as though it were `mov Rn, Rn, lsl <#expression>`. The binary form of this can be found in the emulator section, under Data Processing Instructions, when Operand2 is a register, shifted by a constant amount.

By supporting this syntax, you will not need to support the full syntax for shift-modified expressions in Data Processing Instructions, but do have an (ARM assembly compatible) way of shifting registers for the emulator to use.

A Factorial Program

An example assembler program for computing the factorial of 5 and storing the result in memory address 0x100 is given below (it is deliberately not intended to be optimal!):

```
mov r0,#1
mov r1,#5
loop:
mul r2,r1,r0
mov r0,r2
sub r1,r1,#1
cmp r1,#0
bne loop
mov r3,#0x100
str r2,[r3]
```

The assembly process should generate the binary shown below. Note that we are viewing the binary file using the command `xxd`, in binary mode (`-b`), in blocks of 4 bytes (`-c4`).

```
% xxd -b -c4 factorial
0000000: 00000001 00000000 10100000 11100011  ....
0000004: 00000101 00010000 10100000 11100011  ....
0000008: 10010001 00000000 00000010 11100000  ....
000000c: 00000010 00000000 10100000 11100001  ....
0000010: 00000001 00010000 01000001 11100010  ..A.
0000014: 00000000 00000000 01010001 11100011  ..Q.
0000018: 11111010 11111111 11111111 00011010  ....
000001c: 00000001 00111100 10100000 11100011  .<..
0000020: 00000000 00100000 10000011 11100101  . ..
```

Notice that the LSB (Least Significant Byte) of each instruction is stored in the lowest memory position, and the MSB (most significant byte) is stored in the highest memory position – i.e. ARM stores instruction and data words using a little-endian encoding.

Also note that `xxd` formats its output into columns - the spaces within the words are purely to aid readability. The first column contains the address of the word in the file, and thus increments in multiples of four.

What to do

As with the emulator, your code should be broken down hierarchically to match the structure of the problem. You may assume that the assembly program being processed is syntactically correct and all instruction mnemonics are written lower case. Constructing the assembler involves the following key tasks, which you should consider dividing up among your group members:

- Constructing a binary file writer.
- Building a symbol table abstract data type (ADT).
- Designing and constructing the assembler, comprising:
  - A tokenizer for breaking a line into its label, opcode and operand field(s) (you might find the `strtok_r` and `strtol` functions helpful here).
  - An instruction assembler comprising, for example:
    - A function for assembling Data Processing instructions.
    - A function for assembling Multiply instructions.
    - A function for assembling Single Data Transfer instructions.
    - A function for assembling Branch instructions.

- An implementation of the one or two-pass assembly process.

Notice that the symbol table can be used to map labels into addresses and also to map opcode/condition mnemonics into the opcodes/conditions themselves. Hint: You might want to explore the idea of using C's *function pointers*, similar to higher-order functions in Haskell, in order to get from an integer opcode to a C function for assembling the corresponding machine operation. This will avoid the need to build a large switch statement or nested conditional in your assembler loop. Additionally you could use the symbol table to map each opcode to its expected operand count.

Part III – General Purpose Input/Output on a Raspberry Pi

In this section, you are going to extend your emulator to simulate the turning on and off of an LED on a Raspberry Pi board, and then get the same program (compiled by your assembler) to run on a physical Raspberry Pi.

The Raspberry Pi is a small computer with an ARMv6 processor (ARM11 family). As mentioned in the introduction section, the SoC is a Broadcom BCM2835[14] and it consists of CPU (ARM1176JZF-s), *Graphical Processing Unit* (GPU), *Digital Signal Processor* (DSP), SDRAM (512 MB shared with GPU), a single USB port direct from the BCM2835 and a 10/100 Ethernet connector. Figure 4 illustrates these components. The ARM11176JFZs is based on version 6 of the ARM architecture, the GPU is a low-power mobile multimedia processor architecture and the DSP is a specialised microprocessor with an architecture optimised for digital signal processing operations.There are two models of Raspberry Pi (model A and model B), which have different specifications. In this exercise you will be working with model B.

The Raspberry Pi's SoC has a total of 54 GPIO pins, however only the first 26 are easily accessible. Each pin can be individually enabled or disabled, and configured as an input or output. GPIO inputs are readable (high = 1, low = 0) and GPIO outputs are both readable and writable. Each GPIO pin can be accessed and controlled by modifying control bits in specific memory locations. The following table illustrates the mapping between the GPIO pins and the physical memory address for the relevant pins[15].

Physical address	byte 4	byte 3	byte 2	byte 1	GPIO pins
`0x2020 0008`	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	20-29
`0x2020 0004`	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	10-19
`0x2020 0000`	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	0-9

Each physical address contains three control bits per pin, leaving the last two bits (31-30) reserved. For example, at physical address `0x2020 0008`, you can access the following pins:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	29																													
			28																												
				27																											
					26																										
						25																									
							24																								
								23																							
									22																						
										21																					
											20																				

For example pin 20 has 3 control bits (at positions 0-2). The control bits set the functionality of the pin, writing bit pattern `000` sets the pin as an input pin, writing pattern `001` makes it an output pin. Therefore to set pin 20 as an output, we can write the bit pattern 001 into bits 0-2[16].

Once a pin has been set up as an output, you then have to write a bit to a different memory address to turn the output pin on or off. There are two 4-byte memory addresses that control the output state of the first 32 GPIO pins. The 32-bits starting at `0x2020 0028` control clearing the pins, and the 32 bits starting at `0x2020 001C` control setting (turning on) the pins.

You will always need to clear a pin, by writing to the *clear* area, before you can set it. If you were to write a 1 into bit 20 at address `0x2020 0028` you would clear pin 20. To subsequently turn pin 20 on, you would write a 1 into bit 20 at address `0x2020 001C`. Note that writing a 0 bit has no effect, so it is safe to write `0x0010 0000` into the addresses to only change the state of pin 20. For this exercise, assume that pins are wired to LEDs, so any pin can turn a LED on when the pin is on and turn a LED off when the pin is *clear*. Note also that the GPIO memory addresses (`0x2020 008`, `0x2020 0004`, `0x2020 0000`) can contain any value, as any pin can have a high or low input applied. Therefore, for this exercise, you should assume that reads to GPIO memory addresses return the value of the address that was read. i.e. a read of memory address `0x2020 0004` will return `0x2020 0004`.

What to do

You will first need to prepare your Raspberry Pi, in particular the SD card, for use. You will be downloading and installing onto the SD card an official linux distribution (Raspbian "wheezy"), and then

[14] On CATe for this course you will find a document called `SoC_ARM_peripherals.pdf` where you can find all the hardware information about the SoC

[15] Note that the physical address "`0x2020 ####`" is equivalent to the bus address "`0x7E## ####`" in the hardware information PDF

[16] However think carefully about how to do this, if you were to write 0x00000000 to the four byte address 0x2020008, you will set *all* the above pins as inputs

<50mA	3.3V	1	2	5V	
GPIO 02	SDA 0/1	3	4	NC	
GPIO 03	SCL 0/1	5	6	GROUND	
GPIO 04		7	8	UART TXD	GPIO 14
	GROUND	9	10	UART RXD	GPIO 15
GPIO 17		11	12		GPIO 18
GPIO 27		13	14	GROUND	
GPIO 22		15	16		GPIO 23
	3.3V	17	18		GPIO 24
GPIO 10	SP10(MOSI)	19	20	GROUND	
GPIO 9	SP10(MISO)	21	22		GPIO 25
GPIO 11	SP10(SCKL)	23	24		SP10(CE0)
	GROUND	25	26		SP10(CE1)

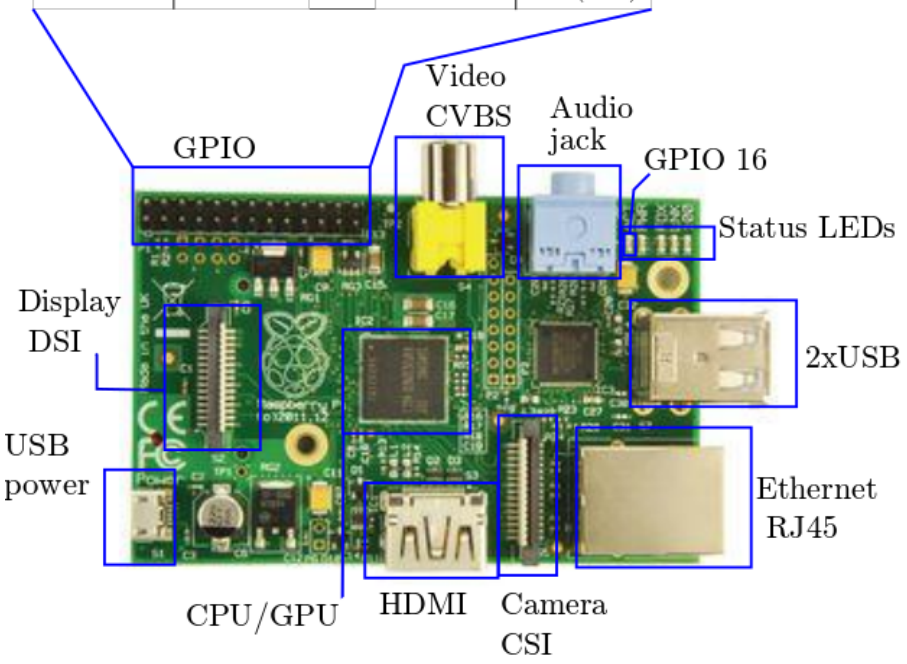


Figure 4: Raspberry Pi (Revision 2) with GPIO pin-outs.

completely replacing the kernel (contained within the file `kernel.img`) with your own program. To get Raspbian installed, please visit http://www.raspberrypi.org/downloads and follow the instructions.

You should write an ARM assembly program, `gpio.s`, which repeatedly switches on and off the LED corresponding to GPIO pin 16 of the Raspberry Pi. This LED can be found on the Raspberry Pi board. You should be able to create the assembler program using the ARM instructions implemented in Part-I and Part-II. Once you create the assembler program, produce a binary file called `kernel.img` using your assembler, viz:

```
% ./assemble gpio.s kernel.img
```

Assuming that the correct binary has been produced, remove the `kernel.img` from your prepared SD card and replace it with the one you have assembled. When the Raspberry Pi is powered on, the LED should start flashing (i.e. the LED should be on for a few seconds and then it should go off for a few seconds in an endless loop). For this assignment you will need to pause the Raspberry Pi for a few seconds, one way to do this is to make a very long loop.

To help you with this section, you should extend your emulator to detect writes to the key memory addresses above, and print out when pins are enabled, disabled, turned on and turned off.

Part IV – A LaTeX-documented extension

Once you have implemented your emulator and assembler, you should design, implement, and document an extension. The implementation may involve changes to your assembler and emulator, but make sure **that the resulting program is backwards-compatible with the ARM specification**. Alternatively, it could be a separate tool (e.g. a debugger, visualiser, high-level compiler, etc.), or an ARM assembly program that makes the Raspberry Pi "do something interesting". At the end of the project you will write a short report and give a presentation detailing your extension.

Please note that this extension should be in C and/or ARM assembly. Even though you are allowed to use additional languages as helpers, these should **not** be the central point of your extension.

Some suggested extensions follow, but you are encouraged to design your own:

- Extend the instruction set with more ARM instructions, for example software interrupts.
- Implement a stack, with associated operations such as push and pop (in ARM, these are known as Block Data Transfer Instructions, `ldm` and `stm`). You will need to define space for the stack itself, as a default component of the binary file. You will need to implement the Block Data Transfer instruction. Although ARM supports ascending stacks (i.e. where the stack structure grows up through memory) and descending stacks, you should implement a descending stack.
- Build a *relocating* loader. The assumption of the current loader is that programs are loaded into memory starting at address 0. Rewrite it so that it can load the code starting at a different location. This requires replacing all absolute memory addresses accordingly.
- Extend your assembler and binary loader to allow programs to be developed in different files with arbitrary cross-references. To do this you will need to provide a mechanism for identifying labels that can be referenced from outside the file (e.g. an *export list*) and, similarly, a way of importing labels from other files. You will then need to extend the binary format to include some form of header data, in order that all external label references can be resolved. You will also need to research the ARM instruction set in order to extend your implementation of the Branch instruction to include Branch With Link (i.e. bit 24 is set to 1).
- Create a 3-bit counter using the Raspberry Pi and three LEDs. The counter should count from zero to seven using the three LEDs.

Working and Submission

There are five deadlines you must meet. Three of the deadlines require that you push your current work back to the master repository and submit to CATe a `cate_token.txt` file containing the SHA1 identifier of the submission's revision. The other two deadlines correspond to the WebPA online group assessments.

Skeleton Repository

Each group will be given a master git repository containing an initial skeleton for this exercise.

Your group repository can be cloned via:

`https://gitlab.doc.ic.ac.uk/lab1617_summer/arm11_<groupno>.git`

where *groupno* is your 1 or 2-digit group number. You can find out your group number by visiting gitlab, or the 120.3 ARM website and clicking on "The Groups" link under Resources:

`https://www.doc.ic.ac.uk/~mvalerae/arm11/`

Note: the entire group will be sharing this repository, so make sure you manage it properly.

As usual, use git commands in order to send your code to gitlab, making sure that your commit messages are **clear and meaningful**.

When you are ready to submit your project, **the Group Leader** should log into the gitlab website https://gitlab.doc.ic.ac.uk, and click through to your group's `arm11_<groupno>` repository. The leader should view the `Commits` tab, find the list of the different versions of your work that you have pushed, select the right version of your code and ensure it is working before submitting to CATe. However, **you will be unable to test your code on LabTS**; you will be provided with a separate **Test Platform** for this purpose (*see next section*). **Only the Group Leader has to submit**; the remaining members will only need to sign on CATe. Please note that the leader **will also need to add your PDF report** to your group's CATe submission before the deadline.

Within the skeleton repository you will find the following files and directories:

`src/`
: You should write the source and header files for your programs in this directory. It contains stub `assemble.c` and `emulate.c` files which you should edit during the completion of Parts I–III. There is also a skeleton `Makefile` which you may need to edit as you work on Parts I–III. To aid with testing, you should ensure that invoking `make` in this directory builds your `assemble` and `emulate` binaries, as well as any additional binaries you create as part of Part III.

`doc/`
: You should write the source for your checkpoint and final report in this directory. Invoking `make` in this directory should build a `Checkpoint.pdf` file and a `Report.pdf`. There are skeleton `Checkpoint.tex` and `Report.tex` latex files, as-well as a `Makefile` to help get you started.

`programs/`
: Your solution to Part IV should reside in this directory in a file named `gpio.s`.

`.gitignore`
: This is not a directory per-se, but a file understood by Git which describes files that it should ignore (i.e. *never* add to the repository). An example of its usefulness is in preventing the addition of temporary files (e.g. object files built during the compilation of your programs or report) to the repository. Note that the `.` at the start of `.gitignore`'s name marks it as a hidden file.[17] You may thus need to configure your file manager to show you such files or, for example, pass the `-a` or `-A` options to `ls`:

```
% ls -a
```

[17] Such files are commonly referred to as "dotfiles" for this reason.

Test platform

Even though you will be submitting your work via LabTS, you will **not** be testing your code there. You will be able to clone a separate test platform from Gitlab, to help you test your project for Parts I and II. You can get this via:

```
git clone https://gitlab.doc.ic.ac.uk/mvalerae/arm11_1617_testsuite.git
```

The test platform has a `test_cases` directory containing test cases for many of the instructions you need to implement. For each test case, there is a source assembly file (ending in `.s`), an expected binary from assembling the source file (no file extension), and an expected output file that should be produced on standard output by your emulator (ending in `.out`).

Some tests are prefixed `opt_`, indicating they are optional.

To help you run all the tests over your assembler and emulator, there is a `testserver.rb` script. If you run this program, passing it as an argument the path to your `assemble` and `emulate` binaries, you will be able to point a web-browser at `localhost:18000` to run and view the entire test suite.

Competitions and Prizes

There are 4 group prizes + 1 individual prize on offer, awarded for:

- The ARM11 16/17 "Most Helpful Student on Piazza", awarded to one student for successfully and **constructively** answering the most questions on Piazza.
- The ARM11 16/17 "Best Group Reflection", as contained in the group's Final Report. c
- The ARM11 16/17 "Most Interesting Extension" (x2), for two groups with *interesting* Part IV extensions. An extension's "interest" will be judged on its code (or how the code was produced) and of any output generated from the Raspberry Pi.
- The ARM11 16/17 "Best Overall Project", as judged by a combination of the overall quality of the code, the Final Report and the Presentation.

Assessment

This exercise forms part of the assessment of the C Programming course, whose marks are distributed as follows:

- Checkpoint (report and interview): 5%
- Emulator and Assembler: 10%
- Completing the WebPA online group assessments: 5%
- Extension: 4%
- Final report: 11%
- Presentation: 5%

The marking for each section above (apart from the presentation) will focus on the following categories:

- Design, implementation, style and readability (up to 13.5 marks): this will be judged by examining your C code and reading your reports. In addition to the usual good practices expected in your previous laboratories (program layout, elimination of duplicate code, appropriate commenting, meaningful assertions, sensible variable naming etc.) you will also be expected to create a codebase with a uniform style and coherent design, despite the fact that different contributions may originate from different group members.

- Correctness and testing (up to 10.5 marks): progress on test cases will be judged through automated testing and examining how your code handles edge cases using the provided test suite. You should also indicate how you would test your extension.
- Group working and communication (up to 10.5 marks): this will be judged on online group assessments, your group reflections in your reports and by looking at your Git commit logs for the *master* branch. Please ensure that your commit messages on this branch (and indeed all branches if possible) are meaningful and be sure to acknowledge all group members who contributed to a given commit. For example, if pair-programming, the committer should mention their co-worker.
- Project hygiene (up to 0.5 marks): a very small number of marks will be used to judge the quality of the revisions you submit for marking. We will consider whether build artifacts (e.g. `.o` files) have been correctly ignored, if your `Makefile`s compute the correct dependencies and perform the minimal amount of work, etc.

For the reports and presentations, remember that we want to see what you have learned, and any insights you can give us into your project and experiences. Simply stating what you have done without *reflecting* on the assignment will not be sufficient for top grades!

Checkpoint: Due on the 2nd June

On or before the 2nd of June, you should submit to CATe a PDF (not more than one sheet, i.e. 2x A4 sides maximum) containing a summary that outlines your group working and implementation of the emulator. This report should be in LaTeX and must include:

- A statement on how you've split the work between group members and how you are co-ordinating your work.
- A discussion on how well you think the group is working and how you imagine it might need to change for the later tasks.
- How you've structured your emulator, and what bits you think you will be able to reuse for the assembler.
- A discussion on implementation tasks that you think you will find difficult / challenging later on, and how you are working to mitigate these.

You will also need to complete an interim peer-assessment on or before the 1st of June through the WebPA system. The feedback on this will be returned to you on the 5th of June.

During the lab sessions in Week 6 we will spend 20 minutes with each group, looking at your progress and discussing your report and your reactions to your WebPA feedback. A timetable for these interviews will be organised closer to the time.

Assembler and Emulator, Raspberry Pi program, Extensions and Report: Due on the 16th June

You should complete your assembler and emulator, gpio program and chosen extension, and then write your work in a report, as detailed in Part IV. This Final Report must be written in LaTeX and submitted as a PDF to CATe along with your code.

Ensure you have pushed your repository with all code (and sources to your report) back to Gitlab and that you have pressed the Submit to CATe button for the right commit.

You should write a short 3-page (i.e. maximum 6 A4 sides) final report documenting your project and extension, which must contain:

- How you've structured and implemented your assembler.
- A description of your extension, including an example of its use.
- High-level details of the design and a discussion of any challenges/problems that had to be overcome during the extension's implementation.

- A description of how you have tested your implementation, and a discussion of how effective you believe this testing to be.
- A group reflection on programming in a group. This should include a discussion on how effective you believe your way of communicating and splitting work between the group was, and things you would do differently or keep the same next time.
- Individual reflections (at least one paragraph per group member). Using both sets of WebPA feedback, and other experiences, *reflect* on how you feel you fitted into the group. For example, what your strengths and weaknesses turned out to be compared to what you thought they might be or things you would do differently or maintain when working with a different group of people.

Presentation: Due on the 20th June

At the end of week 8 (21st, 22nd and 23rd June) you will be giving a 15 minute group presentation on your project and extension. Presentation slides for your talk should be submitted to CATe on/before the 20th of June, in PDF format.

Your presentation should present your project and extension, and all group members should participate. Your presentation should include:

- A demonstration of the provided test-suite running against your emulator and assembler. You may use this as an opportunity to briefly discuss any interesting or failing test cases if appropriate.
- An overview and demonstration of your extension. Also include any testing of your extension that you undertook.
- A reflection on the assignment. For example, how well do you think it went, how well did you work as a group. Are there any particular experiences or insights that this exercise has given you into programming with your peers? What would you do differently next time, or seek to maintain for future group programming assignments?