ARMv8 AArch64 – Assembler and Emulator

COMP40009, Programming III, Assessed C Laboratory

26 May – 23 June, 2023

Aims

- To enhance your C programming skills
- To reinforce your understanding of instruction set architectures and low-level aspects of machine operation.
- To gain experience of group work.
- To prepare and deliver a report and presentation summarising your design and implementation.
- To design, implement and document an extension of your own choice.

Introduction

In this exercise you will be working with a subset of the A64[1] Reduced Instruction Set Computer (RISC) architecture for AArch64, the 64-bit execution mode of the ARMv8-A architecture. You will also be working with a Raspberry Pi 3B[2], which contains a Broadcom BCM2837 system on a chip (SoC), incorporating a Quad Core ARMv8 Cortex-A53 1.2 GHz processor. There are four parts to the exercise:

Part I
An AArch64 emulator that simulates the execution of an AArch64 *binary file* on a Raspberry Pi, and describe how you structured it.

Part II
An AArch64 assembler that translates an AArch64 assembly *source file* containing A64 instructions into a binary file that can subsequently be executed by the emulator.

Part III
An A64 assembly program that flashes an LED on a Raspberry Pi. You will be provided with a Raspberry Pi along with an SD card, power supply and HDMI cable, and will be reusing your work from Parts I and II.

Part IV (Optional for JMC)
An extension of your own choice. For example, you might add new instructions to the AArch64 assembler or emulator to take advantage of features found on the Raspberry Pi, or construct a debugger or high-level compiler. This extension must use the C programming language, and may also exploit additional hardware, like LEDs, that can be provided.

[2]See developer.arm.com/documentation/ and developer.arm.com/documentation/den0024/a.
[2]See raspberrypi.org/.

The programs that you will write in Parts I and II can be used together to execute AArch64 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 four. All members of your group should contribute approximately equally 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. Pair programming is highly recommended.

This is also an exercise in small-group software development. 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 Peer 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 A64 binaries (`add01.bin`, `bne02.bin` and `ldr08.bin`) 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.

Contents

1 The Emulator

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

```
$ ./emulate <file_in>
```

The emulator should also support an optional *output* file, supplied as the second argument:

```
$ ./emulate <file_in> <file_out>
```

When no output file is specified, the emulator should print the results to `stdout`; when one is specified, the results should be saved in `<file_out>`.

For example, the following command emulates the binary file `add01.bin` and prints the results to `stdout`:

```
$ ./emulate add01.bin
```

The following command emulates the file and write the results to `add01.out`:

```
$ ./emulate add01.bin add01.out
```

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

1.1 An ARMv8 Emulator

The CPU on the Raspberry Pi has an ARMv8-A architecture, which is capable of running in 32-bit[3] and 64-bit execution modes. The architecture of the 64-bit execution is called AArch64, and its associated instruction set is called A64. This project focuses only on the AArch64 architecture.

The emulator should implement the behaviour of an ARMv8 machine running in 64-bit execution mode (AArch64), with A64 instructions loaded into memory. It should be able to decode and execute various A64 instructions, which will act on the programs status and memory. The emulator may assume that ARMv8 machine memory has a capacity of 2MB ($2 * 2^{20}$ bytes), so that all addresses can be accommodated within 21 bits. The ARMv8 memory is byte-addressable. All instructions are 32 bits (4 bytes) long and aligned on a 4-byte boundary, such that all instruction addresses are multiples of 4.

An ARMv8-AArch64 system has thirty-one general purpose 64-bit registers, and four 64-bit special registers ZR, PC, `PSTATE` and SP. SP, the stack pointer register, can be ignored for this exercise. ZR is the *zero register*, which ignores writes and always returns the value 0. The PC is the (64-bit) program counter, which may be changed by the execution of a branch instruction. See Section 1.1.1 for more information on the registers and the `PSTATE`. Instructions may modify values in both the ARMv8 memory and the general-purpose registers. In addition to the memory structure, you will also need data structures to store the values of the general and special registers. You might want to separate the structures of the general and special registers. A good idea would be to define a C `struct` that captures the state of an ARMv8 machine.

When the emulator starts, all memory locations and register values should be initialised to 0, reflecting the state of an ARMv8 machine when it is first turned on. For the purpose of this exercise, you may assume that the emulator begins by starting with the instruction at address `0x0`, the initial value of the *program counter* (PC) register, and then simulating the execution of each instruction in turn[4].

[3] AArch32 defines the architecture of the 32-bit execution state, and has instruction sets A32 (called ARM in previous versions) and T32 (Thumb). AArch32 is primarily used to preserve backwards compatibility with ARMv7, which was a 32-bit architecture.

[4] In practice, this is not always the case; for example, the assembly kernel file on the Raspberry Pi starts at address `0x80000`

The A64 instruction set does not have a halt instruction; the processor will run forever. To make testing of the emulator easier, we will deviate slightly from A64 and interpret the instruction with value `8a000000` (which, as you will see when making your assembler, corresponds to '`and x0, x0, x0`') as a signal to your emulator that it should terminate. Upon terminating, you should print to the standard output the value of each general purpose and special register, and the contents of any non-zero memory location (this includes the memory occupied by the instructions).

1.1.1 Registers

General Purpose Registers There are 31 general purpose, 64-bit registers, called R0, ..., R30. Each register Rn can be used in: 64-bit mode, in which case it is referred to as Xn; or, 32-bit mode, referred to as Wn. A *read* from Wn will use only the lower 32 bits of Rn. *Writing* a 32-bit value x to Wn will set the lower 32 bits of Rn to x, and the upper 32 bits to 0. A 64-bit value cannot be written to Wn, it must be written using Xn. It follows that using Xn or Wn can lead to different results.

Special Registers AArch64 also has *special registers*[5]:

- Zero Register (ZR): Reads from ZR always return 0; writes to ZR are ignored – this is useful when executing an instruction that changes the program state (see below) without storing the computed value. The zero register can be used by most data processing instructions. In instructions, `xzr` is the 64-bit zero register, and `wzr` is the 32-bit wide version.
- Program Counter (PC): This contains the address of the current instruction. The PC is not a general purpose register, and can't be used with data processing instructions (see Section 1.4). The PC is modified by branch instructions (see Section 1.8), which will add a (signed) offset to the current PC value. For other instructions, the PC will increment by 4 after each execution of an instruction.
- Stack Pointer: (SP): The stack pointer register[6]. *You do not need to implement the stack pointer*. In instructions, `sp` is used to access the stack pointer register in 64-bit mode (notice it is not called `xsp`); `wsp` is the 32-bit stack pointer, using just the lower 32 bits of the stack pointer register. This means that the address range of `wsp` is much smaller than that of `sp`. The stack pointer register cannot be referenced by most instructions; it is only used in particular cases.
- The Processor State Register: (see below).

The *processor state* (PSTATE) register[7] is used to configure the operating mode of the processor, and to check conditions for conditional branch instructions, which will execute only when the flags satisfy a particular configuration. The PSTATE fields to be implemented are[8]:

- `N` - **N**egative condition flag (the last result was negative).
- `Z` - **Z**ero condition flag (the last result was zero).
- `C` - **C**arry condition flag (the last result caused a bit to be carried out)
- `V` - o**V**erflow condition flag (the last result overflowed).

The PSTATE register cannot be directly modified by any instruction; precisely when and how to read and update PSTATE is discussed in the later section on interpreting instructions. When the CPU starts, the initial value of PSTATE has the Z flag set, and the rest are clear.

Although the PSTATE is a register, you might find it easier to create a special C `struct` specifically for the PSTATE, containing four boolean fields for the condition flags.

[5] See https://developer.arm.com/documentation/102374/0100/Registers-in-AArch64---other-registers

[6] There are actually four stack pointers, SP_EL0 through SP_EL3 – one for each 'exception' level, which manage what parts of memory and hardware can be used by the current process. You don't need to implement any of these

[7] In previous ARM architectures, this was called the program status (CPSR) register.

[8] https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/Processor-stat6e

1.2 Execution Pipeline

The execution pipeline of the emulator can broadly be described in three phases: fetch, decode and execute.

1. Fetch: the 4-byte instruction must be fetched from memory.
2. Decode: the 4-byte word should be decoded into its respective instruction. This will involve inspecting the different bits of the word to determine exactly which instruction is encoded (see Section 1.3).
3. Execute: Emulate the behaviour of executing the instruction on an ARMv8 processor in the 64-bit execution state (AArch64). The effects of the instruction should update the emulator's registers, `PSTATE` and memory accordingly.

This execution pipeline can be assumed to occur within a single CPU cycle, so that the PC is pointing to the instruction currently being executed (which is the same instruction that has been fetched and decoded). This means that offsets in, for example, branch instructions, can be directly applied to the PC value.

In previous versions of ARM, each phase of the pipeline executed per CPU cycle, meaning that at the same time: an instruction was being executed, the next instruction was being decoded, and the instruction after that was being fetched. This had a side-effect, that the instruction being executed is two instructions behind the instruction currently being fetched at PC, which meant that the PC was always 8 bytes ahead of the current instruction being executed. This meant that instructions using PC offsets, like branches, needed to take into account the 8-byte offset.

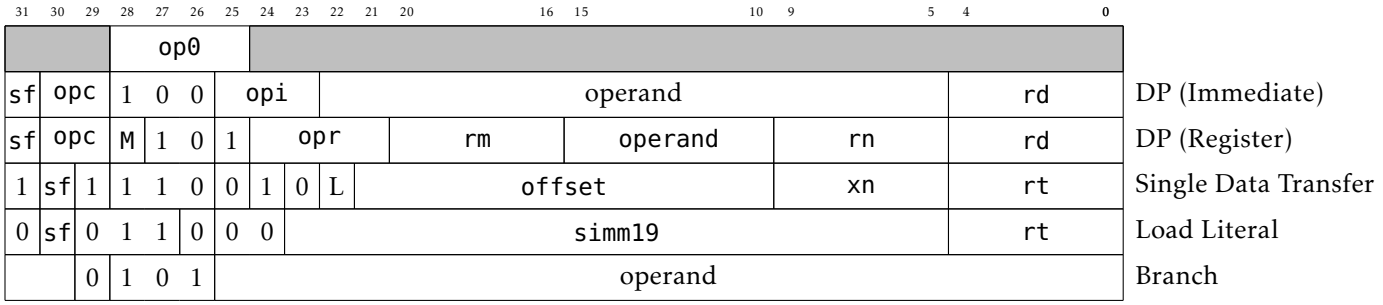
In AArch64 this is no longer the case, and the PC can be assumed to be pointing to the instruction being executed. This lets different AArch64 processors have different *actual* pipeline lengths: the processor on the Pi, a Cortex-A53, has an 8-stage pipeline, which executes each stage in order; a Cortex-A57 has a 15+-stage pipeline that can execute out-of-order.

1.3 A64: The AArch64 Instruction Set

A64 is the 64-bit instruction set used by AArch64 architecture. Instructions in A64 are 32-bit words encoded in a *little-endian*[9] format. A program is encoded as a sequence of instructions stored in memory; the processor runs this program by reading and executing the instructions one by one.

The full A64 instruction set is divided into 4 categories[10], three of which you will need to support:

- *Data Processing*: instructions that perform arithmetic and logical operations on the registers, which can indirectly modify the condition flags based on the outcomes of these operations. Data Processing instructions are further categorised by whether their last operand is an *immediate* (a constant number) or a *register*.
- *Single Data Transfer*: instructions that read or write from/to memory.
- *Branch*: instructions that affect the flow of the program by modifying the PC.



[9] This means the lowest byte of the word is stored at the lowest point in memory.

[10] The category not listed here is *Exception/System* instructions, which enable interaction with system-level operations, and for generating exceptions. These will not be implemented.

This subset of commands will be enough to implement an assembly program that can turn on the power LED on a Raspberry Pi 3B (see Section 3 (Raspberry Pi)).

The group of a particular encoded instruction is determined by the bits 28-25.

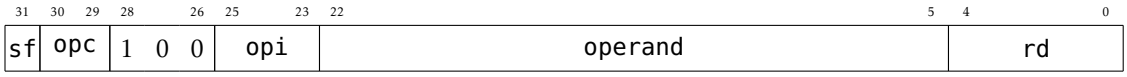
`op0`	Group
`100x`	Data Processing (Immediate) (Section 1.4)
`x101`	Data Processing (Register) (Section 1.5)
`x1x0`	Loads and Stores (Section 1.7)
`101x`	Branches (Section 1.8)

The following sections will make use of the following shorthands:

- `immN` will refer to an `N`-bit unsigned immediate value.
- `simmN` will refer to an `N`-bit signed value.
- In instructions, arguments enclosed in braces, `{ ... }`, are optional.

1.4 Data Processing Instruction (Immediate)

A data processing instruction with a shifted immediate takes the following form:

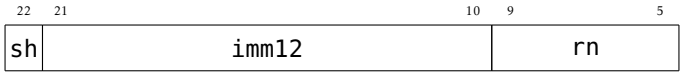


DP (Immediate)

Where each part means:

- `rd`, The encodings of Rd, the destination register, which the result of the operation is written to. A value of `11111` encodes the zero register, ZR, unless the instruction is an arithmetic instruction that does not set condition flags, in which case it encodes the stack pointer SP (optional). You do not need to implement the case when Rd is the stack pointer.
- `sf`: The bit-width of all the registers in the instruction: 0 for 32-bit, 1 for 64-bit. When `sf` is 0, this means Rd is accessed as a W-register, i.e. Wd.
- `opc`: The operation code, determining the operation to be performed.
- `operand`: The value to be saved into Rd.
- `opi`: This determines the type of data processing operation, which determines the interpretation of `opc` and `operand`. When `opi` has binary value: `010`, the instruction is an *Arithmetic* instruction; `101` means the instruction is a *Wide Move*. You do not have to handle other values of `opi`.

Arithmetic When `opi` is `010`, then the instruction is an arithmetic instruction. This means that `operand` will have the following structure



`imm12` is a 12-bit unsigned immediate value; an integer value in the range 0 to $2^{12} - 1$. `sh` is a flag that says whether the immediate is to be left-shifted by 12-bits; the shift is applied when `sh` is 1. See Section 1.6 for the meaning of a left shift. If the shift is applied, the maximum possible value will be $2^{24} - 2^{12}$. `rn` is the first operand register; it will have the same bit-width as `rd`, according to `sf`.

The arithmetic operation is then performed between the value of `rn` and the second operand, which is `imm12` possibly shifted by 12-bits.

opc	*Mnemonic*	*Instruction*	*Action*
00	add	Add	Rd := Rn + Op_2
01	adds	Add and set flags	Rd := Rn + Op_2, update condition flags
10	sub	Subtract	Rd := Rn – Op_2
11	subs	Subtract and set flags	Rd := Rn – Op_2, update condition flags

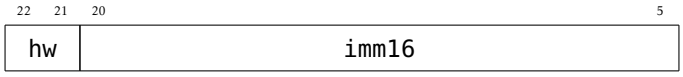
If rn is 11111, then it encodes the stack pointer, SP (optional; you do not need to handle this case). If rd is 11111, then it also encodes the stack pointer, SP (optional), unless the instruction sets condition flags, in which case it encodes ZR. You do not need to handle the case when rd is the stack pointer (i.e. rd value is 11111 and the instruction doesn't set condition flags).

When the PSTATE flags are to be updated, they are done so according to the following:

- N is set to the sign bit of the result.
- Z is set only when the result is all zero.
- C: In an addition, C is set when the addition produced a carry (unsigned overflow), or zero otherwise. In a subtraction, C is set to zero if the subtraction produced a borrow, otherwise it is set to one.
- V is set when there is signed overflow/underflow.

The rules for N, C and V depend on the bit-width of the instruction.

Wide Move When opi is 101, then the instruction is a wide move. operand then has the following structure:



imm16 is the 16-bit immediate value to move, so is in the range of 0 to $2^{16}-1$. hw encodes a logical shift left by hw∗16 bits. The operand's value is thus imm16 « (hw * 16). In the 32-bit version of the move instruction, hw can only take the values 00 or 01 (representing shifts of 0 or 16 bits). The move operation that is performed between Rd and the operand is then determined by the opcode, opc.

opc	*Mnemonic*	*Instruction*	*Action*
00	movn	Move wide with NOT	Rd := ∼Op
10	movz	Move wide with zero	Rd := Op
11	movk	Move wide with keep	Rd[shift + 15 : shift] := imm16.

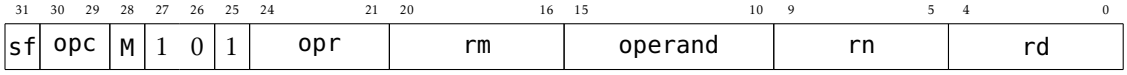
These operations mean, in detail:

- movn: Mode wide with NOT. This sets the value of Rd to the bitwise negation of Op. The bitwise negation of Op will be the 32 or 64-bit value with all bits equal to 1, except for the bits between shift and shift+15, which will have value ∼imm16. This is equivalent to the value ∼(imm16, lsl #shift)

  When the bit-width of the instruction is 32-bit, the upper 32 bits of Rd are zero.
- movz: Move wide with zero. This simply sets the value of Rd to Op.
- movk: Move wide with keep. This moves the 16-bit value into Rd, in between the bits hw∗16 and (hw∗16) + 15, keeping the other bits unchanged. Rd[$u : l$] := x denotes setting the bits of Rd in the range l (lower) to u (upper), to the value x. For example, if imm16 is 0xFF and hw is 1, then the value of Rd wil be Rd[63 : 32] : 00FF : Rd[15 : 0], where $x : y$ denotes the concatenation of the bits of x and y. If sf is 0, then Rd[63 : 32] will be set to zero.

1.5 Data Processing Instruction (Register)

A data processing operation with register operands has the following structure:



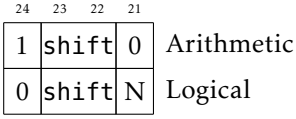
DP (Register)

Where each part means:

- `rd`, `rn`, `rm`: The encodings of Rd, Rn and Rm. The value `11111` encodes the zero register, ZR. If the instruction has a shift, it is performed on Rm.
- `sf`: The bit-width of all the registers: 0 for 32-bit, 1 for 64-bit.
- `opc`: The operation code, determining the operation to be performed.
- `operand`: The last operand of the instruction, which has different meanings depending on the instruction type.
- `M` and `opr` determine the type of data processing instruction to perform, according to the table below.

`M`	`opr`	Group
`0`	`1xx0`	Arithmetic
`0`	`0xxx`	Bit-Logic
`1`	`1000`	Multiply

Arithmetic and Logical Arithmetic and logical instructions with a shifted register operand have similar structures. See Section 1.6 for the meaning of the different shifts. `operand` (bits 15-10) is a 6-bit unsigned immediate value, and `opr` has the following structure:



- `shift`: The type of shift to perform on Rm: the encodings are `00`, `01` and `10` for shifts `lsl`, `lsr` and `asr` (respec.). The value `11` is only valid for logical instructions, in which case it refers to a `ror` shift.
- `operand` is a 6-bit immediate determining the shift amount. For the 32-bit variant, this is in the range 0-31. In the 64-bit variant, the range is 0-63.
- `N` (logical only): whether the shifted register is to be bitwise negated.

The instruction then performs the arithmetic/logical operation with the first operand being Rn, and the second operand, Op_2. Op_2 is Rm shifted `operand` many bits by the shift encoded in `shift`.

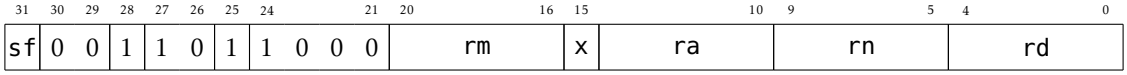
The operation code in `opc` for arithmetic instructions is the same as for the data processing with immediate operand. For logical instructions, the operation code and `N` determine the operation to perform;they take the following values and meanings:

Mnemonic	`opc`	`N`	*Instruction*	*Action*
`and`	`00`	`0`	Bitwise AND	$\mathsf{Rd} := \mathsf{Rn} \,\&\, \mathsf{Op}_2$
`bic`	`00`	`1`	Bitwise bit clear	$\mathsf{Rd} := \mathsf{Rn} \,\&\, \sim\mathsf{Op}_2$
`orr`	`01`	`0`	Bitwise inclusive OR	$\mathsf{Rd} := \mathsf{Rn} \mid \mathsf{Op}_2$
`orn`	`01`	`1`	Bitwise inclusive OR NOT	$\mathsf{Rd} := \mathsf{Rn} \mid \sim\mathsf{Op}_2$
`eor`	`10`	`0`	Bitwise exclusive OR	$\mathsf{Rd} := \mathsf{Rn} \wedge \mathsf{Op}_2$
`eon`	`10`	`1`	Bitwise exclusive OR NOT	$\mathsf{Rd} := \mathsf{Rn} \wedge \sim\mathsf{Op}_2$
`ands`	`11`	`0`	Bitwise AND, setting flags	$\mathsf{Rd} := \mathsf{Rn} \,\&\, \mathsf{Op}_2$ (update condition flags)
`bics`	`11`	`1`	Bitwise bit clear, setting flags	$\mathsf{Rd} := \mathsf{Rn} \,\&\, \sim\mathsf{Op}_2$ (update condition flags)

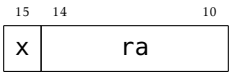
For arithmetic instructions, the PSTATE flags are updated in the same way as with the immediate operand instruction. For logical instructions, when the PSTATE flags are to be updated, they are done so according to the following rules:

- N is set to the sign bit of the result.
- Z is set to 1 if the result of the operation is zero.
- C and V are set to 0.

Multiply A multiply instruction has the encoding:



This means that, for a multiply instruction, opc is 00, and opr is 1000. The structure of operand is:



The meaning of these are:

- ra: the encoding of Ra. The value 11111 encodes the zero register.
- x: whether the to negate the product. This means the instruction is madd (Multiply-Add) (x = 0) or msub (Multiply-Sub) (x = 1).

The meaning of the other fields, sf, rm, rn and rd are the same as for the Arithmetic/Logical instruction.

madd and msub, multiplies two register values and adds/subtracts it from a third register value. There meanings are given below:

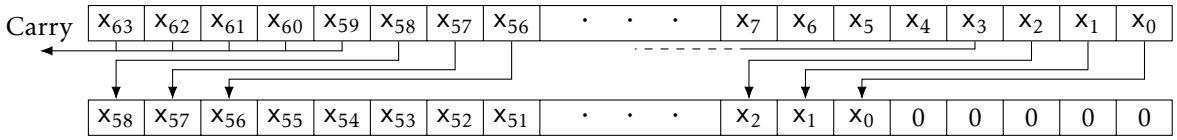
Mnemonic	*Instruction*	*Action*
madd	Multiply-Add	Rd := Ra + (Rn * Rm)
msub	Multiply-Sub	Rd := Ra – (Rn * Rm)

1.6 Bitwise Shifts

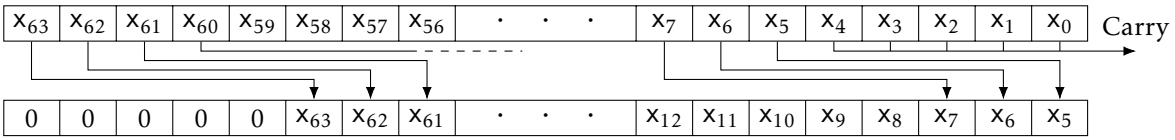
Many instruction operands can be given a *shift*, which performs a bitwise operation on the operand before it is used by the instruction. The possible shifts are:

- lsl - Logical Shift Left: Shifts the bits of the operand to the left by shift_amount bits.
- lsr - Logical Shift Right: Shifts the bits of the operand to the right by shift_amount bits, filling the vacated bits with 0.
- asr - Arithmetic Shift Right: Shifts the bits of the operand to the right by shift_amount bits, filling the vacated bits with the sign bit of the operand.
- ror - Rotate Right: Rotates the bits of the operand to the right by shift_amount bits. This means the bits are shifted right, and those carried out at the bottom are rotated back into the left-hand end.

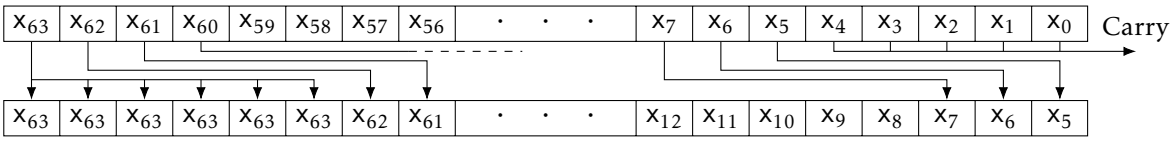
The operand size is important when shifting. In particular, when a shift is applied to a value coming from a register Rn, the width of the Rn must be taken into account. For lsl, lsr, asr, when the register is 64-bit (Xn), the shifts are performed on the 64-bit value of Rn as in Figure 1. When the register is in 32-bit mode (Wn), the higher 32-bits of the value are set to zero, and the shifts are performed on the lower 32-bits, with the result truncated to 32-bits. For example, if Rn = 0x000a 0001 f000 0000, then the value of Wn left-shifted by 1 bit will be 0xe0000000.



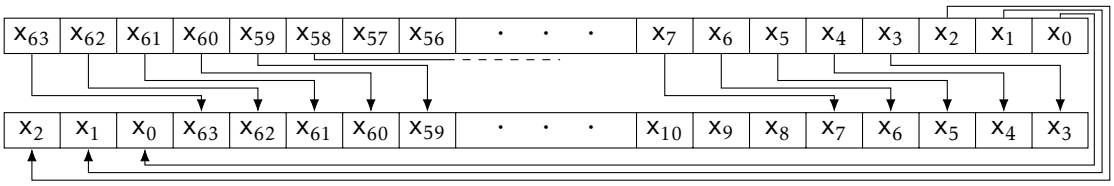
(a) `lsl`: Logical Shift Left by 5 bits



(b) `lsr`: Logical Shift Right by 5 bits



(c) `asr`: Arithmetic Shift Right by 5 bits



(d) `ror`: (64-bit) Right Rotate by 3 bits

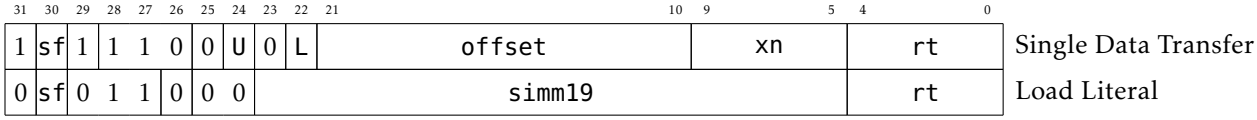
Figure 1: Example 64-bit Shifts

`asr` needs to know the register width to determine where the sign bit is. The sign bit is indicated by the most significant bit: in 64-bit mode this is bit 63; in 32-bit mode, this is bit 31. `ror` also needs to know the width before rotating. If the operand is 64-bits wide, then the rotation is from bits 63-0 (across the whole value). If the operand is 32-bits wide, then the rotation is only performed on the lower 32 bits – this means the top 32 bits are ignored.

To implement any the shifts/rotations, you can either truncate the value to the bit-width first and then perform the shift/rotation, or perform the shift/rotation and then truncate. The important thing is to choose the correct bits to manipulate during `asr` and `ror`.

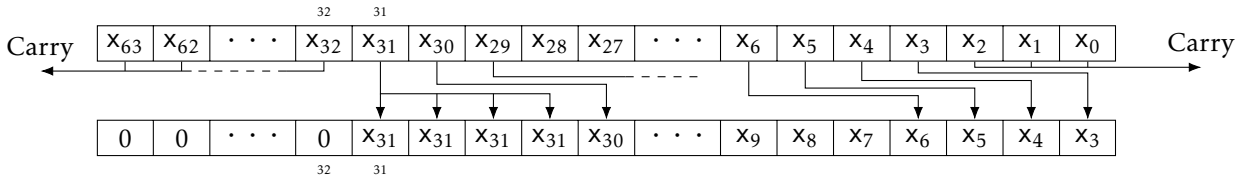
Figures 1 and 2 show examples of 64- and 32-bit shifts.

1.7 Single Data Transfer Instructions

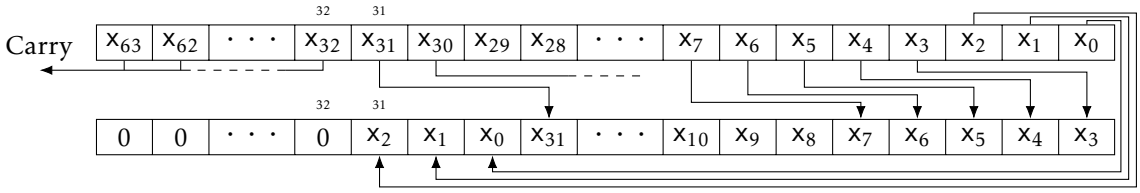


Where each part means

- `sf`: The size of the load. When `sf` is 0, the target register is 32-bit. Otherwise, the target register is 64-bit.
- `L`: The type of data transfer. When `1`, this is a load operation. Otherwise it is a store.



(a) `asr`: 32-bit Arithmetic Shift Right by 3 bits



(b) `ror`: 32-bit Right Rotate by 3 bits

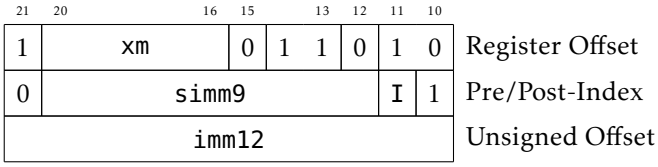
Figure 2: Example 32-bit Shifts

- `U`: Unsigned offset flag. When this bit is set, the addressing mode is an *Unsigned Offset*
- `offset`: The meaning of the `offset` depends on the addressing mode, discussed in Section 1.7.1.

Single data transfers are used to load and store words and double-words from/to memory. AArch64 does not support memory-to-memory operations, so data must be loaded into registers before use. All load and store instructions have a *target register*, Rt; which determines the register that data is stored from or loaded into. Apart from the load from literal, all loads and stores also have a *base register*, Xn, which contains the base address to calculate the offset from. Notice that this is always a 64-bit X-register, and can be either general purpose X0,...,X30 or the stack pointer, SP(optional; you don't need to handle the case when Xn is the stack pointer).

1.7.1 Addressing Modes

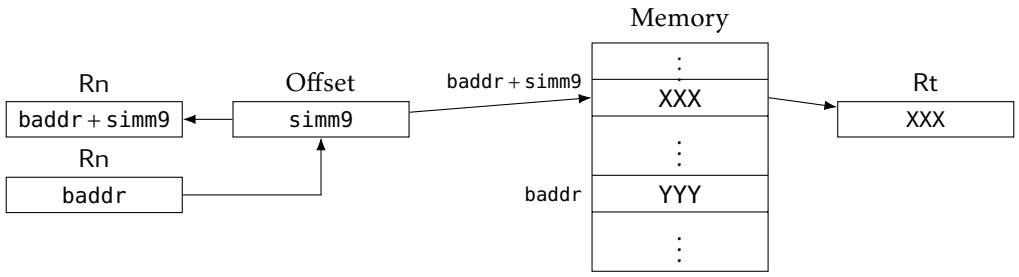
There are four main ways the memory address to access is computed: a base register plus an unsigned immediate offset, pre/post-indexed access, a base plus a register, and literal address (loads only).



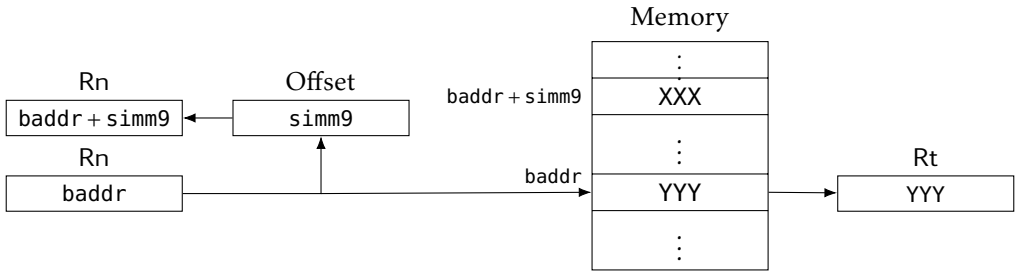
Unsigned Immediate Offset When `U` is 1. The unsigned immediate offset `uoffset` is encoded by `imm12`; the value of `uoffset` depends on the bitwidth of Rt. In the 64-bit variant, when Rt is an X-register, then `uoffset` is given by `imm12*8`. In the 32-bit variant, then `uoffset` is equal to `imm12*4`. The target address is given by Xn plus the `uoffset`; that is, the address is `uoffset` zero-extended to 64-bit plus the value in Xn.

Pre-Indexed When `I` is 1. The address is given by the the sum of the value of Xn and the signed value `simm9`, which is in the range -256 to 255. This calculated address is written back to be the new value of Xn.

Post-Indexed When `I` is 0. The address is given by the value of Xn. After performing the data transfer, the value of Xn is updated by adding the signed value `simm9`, which must be in the range -256 to 255.



(a) Pre-Indexing



(b) Post-Indexing

Figure 3: Pre- and Post-Indexing

Register Offset The address used by the instruction is the value of Xn plus the the value of Xm[11]. Notice as well that the offset register must be an X-register.

Literal (Load Only) A load from a literal address loads from the address calculated by adding the PC and the offset, which is equal to `simm19` $*$ 4. Remember that this offset will have to be sign extended to 64-bit.

Summary The different address types are summarised in the table below.

Addressing Mode	*Transfer Address*	*Write-back*
Unsigned Offset	Xn + `uoffset`	
Pre-Indexed	Xn + `simm9`	Xn := Xn + `simm9`
Post-Indexed	Xn	Xn := Xn + `simm9`
Register Offset	Xn + Xm	
Load from Literal	PC + `simm19` $*$ 4	

In the unsigned offset `uoffset` is equal to: `imm12` $*$ 4 when Rt is a W-register; `imm12` $*$ 8 when Rt is an X-register. Figure 3 shows the difference between pre- and post-indexing, where `baddr` is the *base address* stored in Rn.

1.7.2 Load Size

The size of the load/store at address `addr` is determined by if the target register, Rt, is accessed as a 32-bit or 64-bit register.

- When Rt is accessed as a 32-bit W-register, Wt: `ldr` will load the 4-byte word from `addr` to `addr+3`; the byte at `addr` is loaded into the lowest 8-bits of Rt. `str` will store the 4-byte value in Rt into memory at location `addr`; the lowest word in Rt is stored at `addr`. `str` will store the lower word in Wt into memory at the address `addr` to `addr` + 3.

[11] A `lsl` or extension can be applied to this register, but you don't need to implement this.

- When Rt is accessed as a 64-bit X-register, Xt: `ldr` will load a double-word (8-bytes) from `addr` to `addr`+7; the byte at `addr` is loaded into the lowest 8-bits of Rt. `str` will store the entire entire 8-bytes of Rt into memory, with the lowest byte of Rt being stored at `addr`.

1.8 Branch Instructions

Branch instructions modify the PC, causing the CPU to jump to executing the instruction the location that the new PC value points to. Branches can be unconditional, meaning they always execute, or conditional, when they only execute when the `PSTATE` satisfies certain requirements. Unconditional branches can use either a signed immediate offset, or can be to an address stored in a register. Conditional branches always use a signed immediate offset.

- (Unconditional) `simm26`: The `offset` to apply to the PC is encoded as `simm26` $*$ 4 (sign extended to 64-bit).
- (Register) `xn`: The encoding of Xn, the register containing the address to jump to. When `xn` is `11111`, this encodes the zero register, `xzr`; you don't need to handle this case.
- (Conditional) `simm19`: The `offset` to apply to the PC is encoded as `simm19` $*$ 4 (sign extended to 64-bit).
- (Conditional) `cond`: a 4-bit encoding of the condition that must be satisfied for the branch to be executed.

Mnemonic	*Meaning*	*PSTATE Flags*	*Encoding*
`EQ`	equal	`Z == 1`	`0000`
`NE`	not equal	`Z == 0`	`0001`
`GE`	signed greater or equal	`N == V`	`1010`
`LT`	signed less than	`N != V`	`1011`
`GT`	signed greater than	`Z == 0 && N == V`	`1100`
`LE`	signed less than or equal	`!(Z == 0 && N == V)`	`1101`
`AL`	always	any	`1110`

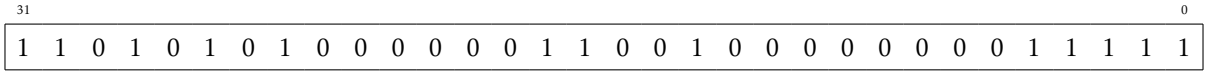
When encoding the offset (divided by 4) into 19 or 26 bits, the sign bit must be preserved to ensure the correct value is encoded. When decoding an offset, care must be taken to properly sign extend it to 64-bit.

Mnemonic	*Instruction*	*Action*
`b`	Branch to the address encoded by `literal`.	PC := PC + `offset`
`br`	Branch to the address in Xn.	PC := Xn
`b.cond`	Branch to `literal` only when `PSTATE` satisfies `cond`.	If `cond`, PC := PC + `offset`

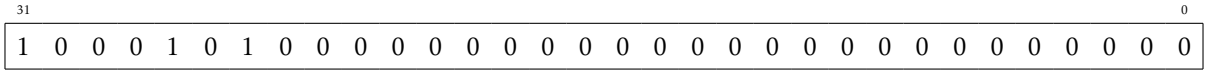
1.9 Special Instructions

In Part III, you will need to interact with the Raspberry Pi's GPU through its Mailbox. As part of this process, the assembly code will need to have aligned some instructions/data to the nearest 4 byte boundary. The easiest way to do this is by padding with the `nop` instruction, which is a *no operation*.

nop No Operation just advances the PC by 4.



Halting As mentioned at the start of this section, A64 does not feature a halt instruction. However, as it will make testing easier the instruction with (little-endian) value `8a000000` (corresponding '`and x0, x0, x0`') should be interpreted as a signal for the emulator to halt.



1.10 What to Do

Object Code File Format The object code of A64 instructions are called AArch64 binary files. AArch64 binary files store each word in a *little-endian* byte order. You should make sure that your memory layout also follows this convention, otherwise loading from and storing to memory will not work correctly.

Output File Format When the emulator finishes, it should write the state of its registers, condition flags and non-zero memory to either `stdout` or an *output* file (with the extension `.out`) specified by the second command-line argument. The output (whether to `stdout` or a `.out` file) should have the following format:

- First, for each general register (in order), it should display the register's name, an '`=`' symbol, and its hexadecimal value and then a newline. White space is not important. For example, if `X9` has value `0x8af00`, then its corresponding line in the `.out` file should be:

  ```
  X09  = 000000000008af00
  ```

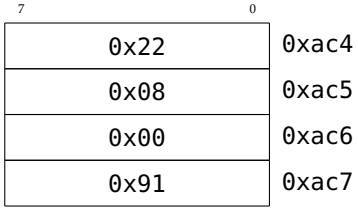
- The next line after all the registers should have the value of the PC, displayed similarly to the general registers:

  ```
  PC   = XXXXXXXXXXXXXXXX
  ```

- The `PSTATE` should output the `PSTATE`, a colon, and then condition flags in the order `NZCV`. Again, white space is not important, but this must be on a separate line. When a flag is set, it should display its corresponding name; if the flag is clear, it should print as a '`-`'. For example, if the final `PSTATE` has the `C` and `V` flags set, then the output should be:

  ```
  PSTATE : --CV
  ```

- Finally, any non-zero memory should be reported by displaying its address, a colon and then its value. Once again, white space is not important. This memory should be reported in 4-byte aligned, 4-byte chunks, each on its own line. For example, if the memory at address `0x00000ac4` has the following state:

  then the output should contain the line:

```
0x00000ac4: 0x91000822
```

An example output file could look like:

```
Registers:
X00    = 0000000000000000
X01    = 0000000000000001
        ⋮
X30    = 0000000000000000
PC     = 0000000000000008
PSTATE : -Z--
Non-zero memory:
0x00000000: 0xd2800021
0x00000004: 0x91000822
```

The test suite does not care about whitespace in your `.out` file, but each of the key/value entries should be on a separate line.

Emulator 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 2MB ($2 * 2^{20}$). Writing the emulator involves the following key tasks:

- Building a binary file loader.
- Writing the emulator loop, comprising:
  - Simulation of the ARMv8 64-bit execution mode (AArch64) pipeline phases, with the fetch, decode and execute cycle.
    - Remember: the initial value of the `PSTATE` has the `Z` bit set, and the other flags clear. This will look like `-Z-` in your `.out` file
  - Simulated execution of the different instructions outlined in Section 1.3. 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 the instruction with encoding `0x8a000000`.
  - Upon termination, output the registers values, the `PSTATE` condition flags, and any non-zero memory to a `.out` file.

1.10.1 Suggestions

Assembler Look at Section 2 (especially Section 2.4) before beginning with the emulator – you might find the suggestions there will inform how you design the emulator.

Testing A test suite is provided to help with testing the emulator, see Section 5.2.

Order of Implementation All the tests use the halt instruction, `0x8a000000`, so it would be a good idea to implement this at the start (even if it's just hardcoded). Many tests also use add instructions with immediate operands to set up the values of registers before performing an operation between them; it would be worth implementing add (immediate) instructions early on.

2 The Assembler

The assembler's main program, `assemble.c`, should read in source code from an AArch64 source file whose filename is given as the first command line argument, and output AArch64 binary code to a file, whose filename is given as the second command line argument:

```
$ ./assemble <file_in> <file_out>
```

For example, to assemble the AArch64 source file `add01.s` to the AArch64 binary file `add01.bin.bin`, one would run:

```
$ ./assemble add01.s add01.bin
```

2.1 Two-pass assembly

One source of difficulty in the assembler is found in handling labels. For example, in the program

```
b myLabel

myLabel:
    add x0, x1
    ...
```

there is an instruction to branch to `myLabel` that occurs before this label has been defined; this is called a *forward reference* to the label[12]. As the assembler will most likely read the `asm` file starting from the first line, then the assembler needs to be able to handle labels that are not yet defined. There are several solutions; we will highlight two.

Two Pass Arguably the simplest way is to perform *two* passes over the source code. The first pass creates a *symbol table* which associates *labels* (strings) with memory addresses (integers). In the second pass the assembler reads in each instruction and `.int` directive and generates the corresponding binary encoding. 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[13].

One Pass The assembly *can* be performed in a single pass, but handling forward references is more difficult. To solve this, you need to maintain a collection of addresses 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.

2.2 Assembler file format

Each (non-empty) line of an assembler file contains either an *instruction*, *directive*, or a *label*. Labels are strings that begin with an alphabetical character (`a`–`z` or `A`–`Z`) and end with a '`:`' ; for example, "`label:`". The value of the label is the address of the machine word corresponding to the position of the first instruction or directive below the label. Directives are strings that start with a '`.`'; the only one you will need to implement is '`.int`' (see Section 2.3.4).

Each instruction is mapped to exactly one 32-bit word during the assembly process. 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 `x0`, `x1`, ..., `x30` and `xzr` for the 64-bit versions, and `w0`, `w1`, ..., `w30` and `wzr` for their 32-bit versions.

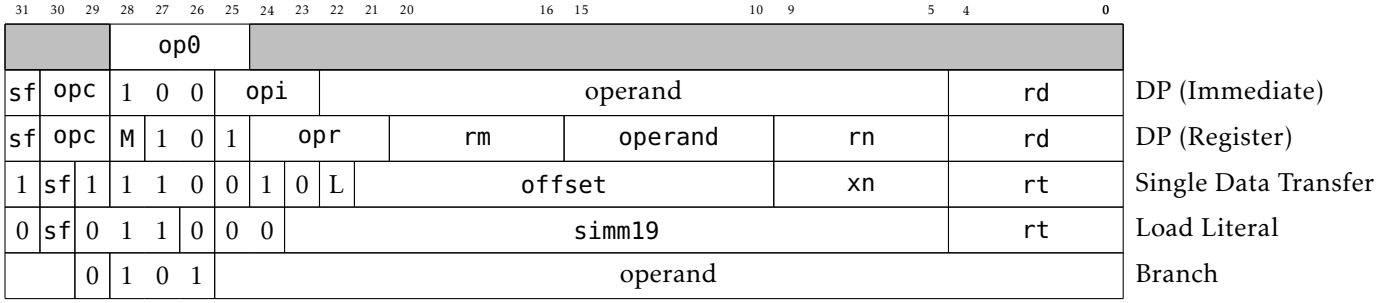
[12] The opposite case, when a label is defined before it is referenced, is called a *backward reference*.

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

2.3 Assembling Instructions

Broadly speaking, all instructions in A64 have the form of an instruction mnemonic, followed by a list of operands (usually comma separated). These operands can be registers, immediate values, or memory addresses/labels; the former two can also have arithmetic/logical shifts applied to them. The following subsections cover the syntax and behaviour of the instructions to be implemented.

Below is repeated the instruction type summary form the emulator. A summary of all the operations your assembler should support is in Table 1.



Addresses and Offsets Branch and Load instructions can have a *literal* address argument, which is either a label or an unsigned immediate (preceded by a `#` symbol). Labels are strings that start with a letter, underscore or period (a full-stop), followed by alphanumeric characters, dollar signs, underscores and/or periods. The regex for labels is `[a-zA-Z_\.][a-zA-Z0-9$_\.]*`. The encoding of these addresses is always the offset from the address of the instruction to the literal address. For example, if the instruction `b 0x500` is at address `0x10`, then the offset encoded is `0x500` – `0x10` = `0x4f0`. In general, when an instruction is at address `x`, and the literal address to branch to or load from is `y`, then the offset is `y - x`. Notice that the offset can be negative – this means the target address is before the instruction's address.

Aliases You might notice some instruction mnemonics that were not in the emulator. Don't worry – you don't have to implement the emulation of any more instructions; these instructions are *aliases* of other instructions. This means that they compile to the instructions you have already implemented, and fill in default values of some of the operands. The following table shows all the aliasing mnemonics you should implement and the instructions they alias.

Instruction	*Aliases*
`cmp rn, <op2>`	`subs rzr, rn, <op2>`
`cmn rn, <op2>`	`adds rzr, rn, <op2>`
`neg(s) rd, <op2>`	`sub(s) rd, rzr, <op2>`
`tst rn, <op2>`	`ands rzr, rn, <op2>`
`mvn rd, <op2>`	`orn rd, rzr, <op2>`
`mov rd, rm`	`orr rd, rzr, rm`
`mul rd, rn, rm`	`madd rd, rn, rm, rzr`
`mneg rd, rn, rm`	`msub rd, rn, rm, rzr`

Registers Although we have been calling the AARch64 registers R0 through R30, the only syntactically valid registers are the X-registers, `x0`, ..., `x30` and the W-registers, `w0`, ..., `w30`. The special registers have the forms: `sp` and `wsp` for the stack pointer (optional); `xzr` and `wzr` for the zero register; and the program counter can only be used in 64-bit form, PC. The `PSTATE` register cannot be used in instructions.

Mnemonic	*Instruction*	*Action in Emulator*	*Type*
`add(s) rd, rn, <op2>`	Add (set flags)	Rd := Rn + Op_2	Data Processing
`sub(s)`	Subtract (set flags)	Rd := Rn – Op_2	(Arithmetic)
`cmp rn, <op2>`	Compare	`PSTATE` flags Rn – Op_2	*Alias* of `subs`
`cmn`	Compare Negate	`PSTATE` flags Rn – Op_2	*Alias* of `adds`
`neg(s) rd, <op2>`	Negate (set flags)	Rd := –Op_2	*Alias* of `sub(s)`
`and(s) rd, rn, <op2>`	And (set flags)	Rd := Rn & Op_2	(Logical)
`bic(s)`	Bit Clear (set flags)	Rd := Rn & ∼Op_2	
`eor`	Exclusive Or	Rd := Rn ∧ Op_2	
`eon`	Exclusive Or Not	Rd := Rn ∧ ∼Op_2	
`orr`	Or	Rd := Rn | Op_2	
`orn`	Or Not	Rd := Rn | ∼Op_2	
`tst rn, <op2>`	Test bits	`PSTATE` flags Rn & Op_2	*Alias* of `ands`
`mvn rd, <op2>`	Move negate	Rd := ∼Op_2	*Alias* of `orn`
`mov rd, rm`	Move register	Rd := Rm	*Alias* of `orr`
`movn rd, imm, lsl #sh`	Move Wide with Not	Rd := ∼(`imm « sh`)	(Wide Moves)
`movk`	Move Wide with Keep	Rd[sh + 15 : sh] := `imm`	
`movz`	Move Wide with Zero	Rd := `imm « sh`	
`madd rd, rn, rm, ra`	Multiply-Add	Rd := Ra + (Rm × Rn)	(Multiply)
`msub`	Multiply-Subtract	Rd := Ra – (Rm × Rn)	
`mul  rd, rn, rm`	Multiply	Rd := Rm × Rn	*Alias* of `madd`
`mneg`	Multiply-Negate	Rd := –(Rm × Rn)	*Alias* of `msub`
`b <literal>`	Branch	PC := `<literal>`	Branch
`br xn`	Branch	PC := Xn	
`b.cond <literal>`	Conditional Branch	If `cond`, PC := `<literal>`	
`ldr rt <address>`	Load into register	Rt := ∗(address)	Single Data Transfer
`str rt <address>`	Store from register	∗(address) := Rt	
`nop`	No-op	-	Special
`and x0, x0, x0`	Halt	Halt	
`.int x`	Directive, compile to `x`	n/a	

Table 1: Instruction Behaviour

2.3.1 Data Processing Instructions

The data processing instructions can be broken down into a few types depending on the opcode:

- Multiply: `<mul_opcode> rd, rn, rm, ra` Applies to `madd` and `msub`.
- Two operand: `<opcode> rd, rn, <operand>` Applies to arithmetic and bit-logic operations.
- Single operand with destination: `<opcode> rd, <operand>` Applies to `mov`, `mul`, `mneg`, `neg(s)`, `mvn`, and wide move instructions (`movz`, `movn`, `movk`).
- Two operands, no destination: `<opcode> rn, <operand>` Applies to `cmp`, `cmn`, `tst`.

where `opcode` is the instruction mnemonic, `rd` is the destination register, `rn` is the first operand register, and `operand` is the (second) operand. `op` defines which operation to perform, and how to interpret the operands; the result of the operation is stored in Rd. In the case of instructions with two operands, the operation uses the value stored in Rm. The (second) operand `<operand>` can be:

- A register, Rn, in which case the operation will use the value stored in Rn.
- An immediate value, `imm`.

The registers Rd, Rm and Rn each can be any of a general register (R0, ..., R30) or, depending on the instruction, ZR or SP.

`<operand>` can also have a shift applied to it, so that the value used by the operation is value obtained by applying a shift to `op2`. The shifts that can be applied to the operand depend on the type of operand and the operation. In general, shifts have the form:

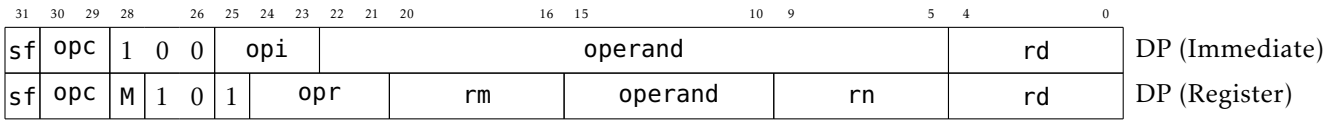
```
<shift> #<shift_amount>
```

Where `shift` is the type of shift (one of `lsl`, `lsr`, `asr` and `ror`), and `shift_amount` is the number of bits to shift by.

Thus an `<operand>` can have the forms:

```
rm{, <shift> #<amount>}
#<imm>{, <shift> #<amount>}
```

The binary format of a Data Processing Instruction that your assembler should target is described in Sections 1.4 and 1.5 (depending on the type of operand), and is summarised below:

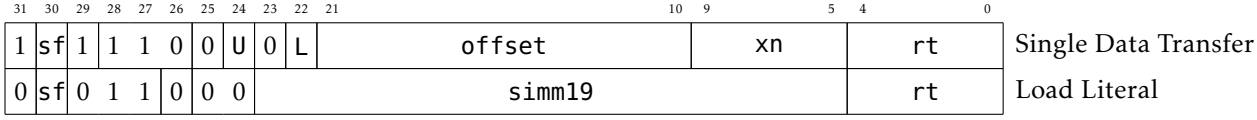


2.3.2 Single Data Transfer Instructions

There are two single data transfer operators, `ldr` and `str`, and all single data transfer instructions have the following form:

```
<ldr/str> rt, <address>
```

Recall the binary encoding for such instructions:



`ldr` loads from memory into a register, and requires the `L` bit to be set. `str` stores a register's value into memory, which will require the `L` bit to be clear. When Rt is an X-register, Xt, then `sf` will be set. Otherwise, when Rt is a W-register, Wt, then `sf` will be clear.

The syntax of `<address>` determines the addressing mode:

- Unsigned Immediate Offset : `[xn{, #<imm>}]`
  - How `<imm>` is encoded in the field `imm12` depends on the bitwidth of Rt. When Rt is an X-register, then `imm12` is equal to `<imm>`/8; when Rt is a W-register, then `imm12` is equal to `<imm>`/4.
- Pre-Indexed: `[xn, #<simm»]!`
- Post-Indexed: `[xn], #<simm>`
- Register: `[xn, xm]`
- Literal (Load only): `<literal>`. A literal can be either an integer, `#N`, or a label. The address represented by the integer or label must be within 1MB of the location of this instruction (as it is encoded as an offset), and must be 4-byte aligned. The assembler should compute the offset

Summary The different address types are summarised in the table below.

Address Code	*Addressing Mode*	*Transfer Address*	*Write-back*
`[xn]`	Zero Unsigned Offset	Xn	
`[xn, #<imm>]`	Unsigned Offset	Xn + `imm`	
`[xn, #<simm>]!`	Pre-Indexed	Xn + `simm`	Xn := Xn + `simm`
`[xn], #<simm>`	Post-Indexed	Xn	Xn := Xn + `simm`
`[xn, xm]`	Register Offset	Xn + Xm	
`<literal>`	Load from Literal	Encoded by `<literal>`	

2.3.3 Branching Instructions

Unconditional branches can be to a literal, or can be from a register value. Conditional branches always take a literal argument. You can assume the literals will be labels.

Branch instructions take one of the forms described in the following table.

Mnemonic	*Instruction*	*Action*
`b <literal>`	Branch to the address encoded by `literal`.	PC := `<literal>`
`br xn`	Branch to the address in Xn.	PC := Xn
`b.cond <literal>`	Branch to `literal` only when `PSTATE` satisfies `cond`.	If `cond`, PC := `<literal>`

The conditions that a `b.cond` instruction can take, and their condition code `cond`, are defined below.

Mnemonic	*Meaning*	*PSTATE Flags*	*Encoding*
`eq`	equal	`Z == 1`	`0000`
`ne`	not equal	`Z == 0`	`0001`
`ge`	signed greater or equal	`N == 1`	`1010`
`lt`	signed less than	`N != 1`	`1011`
`gt`	signed greater than	`Z == 0 && N == V`	`1100`
`le`	signed less than or equal	`!(Z == 0 && N == V)`	`1101`
`al`	always	any	`1110`

Calculating Offsets The encoding of the branch to literal instructions doesn't store the value of the literal, it instead stores the offset from the address of the instruction to the literal.

```
b rest

my_value:
    .int 0x3f
    .int 0

rest:
    ldr x0, my_value
```

(a) Assembly

```
0000000000000000 <my_value-0x4>:
    0:   14000003        b       c <rest>

0000000000000004 <my_value>:
    4:   0000003f        .word   0x0000003f
    8:   00000000        .word   0x00000000

000000000000000c <rest>:
    c:   58ffffc0        ldr     x0, 4 <my_value>
```

(b) Disassembly

Figure 4

Example 2.1. If `label1` is a branch label at address `0xff30`, and the address of the branch instruction is `0x1c4`, then the offset is `0xff30` − `0x1c4` = `0xfd6c`.

If instead the instruction is at `0xff1c4`, then the offset will be `0xff30` − `0xff1c4` = −`0xef294`.

2.3.4 Special Instructions/Directives

`nop` The assembler should compile a `nop` instruction to its encoding as given in Section 1.9.

`.int` Directives Directives are instructions for the *assembler* (not the CPU) to perform. These all take the form of a '`.`' followed by the directive name. For example, the assembler will read `.int N`, and encode the 32-bit integer `N` into the word at the address of the directive. The expected behaviour is that `.int x` is assembled to `x`, taking up 4 bytes of memory. This is only valid for `x` a 32-bit integer.

`.int` is the only directive you will need to implement. Your assembler will need to be able to parse decimal and hexadecimal values. Labels can be used to store the address of these. This directive will let you store specific values and addresses in the assembly files, from which registers can load.

Example 2.2. Figure 4 shows a short assembly program, and its disassembly, that will load the double-word starting at `my_value` into `x0`

2.3.5 Summary

Table 2 summarizes the syntax the assembler will need to be able to parse.

2.4 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 labels and 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:

Data Processing

add, adds, sub, subs	<Rd|SP>, <Rn|SP>, #<imm>{, lsl #(0|12)} <Rd>, <Rn>, <Rm>{, <shift> #<imm>}
cmp, cmn	<Rn|SP>, #<imm>{, <lsl #(0|12)>} <Rn>, <Rm>{, <shift> #<imm>}
neg, negs	<Rd|SP>, #<imm>{, lsl #(0|12)} <Rd>, <Rm>{, <shift> #<imm>}
and, ands, bic, bics eor, orr, eon, orn	<Rd>, <Rn>, <Rm>{, <shift> #<imm>}
tst	<Rn>, <Rm>{, <shift> #<imm>}
movk, movn, movz	<Rd>, #<imm>{, lsl #<imm>}
mov	<Rd>, <Rn>
mvn	<Rd>, <Rm>{, <shift> #<imm>}
madd, msub	<Rd>, <Rn>, <Rm>, <Ra>
mul, mneg	<Rd>, <Rn>, <Rm>

Branching

b, b.cond	<literal>
br	<Xn>

Loads and Stores

str, ldr	<Rt>, [<Xn|SP>], #<simm> <Rt>, [<Xn|SP>, #<simm>]! <Rt>, [<Xn|SP>{, #<imm>}] <Rt>, [<Xn|SP>, <Xm>]	(Post-index) (Pre-index) (Unsigned Offset)
ldr	<Rt>, <literal>	

Special

nop	
.int	<simm>

Key

<Ri>,	Either an X-register (<Xi>) or aw W-register (<Wi>)
<Xi>,	Registers x0, . . . , x30 and xzr.
<Wi>,	Registers w0, . . . , w30 and wzr.
{ operand }	operand is optional.
<Rn|SP>	A general purpose register <Rn> or the stack pointer SP (optional).
#(0|12)	Either #0 or #12
<imm>	An unsigned immediate value
<simm>	An signed immediate value
<shift>	A type of shift
<literal>	A label or immediate address

Table 2: The Syntax of Instructions

  - An assembly (`.s`) file reader, that is able to send each line to the parser.
  - An instruction parser, that is able to convert a string line into an instruction.
  - 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. for example:
    * A function for assembling Data Processing instructions.
    * A function for assembling Load and Store instructions.
    * A function for assembling Branch instructions.

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

The behaviour of the instructions is recalled in Table 1. Some notes on this table:

- Rd$[x:y] := z$ – In Rd, replace bits $x, x-1, \ldots y$ with z. It is implicit that z is an $x-y$-bit value.
- $*$(address) – is the value of the memory at address address.
- If an instruction 'sets flags', this means that the `PSTATE` fields are updated according to the result of the calculation.

2.4.1 Testing

The test suite provided (Section 5.2) will also help with testing the assembler. Also, see Section B.1 on how to use the AArch64 toolchain to produce correct binaries and listings that you can check your assembler against.

2.4.2 Suggestions

Notice that the symbol table can be used to map labels into addresses and also to map opcode/shifts into the opcodes/shifts themselves.

Parsing The C standard library contains many useful functions for working with strings in `<string.h>`. `strtok_r` can be used to split a string by a set of character delimiters. `strcmp` compares two strings, returning `0` when they are equal. `sscanf` can read formatted input from a string; this will be useful when parsing immediate values.

Compiler Construction An assembler is a type of compiler. A common way compilers are constructed is that the raw strings are parsed to an *internal representation* (IR) of the code that has been parsed, and then the IR translated into a lower level language. You could construct your assembler similarly: parse each line into an internal representation of an instruction, which could be an abstract data type of instructions, and then converting each instruction into its binary encoding according to Section 1.3. You need not have just a single IR – some compilers have a sequence of different IRs that a program is translated into, in order.

One advantage of this structure is it makes debugging easier, as it separates the parsing of instructions from their encoding, which isolates bugs in parsing from those of encoding. Another advantage is that the emulator could reuse this IR by decoding 4-byte words into an IR instruction before emulating it.

Function Pointers C's *function pointers* (similar to higher-order functions in Haskell), are a possible alternative to a big switch statement or series of if statements in the instruction parser. The idea is that, after reading the operation mnemonic, instead of checking its string value or an associated enum value, you could create a table or array mapping the string/enum to a pointer to a function that will be able to parse the rest of the instruction. The advantage of using a table over an array is that the table can use any type of data as a key/index, whereas an array can only use an unsigned integer type for indexing.

Abstraction Abstracting common processes (like sign extension and offset calculation) will greatly simplify your code. You are strongly encouraged look at the ARMv8 Reference Manual (Part C) on what processes can be abstracted, and how different instruction cases can be executed under the same code.

3 Raspberry Pi

The aim of this section is to write an A64 program to repeatedly turn on and of an LED on a Raspberry Pi 3B board, and to compile it with your assembler to run on a physical Pi. The Raspberry Pi 3B is a small computer with an ARMv8-A processor. The SoC is a Broadcom BCM2837 has a quad-core Cortex-A53 CPU, a Broadcom VideoCore IV GPU, 1GB of SDRAM, and various IO ports. Unfortunately, documentation for the BCM2837 is limited. Included on Scientia is an unofficial update to the documentation of the BCM2835 to BCM2837, which was the SoC of the Pi 1. There is also very incomplete documentation of the BCM2836 architecture, which is more similar to the BCM2837. The rest of this section should contain enough information to be able to interact with the onboard LEDs of the Pi.

3.1 Mailboxes

The onboard LEDs are not in direct control of the CPU, they are instead managed by the GPU. In order to control the LEDs, the CPU must then interface with the GPU – this is achieved with the *mailbox* system. This allows the CPU to send and receive messages to/from the GPU, where these messages can be commands or queries about the different parts of hardware managed by the GPU.

Peripherals are hardware devices on the Pi, with a specific address in memory that it can read and/or write from/to. A peripheral is described by an offset from the (Overall) Peripheral Base Address, which is `0x3f000000`. Each peripheral can have *peripheral registers*. A register is a 4 byte piece of memory (i.e. a 32-bit register) with a predefined offset from the peripheral's base address[14]that the peripheral can read/write to. Although these are called peripheral registers, they can be treated as just a space in memory.

The Mailbox peripheral facilitates communication between the CPU and the GPU. As the GPU controls the power LED, the CPU must use mailboxes to control the LED. The mailboxes start at offset `0xb880` from the peripheral base address (so starts at address `0x3f000000+0xb880`). There are two mailboxes: Mailbox 0 (MB0) for `GPUCPU` communication; and Mailbox 1 (MB1) for `CPUGPU`. Switching the LED is achieved through two registers from each mailbox, with offsets from the Mailbox base address:

- Mailbox 0 Registers (`GPUCPU`):
  - MB0 Read Register (offset `0x00`): This facilitates reading messages sent from the GPU. The register will contain the message, with the structure of a message given in Figure 6. More on the structure of messages is explained below.
  - MB0 Status Register (offset `0x18`): This register contains information about MB0. You need only worry about bit 30, the empty flag (`E`), which is set when the read register is empty.
- Mailbox 1 Registers (`CPUGPU`):
  - MB1 Write Register (offset `0x20`) : For sending messages to the GPU.
  - MB1 Status Register (offset `0x38`): This register contains information about MB1. Bit 31 (the full flag, `F`) is set when the write register is full.

Figure 5 shows the structure of these registers, where, for the status register, `E` is the read-empty flag, and `F` is the write-full flag. The CPU should never write to Mailbox 0 or read from Mailbox 1.

Messages As mentioned previously, the CPU communicates back and forth with the mailboxes through *messages*. Messages are four bytes long, and have the following structure: the upper 28 bits are the actual data to be sent; the lower four bits dictate the *channel*, which tells the CPU/GPU what type of message is being sent. Figure 6 shows the general structure of a message, which is used in the read and write registers.

The CPU sends a message to the GPU by writing it into the write register, and receives one from the GPU by reading the four bytes of the read register.

[14]A specific peripheral's base address is its offset plus the (Overall) Peripheral Base Address.

Channels The `channel` component of a message tells the recipient what type of message has been sent. There are nine defined channels for the Mailbox[15]; channel 8 (`0b1000`), called the Property Tags channel, is the channel used to interact with the LED.

Messaging There are general procedures for writing and reading to and from the mailboxes. Both must first ensure the appropriate flag in the corresponding status register is clear. If it is still set, then the program must wait until it is unset, by repeatedly polling the mailbox's status register.

- To read a message from MB0:
  1. Wait for the read-empty flag (`E`) of the MB0 status register to be clear.
  2. Once `E` is zero, load the message from the MB0 read register.
- To send a message to MB1:
  1. Wait for the write-full flag (`F`) of the MB1 status register to be clear.
  2. Once `F` is zero, store the message (which has data in the upper 28 bits, and the channel in the lower 4 bits) in the MB1 write register.

Messages that are sent to MB1 are often called *requests*. Each request will, once enacted upon, generate a *response*; a message that is to be read from MB0. This response will contain information about the outcome of the request.

Mailbox Queues MB1 maintains a queue of request messages, and a queue of their responses is maintained by MB0. Each queue has a limit of less than ten messages. The front of the response queue will be the message in the read register. The request queue increases each time a message is stored in the write register, and decreases once a request has been executed and its response generated. The response queue thus increases each time a request has been executed, and decreases when the response at the front of the queue is read from (by loading from the read register).

The mailbox will only accept a new request if there is space to send its response in the response queue. If many requests are sent without their responses being read, then after a few requests the mailbox will ignore any new requests. To ensure that a request is not ignored, make sure to always read the response after making a request. Remember that reading the response will first require waiting for the read-empty flag of the MB0 status register to be not set.

3.1.1 The Mailbox Property Channel Interface

The 28-bits of data in the mailbox messages are an address of a buffer (location of contiguous data in memory), called the *request*. The request structure is discussed below, and illustrated in Figure 7a. This address must be 16-byte aligned, meaning it ends in `0x0` (in BCD Hexadecimal), as only the upper 28-bits of the address can be sent in the message.

The GPU will send a response by writing data back into the buffer – this means the original buffer is overwritten. For this exercise, the response can be ignored.

[15] See `github.com/raspberrypi/firmware/wiki/Mailboxes` for more information on the other channels.

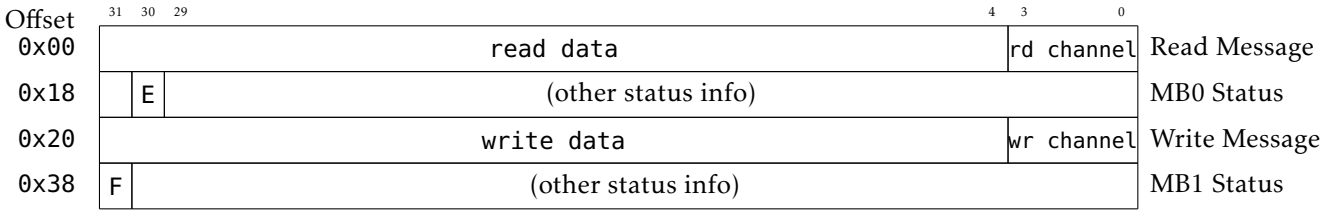


Figure 5: The Structure of the Mailbox Registers

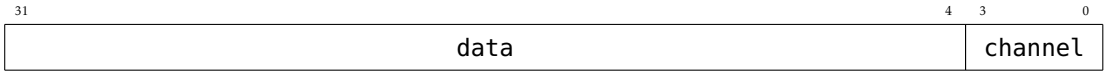
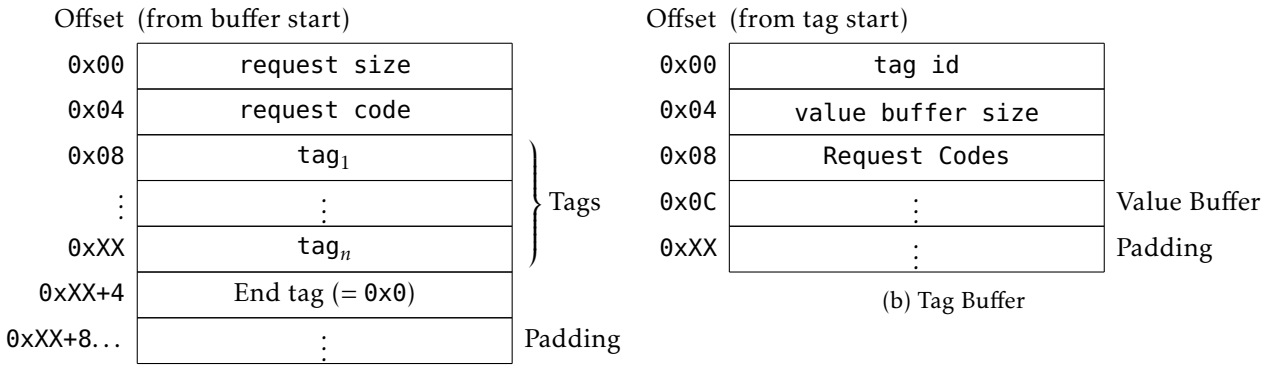


Figure 6: The Structure of a Mailbox Message



(a) Request Buffer

(b) Tag Buffer

Figure 7: Buffer Structures

Request Buffers A request is a small continuous section of memory containing a series of tags (and their parameters), which request various actions for the GPU to perform. The size of a request can vary, so the first 32-bits of the buffer will determine its size in bytes; this lets the GPU know how large any particular request is. The next 32-bits of the buffer are a request code, which is always `0x00000000`. Following this are the tags, which are themselves 32-bit aligned buffers (see below, and Figure 7b). The end of the message is denoted by a 32-bit `0x0` value, the end tag. The end tag can be followed by padding bytes, in case the response is known to be larger than the request (the response is always truncated to the request's size). This structure is summarised in Figure 7a.

Message Tags Tags are commands that get or set specific data about the hardware that the VideoCore manages. Each tag has a 4-byte tag id, which determines the command to run. The tag starts with its 4-byte id. Tags can have arguments, so the size of a tag is variable, and thus is a buffer. The next part of the tag is then a 4-byte value determining the size of the rest of the tag (including padding), called the value buffer. This size should be large enough to include all the parameters of the command, and the size of the result associated with that command. The value buffer itself is a series of single-byte values, which must be padded to align the end of the buffer to 4 bytes. This structure is summarised in Figure 7b.

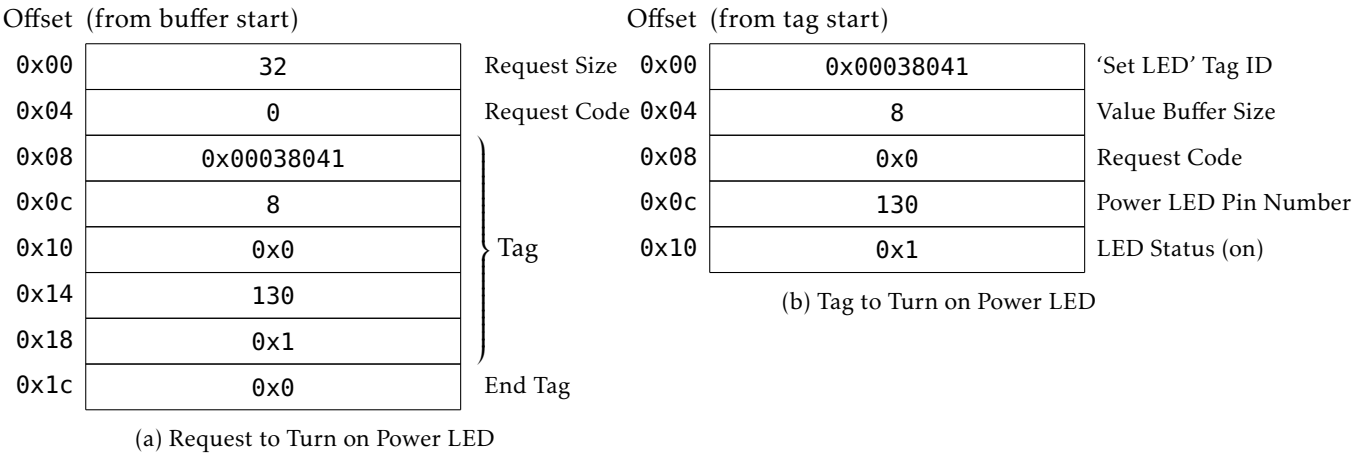
Finally, there is the tag for querying and controlling the onboard LED[16].

Set Onboard LED The tag id to turn on or off one of the onboard LEDs is `0x00038041`. This tag has two 32-bit parameters, which are put, in order, in the value buffer (offset `0x0c`) of the tag buffer. The first parameter is the pin number of the LED to set; 130 (in decimal) refers to the power LED[17]. The next parameter is the value the LED status should be set to: `0x0` is off, `0x1` is on. The response will change the status parameter to the LED's new status.

The Request Structure Putting this all together, the structure of the request to turn the LED on is given in 8. The request to turn the LED off is exactly the same, except the LED Status value (offset `0x38` in the

[16] Tags to interact with other parts of the hardware can be found at `github.com/raspberrypi/firmware/wiki/Mailbox-property-interface`

[17] 42 is for the ACT LED



(a) Request to Turn on Power LED

(b) Tag to Turn on Power LED

Figure 8: Turn LED On Request Structure

request buffer, and offset `0x10` in the tag buffer) will be `0x0`, instead of `0x1`. The response will write over the tag's parameters, which are at offset `0x14` and `0x18` in the request buffer.

3.2 What to Do

3.2.1 Setting up the Pi

To set up the raspberry pi to run your assembled binary, you will need to follow these steps. Use the files `start.elf`[18] and `bootcode.bin`[19], which you should download and save onto the SD card. The binary your assembler produces should be called `kernel8.img`, as the Pi executes this file on startup. If your assembly program to blink the led is called `led_blink.s`, then you would use your assembler as follows:

```
./assemble led_blink.s kernel8.img
```

Your `kernel8.img` should then also be saved onto the SD card. Each of `start.elf`, `bootcode.bin` and `kernel8.img` should be in the base directory of the SD card. You won't need any other files.

3.2.2 Blinking the LED

You need to write an assembly program in A64 that will make the Pi's onboard (green) power LED blink. This program should, in an endless loop:

- Send a message to MB1, containing the address of a request buffer with the tag to the power LED on.
- Wait for a fixed amount of time.
- Send another message to MB1, with the address of a request buffer asking to turn the power LED off.
- Wait again for some time.

Remember, the Mailbox has a limited queue of responses that are put in the read register. If these responses aren't read then the queue will fill up, and no more requests will be accepted by MB1 until the queue is reduced. Also remember that when writing or reading a message from the mailboxes, you must first wait until the write-full or read-empty bits of the status registers are cleared.

[18] github.com/raspberrypi/firmware/blob/master/boot/start.elf
[19] github.com/raspberrypi/firmware/blob/master/boot/bootcode.bin

The Request Buffers The structure of the requests to turn on and off should follow the guide earlier in this section. The start address of the buffer must be 16-byte aligned. An important thing to remember is that the response will write over this buffer, possibly changing its meaning. If this same request buffer is used again, you should ensure it has the correct values. One way to achieve this would be to have a template request buffer in the assembly file, code that copies it to a fixed location with space for a message (the end of the assembly file is a good idea!), and then using this copied buffer as the 'current' request buffer. The message should be copied to the same location each time; copying to a different location each time will exhaust the Pi's memory.

Request Address `kernel8.img`, produced by your assembler, is loaded onto the Pi starting at address `0x80000`. As you will have to load the address of your buffer into a register, remember that this value must include the `0x80000` offset. For example, if your buffer is at address `0xc0` relative to the start of the assembly file, then the address that should be loaded into a register (and then stored in the mailbox write register) will be `0x80000` + `0xc0`.

To summarise:

- Make sure to poll the MB0 status register before using the read, and the MB1 status register before using the write register.
- Read responses to your requests, otherwise MB1 will stop accepting requests.
- The request buffer must start at a 16-byte aligned memory address.
- The responses will overwrite the requests – make sure to rewrite the requests to ensure they have the correct structure.

3.2.3 Suggestions

Mailbox Emulation You do not need to emulate the Mailbox. However, you may find it useful to partially emulate it to help you debug your `.s` file to make the LED blink. To check that loads/stores are being performed at the correct address, you could implement a way to have a block of memory starting at `0x3f000000`, so that your emulator can check reads and writes from here. Instead of allocating a large block of memory (`0x0` to `0x3f000000` would be more than a gigabyte!), you could abstract reads and writes to memory, and only store smaller blocks of memory that will be needed. Bear in mind that on the Pi, the binary will run starting from address `0x80000`, so you might want to let your emulator have the option to also start from this address. This is entirely optional.

No `mov`? In A64, `mov` is a pseudo-instruction that aliases one of many different instructions depending on the type, size and values of the operands. Although you have implemented `mov` with two register operands, the cases for an immediate operand are annoying and difficult to check, so you have not been asked to implement them. If you want to achieve similar functionality to `mov`, you can use one of the following patterns depending on the type of operand:

- 12-bit (Shifted) Immediate: To set Rd to a 12-bit immediate shifted to an arbitrary amount, perform

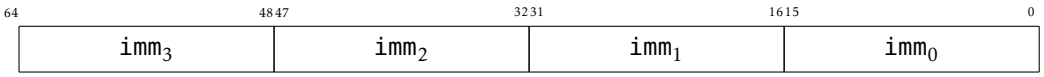
  ```
  add xd, xzr, <imm12>, lsl #<amount1>
  orr xd, xzr, xd, lsl #<amount2>
  ```

  This will perform Rd := (`imm12` « `amount1` + `amount2`). The last instruction could also be an `add`.

- 64-bit Immediate to Register (with `ldr`): To load a 64-bit immediate to a register, you can create a label followed by two `.int` directives, and then `ldr` from that label.

```
ldr xd, my_imm
⋮
my_imm:
    .int <lower_word>
    .int <upper_word>
```

- 64-bit Immediate to Register (wide moves): Finally, a 64-bit immediate can be loaded by a series of wide moves. If `imm` is the immediate to load, and has the following value:



  then the following code achieves Rd := `imm`,

```
movz xd, imm3, lsl #48
movk xd, imm2, lsl #32
movk xd, imm1, lsl #16
movk xd, imm0
```

Immediate Logic You have not been asked to implement bit-logic instructions with an immediate operand; this is due to the complicated way that immediates are encoded. To test a register Rn against an immediate value, you can instead first load the immediate into a register Rm (using the move techniques above), and then using the bit-logic instruction with register operands Rn and Rm.

4 A LaTeX-documented extension

Once you have implemented your emulator and assembler, you should design, implement and document an extension for your project using C as your main programming language. The implementation may involve changes to your assembler and emulator, but make sure that the resulting program is backwards-compatible with the ARM specification provided for the exercise. Alternatively, it could be a separate tool (e.g. a debugger, visualiser, high-level compiler, etc.). If you would like to base your extension on Part III, you can either use the actual Raspberry Pi from your kit, or any equivalent software-based emulators (such as Qemu[20] or Unicorn[21]) and write a program (mainly in C) 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. If you are out of ideas, the suggested "theme" for the extension of this academic year is:

"**Use technology to help the world heal and unite**"

Remember that this extension should use the C programming language. Even though you are allowed to use additional languages as helpers, these should **not** be the central point of your extension.

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

- Extend the instruction set with more A64 instructions, for example software interrupts, exception levels.
- Look into implementing a similar instruction set in ARM, the instruction set of the 32-bit mode architecture, AArch32. You could even implement how the CPU switches between these execution modes.
- Interact with other parts of the Pi's hardware. For example, switching LEDs on/off on a connected breadboard via the GPIO pins, or controlling other parts of the Pi with different mailbox messages.
- 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 A64 instruction set in order to extend your implementation of the Branch instruction to include Branch With Link.
- Extend the assembler to allow comments in `.s` files. Comments are C-style, so can be block comments `/* */`, or line comments `//`. In block comments, everything between `/*` and `*/` should be ignored. For line comments, everything after `//` up until the next line should be ignored. The difficulty will be when a `*/` appears after a `//`, for example `/* erm // */`. In this case, this should be treated as a single block comment, and `*/` should end the block.
- Use the Pi to implement something innovative based on the suggested theme. Remember the use of C for your extension.

[20] qemu.org/
[21] unicorn-engine.org/

5 Working and Submission

There are two important dates: Friday 09/06/2023 and Friday 23/06/2023. All deadline times have been set to 19:00 BST.
By Fri 09/06 19:00 you will need to submit:

- Your Interim Checkpoint Report (PDF)
- Your First Peer Assessment Groupwork Feedback

By Fri 23/06 19:00 you will need to submit:

- Your Final Report (PDF)
- Your Final Master Branch Repo Commit (hash via LabTS)
- Your Second Peer Assessment Groupwork Feedback
- Your Presentation Slides (PDF) and Video URL (TXT) pointing to your video that has been uploaded on Panopto *before the deadline*

5.1 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/lab2223_summer/armv8_<groupno>.git
```

where *groupno* is your 2-digit group number. You can find out your group number by visiting gitlab, or the COMP40009 ARMv8 website:

```
https://www.doc.ic.ac.uk/~kgk/armv8/_files/armv8groups2223.pdf
```

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**. We will be reading these.

When you are ready to submit your project, **the Group Leader** should log into the gitlab website `gitlab.doc.ic.ac.uk`, and click through to your group's `armv8_<groupno>` repository. The leader should view the `Commits` tab, find the list of the different versions of your work that you have pushed, and make sure that everything has been merged into the master branch. The leader should then go on LabTS, select the right commit of your code and ensure that it is working before submitting it to Scientia. 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 Scientia. Please note that the leader **will also need to add your PDF report** to your group's Scientia submission before the deadline; otherwise, the submission is marked as incomplete.

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–II. There is also a skeleton `Makefile` which you may need to edit as you work on Parts I–II. To aid with testing, you should ensure that invoking `make` in this directory builds your `assemble` and `emulate` binaries.

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 III should reside in this directory in a file named `led_blink.s`.

extension/
: You will *not* find this folder in your skeleton, but you should create it if you are going to submit an Extension with your project. This is where your Extension goes. You can create and arrange its subfolders at will.

.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.[22] 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`.

5.2 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/kgk/armv8_testsuite.git
```

Very briefly, you can set up the test suite with `./install`, and run with `./run`. More information can be found in the `README.md` and below.

Requirements Python 3.10.6 (this is the version on the lab machines). The test suite should work for any python version above 3.8 (inclusive), but it is only guaranteed to work for 3.10.6. If your python version is different, you have a few options:

- Your version might work anyway!
- Install the correct version (and optionally replace your version, or use `pipenv` – see below).
- Use `pyenv`, which manages different python versions on the same system

The python library dependencies are:

- `Flask`: a web framework, for the front-end of the test-suite (`flask.palletsprojects.com`).
- `result`: A simple `Result` type, which represents either success (`Ok`) or failure (`Err`). Based on Rust's `Result` type, similar to Haskell's `Either` type (`pypi.org/project/result`).
- `colorama`: For colours in the terminal output of the tests (`pypi.org/project/colorama`).

You can install these to your system python manually (with `pyrhon3 -m pip install <library>`), or using the `requirements.txt` file in the test suite folder, using `python3 -m pip install -r requirements.txt`. If you don't want to have these installed to your system python, the test suite includes a `Pipfile`, which is used by `pipenv` to create an isolated python environment. `pipenv` is a a dependency management tool for python (`pipenv.pypa.io`). If you install and use `pipenv`, then you can run:

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

```
python3 -m pipenv install
```

to install all the requirements.

Running the Test Suite Clone the `testsuite` from GitLab. The easiest way to run the test suite is with the `run` script in the main directory of the test suite:

```
./run <args>
```

To instead run the test suite with your system python, use:

```
python3 run.py <args>
```

To instead run with `pipenv`, use

```
python3 -m pipenv run python3 run.py <args>
```

The suite will then run the tests according to the arguments supplied. This will produce the output of your assembler/emulator under `testsuite/actual_results`, and (optionally) print a result summary to the console.

Web Front-end The script `testserver.py` can be used to run a local web server that lets you interactively run the tests; however, its functionality is quite limited (at the moment).

Once you have run `python3 testserver.py`, the suite should print out a url to `stdout` that you should be able to click on or copy/paste into your browser. `run` will accept command-line arguments – see what these are by running `./run -h`.

File Organisation Your `assemble` and `emulate` executables should be in the `/testsuite/solution` directory. The tests and expected outputs are stored in `testsuite/test_cases` and `testsuite/expected_results` (respectively) – ensure these folders are present.

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 (ending in `.bin`), 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.

When a test fails (either assembly or emulation), a diff file (ending in `.diff`) is produced to quickly show you where the results were different from expected. If the assembler produces an incorrect binary or is not run, the emulator uses the expected binary, so that you can test the two parts in isolation. If you have the arm toolchain installed, the test suite will produce listings of your actual outputs as well (see Section B.2).

The test-suite allows you to run just the assembler or emulator, and on specific files or files matching a glob string[23].

If you wish to add your own test cases, or use the testing scripts directly (for example on the command-line), see Section B.2.

Install/Run Scripts To save you effort, you can install requirements by running `./install`, and then run the test suite with `./run <args>`. The `install` script follows the `pipenv` set-up outlined above: it installs `pipenv` for you, and, if your system doesn't have a python version compatible with the project, the script will also install `pyenv` for you.

[23] wikipedia.org/wiki/Glob_(programming)

6 Assessment

This exercise forms part of the assessment of the COMP40009 C Programming submodule, whose 20% marks are distributed as follows:

- 18% comes from:
  - C Project Interim Checkpoint: 1%
  - C Project Group Feedback: 1%
  - C Final Test: 12%
  - C Project Final Report/Source: 3%
  - C Project Presentation: 1%
- 2% comes from:
  - Extension (for Computing)
  - Architecture (for JMC)

The marking will focus on the following categories:

- Design, implementation, style and readability: 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: 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 (for Computing groups).
- Group working and communication: this will be judged on 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: 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!

6.1 Checkpoint: Due on 09/06/2023

On or before Friday 9 June 2023 at 19:00, you should submit on Scientia under COMP40009 a PDF not more than one sheet of paper, i.e. 2x A4 pages maximum, containing a summary that outlines your group working and implementation of the Emulator (Part I) only. This report must be in LaTeX (*this is assessed*) and include:

- A statement on how you have split the work between group members and how you are coordinating 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 have 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 Friday 09/06/2023 19:00, through the Peer Assessment system at https://peer-assessment.doc.ic.ac.uk/. You will need to complete another peer-assessment on or before Friday 23/06/2023 19:00.

During the Week 7 session, your Group Mentor will look at your progress and discuss your report and your Peer Assessment feedback. This will be an assessed meeting so attendance of all members is mandatory.

6.2 Report, Assembler and Emulator Source Code, Extension Source Code, Presentation Slides and Video: Due on Friday 23 June 2023 19:00

You should complete your Emulator, Assembler, LED/Mailbox program, and chosen Extension, and then write up your work in a report. This Final Report must be written in LaTeX and submitted as a PDF to Scientia manually, along with your code (*which needs to be submitted via LabTS*).

Ensure that you have pushed your repository with all code (and sources to your report) back to Gitlab, merged into your master branch, and that you have pressed the Submit to Scientia button for the right commit.

You should write a short 3-sheet (i.e. maximum 6x A4 pages) LaTeX final report documenting your project (and its Extension, for Computing groups), which must contain:

- How you have structured and implemented your Assembler. (Please note that the Emulator has already been described in the Interim Checkpoint document and should not be re-added here).
- How you have implemented Part III, i.e. making the green LED blink, on the provided Raspberry Pi.
- 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 your first Peer Assessment 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.

Along with your report, you should also prepare a pre-recorded 15 minute video of your group presenting your project and extension. Sticking to the 15 minute duration is an assessed aspect of your presentation. The presentation slides that you used for your talk should be submitted to Scientia in PDF format. You should also submit a URL.txt file with the address of your video. You will be uploading your presentation video on Panopto, under the Assignment folder of COMP40009, here: https://imperial.cloud.panopto.eu/Panopto/Pages/Sessions/List.aspx#folderID=4fcb49d8-a808-417f-a7dd-b00600bf2c08, adding "Group ##" in the title. On Week 9, you will have your final assessed meeting with your Mentor, which will involve a brief Q&A session on your submitted presentation and project. Your presentation should discuss your entire project and extension, and all group members must participate and present. The 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 (approx. 6 minutes).
- An overview and demonstration of your Extension (where applicable). Also include any testing of your extension that you undertook (approx. 6 minutes). JMC groups without an Extension may use up to 12 minutes for their first part.
- 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? (approx. 3 minutes)

7 Competitions and Prizes

There are 5 group prizes + 1 individual prize on offer, awarded for:

- The ARMv8 22/23 "Most Helpful Student on EdStem", awarded to one student for successfully and **constructively** answering the most questions on the EdStem COMP40009 forum during the Summer Term Group Project.
- The ARMv8 22/23 "Best Group Reflection", as contained in the group's Final Report.
- The ARMv8 22/23 "Best Project Presentation", based on the Panopto video submission.
- The ARMv8 22/23 "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 on any output generated (*where applicable*). Groups without an Extension will not be participating under this category.
- The ARMv8 22/23 "Best Overall Project", as judged by a combination of the overall quality of the code, the Final Report and the Presentation.

Appendices

A Examples

A.1 Load Sizes

Example A.1. If the memory layout is as in Figure 9a, and loads from address `0x1C2` into X0 and W1 are executed, the registers R0 and R1 will have layouts:

63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0	
Byte 10	Byte 9	Byte 8	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	R0
`0x00`	`0x00`	`0x00`	`0x00`	Byte 6	Byte 5	Byte 4	Byte 3	R1

If a store at address `0x1C0` is performed from X1, the memory layout will be that in Figure 9b; notice that it stores the entire double-word into memory. If instead a store at address `0x1C0` is performed from W1, the memory layout will now be that in Figure 9c; this time it only stores the lower word into memory.

A.2 W-& X-Registers

Example A.2. Assume that the value of R0 is `0x0`, `0x0000 1000 0010 0000` is the current value of R1, and that `0x1000 0000` is the current value of R2. Then executing the following instruction,

```
add x0, x1, x2
```

will set the value of R0 to `0x0000 1000 1010 0000`. If instead, this next instruction is executed,

```
add w0, w1, w2
```

then the value of R0 will be `0x0000 0000 1010 0000`. This is because the reads from W1 and W2 will give only the lower 32-bits of R1 and R2.

	(a)	(b)	(c)
`0x1C0`	Byte 1	Byte 3	Byte 3
	Byte 2	Byte 4	Byte 4
	Byte 3	Byte 5	Byte 5
	Byte 4	Byte 6	Byte 6
	Byte 5	`0x00`	Byte 5
	Byte 6	`0x00`	Byte 6
	Byte 7	`0x00`	Byte 7
	Byte 8	`0x00`	Byte 8
	Byte 9	Byte 9	Byte 9
`0x1CA`	Byte 10	Byte 10	Byte 10

Figure 9: Memory Layouts

B Testing

B.1 Using the ARM Toolchain

You could also use the `aarch64-none-elf` toolchain to generate correct binaries to test your emulator with, it can be downloaded from `developer.arm.com/downloads/-/ arm-gnu-toolchain-downloads`. The exact one to download will be under the section corresponding to your machine (e.g. linux x86_64), and then the subsection 'AArch64 bare-metal target (aarch64-none-elf)'.

Linux and WSL users can add the commands from this toolchain to the terminal by unpacking the download `arm-gnu-toolchain-12.2.rel1-x86_64-aarch64-none-elf.tar.xz`[24] to a folder `<folder>`, and then adding `<folder>/bin` to the terminal `PATH`. macOS users should be able to achieve a similar effect with `arm-gnu-toolchain-12.2.rel1-darwin-arm64-aarch64-none-elf.pkg`.

The tools that will be particularly useful are:

- `aarch64-none-elf-as`: The reference assembler.
- `aarch64-none-elf-objcopy`: This lets you turn assembled object files into raw binary (using the `-O binary` flag), stripping the ELF header information and symbol tables, leaving only the instructions and data.
- `aarch64-none-elf-objdump`: This lets you produce a listing of the assembled file (using `-d` or `-D`), making it easy to read the output of the reference assembler. The `-z` flag will mean the listing doesn't skip any blocks of zeroes. Using the flags `-b binary -m aarch64 -D` will let you produce a listing from a raw binary. This will provide a more visual way to compare the output of your assembler with that of the official one, as you can directly compare the expected and actual listings.

The binary produced by performing the `-as` and `-objcopy` tools as above on a `.s` file should be runnable with your emulator. This will make it easier for you to produce extra test files for your emulator without relying on your assembler.

B.2 The Testing Suite

Adding Test Cases If you wish to add test cases, you just need to put the `.s` file in the `test_cases` folder, and the corresponding `.bin` and `.out` files into the `expected_results` folder. You can produce the `.bin` using the AArch64 toolchain (see Section B.1). To produce the `.out` file, you can either work out the expected output by hand, or look into using the `unicorn` or `qemu` emulators.

Listings If you have the arm toolchain installed (see Section B.1), then the testing server will also produce listings of your assemblers output. If the toolchain is on the system `PATH`, then the script will use that one. If it is not on the `PATH`, you can supply the command-line argument `-toolchain <path_to_bin>`, which will be the path to the `bin` folder of the arm toolchain.

Testing Scripts The test-suite front-end is separate from the scripts that run your tests. If you wish to avoid the front-end and directly use the scripts (for example, in a Makefile or devops), then you can use the `run_tests.py` file. The script also accepts glob arguments. Use `pipenv run python3 test/run_tests.py -h` for the options available on the command-line. Some arguments to the script you might find particularly useful:

- `--expdir` – directory to to find expected results (defaults to 'test/exp').
- `--actdir` – directory to place actual results (defaults to 'out').

[24] The version numbers may be subject to change, but as long as it ends with `-aarch64-none-elf.tar.xz`, you should be fine.

- `-A` – only run assembler tests
- `-E` – only run emulator tests
- `-p` – pretty print results to ‘stdout‘
- `-f` – only report failures.

For example, `pipenv run python3 test/run_tests.py -Apf test/gen/br/*.s` will run the assembler tests on the `.s` files under the folder `test/gen/br/`, and will report any failures to the console. The results are collected and saved in a `json` file, by default this is `out.json` under the `out` folder.