Convert $0xF0$ to decimal.

1

Convert 18 to binary using 2's complement.

2

Where do the values that are fed into an operand come from when an instruction is executed using the three address style?

3

Why are registers faster than memory?

4

What is the one-address style of instruction?

5

Describe the load-store style of instruction.

6

How is code written in ARM Assembly run?

7

What is the instruction to load a value at a memory address into a register?

8

What is the instruction to store a value in a register to a memory address?

9

What is the instruction to sum two numbers?

10

What is the instruction to branch upon a condition?

11

What does the program counter do? What register is it?

12

Memory.

The resulting value is also copied to a destination address in memory.

3

010010

2

240

1

The only operations on memory are load and store operations. This means each instruction is very fast and very simple, but there are many instructions.

6

Where only one memory address may be used in any one instruction. The other operands must be registers.

5

- *Implemented using a flip flop or some other very fast volatile storage (rather than smaller, cheaper SRAM).*
- *Situated inside the processor, so there's less distance for the data to travel, which takes less time.*
- *Fewer of them so address decoding takes less time*
- *Data doesn't need to be transferred over a bus.*

4

`STR <register> <memory_address_alias>`

9

`LDR <register> <memory_address_alias>`

8

It is first assembled using an assembler into machine code. Then it is loaded into memory and executed sequentially.

7

It is used to store the memory address of the next instruction to be executed.

It's register 15.

12

`B <condition> <branch_name>`

11

`ADD <destination_register> <operand_register1> <operand_register2>`

10

When is the `DEFW` *command executed. What does it do? What's its syntax?*

13

What does `DEFB` *do?*

14

What's the syntax of DEFS?

15

What does `STRB` *do? What is it's syntax?*

16

What does `LDRB` *do? What is it's syntax?*

17

When using Little Endian, bytes are read from [redacted] *to* [redacted].

18

When using Big Endian, bytes are read from [redacted] *to* [redacted].

19

How many bits are assigned to a literal in an ARM instruction

20

What command should be used to load a literal into a register

21

How do you load an address into a register so you can use it as a pointer?

22

What does `DEFW <number>` *do?*

23

What does `DEFB <value>` *do?*

24

`<alias> DEFS <number_of_bytes>, <value_of_bytes>`

15

`DEFB` *stores a single byte in memory. If you give it a string, the whole string will be stored (in multiple bytes).*

14

It is executed before the program runs.

It stores a value at a memory address and assigns the address an alias.

`<alias> DEFW <value_1>, ... <value_n>`

13

When using Little Endian, bytes are read from left to right.

18

Loads the lowest eight bits of a specific memory address into a register. The other bits in the register are set to zero.

`LDRB <register> <memory_alias>`

17

Stores the lowest eight bits of a register into memory.

`STRB <register> <memory_alias>`

16

`LDR <register> =<literal>`

Note, this is a pseudo instruction, that is converted to either `MOV <register> #<literal>` *or it will define a constant and load that in from memory.*

21

12

20

When using Big Endian, bytes are read from right to left.

19

It reserves a byte(s) of memory with the value `<value>`*. If a string is passed as the value, multiple bytes will be reserved, each with the value of a character.*

24

It reserves a word of memory initialised to `<number>`

23

`ADR <register>, <alias>` *Now the* `<register>` *will hold the memory location (and is therefore a pointer to) the alias.*

22

What does `DEFS <size>, <fill>` *do?*

25

What does `ALIGN` *do?*

26

What does `ENTRY` *do?*

27

What does `EQU` *do?*

28

What are the four status flags provided by the ARM architecture?

29

How can you combine a CMP instruction with another instruction?

30

What does `RSB` *do?*

31

What does `MLA` *do?*

32

Condition code	*Meaning (for cmp or subs)*
eq	
ne	
ge	
le	
lt	
gt	

33

In the following instruction, what method of addressing is used, what will `R0` *contain and what will happen to* `R1`*?*

```
LDR R0, [R1]
```

34

In the following instruction, what method of addressing is used, what will `R0` *contain and what will happen to* `R1`*?*

```
LDR R0, [R1, #4]
```

35

In the following instruction, what method of addressing is used, what will `R0` *contain and what will happen to* `R1`*?*

```
LDR R0, [R1, #4]!
```

36

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

Leaves blank bytes in memory so that the next `DEFINE` *command will start on a word boundary.*

It reserves a block of memory `<size>` *bytes long initialised to the value* `<fill>`.

Append `S` *to an instruction. E.g.:*
`SUBS R0, R1, R2`
If the result in `R0` *was negative, then the negative flag would be set etc etc.*

- *Negative*
- *Zero*
- *Carry*
- *Overflow*

`EQU` *allows us to name a literal. You can then refer to the literal (still with a hash before it) by name in your code which is easy to read.*

`<alias> EQU #<value>`

Condition code	*Meaning (for cmp or subs)*
eq	*Equal*
ne	*Not equal*
ge	*Signed greater than or equal*
le	*Signed less than or equal*
lt	*Signed less than*
gt	*Signed greater than*

Multiply and add. `MLA R0, R1, R2, R3`*:*
`R0 = (R0 * R1) + R2`

Reverse subtract. `RSB R1, R0, #0`*:*
`R1 = 0 - R0 = -R0`

This is called **pre-indexed autoindexed addressing**.

The value loaded into `R0` *will be the 32 bits stored at the memory address that is equal to the value in* `R1 + 4`.

`R1` *will be incremented by* `4` *before the load operation.*

This is called **pre-indexed addressing**.

The value loaded into `R0` *will be the 32 bits stored at the memory address that is equal to the value in* `R1 + 4`.

`R1` *won't be altered at all.*

This is called **register-indirect addressing**.

The value loaded into `R0` *will be the 32 bits stored at the memory address that is equal to the value in* `R1`.

`R1` *won't be altered at all.*

In the following instruction, what method of addressing is used, what will `R0` *contain and what will happen to* `R1`*?*

```
LDR R0, [R1], #4
```

37

In the following instruction, what method of addressing is used, what will `R0` *contain and what will happen to* `R1` *and* `R2`*?*

```
LDR R0, [R1, R2]
```

38

In the following instruction, what method of addressing is used, what will `R0` *contain and what will happen to* `R1` *and* `R2`*?*

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

39

How do you load a String into a register?

40

What does the `ADRL` *instruction do?*

41

How do you find the length of a string defined by the alias `message`*?*

42

What are the four ARM bit shifting/rotation instructions?

43

What is the syntax of a shifting or rotation operation in ARM?

44

How can `LSL` *be used to load words from a table/array in memory?*

45

What is the command to push something to the stack?

46

What is the command to pop something off the stack?

47

What is a different way of accessing the top value of the stack rather than using `POP`*?*

48

This is called **scaled register-indexed addressing**.

The value loaded into `R0` *will be the 32 bits stored at the memory address that is equal to the value in* `R1 + (R2 * 4)`.

`R1` *and* `R2` *will stay the same.*

39

This is called **register-indexed addressing**.

The value loaded into `R0` *will be the 32 bits stored at the memory address that is equal to the value in* `R1 + R2`.

`R1` *and* `R2` *will stay the same.*

38

This is called **post-indexed autoindexed addressing**.

The value loaded into `R0` *will be the 32 bits stored at the memory address that is equal to the value in* `R1 + 4`.

`R1` *will be incremented by* `4` *after the load operation.*

37

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

42

It is a psudo instruction that loads a program relative address into the register. It is compiled into two `ADD` *instructions.*

41

```
msg DEFB "Hello"
ALIGN
ADRL R0, msg
SVC 3
```

40

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

45

```
INSTRUCTION destination operand (#)shift
```

44

Mnemonic	Meaning
`LSL`	*Logical shift left*
`LSR`	*Logical shift right*
`ASR`	*Arithmetic shift right*
`ROR`	*Rotate Right*

43

```
LDR R1, [SP], #4
```

48

```
POP <register/literal>
```

47

```
PUSH <register/literal>
```

46

What is a different way of adding a new element to the stack rather than using the `PUSH` *command?*

49

How do you push or pop multiple elements from the stack without using the `PUSH` *or* `POP` *commands?*

50

What does the `BL` *command do?*

51

After a method called using the `BL` *command has finished executing, what command does it use to tell the processor go back to what it was doing?*

52

If a method is using registers as temporary stores, what should it do before and after it executes? What instructions should it use to do this?

53

How do you pass a parameter to a variable?

54

How should a method access stacked parameters?

55

What is a stack frame?

56

How do you convert a Java switch statement into ARM Assembly?

57

What are the ARM comparison operators for unsigned integers?

58

How is a boolean represented in ARM assembly?

59

How do you test if an ARM boolean is true or false?

60

- Moves the current value of the program counter (PC) into the link register (LR).
- Branch to the label defined in the instruction by moving the memory address of that instruction into the PC.


```
STMFD SP!, R0, R1
LDMFD SP!, R0, R1
```



```
STR R1, [SP, #-4]!
```


Either put it in a register (this is the stupid way), or put it on the stack (this is a good way since you can pass lots of paramaters like this.)

It should `PUSH` *the value of the registers to the stack and then* `POP` *them again after:*

```
methodname
PUSH {registers_used}
; Do stuff
POP {registers_used}
MOV PC, LR
```



```
MOV PC, LR
```


1. Create a table with the values triggered by the case statement as the index of the table.
2. Load the address of the table into a register (`ADR R1, table`*)*
3. Get the value of the case variable in a register (we'll use `R0`*).*
3. Use the `LDR` *command to load the correct method to call (*`LDR PC, [R1, R0, LSL, #2]`*)*
4. Make sure you've got a default case
5. Make sure you branch to the end of the case statement after each method.

A stack frame is a set of values on the stack that relate to a single method. They may contain saved registers, temporary values used by the method, a pointer to the parent method etc.

Just read the stack using the `STR` *command and add an offset depending on what parameter you want. E.g:*
`LDR R0, [SP, #12]`

`CMP R1, #0`
If it's equal, then it's false.

As an 8-bit set of values that is either `00000000` *or* `00000001`*.*

`HI`	*Higher*
`HS`	*Higher or same*
`LO`	*Lower*
`LS`	*Lower or same*

What are the logical operators that ARM has?

61

What is the syntax of an ARM logical operation such as `EOR`*?*

62

Fill in the truth table:

A	B	BIC A, B
0	*0*	
0	*1*	
1	*0*	
1	*1*	

63

What are the two methods of interfacing with peripherals?

64

When the CPU is polling a peripheral, what does the poll actually do?

65

Define 'memory mapped register'

66

What does `TST` *do?*

67

What is the syntax for the `TST` *instruction?*

68

What happens when an interrupt occurs?

69

Write some sample code for checking a status register against a pattern.

70

What is an interrupt vector table?

71

What does `SVC` *stand for?*

72

Logical AND `AND`
Logical OR `ORR`
Logical XOR `EOR`
Bit clear `BIC`
Logical NOT `MVN`

61

- Polling
- Interrupts

64

It performs a bitwise `AND` *on it's operands and then compares it to zero, setting the comparison flags as it does so.*

67

```
ADR R1, status_reg
LDR R0, [R1]
TST R0, 0x40
BEQ correct
B incorrect
```

70

A table of pointers to various methods for handeling different peripherals. It's implemented in the same way as a case statement is.

71

SuperVisor Call.

72

`EOR <reg_1>, <reg_2>`

62

- Read the status register
- Use the `TST` *command to check the value of the status register against a bit pattern*
- If the test passes, then run a method to handle what to do next.

65

`TST <register>, <pattern>`

68

1. Stop the program execution (like a `BL`*)*
2. Save important registers by stacking them (including the CSPR)
3. Run the interrupt handler
4. Restore the saved registers from the stack
5. Copy the LR into the PC (and add 4) so the next instruction is exected.

Remember you can use `STMFD` *and* `LDMFD` *to efficiently manage the stack*

69

A memory location that appears to be an actual memory location, but is actually situated inside a peripheral (so it is mapped to a different address).

66

Fill in the truth table:

A	B	BIC A, B
0	*0*	*0*
0	*1*	*0*
1	*0*	*1*
1	*1*	*0*

63

How do we get the parameter from `SVC X`*?*

73

What is direct memory access?

74

List advantages and disadvantages of DMA.

75

What does the kernel do?

76

What is a kernel mode?

77

What are the five steps of compilation?

78

What are the pros and cons of an interpreted language?

79

Why does Java use zero address and one address instructions?

80

Code that is written using zero or one address instructions is called a [redacted].

81

What would a zero address `ADD` *instruction do?*

82

How would $x = (a + b) * c$ *by represented in a stack machine?*

83

What does Dynamic Class Loading do?

84

Advantages	Disadvantages
·*The load on the CPU is lessened.*	·*The load on the memory and the peripheral is the same as before.* ·*The motherboard will be more comlplicated with the extra hardware.*

75

A hardware chip on the motherboard that allowd peripherals to write diretly to memory rather than going through the CPU first.

74

- Load the `SVC X` *instruction into a register with* `LDR R0, [LR, #-4]`
- Clear the top 8 bits (`BIC R0, R0, 0xFF000000`*)*
- Logic shift right by 8 places to get the parameter (`LSR R1, R0, #8`*).*

73

1. *Lexical analysis*
2. *Syntaxic analysis*
3. *Semantic analysis*
4. *Code generation*
5. *Optimisation*

78

A kernel mode conveys privaleges to a program. The default (user) mode only allows memory locations within the allocated space to be accessed. Other modes convey different privilages, e.g. the privelaged mode which allows any address to be accessed.

77

It abstracts the implementation of the hardware from the running programs. It also protects programs from eachother by letting them only access memory locations that have been assigned to them.

76

Code that is written using zero or one address instructions is called a stack machine.

81

So that the bytecode is very small in size and can be quickly transferred over the internet.

80

Advantages	Disadvantages
·*Simple to implement* ·*Portable code* ·*Increased security (since applications can be sand-boxed).* ·*Ease of debugging since the state is within the interpreter*	·*Slow.* ·*Hard to simulate a real computer on them.*

79

Only loads java classes just before they are executed to minimise memory useage and improve loading times of programs.

84

```
PUSH a
PUSH b
ADD
PUSH c
MUL
POP x
```

83

- Pop the two top values off the stack.
- Add them together.
- Push the result onto the stack again.

82

What does the JIT compiler do?

85

What are the three areas of memory usage in Java programs?

86

Where do Java objects reside in memory?

87

Where does the heap pointer point to?

88

What are the three parts that each item in the heap is composed of?

89

What is a stale object?

90

What are the stages of garbage collection?

91

How do you do array access in ARM?

92

The heap.

87

The class area (containing method code and class variables), stack area (containing the stack, which includes method parameters, local variables and return pointers) and heap area (containing the objects used by the program).

86

It compiles classes into machine code in real time as they are loaded. It uses **dynamic compilation** *to refine it's compilation so that code that is run multiple times is optimised more than code that is run only once.*

85

An object in the heap that is no longer referenced by any pointer.

90

A header (containing information about the object such as it's size), storage for instance variables and a table of method parameters that the object contains.

89

The next free memory location in the heap.

88

```
LDR R0, index
LDR R1, basePointer
LDR R2, [R1, R0, LSL #2]
STR R2, element
```

92

- Stop the program execution
- Walk through the heap and mark objects as live or unreferenced.
- Delete unreferenced objects in the heap.
- Compact the heap by moving the live objects together (remembering to update the pointers to the moved objects).

91