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

1

Why are registers faster than memory?

2

What is the one-address style of instruction?

3

Describe the load-store style of instruction.

4

How is code written in ARM Assembly run?

5

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

6

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

7

What is the instruction to sum two numbers?

8

- *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.*

2

Memory.

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

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.

4

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

3

```
LDR <register> <memory_address_alias>
```

6

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

5

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

8

```
STR <register> <memory_address_alias>
```

7

What is the instruction to branch upon a condition?

9

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

10

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

11

What does `DEFB` *do?*

12

What's the syntax of DEFS?

13

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

14

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

15

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

16

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

It's register 15.

10

`B <condition> <branch_name>`

9

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

12

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>`

11

Stores the lowest eight bits of a register into memory.

`STRB <register> <memory_alias>`

14

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

13

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

16

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>`

15

When using Big Endian, bytes are read from to .

17

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

18

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

19

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

20

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

21

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

22

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

23

What does `ALIGN` *do?*

24

12

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

18

17

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

`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.*

20

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.*

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

22

21

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>`*.*

24

23

What does `ENTRY` *do?*

25

What does `EQU` *do?*

26

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

27

How can you combine a CMP instruction with another instruction?

28

What does `RSB` *do?*

29

What does `MLA` *do?*

30

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

31

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

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

32

`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>
```

26

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

25

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.*

28

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

27

Multiply and add. `MLA R0, R1, R2, R3`*:*

```
R0 = (R0 * R1) + R2
```

30

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

```
R1 = 0 - R0 = -R0
```

29

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.*

32

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*

31

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

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

33

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

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

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` *and* `R2`*?*

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

36

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]
```

37

How do you load a String into a register?

38

What does the `ADRL` *instruction do?*

39

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

40

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.*

34

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.*

33

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.*

36

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.*

35

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

38

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.*

37

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

40

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

39

What are the four ARM bit shifting/rotation instructions?

41

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

42

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

43

What is the command to push something to the stack?

44

What is the command to pop something off the stack?

45

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

46

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

47

What does the `BL` *command do?*

48

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

42

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

41

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

44

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

43

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

46

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

45

- 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.

48

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

47

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?*

49

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?

50

How do you pass a parameter to a variable?

51

How should a method access stacked parameters?

52

What is a stack frame?

53

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

54

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
```

50

```
MOV PC, LR
```

49

Just read the stack using the STR *command and add an offset depending on what parameter you want. E.g:*

```
LDR R0, [SP, #12]
```

52

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.)

51

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.

54

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.

53