

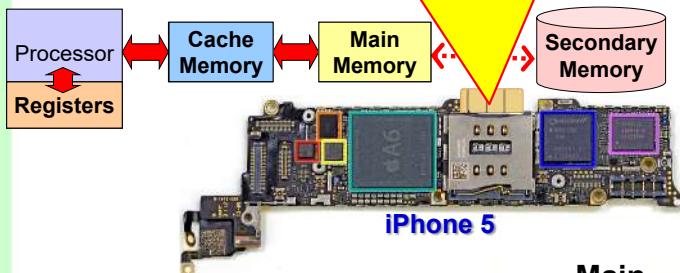
Data Organisation in Memory

Quiz Questions in Video Lectures

©2020, SCSE/NTU

Quiz: Memory Hierarchy

Which memory type is the data storage on a nano SIM card?



Registers



Cache



Main Memory



Secondary Memory



pause

Data Organisation in Memory

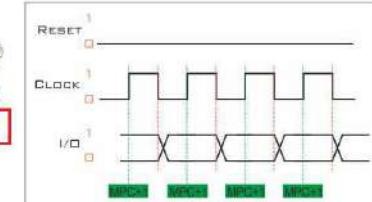
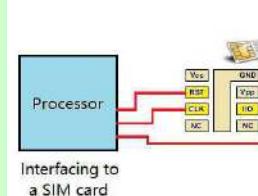
Role of Memory in Computing

Learning Objectives (2.1)

1. Describe the concept of programming in software.
2. Describe the von Neumann's stored program concept.
3. Describe the role of memory in computing.
4. Describe the characteristics and function of different data storage elements in the memory hierarchy.

©2020, SCSE/NTU

Quiz: Memory Hierarchy



SIM cards storage capacity various from 8KB - 256KB. Stores up to 250 contacts and numerous Network Identification Code.



Secondary Memory



©2020, SCSE/NTU

Chapter 2

Data Organisation in Memory

Number Representation

Learning Objectives (2.2)

1. Describe the different C numeric data types and their characteristics.
2. Describe the concept of numeric range and its implications to data size.
3. Describe how multi-byte numbers are stored in memory.

©2020, SCSE/NTU

Quiz: Byte-ordering

- How would a 32-bit hexadecimal value of **0x23AB45CD** be stored in memory using the Big Endian format?

In Big Endian, the **most significant byte** (i.e. **0x23**) is stored at the **lowest address**. The rest the bytes are ordered into consecutive increasing memory addresses.

Address		Memory	
N	0x23	N+1	0x45
N+2	0xAB	N+3	0xCD
← 8 bits →			

Address		Memory	
N	0x23	N+1	0x45
N+2	0xAB	N+3	0xCD
← 8 bits →			

Address		Memory	
N	0xCD	N+1	0xAB
N+2	0x45	N+3	0x23
← 8 bits →			

Address		Memory	
N	0xCD	N+1	0x45
N+2	0xAB	N+3	0x23
← 8 bits →			

a

b

c

d

©2020, SCSE/NTU

Quiz: Byte-ordering

- How would a 32-bit hexadecimal value of **0x23AB45CD** be stored in memory using the Big Endian format?

pause

Address		Memory	
N	0x23	N+1	0x45
N+2	0xAB	N+3	0xCD
← 8 bits →			

a

Address		Memory	
N	0x23	N+1	0x45
N+2	0xAB	N+3	0xCD
← 8 bits →			

b

Address		Memory	
N	0xCD	N+1	0x45
N+2	0xAB	N+3	0x23
← 8 bits →			

c

Address		Memory	
N	0xCD	N+1	0x45
N+2	0xAB	N+3	0x23
← 8 bits →			

d

©2020, SCSE/NTU

Quiz: Defining Variable Size

- How many bytes are allocated for an **unsigned long long int** variable?

pause

4

a

8

b

32

c

64

d

©2020, SCSE/NTU

Quiz: Defining Variable Size

- How many bytes are allocated for an `unsigned long long int` variable?

same for signed long long int

The `unsigned long long int` is an unsigned integer represented by 8 bytes. This is equivalent to 64 bits of data. If you have selected 64, you did not look at the question carefully.

64

32

8

4

a

b

c

d

Data Organisation in Memory

Character and Boolean Representation

Learning Objectives (2.3)

- Describe the `char` data type and its representations.
- Describe the Boolean data type and its implementation.

Quiz: Text File in Memory

- Which of the memory contents below is **least likely** to be a segment of memory containing a plain text file?

LS	MS	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	'	p	q
1	SOH	DC1	!	1	A	Q	a	b	r
2	STX	DC2	"	2	B	R	s	t	s
3	ETX	DC3	#	3	C	S	c	d	t
4	EOT	DC4	\$	4	D	T	e	f	u
5	ENQ	NAK	%	5	E	U	v	w	v
6	ACK	SYN	&	6	F	V	g	h	w
7	BEL	ETB	'	7	G	W	g	i	x
8	BS	CAN	(8	H	X	h	j	y
9	HT	EM)	9	I	Y	i	k	z
A	LF	SUB	*	:	J	Z	j	l	{
B	VT	ESC	+	;	K	[k	m	}
C	FF	FS	,	<	L]	l	n	
D	CR	GS	,	=	M	^	m	o	-
E	SO	RS	/	>	N	O	n	o	DEL
F	SI	US	?	O	-	-	-	-	-

7-bit ASCII Table

character i cannot be printed (suppose to be nt)

pause



Address	Memory	N	0x4E	N	0x2B	N	0x42	N	0x36	N	0x35	N	0x0D	N	0x0A	N	0x07	
		N		N		N		N		N		N		N		N		N
		O		O		O		O		O		O		O		O		O
		CR		CR		CR		CR		CR		CR		CR		CR		CR
		LF		LF		LF		LF		LF		LF		LF		LF		LF

a

b

c

Quiz: Text File in Memory

- Which of the memory contents below is **least likely** to be a segment of memory containing a plain text file?

LS	MS	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	'	p	q
1	SOH	DC1	!	1	A	Q	a	b	r
2	STX	DC2	"	2	B	R	s	t	s
3	ETX	DC3	#	3	C	S	c	d	t
4	EOT	DC4	\$	4	D	T	e	f	u
5	ENQ	NAK	%	5	E	U	v	w	v
6	ACK	SYN	&	6	F	V	g	h	w
7	BEL	ETB	'	7	G	W	g	i	x
8	BS	CAN	(8	H	X	h	j	y
9	HT	EM)	9	I	Y	i	k	z
A	LF	SUB	*	:	J	Z	j	l	{
B	VT	ESC	+	;	K	[k	m	}
C	FF	FS	,	<	L]	l	n	
D	CR	GS	,	=	M	^	m	o	-
E	SO	RS	/	>	N	O	n	o	DEL
F	SI	US	?	O	-	-	-	-	-

7-bit ASCII Table

No CR

LF ↗

+65 Space

Space ↗

BEL X

BEL ↗

Address	Memory	N	0x4E	N	0x2B	N	0x42	N	0x36	N	0x35	N	0x0D	N	0x0A	N	0x07	
		N		N		N		N		N		N		N		N		N
		O		O		O		O		O		O		O		O		O
		CR		CR		CR		CR		CR		CR		CR		CR		CR
		LF		LF		LF		LF		LF		LF		LF		LF		LF

a

b

c

Chapter 2

Data Organisation in Memory

Array, String and Structure Representations

Learning Objectives (2.4)

1. Describe the representation of arrays in memory.
2. Describe the C and Pascal strings storage in memory.
3. Describe the representation of structures in memory

©2020, SCSE/NTU

Quiz: Accessing Array Elements

- What is the start address of the array variable **A[3]** if the base address of the array **A** is hexadecimal **0x100**?

```
:
long int A[5];
A[3] = 1;
```

Since **long int** is 4 bytes, the start addresses of **A[3]** should be:
 $BA + (3 \times 4) = 0x100 + 0x00C = 0x10C$
Note: 12 bytes from **0x100** is **0x10C** (not **0x112**)
(addresses use concise hexadecimal notation)

0x112**0x10C****0x103****0x100****a****b****c****d**

Quiz: Accessing Array Elements

- What is the start address of the array variable **A[3]** if the base address of the array **A** is hexadecimal **0x100**?

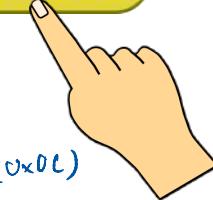
```
:
long int A[5];
A[3] = 1;
```

$$\begin{aligned}
BA &= 0x100 \\
\text{offset} &= 0x100 + (4 \times 3) \\
&= 0x100 + 12 (0x0C) \\
&= 0x10C
\end{aligned}$$

0x112**0x10C****0x103****0x100****a****b****c****d**

©2020, SCSE/NTU

long int = 4 bytes

pause

Chapter 2

Data Organisation in Memory

Pointer Representation

Learning Objectives (2.5)

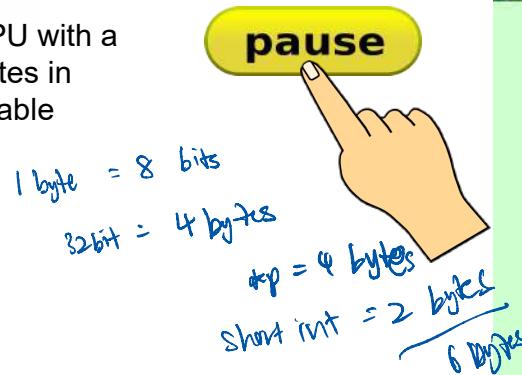
1. Describe the representation of pointers in memory.

©2020, SCSE/NTU

Quiz: Memory Allocation

- A C compiler is compiling for a CPU with a 32-bit address bus. How many bytes in memory will it allocate for the variable declarations shown?

```
:
short int I;
short int *P;
```



- 2 bytes 4 bytes 6 bytes 8 bytes

a

b

c

d

Quiz: Memory Allocation

- A C compiler is compiling for a CPU with a 32-bit address bus. How many bytes in memory will it allocate for the variable declarations shown?

```
:
short int I; 2 bytes
short int *P; 4 bytes
```

The **short int** data type is 2 bytes. A pointer data type represents an address and will require a memory size that matches the number of bits needed to specify a unique address (i.e. 32 bits or 4 bytes). Therefore the total memory allocated by the C compiler will be **2 + 4**, which is **6 bytes**.

- 2 bytes 4 bytes 6 bytes 8 bytes

a

b

c

d

Chapter 2

Data Organisation in Memory

Data Alignment

Learning Objectives (2.6)

- Describe the concept of data alignment.
- Describe data alignment considerations for efficient access and storage of structure variables in memory.

Hand pointing at a yellow 'pause' button.

Quiz: Data Alignment

- How many bytes will be allocated for variables **s1** and **s2**? Assume multi-byte data requires even address data alignment and memory allocation starts at address **0x000**.

```
struct rec {
    char C;      → 1 byte
    short int X; → 2 bytes
}
rec S1,S2;
```

- 2 bytes 4 bytes 6 bytes 8 bytes

a

b

c

d

Quiz: Data Alignment

- How many bytes will be allocated for variables **s1** and **s2**? Assume multi-byte data requires even address data alignment and memory allocation starts at address **0x000**.

```
struct rec {
    char C;
    short int X;
}
rec S1,S2;
```

Character **C** of **s1** occupies byte at **0x000**. But next available memory location **0x001** is an odd address, a padded byte will be added to ensure the two-byte variable **X** starts at an even address **0x002**. Structure **s1** takes 4 bytes. The same is repeated for **s2** starting at **0x004**. Total allocation is 8 bytes.

2 bytes

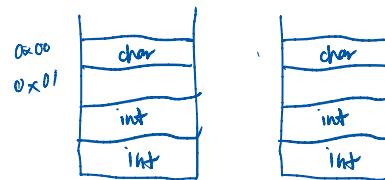
4 bytes

6 bytes

8 bytes

a**b****c****c**

Must start at even address

**s1**

Total: 8 bytes

Chapter 3

Instruction Organisation in Memory

Quiz Questions in Video Lecture

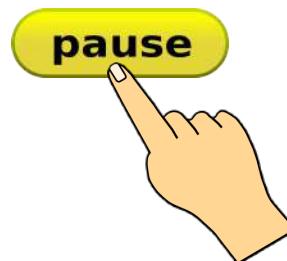
©2020, SCSE/NTU

Quiz: The MOV Instruction

- Based on the initial state shown, which option is the correct register state after executing the ARM mnemonic **MOV R3, R4?**



R3 **0xAABBCCDD**
 R4 **0x00000000**
 Before execution



R3 **0x00000000**
 R4 **0x00000000**
 After execution



R3 **0xAABBCCDD**
 R4 **0xAABBCCDD**
 After execution



R3 **0xAABBCCDD**
 R4 **0x00000000**
 After execution



R3 **0x00000000**
 R4 **0xAABBCCDD**
 After execution



Chapter 3

Instruction Organisation in Memory

Characteristics of Instructions and the ARM Programmer's Model

Learning Objectives (3.1)

- Describe the characteristics and role of an instruction.
- Describe the function of the ARM instruction set **MOV** instruction
- Describe the ARM programmer's model.
- Describe the functions and interpretation of several ARM registers.

©2020, SCSE/NTU

Quiz: The MOV Instruction

- Based on the initial state shown, which option is the correct register state after executing the ARM mnemonic **MOV R3, R4?**

R3 **0xAABBCCDD**
 R4 **0x00000000**
 Before execution

Destination ————— Source

R3 **0x00000000**
 R4 **0x00000000**
 After execution



R3 **0xAABBCCDD**
 R4 **0xAABBCCDD**
 After execution



R3 **0xAABBCCDD**
 R4 **0x00000000**
 After execution



R3 **0x00000000**
 R4 **0xAABBCCDD**
 After execution

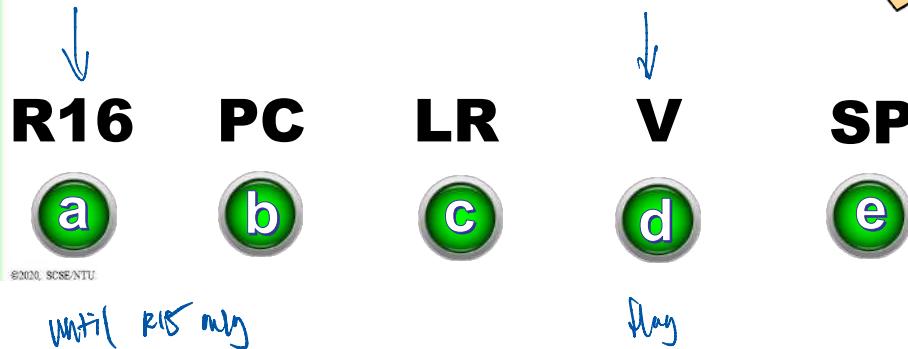


©2020, SCSE/NTU

Quiz: ARM Programmer's Model

- Which of the following is(are) **not** valid register(s) in the ARM programmer's model?

Note: You can indicate more than one selection.

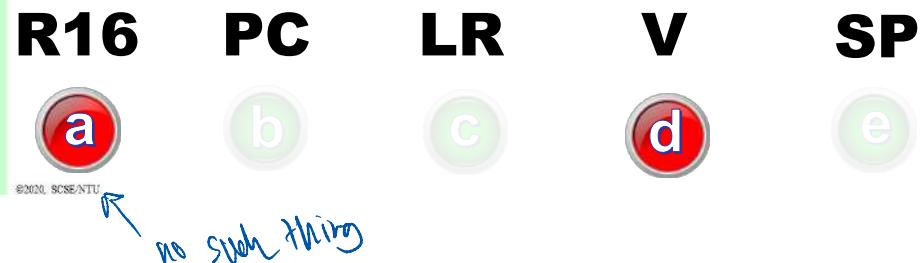


Quiz: ARM Programmer's Model

- Which of the following is(are) **not** valid register(s) in the ARM programmer's model?

Note: You can indicate more than one selection.

ARM has 16 32-bit registers, **R0** to **R15**. As such, **R16** is not a valid register. **R13**, **R14** and **R15** are also known as the **SP**, **LR** and **PC** respectively. The overflow **V** flag is not a register but a condition code bit (or flag) in the CPSR. It indicates if an overflow has occurred during a signed arithmetic operation.



Chapter 3

Instruction Organisation in Memory

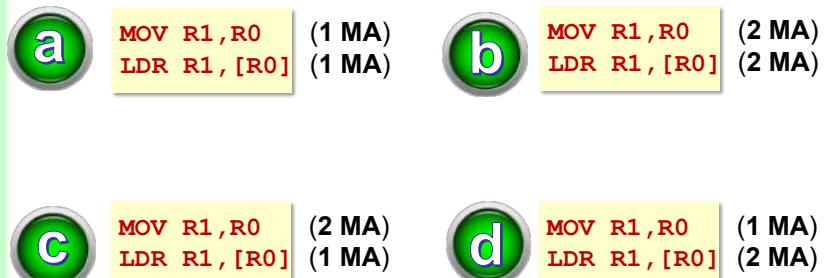
Basic Execution Cycle

Learning Objectives (3.2)

- Describe the function of a simple **LDR** instruction from the ARM instruction set.
- Describe the basic execution cycle of a simple **LDR** instruction.
- Describe the factor that determines the execution speed of an instruction.

Quiz: Instructions & Memory Access

- Select the option that correctly describes the number of memory access (**MA**) required to execute the two ARM instructions shown.



Quiz: Instructions & Memory Access

- Select the option that correctly describes the number of memory access (**MA**) required to execute the two ARM instructions shown.

a

MOV R1, R0
LDR R1, [R0] (1 MA)
(1 MA)

b

MOV R1, R0
LDR R1, [R0] (2 MA)
(2 MA)

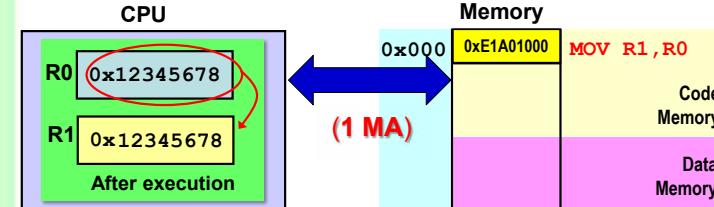
c

MOV R1, R0
LDR R1, [R0] (2 MA)
(1 MA)

d

MOV R1, R0
LDR R1, [R0] (1 MA)
(2 MA)**Quiz: Instructions & Memory Access**

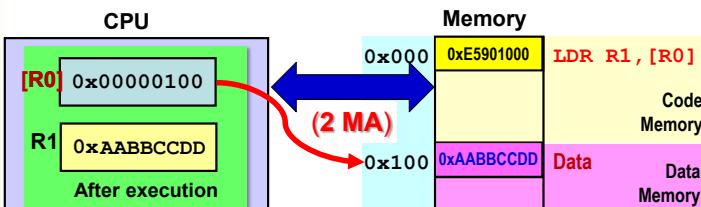
- Select the option that correctly describes the number of memory access (**MA**) required to execute the two ARM instructions shown.



d

MOV R1, R0
LDR R1, [R0] (1 MA)
(2 MA)**Quiz: Instructions & Memory Access**

- Select the option that correctly describes the number of memory access (**MA**) required to execute the two ARM instructions shown.



d

MOV R1, R0
LDR R1, [R0] (1 MA)
(2 MA)

Addressing Modes

Quiz Question for Video Lectures

Addressing Modes

Introduction to Assembly Programming and Addressing Modes

Learning Objectives (4.1)

1. Identify why and when to use assembly language programming.
2. Describe what are addressing modes.

Addressing Modes

Register Direct and Immediate Addressing

Learning Objectives (4.2)

1. Describe what is register direct.
2. Describe what is immediate data and its application.

Quiz: The MOV mnemonic

- Which of the following **MOV** mnemonic(s) is(are) **not valid** ARM instructions?
- You may select more than one answer.

MOV R1 ,R13

a

MOV R1 ,SP

b

MOV #0 ,R0

c

MOV R2 ,CPSR

d

pause



Quiz: The MOV mnemonic

- Which of the following **MOV** mnemonic(s) is(are) **not valid** ARM instructions?
- You may select more than one answer.

Register **R13** is also known as the stack pointer (**SP**). Both notations are valid for register direct. ✓

Immediate values can only be used as source operands. ✗

Current Processor Status Register (CPSR) cannot be accessed using the **MOV** operator. ✗

MOV R1, R13

MOV R1, SP

MOV #0, R0

MOV R2, CPSR

a

b

c

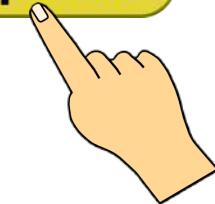
d

©2020, SCSE/NTU

Quiz: Immediate Addressing

- Which of the following immediate value(s) of **I** is(are) **not valid** for the ARM mnemonic **MOV R3, #I**?
- Answers are given in both decimal and hexadecimal notations.
- You may select more than one answer.

pause



$I = \text{M}\text{I}\text{O}$

I = 256

$I = \text{0x}\text{102}$

I = 258

I = 0x258

I = 0x3F0

a

b

c

d

©2020, SCSE/NTU

Quiz: Immediate Addressing

- Which of the following immediate value(s) of **I** is(are) **not valid** for the ARM mnemonic **MOV R3, #I**?

Bit 11	2	5	8	Bit 0
0	0	1	0	0
0	0	1	0	1

Bit 11	0x3F				0	Bit 0
0	0	1	1	1	1	0
0	0	1	1	1	1	0

Decimal value of **0x100** ✓

Decimal value of **0x102** ✗

I = 256

b

8-bit hex value of **0x96** rotated right by 30 ($n=15$) bits ✓

I = 0x258

c

8-bit hex value of **0x3F** rotated right by 28 ($n=14$) bits ✓

I = 0x3F0

d

©2020, SCSE/NTU

Learning Objectives (4.3)

- Describe what is register indirect and the ARM instructions that support this addressing mode.
- Describe the variants and application of register indirect that uses base plus offset and index register.
- Compare the relative pros and cons of register direct and register indirect addressing modes.

Chapter 4**Addressing Modes****Register Indirect with Base Register**

Quiz: Register Indirect

- Given the initial states shown, which are the correct states of **R0** and **R2** after the execution of the instruction **STR R2 , [R0 , #4]**?

Initial States	
R0	0x00000100
R2	0x12345678
Address Memory	
0x100	0x00
0x101	0x00
0x102	0x00
0x103	0x00
0x104	0x88
0x105	0x88
0x106	0x88
0x107	0x88
:	:

R0	0x00000100
R2	0x12345678



R0	0x00000104
R2	0x88888888



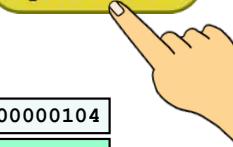
R0	0x00000100
R2	0x88888888



R0	0x00000100
R2	0x00000000



pause



Quiz: Register Indirect

- Given the initial states shown, which are the correct states of **R0** and **R2** after the execution of the instruction **STR R2 , [R0 , #4]**?

no exclusion mark, not encouraged

Initial States	
R0	0x00000100
R2	0x12345678
Address Memory	
0x100	0x00
0x101	0x00
0x102	0x00
0x103	0x00
0x104	0x88
0x105	0x88
0x106	0x88
0x107	0x88
:	:

R0	0x00000100
R2	0x12345678



R0	0x00000100
R2	0x88888888



R0	0x00000104
R2	0x1245678



R0	0x00000100
R2	0x00000000



STR operator copies register content to memory. So source register is not altered after execution. Only destination memory location is modified.

Quiz: Execution Speed

- With the initial states shown, which instruction will be **able** to load the value **0x102** into register **R2** with the **least** number of clock cycles?

Initial States	
R0	0x00000102
R2	0x12345678
Address Memory	
0x100	0x02
0x101	0x01
0x102	0x00
0x103	0x00
0x104	0x00
0x105	0x00
:	:

MOV R2 , #0x102



MOV R2 , R0

invalid



LDR R2 , [R0]

LDR R2 , [R0 , #-2]



Note: Multi-byte data storage in memory is in Little Endian format.

pause



Quiz: Execution Speed

- With the initial states shown, which instruction will be **able** to load the value **0x102** into register **R2** with the **least** number of clock cycles?

Initial States	
R0	0x00000102
R2	0x12345678
Address Memory	
0x100	0x02
0x101	0x01
0x102	0x00
0x103	0x00
0x104	0x00
0x105	0x00
:	:

MOV R2 , #0x102

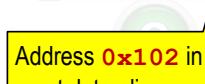


The immediate value #0x102 is not valid.

MOV R2 , R0

0x102 in R0 is copied into R2. This takes only 1 memory cycle.

LDR R2 , [R0]



Address 0x102 in R0 does not meet data alignment constraints for 4-byte memory access.

LDR R2 , [R0 , #-2]

Effective address 0x100 meets data alignment constraints but LDR takes 2 memory cycles.

Addressing Modes

Register Indirect with Autoindexing and Stacks

Learning Objectives (4.4)

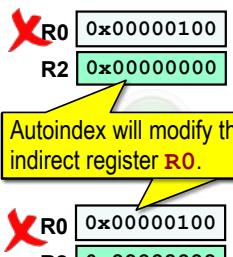
1. Describe what is autoindexing feature of ARM's register indirect addressing mode.
2. Describe the differences between pre-index and post-index addressing modes.
3. Describe the various stack implementations and operations using the ARM addressing modes.

©2020, SCSE/NTU

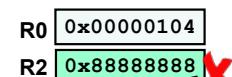
Quiz: Autoindexing

- Given the initial states shown, which are the correct states of **R0** and **R2** after the execution of the instruction **LDR R2 , [R0] , #4?**

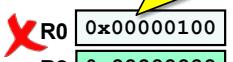
Initial States	
R0	0x00000100
R2	0x12345678
Address Memory	
0x100	0x00
0x101	0x00
0x102	0x00
0x103	0x00
0x104	0x88
0x105	0x88
0x106	0x88
0x107	0x88
:	:



Autoindex will modify the indirect register **R0**.



Offset with **pre-index** autoindexing given by **LDR R2 , [R0 , #4]** ! gives this answer.



This is offset with **post-index** autoindexing.
This mean **EA** is given by **0x100** in **R0** before offset **4** is added to **R0**.

Quiz: Autoindexing

- Given the initial states shown, which are the correct states of **R0** and **R2** after the execution of the instruction **LDR R2 , [R0] , #4?**

R0 0x00000104
R2 0x88888888



R0 0x00000104
R2 0x00000000



Initial States
R0 0x00000100
R2 0x12345678

Address Memory

0x100	0x00
0x101	0x00
0x102	0x00
0x103	0x00
0x104	0x88
0x105	0x88
0x106	0x88
0x107	0x88
:	:

©2020, SCSE/NTU

pause


R0 0x00000100
R2 0x00000000



R0 0x00000100
R2 0x88888888



CZ1106
Push – **STR R0 , [SP , #-4]** !
Pop – **LDR R0 , [SP] , #4**

Full Descending (FD)



Quiz: The ED Stack

- Give the mnemonic to **push** the register **R1** on to an **Empty Descending (ED) stack**.

Full ascending

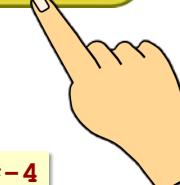
STR R1 , [R13 , #4]



STR R1 , [R13 , #-4] !



SP → 0xFE000003
0xFE000004
0xFE000005
0xFE000006
0xFE000007
0xFE000008 0x44
0xFE000009 0x33
0xFE00000A 0x22
0xFE00000B 0x11
0xFE00000C

pause


Empty descending

STR R1 , [R13] , #4



STR R1 , [R13] , #-4



Full descending

Empty ascending

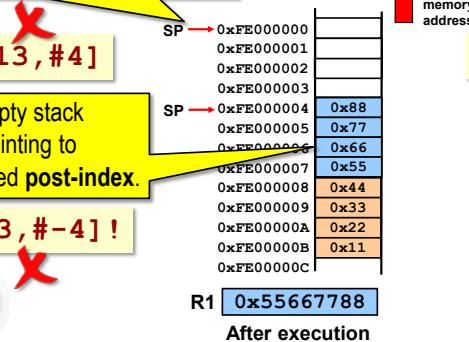
©2020, SCSE/NTU

Quiz: The ED Stack

- Give the mnemonic to **push** the register **R1** on to an **Empty Descending** (ED) stack.

Then, add **-ve** offset of **-4** to **R13** to move **SP** towards lower memory address (descending)

STR R1 , [R13 , #4]



STR R1 , [R13] , #-4



STR R1 , [R13] , #4

**Quiz: Position-Independent**

- Select the instructions below that can be used at label **Q1**, do what is in the comments and are **position-independent** (P-I).
- Note: You may select more than one answer.

MOV R1 , R0 ;copy R0 to R1



MOV PC , #0x20 ;jump to Jmp



Address	ARM code segment
0x00	MOV R0 , #0x20
0x04	Q1 ???
:	:
0x20	Jmp MOV R0 , R2
:	:

B Jmp ;jump to Jmp



MOV PC , R0 ;jump to Jmp



pause

**Quiz: Position-Independent**

- Select the instructions below that can be used at label **Q1**, do what is in the comments and are **position-independent** (P-I).

✓ MOV R1 , R0 ;copy R0 to R1

Changing any register other than the **PC** is always P-I.

✗ MOV PC , #0x20 ;jump to Jmp

Loading **PC** with the absolute start address (i.e. **0x20**) of the instruction to jump to violates the P-I requirement that jump must be relative to the current **PC** value. It does not matter whether immediate or register direct is used to modify the **PC**.

Chapter 4**Addressing Modes****PC-related Addressing Modes****Learning Objectives (4.5)**

- Describe difference between absolute & relative jump.
- Describe the concept of position-independent code and how it is achieved.
- Describe how data can be accessed using PC relative addressing

✓ B Jmp ;jump to Jmp

The ARM branch instruction **B** used a **PC** relative offset to carry out the jump and is therefore P-I.

✗ MOV PC , R0 ;jump to Jmp

Instruction Set

Quiz Questions for Video Lectures

Quiz: The MVN Instruction

- Given the 32-bit hexadecimal value in **R0** after executing the instruction **MVN R0, #0x3F0**.

Invalid mnemonic



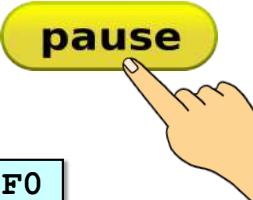
R0 **0x000003F0**



R0 **0xFFFFFC0F**



R0 **0x00000C0F**



Instruction Set

Data Transfer Instructions

Learning Objectives (5.1)

- Describe how data in register and memory can be efficiently transferred.
- Describe how byte-sized data can be access in memory.

Quiz: The MVN Instruction

Quiz: The MVN Instruction

- Given the 32-bit hexadecimal value in **R0** after executing the instruction **MVN R0, #0x3F0**.

MVN will complement the 32-bit immediate value before moving it to the destination register.

Invalid mnemonic



R0 **0xFFFFFC0F**



R0 **0x000003F0**



R0 **0x00000C0F**



32-bit value of **0x3F0** is **0x000003F0**. Its complement is **0xFFFFFC0F**. Reminder:
 $3 = 0011_2$
 $\text{Not } (3) = 1100_2$
 $= 0xC$

Quiz: Data Transfer Instructions

- Based on the initial states, which instruction(s) (after execution) will give this result:
- Note: You may select more than one answer.

After Execution
R0 0x00000100
R2 0x000000F8

pause

MVN R2, #0x7

a

LDRB R2, [R0, #7]

b

MVN R2, R2

c

LDR R2, [R0], #4

d

Note: Multi-byte data storage in memory is in Little Endian format.

Initial States	
R0	0x00000100
R2	0xFFFFFFF07
Address	Memory
0x100	0xF8
0x101	0x00
0x102	0x00
0x103	0x00
0x104	0x00
0x105	0x00
0x106	0x00
0x107	0xF8
:	:

©2020, SCSE/NTU

Quiz: Data Transfer Instructions

- Based on the initial states, which instruction(s) (after execution) will give this result:
- Note: You may select more than one answer.

After Execution
R0 0x00000100
R2 0x000000F8

MVN R2, #0x7

X

LDRB R2, [R0, #7]

✓

Initial States	
R0	0x00000100
R2	0xFFFFFFF07
Address	Memory
0x100	0xF8
0x101	0x00
0x102	0x00
0x103	0x00
0x104	0x00
0x105	0x00
0x106	0x00
0x107	0xF8
:	:

©2020, SCSE/NTU

The 32-bit complement of 0x7 copied into R2 is 0xFFFFFFF8.

EA is 0x107, so 0xF8 zero-extends in R2. Byte access - no need data alignment.

MVN R2, R2

✓

0xFFFFFFF7 in R2 when complemented will become 0x000000F8. This value moves back into itself in R2.

LDR R2, [R0], #4

X

The register indirect with post-index will load 0x000000F8 into R2 but R0 will be changed to 0x104 due to autoindexing.

Chapter 5**Instruction Set****Arithmetic Instructions****Learning Objectives (5.2)**

- Describe the operation and uses of the basic arithmetic instructions in the ARM instruction set.
- Describe how arithmetic operations influence the status of Condition Code flags.

Quiz: Binary Subtraction

- What is the result of the 5-bit binary subtraction $01001_2 - 11100_2$ if both these numbers are signed numbers?

00101₂

a

01101₂

c

00011₂

b

10010₂

d

©2020, SCSE/NTU

Quiz: Logical and Shift Instructions

- Given the initial states shown, which is the correct state of **R2** after executing the instruction **EOR R2, R0, R1, ASR R3**?

Initial States	
R0	0x11223344
R1	0xF0000000
R2	0x12345678
R3	0x00000004

R2 0x12345678

a

R2 0x11223344

b

R2 0xEE223344

c

R2 0x1E223344

d

↓ shift right

pause

**Quiz: Logical and Shift Instructions**

- Given the initial states shown, which is the correct state of **R2** after executing the instruction **EOR R2, R0, R1, ASR R3**?

calculator
does not work

$$\begin{aligned}
 &= (R1 \gg \text{sign } R3) \\
 &= (R1 \gg \text{sign } 4) \\
 &= (0xF0000000 \gg \text{sign } 4) \\
 &= (0xFF000000)
 \end{aligned}$$

R2 0x11223344

b

R2 0x12345678

a

R2 0xEE223344

c

R2 0x1E223344

d



Initial States

R0 0x11223344

R1 0xF0000000

R2 0x12345678

R3 0x00000004

Quiz: An ARM Code Segment

- R0** contains a zero-extended byte read from memory. What would be the value in **R0** after executing the code segment shown?

```

MOV R1, #0
SUB R2, R1, R0, LSR #7
ORR R0, R0, R2, LSL #8
    
```

R0 0x00000080

```

MOV R1, #0
SUB R2, R1, R0, LSR #7
ORR R0, R0, R2, LSL #8
    
```

R0 0x00000000

R0 0xFFFFFFF80

R0 0xFFFFFFFF

R0 0xFFFFFFFF

pause

**Quiz: An ARM Code Segment**

- R0** contains a zero-extended byte read from memory. What would be the value in **R0** after executing the code segment shown?

R0 0x00000080

```

MOV R1, #0
SUB R2, R1, R0, LSR #7
ORR R0, R0, R2, LSL #8
    
```

R0 0x00000000

R0 ...000000000001

$$\begin{aligned}
 R0 &= R0 \text{ OR } (R2 \ll 8) \\
 R0 &= R0 \text{ OR } 0xFFFFFFF00 \\
 R2 &= 0xFFFFFFF80
 \end{aligned}$$

$$\begin{aligned}
 &= (R2 \ll 8) \\
 &= 0xFFFFFFF00
 \end{aligned}$$

R0 0x00000000

R0 0xFFFFFFFF

R0 0x00000080

R0 0xFFFFFFFF

This code sign-extends the zero-extended byte fetched from memory to 32-bits

calculator works

Instruction Set

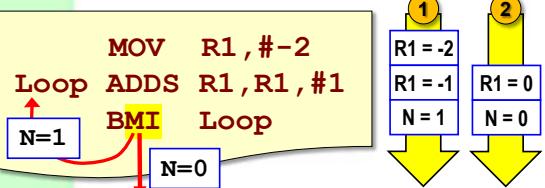
Program Control Instructions

Learning Objectives (5.4)

1. Describe the various conditional branch instructions and its uses.
2. Describe how conditional test can be implemented.

Quiz: Loop Count

- How many times will the **BMI Loop** instruction be executed in the code segment shown?



0 1 2 3

Execute twice. Once branch taken and once branch not taken.

Quiz: Loop Count

- How many times will the **BMI Loop** instruction be executed in the code segment shown?

```

MOV R1, #-2
Loop ADDS R1, R1, #1
BMI Loop
  
```

R1 = -2
R1 = -1
R1 = 0

N=1

- 0 1 2 3
- a b c d

Quiz: The CMP Instruction

- Which of the following values in **R0** and **R1** will cause the **MOV R2, R3** instruction to be executed?
- Note: You may choose more than one correct answer

```

Right == Same  
C = 1
  
```

CMP R0, R1
BHS Skip
MOV R2, R3
Skip :

- | | | | | | |
|----------|---------------|---------------|----------|---------------|---------------|
| a | R0 0xABCDFFFF | R1 0xABCDFFFF | b | R0 0x00000001 | R1 0x00000005 |
| c | R0 0x00000005 | R1 0x00000001 | d | R0 0x00000005 | R1 0xFFFFFFFF |

pause



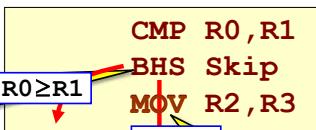
pause



Quiz: The CMP Instruction

- Which of the following values in **R0** and **R1** will cause the **MOV R2 ,R3** instruction to be executed?

BHS test unsigned numbers. If unsigned value in $(R0 \geq R1)$, **BHS** will skip over the **MOV** instruction



Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned
CC or LO	C = 0	Lower, unsigned

a	R0 0xABCDFFFF	R1 0xABCDFFFF
b	R0 0x00000001	R1 0x00000005
c	R0 0x00000005	R1 0x00000001
d	R0 0x00000005	R1 0xFFFFFFFF

©2020, SCSE/NTU

To execute **MOV R2 ,R3** ($R0 < R1$) must be true.

b	R0 0x00000001	R1 0x00000005
d	R0 0x00000005	R1 0xFFFFFFFF

©2020, SCSE/NTU

Program Example

Finding the Largest Number

Learning Objectives (5.5)

- Use appropriate data transfer instructions to retrieve memory arrays efficiently.
- Use appropriate program control instructions to determine flow of program based on desired outcomes.
- Implement a simple find max algorithm in ARM assembly.

Quiz: Code Optimisation

- Which mnemonic(s) in **FindMax** can you remove to optimised the program?
- Note: You may select more than one instructions.

```

MOV R0, #0x100 ;setup pointer to first array element
MOV R1, #9 ;load 9 into counter register
LDR R3, [R0] a ;assume 1st no. in array is current max

Loop ADD R0, R0, #4 b ;increment array pointer to next element
LDR R2, [R0] ;get next no. in array
CMP R2, R3 ;compare R3 and R2 (i.e. R2-R3)
BLS Skip c ;branch if R2 < current max (i.e. R3)
MOV R3, R2 ;update current max. in R3 with R2

Skip SUBS R1, R1, #1 d ;decrement 1 from counter register
BNE Loop ;jump back to Loop if not zero
  
```

pause



Quiz: Code Optimisation

- Which mnemonic in **FindMax** can you remove by using a more optimised addressing mode?

```

MOV R0, #0x100 ;setup pointer to first array element
MOV R1, #9 ;load 9 into counter register
LDR R3, [R0] a ;assume 1st no. in array is current max

Loop LDR R2, [R0, #4]! b ;pre-index autoindex to next element
;then get array element from memory
CMP R2, R3 ;compare R3 and R2 (i.e. R2-R3)
BLS Skip c ;branch if R2 < current max (i.e. R3)
MOV R3, R2 ;update current max. in R3 with R2

Skip SUBS R1, R1, #1 d ;decrement 1 from counter register
BNE Loop ;jump back to Loop if not zero
  
```

©2020, SCSE/NTU

©2020, SCSE/NTU

Conditional Execution

- ARM instructions can be conditionally executed based on the CC flags.

ARM code example

```
; C code
if (R0 == 1)
    R1 = 3;
else
    R1 = 5;

    CMP R0, #1      ; set CC based on r0 -1
    BNE ELSE        ; if (R0 == 1)
    MOV R1, #3      ; then { R1 := 3}
    B SKIP         ; skip over else code seg
ELSE MOV R1, #5    ; else { R1 := 5}
SKIP .....;
```



- The conditional execution feature allows us to make the execution of each instruction dependent on the current status of the **N**, **Z**, **V**, **C** flags.

```
CMP R0, #1      ; if (r0 == 1)
Execute if Z=1  MOVEQ R1, #3    ; then { r1 := 3}
Execute if Z=0  MOVNE R1, #5    ; else { r1 := 5}
SKIP .....;
```

©2020, SCSE/NTU

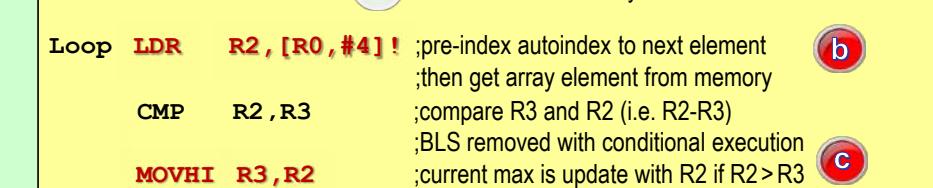
Quiz: Code Optimisation

- Which mnemonic in **FindMax** can you remove by using a more optimised addressing mode?

```
MOV R0, #0x100 ;setup pointer to first array element
MOV R1, #9     ;load 9 into counter register
LDR R3, [R0]    a;assume 1st no. in array is current max

Loop LDR R2, [R0, #4]! ;pre-index autoindex to next element
                        ;then get array element from memory
CMP R2, R3       ;compare R3 and R2 (i.e. R2-R3)
                  ;BLS removed with conditional execution
MOVHI R3, R2     c;current max is update with R2 if R2>R3

Skip SUBS R1, R1, #1 d;decrement 1 from counter register
BNE Loop         ;jump back to Loop if not zero
```



©2020, SCSE/NTU

1 byte = 8 bits

1 MB = 1,048,576

decimal value of 8-bit hex 2's complement 0x80?

$$0x80 = 1000\ 0000$$

$$-(0111\ 1111)$$

$$\begin{array}{r} | \\ -(1000\ 0000) \\ \hline \end{array}$$

$$= -128$$

Find a positive short integer in the memory map if Little Endian byte-ordering is used.

| 0xF3
| 0x72
| 0x82

8272F3

Ans: 0x72F3

short int occupies 2 bytes

Pascal string format

Give the longest string represented by this Pascal string format.

(1106) main()
char s[4] = "123"; // a string constant

A. 128
B. 255
C. 256
D. No limit

Base Address	Address	Content in Memory
BA _s	0x0100	0x00
	0x0101	0x31
	0x0102	0x32
	0x0103	0x33
	0x0104	:
	0x0105	:
	0x0106	0x00

largest byte-sized
value for 8-bits
is 255

Mock Quiz

0x00
0x31
0x00

POSSIBLE interpretations:

- Unicode "6"
- 3 bool values
(2 false, 1 true)
- C string of length 1

Interpreting Data in Memory

Address Memory
N | 0x00
N+1 | 0x41

Which interpretation is not possible for the memory contents shown?

null terminator should
be in higher addr (at the bottom)

A. C string of length 1

B. Short integer with decimal value of 16640

C. One Unicode character representing letter 'A'

D. 2 Boolean variables, one TRUE and one FALSE

41

41

Pointer P

CX1106

Pointers

Based on the given memory map below, what is the size (in bits) of the short int pointer p?

```
main()
{
    short int x;
    short int *p;
    x = 1;
    p = &x;
}
```

12-bit address still needs two bytes to store pointer address.

Address Size	Percentage
8 bits	11%
12 bits	58%
16 bits	21%
32 bits	10%

Content in Memory

Address	Content
0x200	0x00
0x201	0x01
0x202	0x00
0x203	0x02
0x204	0x00
0x205	:
0x206	0x00
...	...

CPU has 12-bit address

A. 8 bits
B. 12 bits
C. 16 bits
D. 32 bits

only possible +

size 8, 16, 32, 64 bits

MOV and LDR instructions

CX1106

MOV and LDR Instructions

LDR R1, [R0]
MOV R0, R1

Given the initial states below, what is the value in R0 after executing the two instructions shown.

Note: Little Endian byte-ordering is used

Address Memory: 0x100 (0xDD, 0xCC, 0xBB, 0xAA)

R0: 0x00000100
R1: 0x12345678

Initial States

A. R0: 0x00000100
B. R0: 0x12345678
C. R0: 0xDDCCBBAA
D. R0: 0xAABBCCDD

$$R1 = AABBCCCDD$$

Memory Cycles

CX1106

Reducing Memory Cycles

LDR R1, [R0] 2~
LDR R2, [R0] 2~
LDR R3, [R0] 2~

What is the number of memory cycles needed to execute of this code segment?
What is the minimum cycle count if this code is fully optimised?

6

A. 3 and 3 (optimised)
B. 6 and 3 (optimised)
C. 6 and 4 (optimised)
D. 6 and 5 (optimised)

LDR R1, [R0] 2~
MOV R2, R1 1~
MOV R3, R1 1~
Fully optimised = 4

Code Segment	Cycle Count	Percentage
3 and 3 (optimised)	2~	6%
6 and 3 (optimised)	2~	33%
6 and 4 (optimised)	2~	33%
6 and 5 (optimised)	2~	22%

Change in ARM Content

Review Previous Session (VISUAL)

Which ARM register(s) contents will **change** after the execution of **MOVS R0,R1** given the initial states shown below?

A. R0
✓ B. R0, PC
C. R0, PC, CPSR
D. R0, R1, PC, CPSR

N V Z C V
1 0 0 0

N V Z C V
1 0 0 0

MOV and LDR instructions

MOV and LDR Instructions

Which of this ARM mnemonic below **will not** result in **R1=0x00000200** given the initial states shown.

Note: Little Endian byte-ordering is used
Must use **MOV R1,R0** to do register direct.

✓ A. LDR R1, R0 ~~X~~
B. MOV R1, #0x200
C. LDR R1, [R0]
D. None of the above

Address Memory
0x200 0x00
0x201 0x02
0x202 0x00
0x203 0x00

R0 0x00000200
R1 0x12345678

Initial States

Register Indirect with Base & Offset

Register Indirect with Base & Offset

Given the initial states below, what is the value in R0 after executing the code segment shown.

MOV R0,R1
MOV R0,#0x100
LDR R0,[R0,#4]

EA = 0x100 + 4 = 0x104

R0 0x00000000
R1 0x12345678

A. R0 0x00000000
B. R0 0x00000100
C. R0 0x12345678
D. R0 0x33221100
E. ✓ R0 0xAABBCCDD

Address Memory
0x100 0x00
0x101 0x11
0x102 0x22
0x103 0x33
0x104 0xD0
0x105 0xC
0x106 0xBB
0x107 0xBA
0x108

Initial States

Review of Addressing Mode (Part 1)

R2 0x00001100 Which of the following instruction will not give this result in R2 after execution.

Note: Byte-ordering is based on Little Endian

A. MOV R2, R0
B. MOV R2, #0x1100
C. LDR R2, [R0, R1]
~~D. LDR R2, [R0, #1]~~
EA = 0x1100 + 1 = 0x1101
E. None of the above

Initial States

data alignment violation

Register Indirect with Autoindexing

Register Indirect with Autoindexing

Which ARM register(s) maybe modified by the instruction below based on the given values in memory.

LDR R1, [R0], #4
R0 = 0x104 After Execution
R1 = (-1)
Program Counter (PC) increments after instruction execution
LDR does not influence CC flags (No "S" suffix option)

CPSR 0 0 0 0
R0 0x000000100

Address Memory
0x100 0xFFFF
0x101 0xFFFF
0x102 0xFFFF
0x103 0xFFFF
0x104 0xDDD
0x105 0xCC
0x106 0xBB
0x107 0xAA
0x108

No such thing as LDRS

Push & Pop Operations

ARM Stack Implementations

Based on the description of the Push and Pop operations shown, which ARM stack type is being implemented here?

Push - STR R0, [SP, #-4]!
Pop - LDR R0, [SP], #4

A. Full Descending (FD)
B. Empty Descending (ED)
C. Full Ascending (FA)
D. Empty Ascending (EA)

Full - SP point to top of stack
(occupied space)

PC-relative Data Access

Address: 0x00200

Code Area

Data Area

Note: The PC points 8 bytes ahead of the current executed instruction.

PC is 8 bytes ahead during execution

ADD R0, PC, Offset

LDR R1, [R0] ; copy VarA

0x0024 - VarA

0x2000 - VarA

What PC offset value is needed to access VarA in a position independent manner?

PC-relative Offset = Var address - [PC value] + 8

A. 0x0000
B. 0x2000
✓ C. 0x2000 - 0x0020 - 8
D. 0x2000 - 0x0020 + 8
E. 0x2000 - 0x0024 - 8

ADDS

Address: 0x00200

Code Area

Data Area

What condition code (CC) flag(s) will be set after executing this instruction given the initial states,

ADDS R0, R0, R1

R0: 0x40000000
R1: 0x40000000

Initial States

A. No flags
B. N
C. N and C
✓ D. N and V
E. N, C, and V

Sub

Address: 0x00200

Code Area

Data Area

Which option does not give the equivalent of this expression?

R0 = R1 - R0

A. SUB R0, R1, R0
B. SUB R0, R0, R1
RSB R0, R0, #0
C. MVN R0, R0
ADD R0, R1, R0
ADD R0, R0, #1
✓ D. None of the above

R0 = NOT(R0)
R0 = R1 + NOT(R0)
R0 = R1 + NOT(R0) + 1 = R1 - R0

Two's Complement operation = Negate

$$\begin{aligned} \text{B. } R0 &= R0 - R1 \\ R0 &= 0 - (R0 - R1) \\ &= R1 - R0 \end{aligned}$$

Logical instruction AND

CX1106 Logical Instruction

AND R3, R3, R0

Given the initial states below, what will be the result of executing this instruction.

Initial States:

- A. R0 **0xFF000000**, R3 **0xFF0000FF**
- B. ✓ R0 **0xFFFF0000**, R3 **0xFF000000** *R0 acts as a mask. For AND, a 0 bit will clear.*
- C. R0 **0xFFFF0000**, R3 **0xFFE000FF** *Correct if ORN*
- D. R0 **0xFFFF0000**, R3 **0x00E000FF** *Correct if EOR*

RRX (Rotate right Extended)

CX1106 Rotate Right Extended

MOV R0, R1, RRX

Given the initial states below, what is the value in R0 after executing this instruction.

RRX can only rotate right 1 bit

Initial States:

- A. R0 **0x80000000**
- B. R0 **0x00000011**
- C. ✓ R0 **0x80000011**
- D. R0 **0x00000045**

C flag extends

0010 0010 → 0011
0001 0001

no change in CPSR
as Mov has NO S

ADDS Loop Construct

CX1106 Analysing a Loop Construct

MOV R1, #0xFFFFFFF
Loop ADDS R1, R1, #1
BNE Loop

How many times is the ADDS instruction executed?

Z = 1

Jump to Loop: 1

A. 0 R1 **0xFFFFFFFF**
B. 1 R1 **0xFFFFFFFF**
C. ✓ 2 R1 **0x00000000**
D. 3

(-2) R1 goes from -2 to -1 (not zero yet).
R1 zero, so exit loop.

CMP Instruction

CX1106

How many times is the CMP instruction executed?

R1 is 0 and not 1, so CMP with #1 does not set Z flag.

R1 is 1, so CMP with #1 will set the Z flag.

A. 0 R1 0x00000000
 B. 1 R1 0x00000001
 C. ✓ 2
 D. 3

BCC Conditional Test

CX1106

The Conditional Test (Bcc)

Which content in R1 will avoid the execution of the MOV R2, R3 instruction?

CMP R1, #0xFFFFFFFF
 BLT Next
 MOV R2, R3

Next :

Read this as:
 Branch to NEXT
 (i.e. avoid MOV R2, R3)
 if R1 is less than -2.
 i.e. (R1 < -2) or (R1 LT -2)

A. ✓ R1=0xFFFFFFF
 B. R1=0xFFFFFFFF
 C. R1=0xFFFFFFFF
 D. R1=0x00000000

CX1106

Another Conditional Test (Bcc)

Which content in R2 will avoid the execution of the MOV R2, R3 instruction?

MOV R1, #0
 CMP R1, R2
 BHS Next
 MOV R2, R3

Next :

Read this as:
 Branch to NEXT
 (i.e. avoid MOV R2, R3)
 if (R1=0) is higher than or same as R2, i.e. (R1 ≥ R2)

A. R2=0x00000001
 B. R2=0xFFFFFFFF
 C. R2=0xFFFFFFFF
 D. ✓ R2=0x00000000

$0 \geq 1$
 $0 \geq \text{big}$
 $0 > \text{big}$
 $0 \geq 0$ ✓

Review Question

Q1 - CALL

Ans: What registers may be affected by executing :

BL RegR1

Ans: PC & LR

Take location of instruction "BL RegR1" ⁺⁴, store into Link Register (LR)

Then take RegR1 make a copy of the above memory location
and put into PC

Stack pointer not affected by BL instruction in this case

Q2 - CALL

What value will be found
at the LR after the
execution of BL MYSUB?

0x000	MOV SP, #0xFFC
:	:
0x010	BL MYSUB
:	:
0x050	MYSUB ...

Ans : 0x014

* Every instruction a fixed 4 bytes.

BitCnt Question

Give the code to use **Count1s** to count the number of bits in the memory variable at address **0x100** (stored in R0).

```
Count1s: MOV R0, #0           ; Initialize R0 with 0
          MOV R1, #22        ; Initialize R1 with 22
          Loop: ADC R0, R1      ; Add the memory value
          SRSB R2, R1      ; Decrement counter by 1
          BNE R2, #0       ; Loop if not zero
          MOV PC, LR      ; Return to the caller
```

slide 30 code



(1)

```
MOV LR, PC
B Count1s > 204 location
```

→ return add here : 208 location

E.g.
Assume MOV LR, PC is in address 200
Value of PC = 20E ← or offset of PC itself

(2)

```
BL Count1s > 200 location
```

→ 204 ⇒ Return memory location

(3)

```
LDR R1, [R0]
BL Count1s
```

① & ② are the same, just that they don't load value in 0x100 into R1

⇒ Branch Link - 4 ahead
PC - 8 ahead

(4)

```
MOV R1, [R0]
B Count1s
```

MOV PC, LR == BX LR

PC point to
ahead during
normal execution
like B instruction

(1) "MOV LR, PC" places PC (address of MOV+8) to LR, followed by a branch
⇒ Actually the correct return address
However, value not loaded to R1 prior to subroutine

$$\Rightarrow \text{MOV LR, PC} = \text{BL Count1s}$$

(2) Have to prepare the parameter first before calling the Count1s subroutine
Count1s requires the parameter to be passed via the R1 register

(3)

1. Set up parameter into R1 register
2. Call Count1s subroutine with BL instruction using the appropriate subroutine label

(3) MOV an invalid instruction; cannot be used to read from memory
Operands can be only

* (1) If another instruction was placed between MOV & branch, it would loop infinitely

Passing params in memory

Given the problem to transfer lower-case to upper case, what is the right syntax to read each letter from memory to register R1 (base address is in R0)?

B = byte

Address	Memory
0x100	"a"
0x101	"p"
0x102	"p"
0x103	"l"
0x104	"e"
0x105	0x000
0x106	:

(1) LDRB R1, [R0]

(2) LDRB R1, [R0], #1

(3) LDR R1, [R0], #1

(4) LDR R1, [R0], #4

(1) Byte is read but does not update R0
for the next instruction

⇒ increment R0 in another instruction

E.g. ADD R0, R0, #1

(2) instruction can read a byte, and
increments R0 with 1 after fetching

⇒ by auto indexing

(3) instruction can read a 32-bit word,
and increments R0 with 4
after fetching

"#1" will cause an error \Rightarrow runtime
cause trying to get unaligned word

"0x100" \rightarrow "0x101" x "0x104"
allowed

(4) instruction can read a 32-bit word,
and increments R0 with 4
after fetching

"#4" will not cause an error

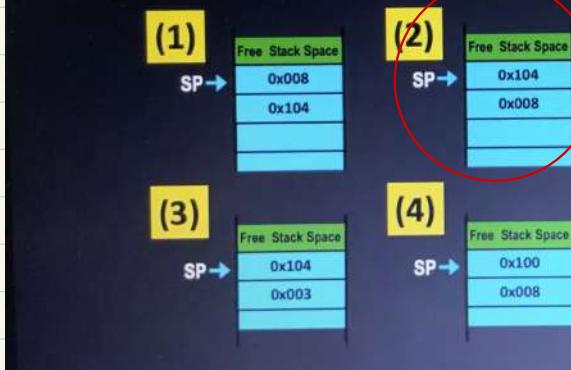
Review Q4 – Stack Passing

Show the known contents on the stack on entering subroutine **SubA**.

Address	Memory
1 0x000	MOV R0, #0x100
2 0x004	LDR R1, [R0], #4
3 0x008	STMFD SP!, {R1,R0}
0x00C	BL SubA
0x010	

post = true while push $\times 4$

R0 gets incremented



After 1

- ① R0 = 0x100
- ② R1 = 0x008
- ③ R0 = 0x104
- ④ R0 0x104
R1 0x008

After 1

R0	0x100
R1	

After 3

R0	0x104
R1	0x008

↓ ↓ put into stack

After 2

R0	0x104
R1	0x008

R0	0x104
R1	0x008

LDR R1,[R0],#4

- 1) go to address in R0 (0x100), take the value (0x008) and put into R1
- 2) increment R0 by 4 ($0x100 \rightarrow 0x104$)

Clock cycles of STMFD

- ⇒ For 1 register, 3 cycles.
- ⇒ +1 cycle for each additional register

What should be the instruction in a

A. LDMFD SP, {R5,R4}

B. LDMFD SP!, {R5,R4}

C. LDMFA SP, {R4,R5}

D. LDMFA SP!, {R4,R5}

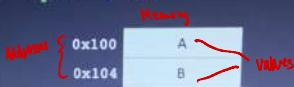
```
SubX STMFD SP!, {R5,R4}
      LDR  R4, [SP, #12]
      LDR  R5, [SP, #8]
      :
      :
      ;(a)
      MOV  PC, LR
```

C & D : Since STMFD was used, C & D not valid here

A : does not update stack pointer

For the shown code segment, how are parameters A and B passed?

```
MOV R0, #0x100 ①
LDR R1, [R0], #4 ②
STMFD SP!, {R0,R1}
BL Sub
```



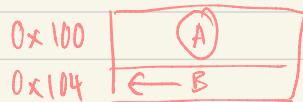
A. A reference, B value

B. A value, B value

C. A reference, B reference

D. A value, B reference

0x104 : reference of B
R1 has the value of A



- ① R0 = 0x100
② R1 = A — value
R0 = 0x104 — reference

① R0 has address of A

② post index

R0 0x104 ← value of 0x100 updated by 4
R1 A

MOV doesn't mean always pass by ref, it is case paired with LDR,
then one value one reference

The segment loads the content of memory address 0x100
to R1 → value of A

We store in the stack then the value of A and the
value 0x104 (address of B)

Review Q1 - IF

What is the equivalent C code segment for the following ARM code.

CMP R2,R3
BLT Next
ADD R3,R3,R2
Next :

(1) if (R2 < R3)
R3 = R3 + R2; X

(2) if (R2 < R3)
R2 = R2 + R3; X

(3) if (R2 >= R3)
R3 = R2 + R3; ✓

(4) if (R3 >= R2)
R3 = R2 + R3; X

$R_2 < R_3$
 \Updownarrow
 $R_2 \geq R_3$

(1) Need to use reverse condition for "<", which is ">=" or BGE for efficient implementation

(2) Same as (1) but in ADD, R3 is destination register.
So ADD R3, R3, R2 gives $R_3 = R_2 + R_3$

(3) Reverse condition correct

CMP R2, R3

↑

comes first

(4) Reverse condition correct but wrong order of R2, R3 in CMP instruction

What is the most optimal implementation of this C-code

```
If (X>10)
    F1(X);
Else x<=10
    F2(X);
```

(1)

Even if
still many → cos states
have changed

```
CMP R0, #10
BLGT F1
BLLF F2
```

both gets executed,
not what we wanted.
suppose to skip



(2)

x>10

```
CMP R0, #10
BLGT F1
BL F2
B Done
Done
```

2x10 (use)



(3)

not a
waste of
the CC flags

```
CMP R0, #10
BLLF F2
CMP R0, #10
BLGT F1
```

if this was
BLLF, it
would have
been most
optimal



(4)

```
CMP R0, #10
BLZ F1
BL F2
B Done
DoElse
Done
```



- ① Check if correct flow
- ② if multiply correct,
choose the one
with the least
instructions

What would be the C equivalent for the shown assembly?

(1)

```
if (R2 == R3 && R3 > 2)
    R3 = R3 + R2;
```

(2)

```
if (R2 != R3 || R3 < 2)
    R3 = R3 + R2;
```

(3)

```
if (R2 == R3 && R3 ≥ 2 )
    R3 = R3 + R2;
```

(4)

```
if (R3 != R2 && R3 ≥ 2 )
    R3 = R3 + R2;
```

```
CMP R2,R3
BNE Next "skip"
CMP R3,#2
ADDGE R3,R3,R2
```

Next : Greater than equal
"skip"

$\# R3 \neq R2$,
CMP & ADD GE
WILL NOT BE
EXECUTED

goto next
if !=
 $\therefore 2 \neq 4 \approx$
why

$R2 == R3$
 \geq

if && : negate first condition

if || : negate last condition

Pretest is used
in both loop construct
 $x \geq 5$ stay looping
if ($x \geq 5$) [$x = x - 1$]

Review Q2 - Loop

What is the equivalent C code segment for the following ARM code.

(X)

```
FOR (X=5; X>0; X--) { }
```

loop counter initialization in
for loop

```
Back CMP R0, #5
BLT Next
SUB R0, R0, #1
B Back
```

Next :

Always

no initialization
here

(2)

```
WHILE (X >= 5) {
    X = X - 1;
}
```



(3)

```
WHILE (X < 5) {
    X = X - 1;
}
```

"stay in loop"
if " $x < 5$ " X → should be REVERSE
OF BLT

(4)

```
DO {
    X = X - 1;
} WHILE (X >= 5)
```

post-test loop
→ test condition
after loop body

discard cause it is post-test

Want
right away

w Q2 - Loop

What is the equivalent C code segment for the following ARM code?

(1)

```
FOR (X=5; X>0; X--) { }
```

```
Back CMP R0, #5
BLT Next
SUB R0, R0, #1
B Back
```

Next :

Pre-test Loop

Test condition occurs
before loop body

(2)

```
WHILE (X >= 5) {
    X = X - 1;
}
```

Post-test Loop

Test condition after
loop body.

(3)

```
WHILE (X < 5) {
    X = X - 1;
}
```

(4)

```
DO {
    X = X - 1;
} WHILE (X >= 5)
```

What is the instruction to create a stack frame for 4 local variables (32-bit each)

A. ADD SP, SP, ~~#4~~ #16

B. SUB SP, SP, #~~4~~ #16

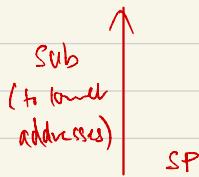
C. SUB SP, SP, #32

D. ADD SP, SP, #32

STR R11, [SP, # -4]!
MOV R11, SP

Each is 4 bytes

$$\therefore 4 + 4 = 16 \text{ bytes}$$



If we are using R11 as frame pointer, what is the instruction to store R4 to local variable c

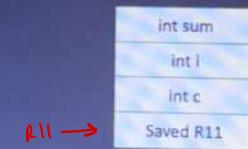


A. STR R4, [R11,#-1]

B. STR R4, [R11,#-4]

C. STR R4, [R11,#-4]! ← *not aligned to relevant*

D. STMFD R11!,{R4}



if r11 was a stack
pointer, then we would
have to update it

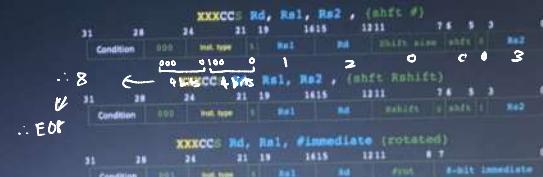
Q1 which of the following hexadecimal encoding of ADD R2,R1,R3

1. ~~0xE0012003~~

~~2. 0xF1021003~~

~~3. 0xE0131200~~

~~4. 0x00012003~~



Code	Condition	Flags	Meaning
0000 RD	Z = 1	Equal	
0001 RI	Z = 0	Not equal	
0002 CS WE RS	C = 1	Higher or same, unsigned	
0003 CC WE LD	C = 0	Lower, unsigned	
0100 MI	N = 1	Negative	
0101 PL	N = 0	Positive or zero	
0110 VB	V = 1	Overflow	
0111 VC	V = 0	No overflow	
1000 ET	C = 1 and Z = 0	Greater than, unsigned	
1001 LT	C = 0 or Z = 1	Lower or same, unsigned	
1002 GE	N < V	Greater than or equal, signed	
1003 LE	N > V	Less than, signed	
1100 WT	Z = 0 and N < V	Greater than, signed	
1101 LT	Z = 1 and N < V	Less than or equal, signed	
1110 AL	Always		
1111 MV	Reserved currently		

Code	Instruction
0000 AND	1000 TST
0001 EOR	1001 TEQ
0010 SUB	1010 CMP
0011 RSB	1011 CHN
0100 ADD	1100 ORR
0101 ADC	1101 MOV
0110 SBC	1110 BIC
0111 RSC	1111 MVH

Code	Instruction
1000 TST	0000 AND
1001 TEQ	0001 EOR
1010 CMP	0010 SUB
1011 CHN	0011 RSB
1100 ORR	0100 ADD
1101 MOV	0101 ADC
1110 BIC	0110 SBC
1111 MVH	0111 RSC

Address# of registers:

R0 - 0000

R1 - 0001

R2 - 0010

R3 - 0011

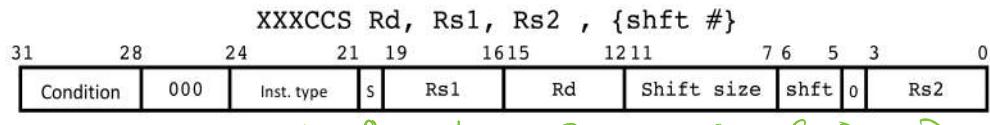
ADD - no condition

⇒ does not mean condition = 0000

↳ cause 0000 indicates EQ

⇒ so must be condition = 1110

↳ aka E, indicates "Always"



1110 000 0100 0 1 2 0 0 0 3
 ↑ ↑ ↑
 aka E,
 always ADD code
 ADD,
 not ADDS

con't is
 no shifts

Q2 which of the following hexadecimal encoding of RSBEQS R2,R1,#5

ECS

← 8 bit immediate value

1. ~~0xF10121005~~

2. ~~0x027211005~~ ✓

3. ~~0X013211005~~

4. ~~0x012121005~~

Condition	008	Inst. type	Rd	Rel	Rs2	(shift #)	7	5	3	0
Condition	008	Inst. type	Rd	Rel	Rs2	(shift #)	7	5	3	0
Condition	001	Inst. type	Rd	Rel	Rs2	#immediate (rotated)	7	5	3	0

Code	Condition	Flags	Meaning
0000 EQ	Z = 1	Equal	
0001 NE	Z = 0	Not equal	
0010 CS or GE	N = 1	Greater or same, unsigned	
0011 CC or LE	N = 0	Lower, unsigned	
0100 VS	V = 1	Negative	
0101 PL	N = 0	Positive or zero	
0110 VS	V = 1	Overflow	
0111 VC	V = 0	Not overflow	
1000 HS	S < 0 and Z = 0	Half signed	
1001 LS	S < 0 or Z = 1	Lower or same, signed	
1010 GS	N < V	Greater than or equal, signed	
1011 LT	N < V	Less than, signed	
1100 GT	Z = 0 and N < V	Greater than, signed	
1101 LS	Z = 1 and N < V	Less than or equal, signed	
1110 AL		Always	
1111 MV		Reserved (unused)	

Code	Instruction
0000 AND	1000 TST
0001 EOR	1001 TEQ
0010 SUB	1010 CMP
0011 RSB	1011 CMOK
0100 ADD	1100 ORR
0101 ADC	1101 MOV
0110 SBC	1110 BIC
0111 RSC	1111 MVN

Code	Instruction
1000 TST	0000 EQ
1001 TEQ	0001 NE
1010 CMP	0010 CS or GE
1011 CMOK	0011 CC or LE
1100 ORR	0100 VS
1101 MOV	0101 PL
1110 BIC	0110 GS
1111 MVN	0111 LT

0 0 0 0 0 0 1 0 0 1 1 | 1 | 2 | 5

XXXCCS Rd, Rs1, #immediate (rotated)

31	28	24	21	19	1615	1211	8	7	0
Condition	001	Inst. type	S	Rs1	Rd	#rot	8-bit immediate		

↳

0001 if

3rd operand

is a

register

Q3 which of the following hexadecimal encoding of MOVLE R7,R4

~~1.0xD0D74000~~

~~2.0xD0D70004~~

3.0XD1A70004

4.0xD1A74000

			XXXCCS	Rd, Rs, (shft #)				
31	28	24	1101	21 19	1615	1211	7 6	5 3
Condition	000	Inst. type	5	0000	Rd	Shift size	shft #	Ra
			XXXCCS	Rd, Rs, (shft #)				
31	28	24	1101	21 19	1615	1211	7 6	5 3
Condition	000	Inst. type	5	0000	#d	Rs	shiftr	0 shft #
			XXXCC	Rd, #immediate (rotated)				
31	28	24	1101	21 19	1615	1211	7	0
Condition	001	Inst. type	5	0000	Rd			#-bit immediate
Code	Condition	Flags		Meaning				
0000	RD	Z = 1		Equal				
0000	RD	Z = 0		Not equal				
0010	CS == MS	C < 0		Higher or same, unsigned				
0011	CS == MS	C > 0		Lower, unsigned				
0100	MI	N = 1		Negative				
0101	PL	N = 0		Positive or zero				
0110	VS	V = 1		Overflow				
0110	VS	V = 0		No overflow				
1000	LT	C < 0		Less than, unsigned				
1001	LS	C < 0 or Z = 1		Less or same, unsigned				
1010	GE	N < V		Greater than or equal, signed				
1011	LT	N > V		Less than, signed				
1100	GT	Z = 0 and N < V		Greater than, signed				
1101	LE	Z = 1 and N > V		Less than or equal, signed				
1110	AL			Always,				
1111	MV			Preserved (unused)				

Q4 If the registers list field is encoded with **0x40F0, what register list this represents?**

31	28	24	21	19	1615	1211	0
Condition	100	P U^W	L	Rs	Register list		

1. {R4, R15}
2. {R4-R7, LR}
3. {R1, R8-R11}
4. Don't know!

31	28	24	21	19	1615	0									
1110	100	10^1	0	1101 (SP)		0x40F0									
PC	LR	SP	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0

0 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0

R4 - R7 , LR

Muck Quiz

①

0xF3

most negative short int

0x72

⇒ short int occupies 2 bytes (2 memory locations)

0x82

⇒ Ans: 82 72

Little endian

= 82 72 F3

②

0x100

0x FF
0x 00
0x 00
0x FF

LDR R0, [R0]

ORR R2, R1, R0

Initial: R0 = 0x100

R1 = 0xAABBCCDD

Use calculator OR

⇒ 32 bit, unsigned

⇒ Ans: FF BB CC FF

③

```

Start MOV R1, # -5
      SUB R0, R0, R0
Loop  CMP R1, #1
      BGT Exit
      ADD R0, R0, R1
      ADD R1, R1, #2
      B Loop
Exit   ADD R0, R0, R1
Done   END
    
```

greater than

What is R0 at the end?

Start: R1 = -5 (not gt, go loop) (-5 > 1)

R0 = 0

Loop 1: R0 = -5 (not gt, go loop) (-3 > 1)

R1 = -3

Loop 2: R0 = -8 (not gt, go loop) (-1 > 1)

R1 = -1

Loop 3: R0 = -9 (not gt, go loop) (1 > 1)

R1 = 1

Loop 4: R0 = -8 (gt, go loop) (3 > 1)

R1 = 3

Exit: R0 = -5

✓

Figure 1

Reg	Value	PC	Value	Address	Contents
R0	0x00000100	PC	0x00000000	0x0100	0x04
R1	0x00000104	CPBR	0x00000000	0x0101	0x01
R2	0xAFFFFFFE	Note: Only bits 31 to 28 can change bit mask: 11 10 29 28 0		0x0102	0x00
R3	0x000008B2	CC flags: N [Z] C V		0x0103	0x00
R4	0x11223344			0x0104	0x00
Note: Multi-byte data in memory are accessed using Little Endian format.					

① R0 = 0x104
 (Part 1) ② R1 = AABBCCCD
 ③ R2 = 0x100
 ④ R3 = 0x882
 ⑤ R4 = R3

$$\begin{aligned} R4 - R3 \\ = 11223344 - 1104 \\ = 11222240 \end{aligned}$$

Given the initial states shown in Figure 1, what is the 32-bit hexadecimal value in R1 after executing the two ARM instructions shown above, one after the other?
 R1 = **0x AABBCCCD** Note: Provide your answer in 32-bit hexadecimal form, e.g. 0x12345678

⑥ Given the initial states shown in Figure 1, what is the 32-bit hexadecimal value in R2 after executing the instruction SUBS R2, R4, R3, LSL #1?
 R2 = **0x11222240** Note: Provide your answer in 32-bit hexadecimal form, e.g. 0x12345678

(Part 2)
 Fill in the missing instructions (11) to (15) in the ARM code segment below based on the given comments.
 Note: In typing your mnemonics, do not use tab, only a single space between operator and operands, and use only decimal numeric values (e.g. MOV R0, #12 or ADD R1, R2, #4)

```

Start: MOV R2, #1
Loop: LDAB R3, [R1], #1      : (11) Move byte pointed by R1 into R3, then increment R1 by 1
      MOV R3, R3, LSR #1      : (12) Divide R3 by 2 using shift or LSR R3, R3, #1
      ANDS R4, R3, #1      : (13) Make R4=1 if R3 is odd value or make R4=0 if R3 is even
      ADDS R0, R0, R3      : (14) Add R3 to R0 only if R3 is even → Z=0
      SUBS R2, R2, #1      : (15) Subtract R2 by 1
      BPL Loop
      END
  
```

Given the initial states shown in Figure 1, what is the 32-bit hexadecimal value in R0 at the end of executing the code segment shown above. Assume execution begins at the label Start.
 R0 = **0x 000 001D4** Note: Provide your answer in 32-bit hexadecimal form, e.g. 0x12345678

(Part 3)

1 MA	MOV R1, R0	①
2 MA	STR R4, [R1, #4]!	②
2 MA	LDR R3, [R1]	③

① R1 = 0x100 ② R1 = 0x104 ③ R3 = 11223344

$$\begin{aligned} R0 &= 0x100 \\ &\text{mem}(R1) \\ &= 0x11223344 \end{aligned}$$

Given the initial states shown in Figure 1, what is the 32-bit hexadecimal value in R3 after executing the ARM code segment shown above?
 R3 = **0x11223344** Note: Provide answer in 32-bit hexadecimal form, e.g. 0x12345678

How many memory accesses in total were incurred in executing the given ARM code segment above.
 5 Note: Give numeric answer without any leading zeroes (e.g. 0, 1, 2, etc.)

(Part 2)

Fill in the missing instructions (I1) to (I5) in the ARM code segment below based on the given comments.

Note: In typing your mnemonics, do not use tab; only a single space between operator and operands; and use only decimal numeric values (e.g. MOV R0, #12 or ADD R1, R2, #4).

Start: MOV R2, #1

Loop: LDRB R2, [R1], #1 : (I1) Move byte pointed by R1 into R2, then increment R1 by 1
 MOV R3, R3, LSR #1 : (I2) Divide R3 by 2 using shift or LSR R3, R3, #1 → right shift
 ANDS R4, R3, #1 : (I3) Make R4=1 if R3 is odd value or else R4=0 if R3 is even
 ADDEQ R0, R0, R3 : (I4) Add R3 to R0 only if R3 is even. → Z=0
 SUBS R2, R2, #1 : (I5) Subtract R2 by 1

BPL Loop

END

Multiply
is
LSL

Given the initial states shown in Figure 1, what is the 32-bit hexadecimal value in R0 at the end of executing the code segment shown above. Assume execution begins at the label `Start`.

R0 = 0x0000001D4

Note: Provide your answer in 32-bit hexadecimal form, e.g. 0x12345678

$$\text{Initial: } R1 = 0 \times 104 \quad R0 = 0 \times 100 \\ R2 = 0 \times 1 \\ R3 = 0 \times 882$$

$$\text{Loop 1: } R1 = \#0 \times 105$$

$$R3 = DD \quad (\text{hex } 221 \text{ (odd)})$$

$$R3 \div 2 = DD \div 2 = 0 \times 6E \quad (\text{right shift})$$

$$Z=0 \rightarrow 0 \times 6E \text{ AND } 1 = 0 = R4 \rightarrow Z=1 \\ R0 = R0 + R3 = 0 \times 16E \\ R2 = 0 \quad (\text{loop break})$$

$$\text{Loop 2: } R1 = \#0 \times 106$$

$$R3 = CC$$

$$R3 \div 2 = CC \div 2 = 0 \times 66$$

$$\leftarrow 0 \times 66 \quad \text{AND } 1 = 0 = R4 \rightarrow Z=1$$

$$R0 = R0 + R3 = 0 \times 16E + 0 \times 66 \\ = 0 \times 1D4$$

$$R2 = -1 \quad (\text{Exit})$$

$$\begin{array}{r} \text{AND } 1100110 \\ 0000001 \\ \hline 00000000 \end{array}$$