ENSC 3213

Homework Review

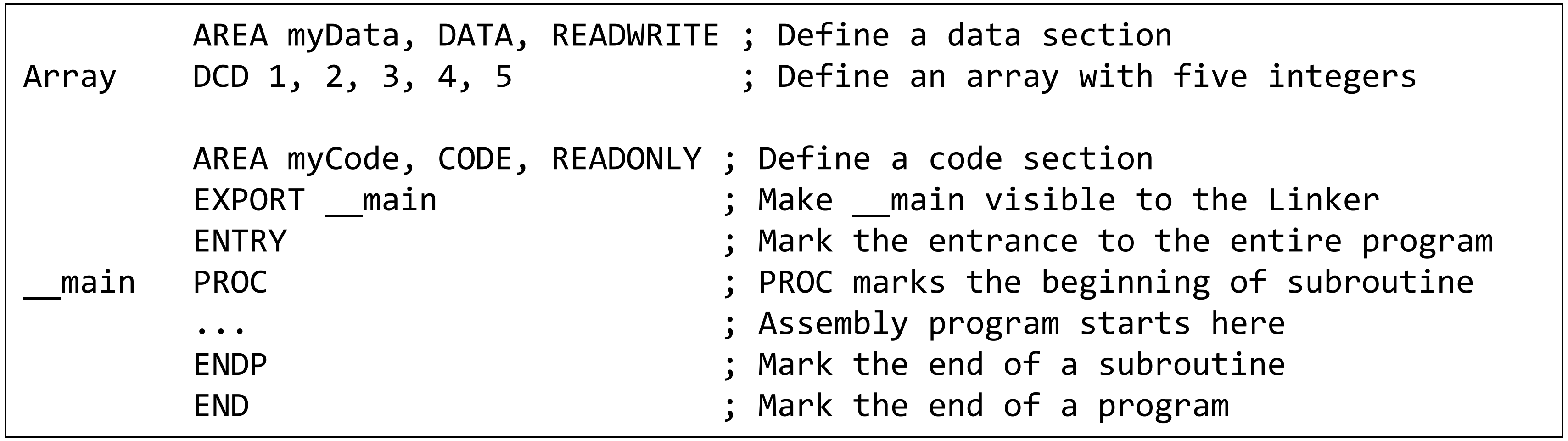
Professor: Dr. Yanmin Gong
TAs: Francisco Fernandes
Khuong Nguyen

Spring 2019

Homework 3

Homework 3 – Question 1

- The default attribute of the **CODE** section of an assembly code is:
  - Read Only
  - Write Only
  - Read and Write
  - None of above

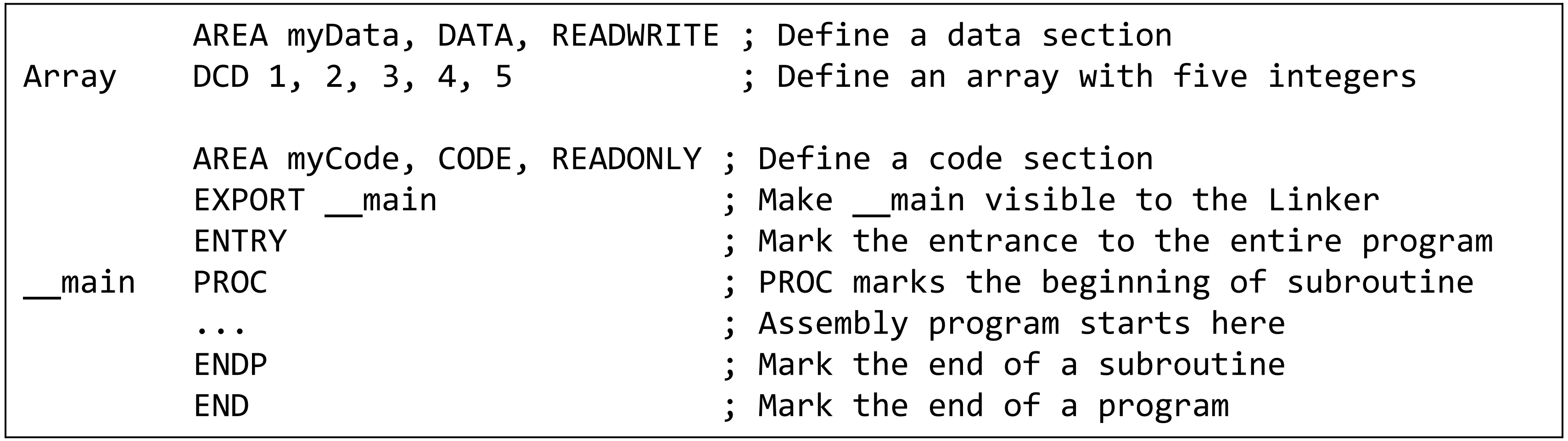


```
         AREA myData, DATA, READWRITE ; Define a data section
Array    DCD 1, 2, 3, 4, 5            ; Define an array with five integers

         AREA myCode, CODE, READONLY ; Define a code section
         EXPORT __main               ; Make __main visible to the Linker
         ENTRY                       ; Mark the entrance to the entire program
__main   PROC                        ; PROC marks the beginning of subroutine
         ...                         ; Assembly program starts here
         ENDP                        ; Mark the end of a subroutine
         END                         ; Mark the end of a program
```

Table 3-3. Skeleton of an ARM assembly program.
Textbook page 69

Homework 3 – Question 2

- The default attribute of the **DATA** section of an assembly code is:
  - Read Only
  - Write Only
  - Read and Write
  - None of above



```
        AREA myData, DATA, READWRITE ; Define a data section
Array   DCD 1, 2, 3, 4, 5            ; Define an array with five integers

        AREA myCode, CODE, READONLY ; Define a code section
        EXPORT __main               ; Make __main visible to the Linker
        ENTRY                       ; Mark the entrance to the entire program
__main  PROC                        ; PROC marks the beginning of subroutine
        ...                         ; Assembly program starts here
        ENDP                        ; Mark the end of a subroutine
        END                         ; Mark the end of a program
```

Table 3-3. Skeleton of an ARM assembly program.
Textbook page 69

Homework 3 – Question 3

- Most ARM processors support both Big Endian and Little Endian. ARM processor is Little Endian by default.

Endian	First byte (lowest address)	Middle bytes	Last byte (highest address)
big	*most* significant	...	*least* significant
little	*least* significant	...	*most* significant

Memory Address	Memory Data
0x8000	0xEE
0x8001	0x8C
0x8002	0x90
0x8003	0xA7
0x8004	0xFF

By default setting, the word stored at address 0x8000 is: A7 90 8C EE

Homework 3 – Question 3

- Most ARM processors support both Big Endian and Little Endian. ARM processor is Little Endian by default.

Endian	First byte (lowest address)	Middle bytes	Last byte (highest address)
big	*most* significant	...	*least* significant
little	*least* significant	...	*most* significant

Memory Address	Memory Data
0x8000	0xEE
0x8001	0x8C
0x8002	0x90
0x8003	0xA7
0x8004	0xFF

Little Endian

By default setting, the word stored at address 0x8000 is: A7 90 8C EE

Homework 3 – Question 3

- Most ARM processors support both Big Endian and Little Endian. ARM processor is Little Endian by default.

Endian	First byte (lowest address)	Middle bytes	Last byte (highest address)
big	*most* significant	…	*least* significant
little	*least* significant	…	*most* significant

Memory Address	Memory Data
0x8000	0xEE
0x8001	0x8C
0x8002	0x90
0x8003	0xA7
0x8004	0xFF

Little Endian

By default setting, the word stored at address 0x8000 is: A7 90 8C EE

32 bits or 4 bytes

Homework 3 – Question 3

- Most ARM processors support both Big Endian and Little Endian. ARM processor is Little Endian by default.

Endian	First byte (lowest address)	Middle bytes	Last byte (highest address)
big	*most* significant	…	*least* significant
little	*least* significant	…	*most* significant

Memory Address	Memory Data
0x8000	0xEE
0x8001	0x8C
0x8002	0x90
0x8003	0xA7
0x8004	0xFF

Little Endian

By default setting, the word stored at address 0x8000 is: A7 90 8C EE

32 bits or 4 bytes

Byte 3	Byte 2	Byte 1	Byte 0

Last byte *1st byte*

Homework 3 – Question 3

- Most ARM processors support both Big Endian and Little Endian. ARM processor is Little Endian by default.

Endian	**First byte (lowest address)**	**Middle bytes**	**Last byte (highest address)**
big	*most* significant	...	*least* significant
little	*least* significant	...	*most* significant

Memory Address	**Memory Data**
0x8000	0xEE
0x8001	0x8C
0x8002	0x90
0x8003	0xA7
0x8004	0xFF

A word (32 bits) will always follow this structure for either little or big endian

Little Endian

By default setting, the word stored at address 0x8000 is: A7 90 8C EE

32 bits or 4 bytes

Byte 3	Byte 2	Byte 1	Byte 0

Last byte

1st byte

Homework 3 – Question 3

- Most ARM processors support both Big Endian and Little Endian. ARM processor is Little Endian by default.

Endian	**First byte (lowest address)**	**Middle bytes**	**Last byte (highest address)**
big	*most* significant	...	*least* significant
little	*least* significant	...	*most* significant

4 bytes

Memory Address	**Memory Data**
0x8000	0xEE
0x8001	0x8C
0x8002	0x90
0x8003	0xA7
0x8004	0xFF

A word (32 bits) will always follow this structure for either little or big endian

Little Endian

By default setting, the word stored at address 0x8000 is: A7 90 8C EE

32 bits or 4 bytes

Byte 3	Byte 2	Byte 1	Byte 0

Last byte

1st byte

Homework 3 – Question 3

- Most ARM processors support both Big Endian and Little Endian. ARM processor is Little Endian by default.

Endian	First byte (lowest address)	Middle bytes	Last byte (highest address)
big	*most* significant	…	*least* significant
little	*least* significant	…	*most* significant

Memory Address	Memory Data
0x8000	0xEE
0x8001	0x8C
0x8002	0x90
0x8003	0xA7
0x8004	0xFF

4 bytes

5th byte. It will not be included in the answer

Little Endian

By default setting, the word stored at address 0x8000 is: A7 90 8C EE

32 bits or 4 bytes

Homework 3 – Question 3

- Most ARM processors support both Big Endian and Little Endian. ARM processor is Little Endian by default.

Endian	First byte (lowest address)	Middle bytes	Last byte (highest address)
big	*most* significant	…	*least* significant
little	*least* significant	…	*most* significant

	Memory Address	Memory Data
Least significant	0x8000	0xEE
	0x8001	0x8C
	0x8002	0x90
Most significant	0x8003	0xA7
	0x8004	0xFF

By default setting, the word stored at address 0x8000 is: A7 90 8C EE

Homework 3 – Question 3

- Most ARM processors support both Big Endian and Little Endian. ARM processor is Little Endian by default.

Endian	**First byte (lowest address)**	**Middle bytes**	**Last byte (highest address)**
big	*most* significant	…	*least* significant
little	*least* significant	…	*most* significant

Memory Address	**Memory Data**
0x8000	0xEE
0x8001	0x8C
0x8002	0x90
0x8003	0xA7
0x8004	0xFF

Least significant

Most significant

Last byte:
Most significant

1st byte:
Least significant

By default setting, the word stored at address 0x8000 is: A7 90 8C EE

Homework 3 – Question 3

- Most ARM processors support both Big Endian and Little Endian. ARM processor is Little Endian by default.

Endian	First byte (lowest address)	Middle bytes	Last byte (highest address)
big	*most* significant	…	*least* significant
little	*least* significant	…	*most* significant

Memory Address	Memory Data
0x8000	0xEE
0x8001	0x8C
0x8002	0x90
0x8003	0xA7
0x8004	0xFF

Last byte:
Least significant

In Big Endian? EE 8C 90 A7

1st byte:
Most significant

Homework 3 – Question 4 – Item A

- Suppose `r0 = 0x8000`, and the memory layout is as follows:

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

- What is the value of `r1` after running `LDR r1, [r0]` if the system is little endian or big endian?

Homework 3 – Question 4 – Item A

- Suppose `r0 = 0x8000`, and the memory layout is as follows:

Address	**Data**
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

- What is the value of `r1` after running `LDR r1, [r0]` if the system is little endian or big endian?

This instruction will load a word (32 bits) from the memory data starting from address `r0`.

Homework 3 – Question 4 – Item A

- Suppose `r0 = 0x8000`, and the memory layout is as follows:

`0x8000` in this case will be used as a memory address. `r0` DOES NOT represent data in this context!

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

- What is the value of `r1` after running `LDR r1, [r0]` if the system is little endian or big endian?

This instruction will load a word (32 bits) from the memory data starting from address `r0`.

Homework 3 – Question 4 – Item A

- Suppose `r0 = 0x8000`, and the memory layout is as follows:

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

Most significant ← 0x8003

Least significant ← 0x8000

4 bytes

Last byte:
Most significant

In Little Endian: `r1 = 0D EB 2C 1A`

1st byte:
Least significant

Last byte:
Least significant

In Big Endian: `r1 = 1A 2C EB 0D`

1st byte:
Most significant

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

- Suppose the system is set as little endian. What are the values of `r1` and `r0` if the instructions are executed separately?
  - `LDR r1, [r0, #4]`
  - `LDR r1, [r0], #4`
  - `LDR r1, [r0, #4]!`

This means that one instruction does not affect the other. So, when you run the next instruction, `r1` and `r0` will be reinitialized.

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

- Suppose the system is set as little endian. What are the values of `r1` and `r0` if the instructions are executed separately?
  - `LDR r1, [r0, #4]`
  - `LDR r1, [r0], #4`
  - `LDR r1, [r0, #4]!`

Let’s look them one by one!

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0, #4]`

In this case, we are accessing the memory data using the *pre-index mode*

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0, #4]`

In this case, we are accessing the memory data using the *pre-index mode*

It means we are going to load the memory data from address *`r0 + 4`* into *`r1`*.

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0, #4]`

In this case, we are accessing the memory data using the *pre-index mode*

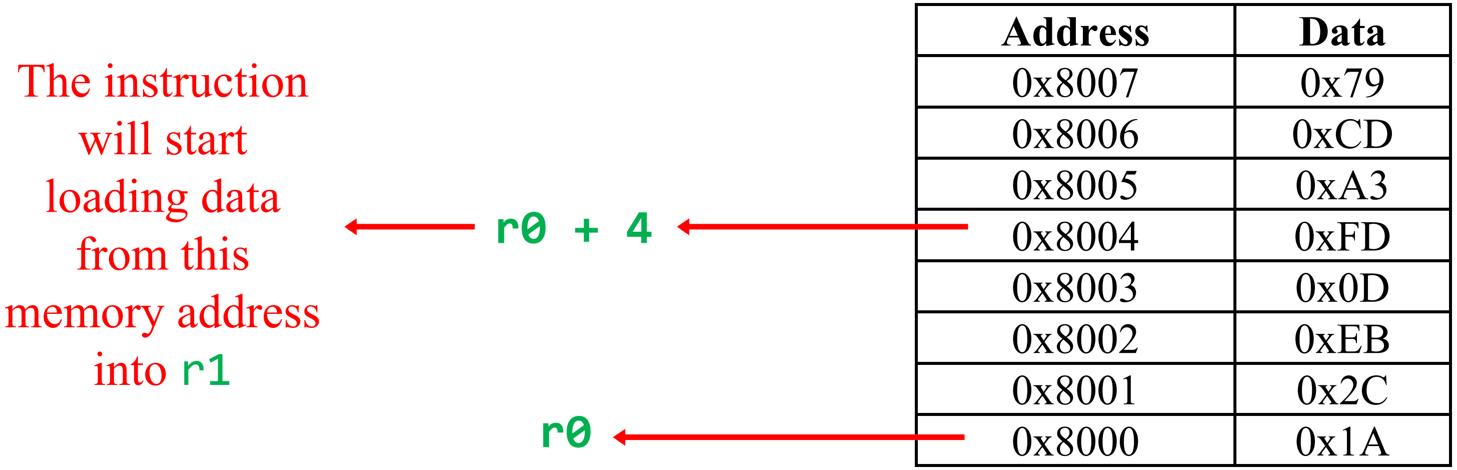
It means we are going to load the memory data from address *r0 + 4* into *r1*.

In the pre-index mode r0 will NOT be modified.

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0, #4]`



The instruction will start loading data from this memory address into `r1`

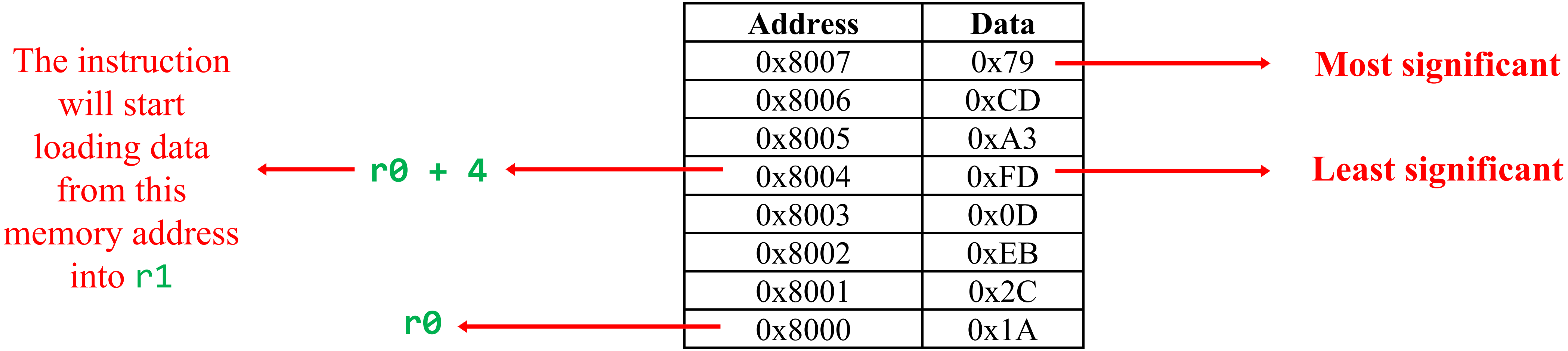
`r0 + 4`

`r0`

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0, #4]`



The instruction will start loading data from this memory address into `r1`

`r0 + 4`

`r0`

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

Most significant

Least significant

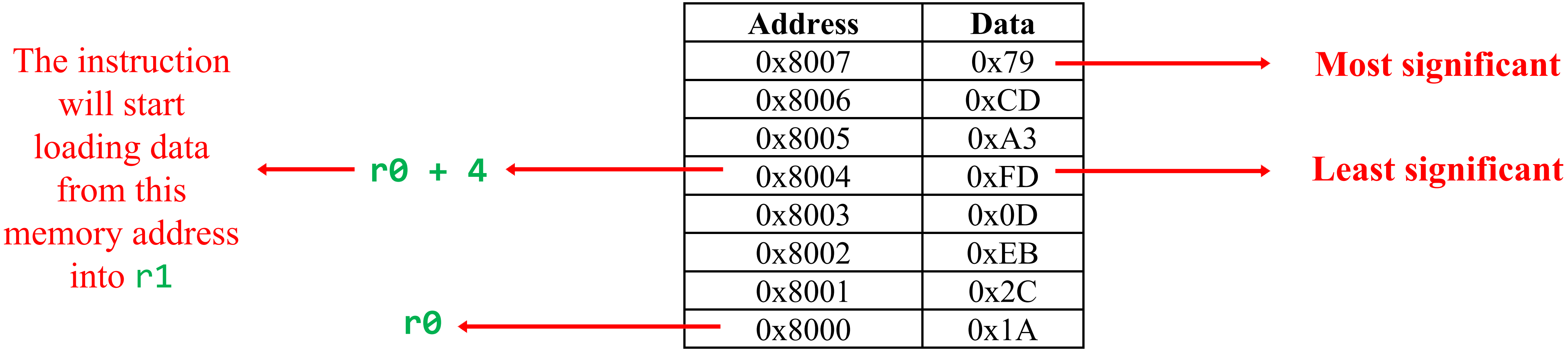
Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0, #4]`

Solution:

`r0 = 0x8000` → **r0 is unchanged!**

`r1 = 79 CD A3 FD`



Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

The instruction will start loading data from this memory address into r1
r0 + 4
r0
Most significant
Least significant

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0], #4`

In this case, we are accessing the memory data using the *post-index mode*

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0], #4`

In this case, we are accessing the memory data using the *post-index mode*

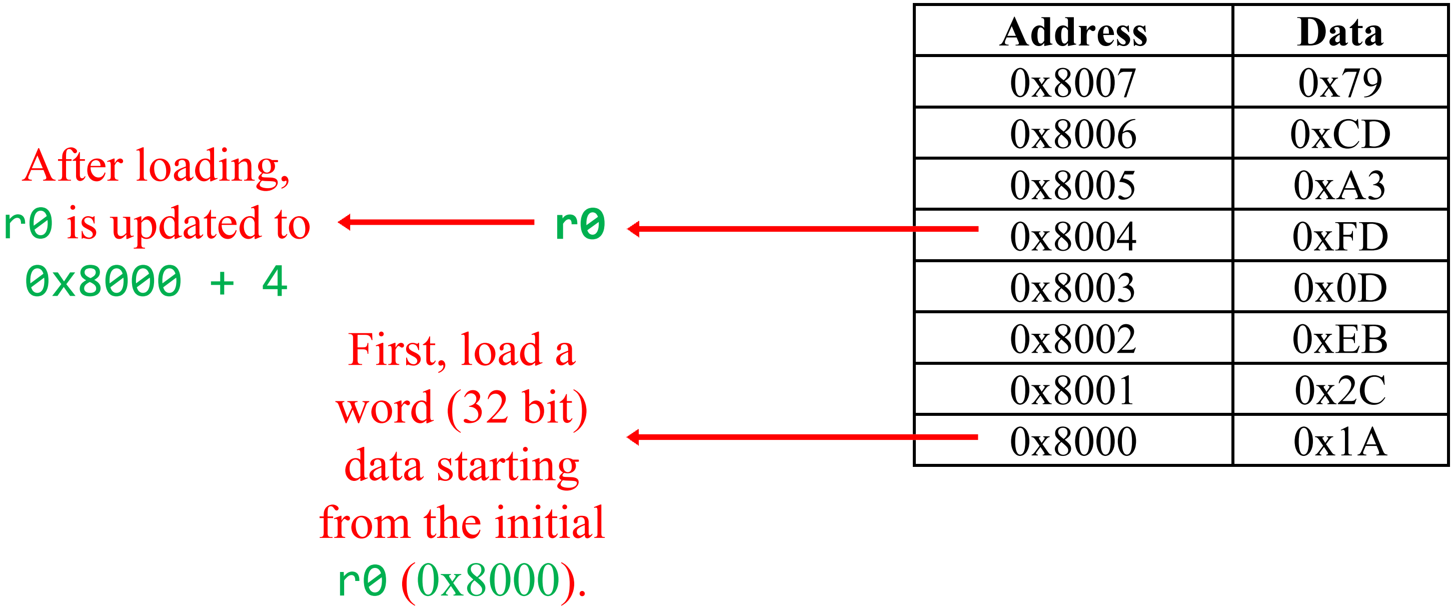
It means we are going to load the memory data from address *`r0`* into *`r1`*.

After loading, `r0` is updated to become `r0 + 4`.

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0], #4`



After loading, r0 is updated to 0x8000 + 4

r0

First, load a word (32 bit) data starting from the initial r0 (0x8000).

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0], #4`

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

After loading, `r0` is updated to `0x8000 + 4` ← `r0` ← 0x8004

First, load a word (32 bit) data starting from the initial `r0` (0x8000). ← 0x8000

0x0D → **Most significant**

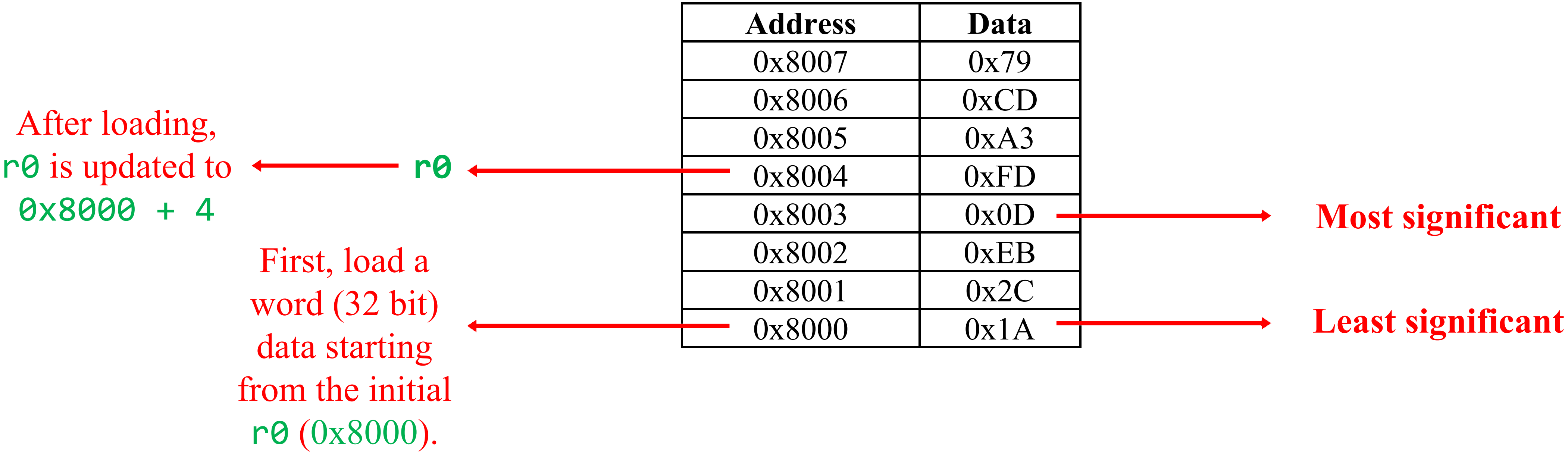
0x1A → **Least significant**

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0], #4`

Solution:

```
r0 = 0x8004 ——> r0 = r0 + 4
r1 = 0D EB 2C 1A
```



Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

After loading, `r0` is updated to `0x8000 + 4`

`r0`

First, load a word (32 bit) data starting from the initial `r0` (0x8000).

Most significant

Least significant

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0, #4]!`

In this case, we are accessing the memory data using the *pre-index with update mode.*

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0, #4]!`

In this case, we are accessing the memory data using the *pre-index with update mode.*

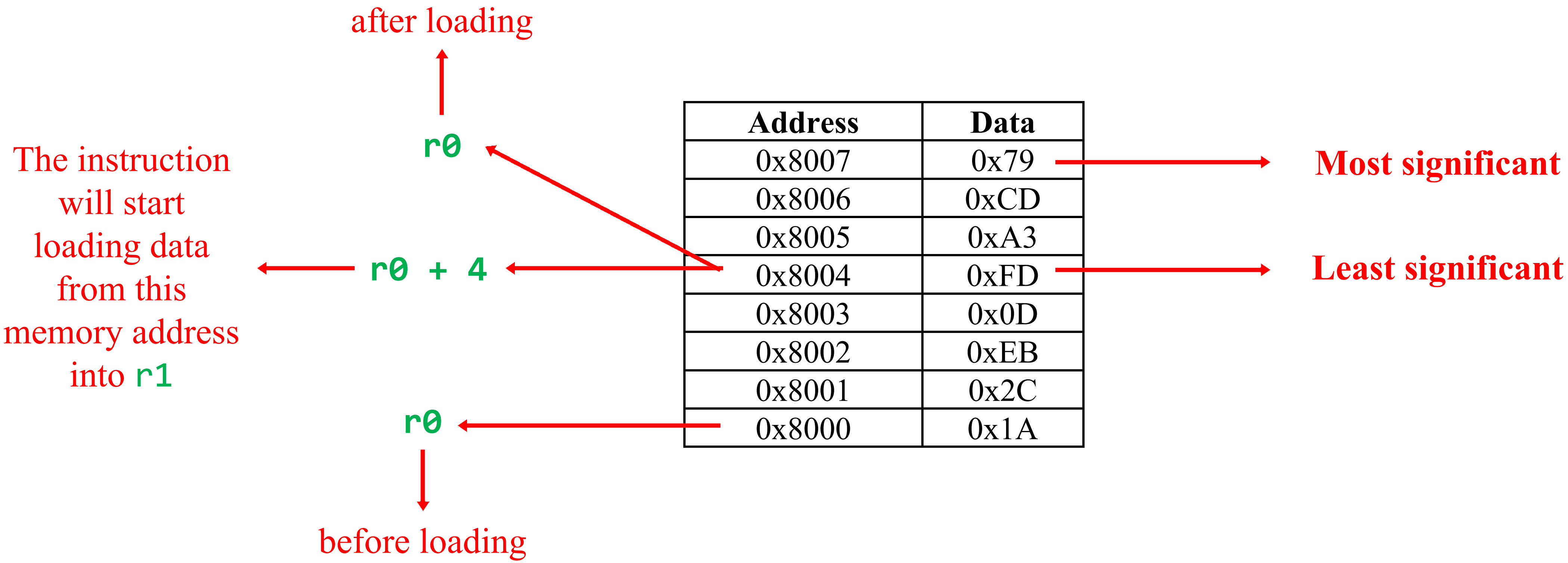
It means we are going to load the memory data from address *`r0 + 4`* into *`r1`*.

After loading, `r0` is also updated to become `r0 + 4`.

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0, #4]!`



after loading

r0

The instruction will start loading data from this memory address into r1

r0 + 4

r0

before loading

Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

Most significant

Least significant

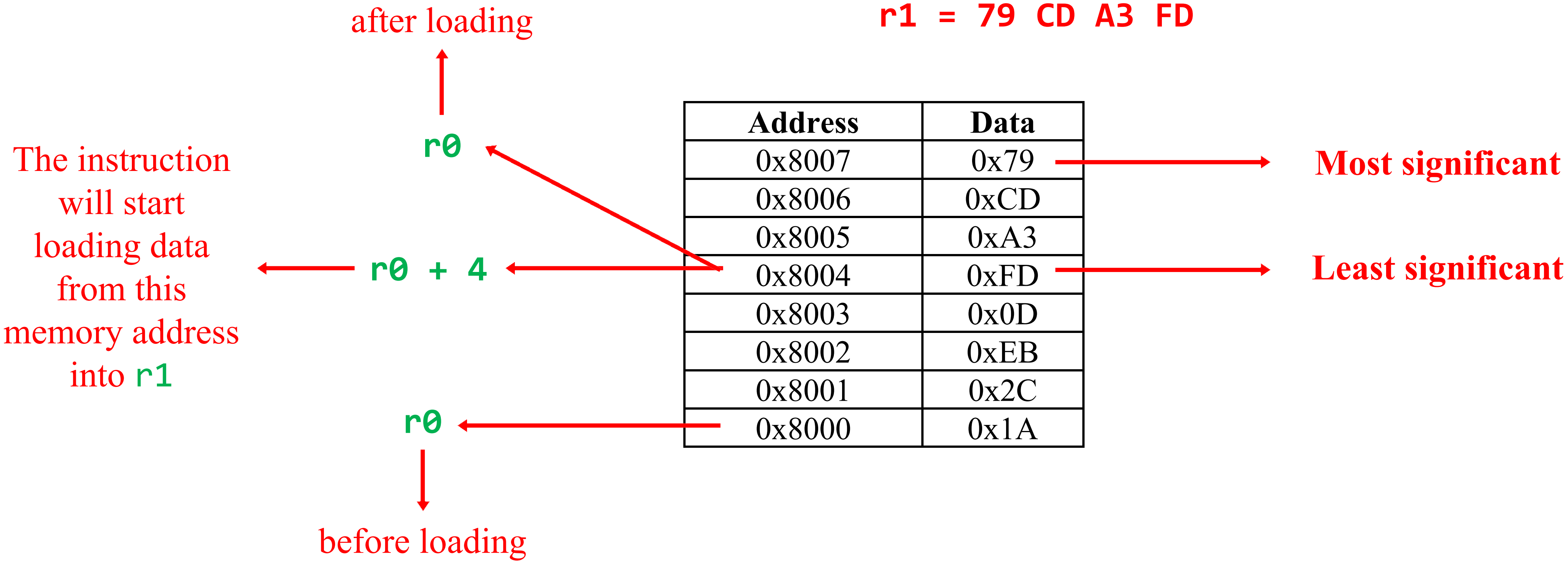
Homework 3 – Question 4 – Item B

- Suppose `r0 = 0x8000`, and the memory layout is as follows:
  - `LDR r1, [r0, #4]!`

Solution:

`r0 = 0x8004` → `r0 = r0 + 4`

`r1 = 79 CD A3 FD`



Address	Data
0x8007	0x79
0x8006	0xCD
0x8005	0xA3
0x8004	0xFD
0x8003	0x0D
0x8002	0xEB
0x8001	0x2C
0x8000	0x1A

Homework 4

Homework 4 – Chapter 5 – Exercise 3

- Suppose `r0 = 0x20000000` and `r1 = 0x12345678`. All bytes in memory are initialized to `0x00`. Suppose the following assembly program has been executed successfully. Draw a table to show the memory value if the processor uses little endian.

```
STR r1, [r0], #4
STR r1, [r0, #4]!
STR r1, [r0, 4]
```

Homework 4 – Chapter 5 – Exercise 3

- Suppose `r0 = 0x20000000` and `r1 = 0x12345678`. All bytes in memory are initialized to `0x00`. *Suppose the following assembly program has been executed successfully*. Draw a table to show the memory value if the processor uses little endian.

```
STR r1, [r0], #4

STR r1, [r0, #4]!

STR r1, [r0, 4]
```

In this case, each line of code is NOT independent of each other. We should consider these three lines as a single program.

Homework 4 – Chapter 5 – Exercise 3

- Suppose `r0 = 0x20000000` and `r1 = 0x12345678`. *All bytes in memory are initialized to `0x00`*. Suppose the following assembly program has been executed successfully. Draw a table to show the memory value if the processor uses little endian.

```
STR r1, [r0], #4
STR r1, [r0, #4]!
STR r1, [r0, 4]
```

In this case, all memory positions will start empty or equal to 0x00.

Address	Data
0x20000007	
0x20000006	
0x20000005	
0x20000004	
0x20000003	
0x20000002	
0x20000001	
0x20000000	

Homework 4 – Chapter 5 – Exercise 3

- Suppose `r0 = 0x20000000` and `r1 = 0x12345678`. All bytes in memory are initialized to `0x00`. Suppose the following assembly program has been executed successfully. Draw a table to show the memory value if the processor uses little endian.

Another important thing to note is that `r1` already contains some data and we are going to store this data back in the memory using the STR instruction.

```
STR r1, [r0], #4
STR r1, [r0, #4]!
STR r1, [r0, 4]
```

Homework 4 – Chapter 5 – Exercise 3

- Suppose `r0 = 0x20000000` and `r1 = 0x12345678`. All bytes in memory are initialized to `0x00`. Suppose the following assembly program has been executed successfully. Draw a table to show the memory value if the processor uses little endian.

```
STR r1, [r0], #4
STR r1, [r0, #4]!
STR r1, [r0, 4]
```

Post-Index:

First, it stores `r1` in the memory address of `r0`. Then, it updates `r0` to `r0 + 4`.

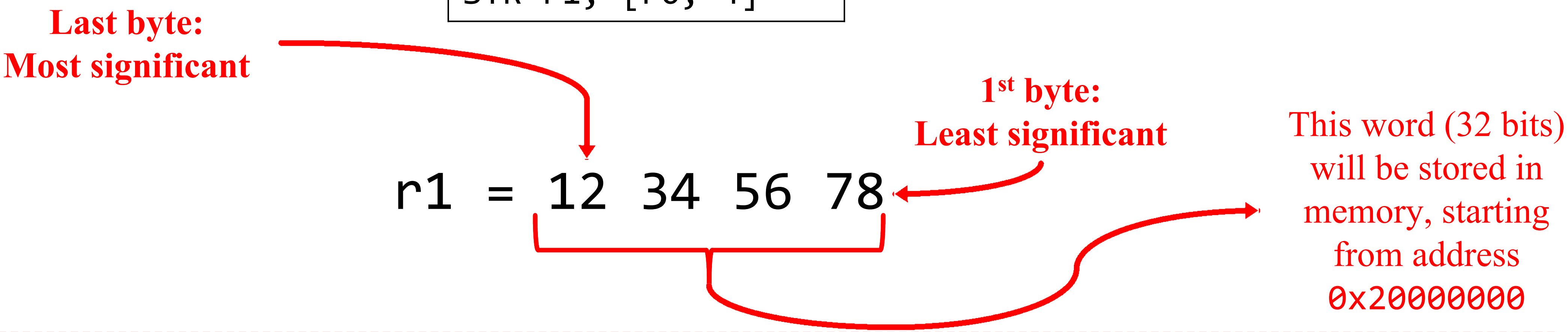
Homework 4 – Chapter 5 – Exercise 3

- Suppose `r0 = 0x20000000` and `r1 = 0x12345678`. All bytes in memory are initialized to `0x00`. Suppose the following assembly program has been executed successfully. Draw a table to show the memory value if the processor uses little endian.

```
STR r1, [r0], #4

STR r1, [r0, #4]!

STR r1, [r0, 4]
```

Post-Index:

First, it stores `r1` in the memory address of `r0`. Then, it updates `r0` to `r0 + 4`.



This word (32 bits) will be stored in memory, starting from address `0x20000000`

Homework 4 – Chapter 5 – Exercise 3

- Suppose `r0 = 0x20000000` and `r1 = 0x12345678`. All bytes in memory are initialized to `0x00`. Suppose the following assembly program has been executed successfully. Draw a table to show the memory value if the processor uses little endian.

Address	Data
0x20000007	
0x20000006	
0x20000005	
0x20000004	
0x20000003	12
0x20000002	34
0x20000001	56
0x20000000	78

r0

Most significant

Least significant

```
STR r1, [r0], #4

STR r1, [r0, #4]!

STR r1, [r0, 4]
```

Post-Index:

First, it stores `r1` in the memory address of `r0`. Then, it updates `r0` to `r0 + 4`.

Homework 4 – Chapter 5 – Exercise 3

- Suppose `r0 = 0x20000000` and `r1 = 0x12345678`. All bytes in memory are initialized to `0x00`. Suppose the following assembly program has been executed successfully. Draw a table to show the memory value if the processor uses little endian.

	Address	Data
	0x20000007	
	0x20000006	
	0x20000005	
r0 →	0x20000004	
	0x20000003	12
	0x20000002	34
	0x20000001	56
	0x20000000	78

```
STR r1, [r0], #4

STR r1, [r0, #4]!

STR r1, [r0, 4]
```

Post-Index:
First, it stores `r1` in the memory address of `r0`. Then, it updates `r0` to `r0 + 4`.

Now, `r0` is updated to `0x20000004`

Homework 4 – Chapter 5 – Exercise 3

- Suppose `r0 = 0x20000000` and `r1 = 0x12345678`. All bytes in memory are initialized to `0x00`. Suppose the following assembly program has been executed successfully. Draw a table to show the memory value if the processor uses little endian.

	Address	Data
	0x20000007	
	0x20000006	
	0x20000005	
r0 →	0x20000004	
	0x20000003	12
	0x20000002	34
	0x20000001	56
	0x20000000	78

```
STR r1, [r0], #4
STR r1, [r0, #4]!
STR r1, [r0, 4]
```

Pre-index with update:

First, it stores `r1` in the memory address of `r0 + 4`. Then, it updates `r0` to `r0 + 4`.

Homework 4 – Chapter 5 – Exercise 3

```
STR r1, [r0], #4
STR r1, [r0, #4]!
STR r1, [r0, 4]
```

Pre-index with update:
First, it stores `r1` in the memory address of `r0 + 4`. Then, it updates `r0` to `r0 + 4`.

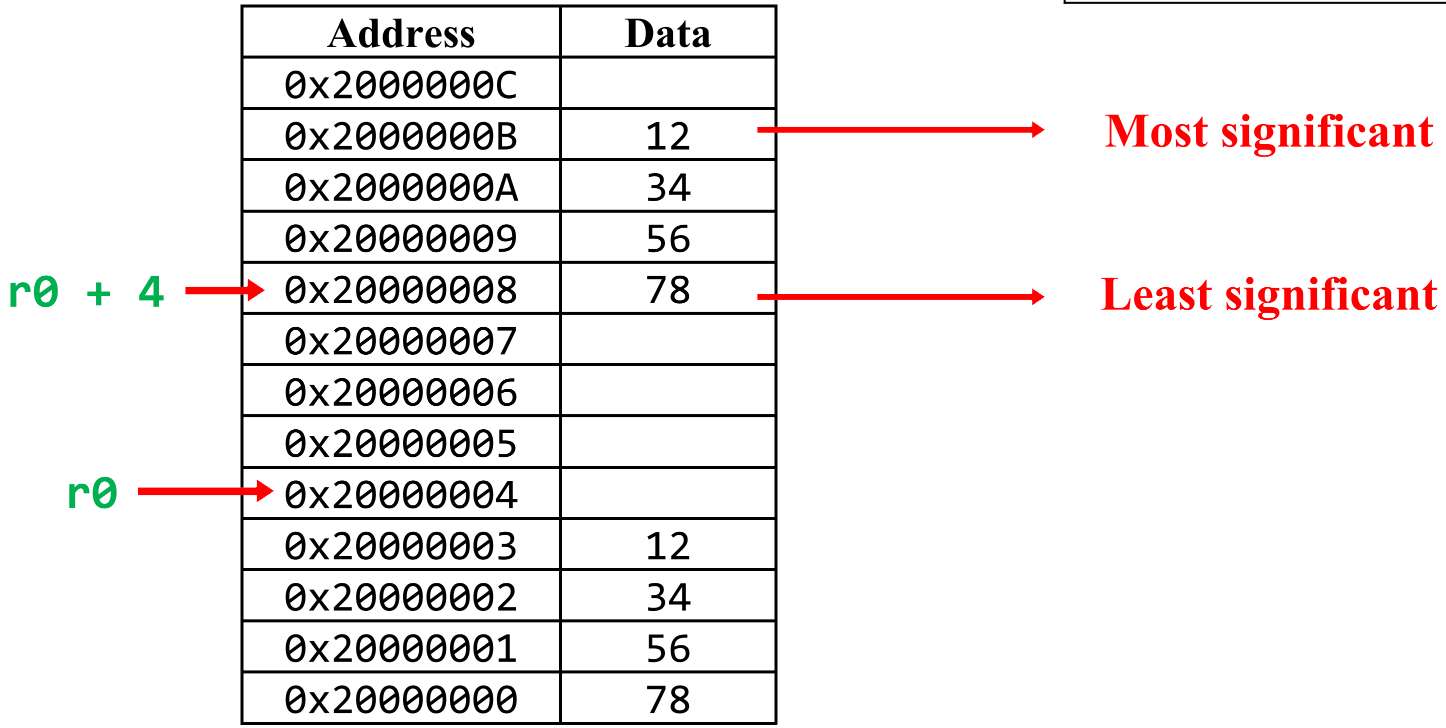
	Address	Data
	0x20000012	
	0x20000011	
	0x20000010	
	0x20000009	
r0 + 4 →	0x20000008	
	0x20000007	
	0x20000006	
	0x20000005	
r0 →	0x20000004	
	0x20000003	12
	0x20000002	34
	0x20000001	56
	0x20000000	78

Homework 4 – Chapter 5 – Exercise 3

```
STR r1, [r0], #4

STR r1, [r0, #4]!

STR r1, [r0, 4]
```

Pre-index with update:

First, it stores `r1` in the memory address of `r0 + 4`. Then, it updates `r0` to `r0 + 4`.



Address	Data
0x2000000C	
0x2000000B	12
0x2000000A	34
0x20000009	56
0x20000008	78
0x20000007	
0x20000006	
0x20000005	
0x20000004	
0x20000003	12
0x20000002	34
0x20000001	56
0x20000000	78

Homework 4 – Chapter 5 – Exercise 3

```
STR r1, [r0], #4

STR r1, [r0, #4]!

STR r1, [r0, 4]
```

Pre-index with update:

First, it stores r1 in the memory address of r0 + 4. Then, it updates r0 to r0 + 4.

	Address	Data
	0x2000000C	
	0x2000000B	12
	0x2000000A	34
	0x20000009	56
r0	0x20000008	78
	0x20000007	
	0x20000006	
	0x20000005	
	0x20000004	
	0x20000003	12
	0x20000002	34
	0x20000001	56
	0x20000000	78

Now, r0 is updated to 0x20000008

Homework 4 – Chapter 5 – Exercise 3

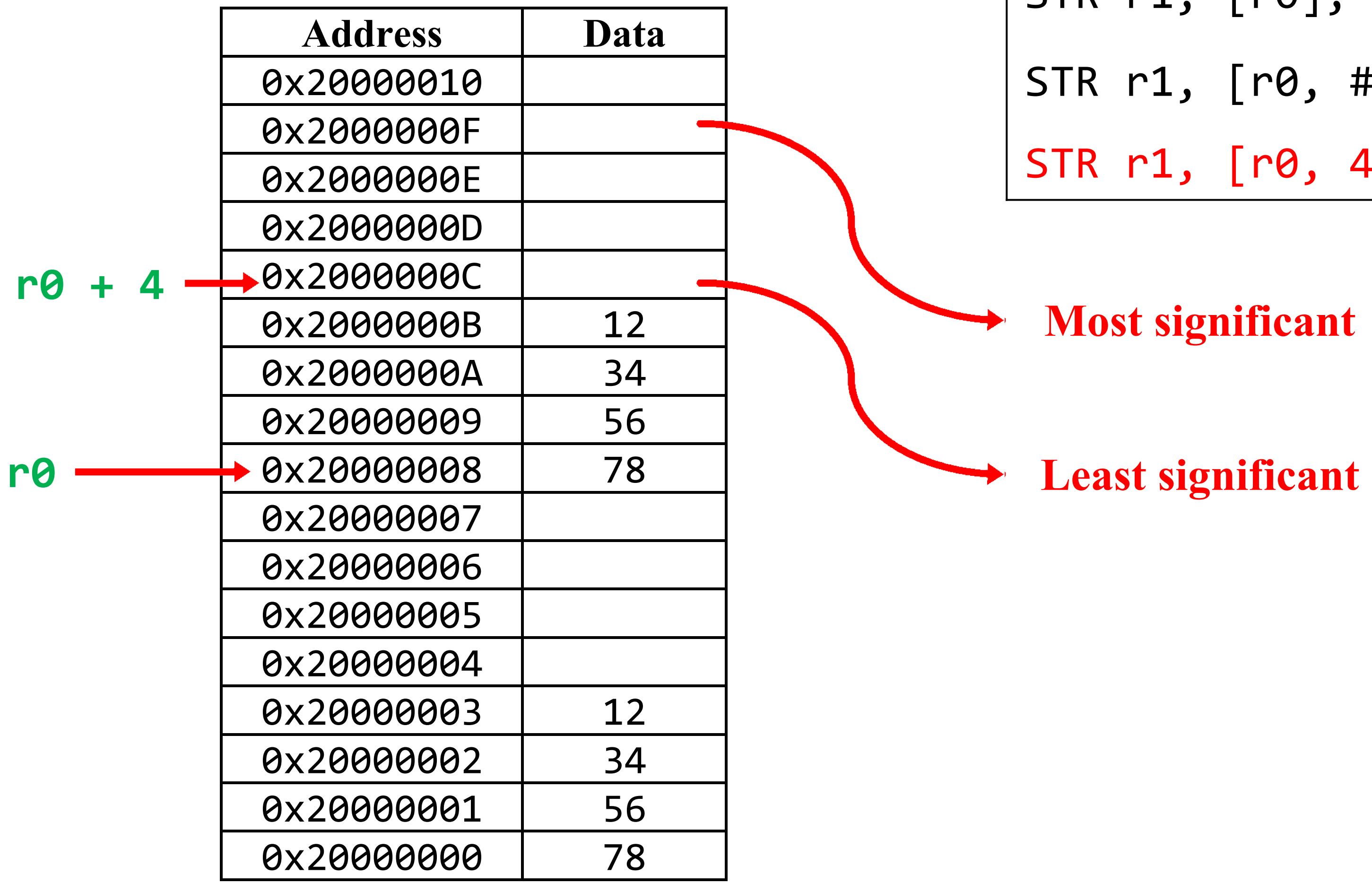
```
STR r1, [r0], #4

STR r1, [r0, #4]!

STR r1, [r0, 4]
```

Pre-index:

First, it stores `r1` in the memory address of `r0 + 4`, and `r0` remains unchanged.

	Address	Data
	0x2000000C	
	0x2000000B	12
	0x2000000A	34
	0x20000009	56
r0 →	0x20000008	78
	0x20000007	
	0x20000006	
	0x20000005	
	0x20000004	
	0x20000003	12
	0x20000002	34
	0x20000001	56
	0x20000000	78

Homework 4 – Chapter 5 – Exercise 3



Address	Data
0x20000010	
0x2000000F	
0x2000000E	
0x2000000D	
0x2000000C	
0x2000000B	12
0x2000000A	34
0x20000009	56
0x20000008	78
0x20000007	
0x20000006	
0x20000005	
0x20000004	
0x20000003	12
0x20000002	34
0x20000001	56
0x20000000	78

```
STR r1, [r0], #4

STR r1, [r0, #4]!

STR r1, [r0, 4]
```

Pre-index:

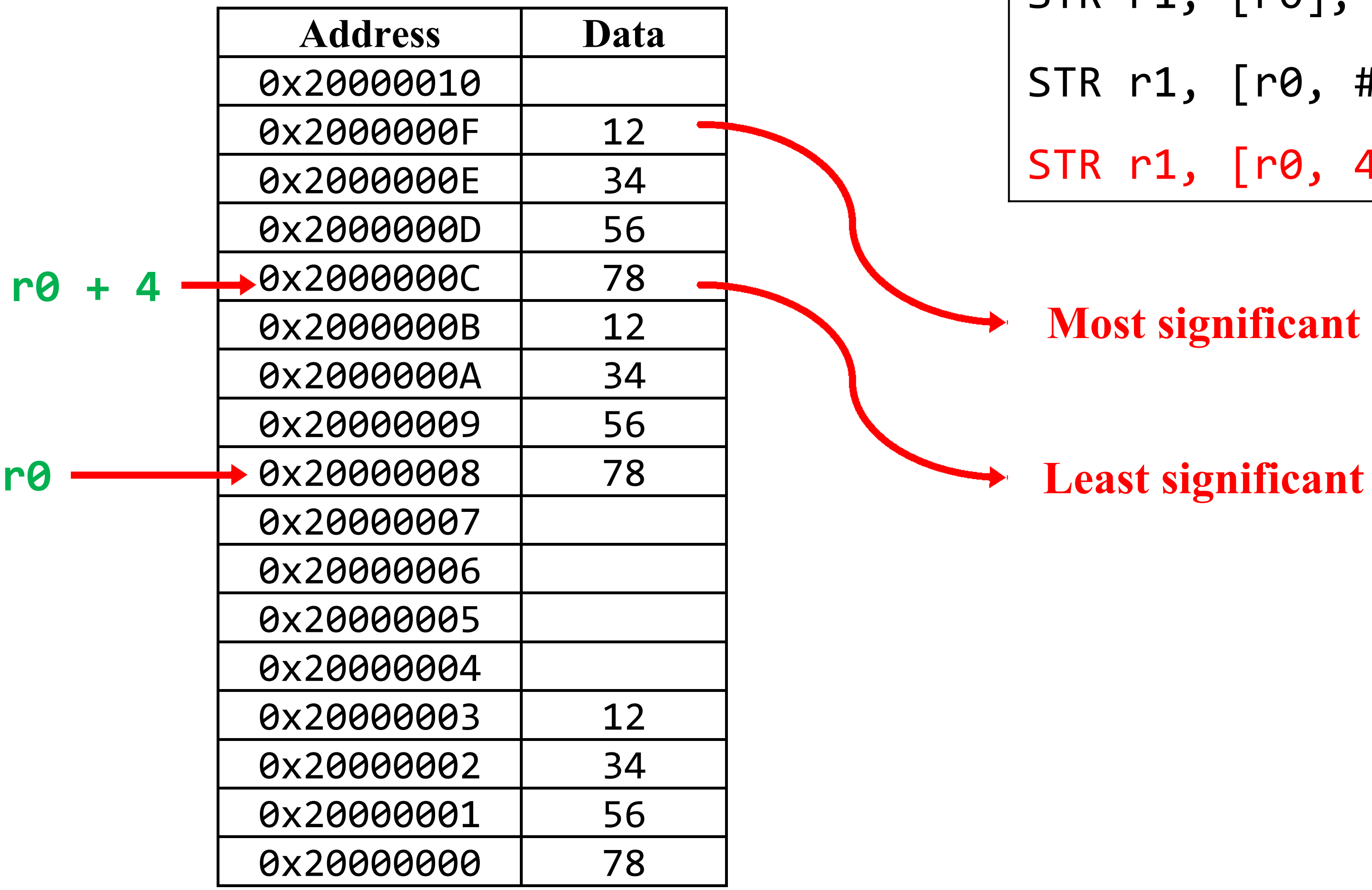
First, it stores `r1` in the memory address of `r0 + 4`, and `r0` remains unchanged.

Homework 4 – Chapter 5 – Exercise 3

```
STR r1, [r0], #4

STR r1, [r0, #4]!

STR r1, [r0, 4]
```

Pre-index:

First, it stores `r1` in the memory address of `r0 + 4`, and `r0` remains unchanged.



Address	Data
0x20000010	
0x2000000F	12
0x2000000E	34
0x2000000D	56
0x2000000C	78
0x2000000B	12
0x2000000A	34
0x20000009	56
0x20000008	78
0x20000007	
0x20000006	
0x20000005	
0x20000004	
0x20000003	12
0x20000002	34
0x20000001	56
0x20000000	78

Homework 4 – Chapter 5 – Exercise 3

```
STR r1, [r0], #4

STR r1, [r0, #4]!

STR r1, [r0, 4]
```

Pre-index:
First, it stores r1 in the memory address of r0 + 4, and r0 remains unchanged.

Address	Data
0x20000010	
0x2000000F	12
0x2000000E	34
0x2000000D	56
0x2000000C	78
0x2000000B	12
0x2000000A	34
0x20000009	56
0x20000008	78
0x20000007	
0x20000006	
0x20000005	
0x20000004	
0x20000003	12
0x20000002	34
0x20000001	56
0x20000000	78

r0

r0 remains unchanged and equal to 0x20000008

Homework 4 – Chapter 6 – Exercise 1

- Translate the following code into a C program and explain what it does.

```
        MOV r2, #1
        MOV r1, #1

loop    CMP r1, r0
        BGT done
        MUL r2, r1, r2
        ADD r1, r1, #1
        B   loop

done    MOV r0, r2
```

Homework 4 – Chapter 6 – Exercise 1

- Translate the following code into a C program and explain what it does.

```
       MOV r2, #1 ; r2 = 1
       MOV r1, #1 ; r1 = 1

loop   CMP r1, r0 ; compare r1 to r0. In this case, r0 is our input variable.
       BGT done   ; If r1 is greater than r0 go to “done”
       ; If r1 is less or equal to r0 the following lines will run.
       MUL r2, r1, r2 ; r2 = r2*r1
       ADD r1, r1, #1 ; r1 = r1 + 1
       B   loop       ; go back to “loop”

done   MOV r0, r2     ; When r1 is greater than r0, store the result r2 in r0
```

Homework 4 – Chapter 6 – Exercise 1

- Translate the following code into a C program and explain what it does.

So, the easiest way to know what is this program computing is to plug in some numbers. We know that `r0` is the input and `r2` is the output.

```
        MOV r2, #1
        MOV r1, #1

loop    CMP r1, r0
        BGT done
        MUL r2, r1, r2
        ADD r1, r1, #1
        B   loop

done    MOV r0, r2
```

```
If r0 = 0 → r2 = 1   → r1 = 1 (the loop will not run)

If r0 = 1 → r2 = 1*1 = 1   → r1 = 1 + 1 = 2

If r0 = 2 → r2 = 1*1 = 1 → r1 = 1 + 1 = 2
             r2 = 1*2 = 2 → r1 = 2 + 1 = 3

If r0 = 3 → r2 = 1*1 = 1 → r1 = 1 + 1 = 2
             r2 = 1*2 = 2 → r1 = 2 + 1 = 3
             r2 = 3*2 = 6 → r1 = 3 + 1 = 4

If r0 = 4 → r2 = 1*1 = 1 → r1 = 1 + 1 = 2
             r2 = 1*2 = 2 → r1 = 2 + 1 = 3
             r2 = 3*2 = 6 → r1 = 3 + 1 = 4
             r2 = 6*4 = 24 → r1 = 4 + 1 = 5
```

Homework 4 – Chapter 6 – Exercise 1

- Translate the following code into a C program and explain what it does.

So, the easiest way to know what is this program computing is to plug in some numbers. We know that `r0` is the input and `r2` is the output.

```
        MOV r2, #1
        MOV r1, #1

loop    CMP r1, r0
        BGT done
        MUL r2, r1, r2
        ADD r1, r1, #1
        B   loop

done    MOV r0, r2
```

```
If r0 = 0 → r2 = 1   → r1 = 1

If r0 = 1 → r2 = 1*1 = 1   → r1 = 1 + 1 = 2

If r0 = 2 → r2 = 1*1 = 1 → r1 = 1 + 1 = 2
            r2 = 1*2 = 2 → r1 = 2 + 1 = 3

If r0 = 3 → r2 = 1*1 = 1 → r1 = 1 + 1 = 2
            r2 = 1*2 = 2 → r1 = 2 + 1 = 3
            r2 = 3*2 = 6 → r1 = 3 + 1 = 4

If r0 = 4 → r2 = 1*1 = 1 → r1 = 1 + 1 = 2
            r2 = 1*2 = 2 → r1 = 2 + 1 = 3
            r2 = 3*2 = 6 → r1 = 3 + 1 = 4
            r2 = 6*4 = 24 → r1 = 4 + 1 = 5
```

The code is computing the factorial of `r0`.

Homework 4 – Chapter 6 – Exercise 1

- Translate the following code into a C program and explain what it does.

```
       MOV r2, #1 ; r2 = 1
       MOV r1, #1 ; r1 = 1

loop   CMP r1, r0 ; compare r1 to r0. In this case, r0 is our input variable.
       BGT done   ; If r1 is greater than r0 go to “done”
       ; If r1 is less or equal to r0 the following lines will run.
       MUL r2, r1, r2 ; r2 = r2*r1
       ADD r1, r1, #1 ; r1 = r1 + 1
       B   loop       ; go back to “loop”

done   MOV r0, r2     ; When r1 is greater than r0, store the result r2 in r0
```

Variables are initialized.

Homework 4 – Chapter 6 – Exercise 1

- Translate the following code into a C program and explain what it does.

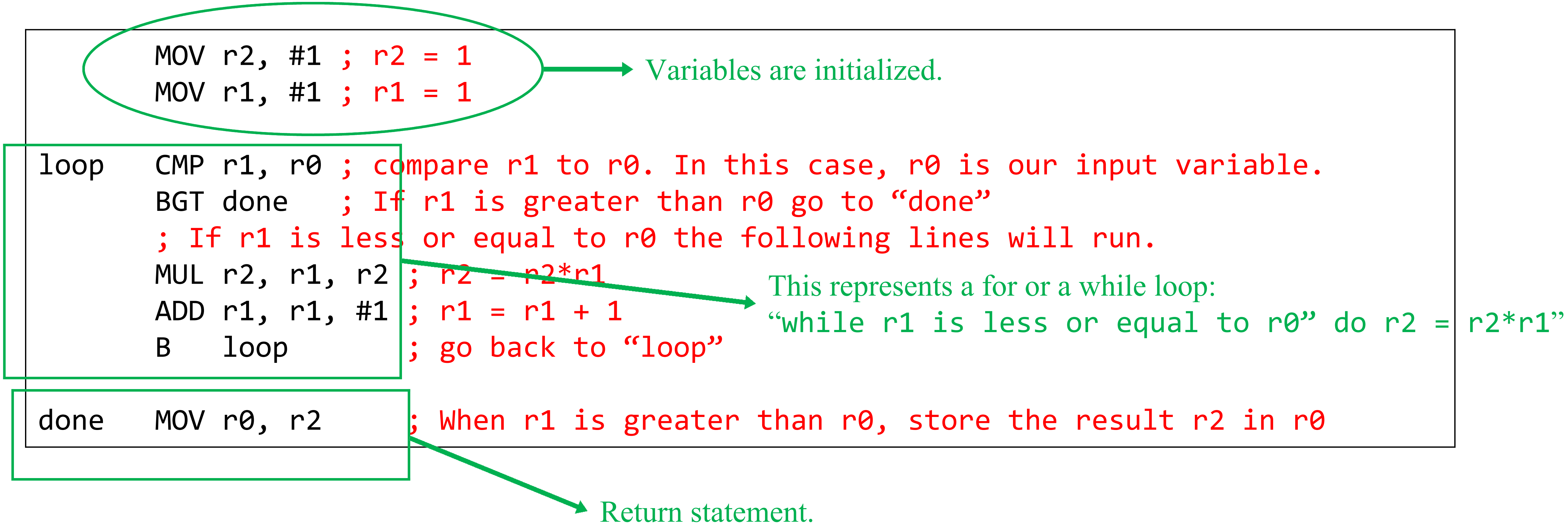
```
        MOV r2, #1 ; r2 = 1
        MOV r1, #1 ; r1 = 1

loop    CMP r1, r0 ; compare r1 to r0. In this case, r0 is our input variable.
        BGT done   ; If r1 is greater than r0 go to “done”
        ; If r1 is less or equal to r0 the following lines will run.
        MUL r2, r1, r2 ; r2 = r2*r1
        ADD r1, r1, #1 ; r1 = r1 + 1
        B   loop       ; go back to “loop”

done    MOV r0, r2     ; When r1 is greater than r0, store the result r2 in r0
```

Variables are initialized.

This represents a for or a while loop:
“while r1 is less or equal to r0” do r2 = r2*r1”

Homework 4 – Chapter 6 – Exercise 1

- Translate the following code into a C program and explain what it does.



```
        MOV r2, #1 ; r2 = 1
        MOV r1, #1 ; r1 = 1

loop    CMP r1, r0 ; compare r1 to r0. In this case, r0 is our input variable.
        BGT done   ; If r1 is greater than r0 go to “done”
        ; If r1 is less or equal to r0 the following lines will run.
        MUL r2, r1, r2 ; r2 = r2*r1
        ADD r1, r1, #1 ; r1 = r1 + 1
        B   loop       ; go back to “loop”

done    MOV r0, r2     ; When r1 is greater than r0, store the result r2 in r0
```

Variables are initialized.

This represents a for or a while loop:
“while r1 is less or equal to r0” do r2 = r2*r1”

Return statement.

Homework 4 – Chapter 6 – Exercise 1

- Translate the following code into a C program and explain what it does.

```
        MOV r2, #1
        MOV r1, #1

loop    CMP r1, r0
        BGT done
        MUL r2, r1, r2
        ADD r1, r1, #1
        B   loop

done    MOV r0, r2
```

```
int factorial(int r0){
       int r1;
       int r2 = 1;

       for(r1 = 1; r1 <= r0; r1++){
              r2 = r2*r1;
       }

       return r2;
}
```

Homework 4 – Chapter 6 – Exercise 4

```
        AREA myData, DATA
array  DCD 2, 4, 7, 3, 1, 2, 10, 11, 5, 13
size   DCD 10
```

Homework 4 – Chapter 6 – Exercise 4

Your code should perform this operation!

The memory addresses of our array can be accessed by using these labels. So, we don't need to know the exactly memory location of the array elements.

```
        AREA myData, DATA
array   DCD 2, 4, 7, 3, 1, 2, 10, 11, 5, 13
size    DCD 10
```

This is your a_i's.

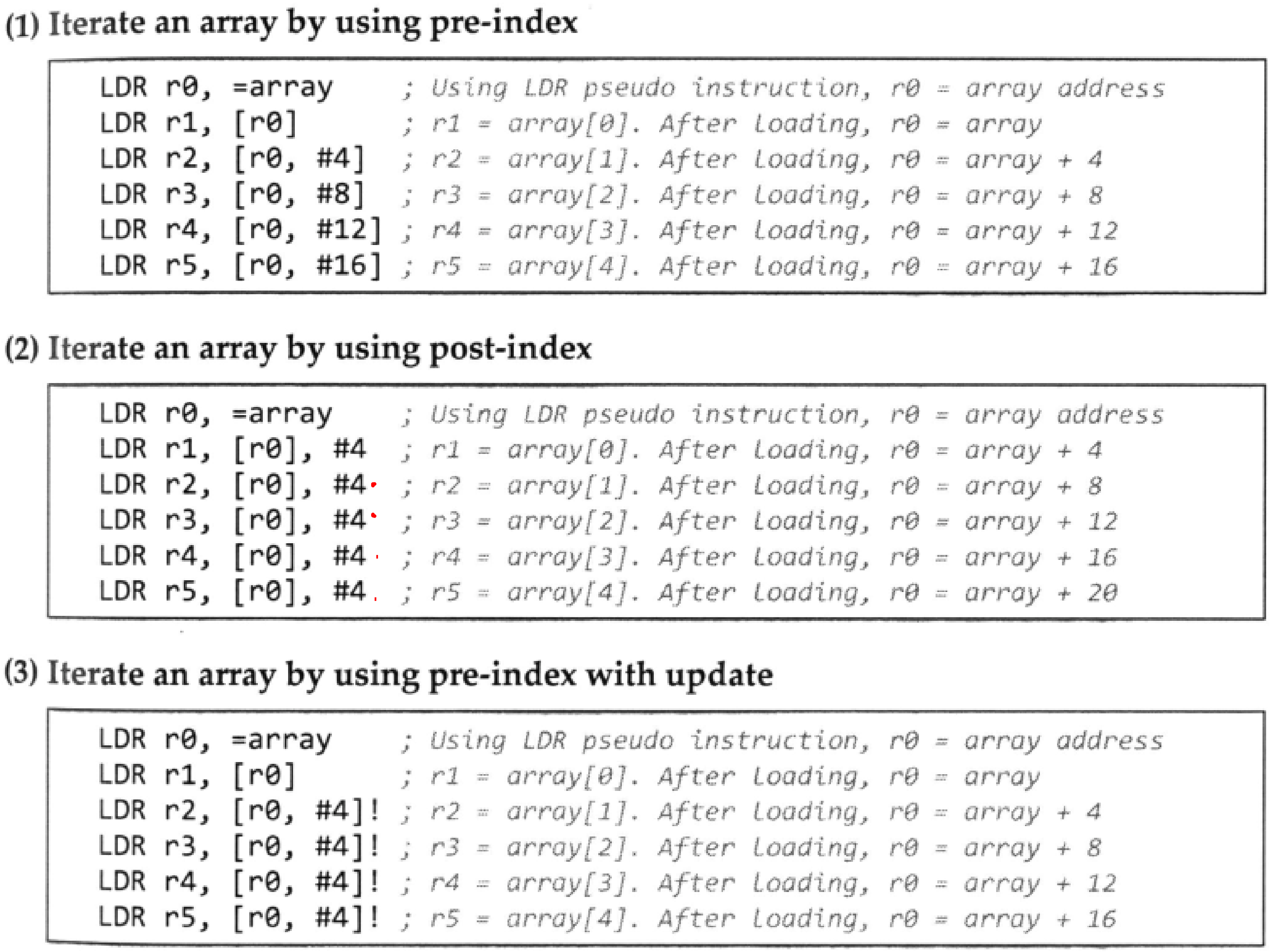
Homework 4 – Chapter 6 – Exercise 4

```
        AREA myData, DATA
array  DCD 2, 4, 7, 3, 1, 2, 10, 11, 5, 13
size   DCD 10
```

The summation indicates to us that we are going to perform some kind of loop.

Homework 4 – Chapter 6 – Exercise 4

Accessing an array in assembly can be found in the textbook section 5.4.4, page 105.



(1) Iterate an array by using pre-index

```
LDR r0, =array     ; Using LDR pseudo instruction, r0 = array address
LDR r1, [r0]       ; r1 = array[0]. After loading, r0 = array
LDR r2, [r0, #4]   ; r2 = array[1]. After loading, r0 = array + 4
LDR r3, [r0, #8]   ; r3 = array[2]. After loading, r0 = array + 8
LDR r4, [r0, #12]  ; r4 = array[3]. After loading, r0 = array + 12
LDR r5, [r0, #16]  ; r5 = array[4]. After loading, r0 = array + 16
```

(2) Iterate an array by using post-index

```
LDR r0, =array     ; Using LDR pseudo instruction, r0 = array address
LDR r1, [r0], #4   ; r1 = array[0]. After loading, r0 = array + 4
LDR r2, [r0], #4   ; r2 = array[1]. After loading, r0 = array + 8
LDR r3, [r0], #4   ; r3 = array[2]. After loading, r0 = array + 12
LDR r4, [r0], #4   ; r4 = array[3]. After loading, r0 = array + 16
LDR r5, [r0], #4   ; r5 = array[4]. After loading, r0 = array + 20
```

(3) Iterate an array by using pre-index with update

```
LDR r0, =array     ; Using LDR pseudo instruction, r0 = array address
LDR r1, [r0]       ; r1 = array[0]. After loading, r0 = array
LDR r2, [r0, #4]!  ; r2 = array[1]. After loading, r0 = array + 4
LDR r3, [r0, #4]!  ; r3 = array[2]. After loading, r0 = array + 8
LDR r4, [r0, #4]!  ; r4 = array[3]. After loading, r0 = array + 12
LDR r5, [r0, #4]!  ; r5 = array[4]. After loading, r0 = array + 16
```

Homework 4 – Chapter 6 – Exercise 4

```
             AREA myData, DATA, READWRITE
             ALIGN
array        DCD 2, 4, 7, 3, 1, 2, 10, 11, 5, 13
size         DCD 10

             AREA myCode, CODE, READONLY
             EXPORT __main
             ALIGN
             ENTRY

__main       PROC
             LDR r0, =size
             LDR r1, [r0]              ; r1 = 10
             LDR r0, =array
             LDR r2, [r0], #4          ; r2 = a_1 --> get the first position of the array

             MOV r3, #1                ; We are going to use r3 as our counter in the loop
             MOV r4, #0                ; The summation will be stored in r4

loop         CMP r3, r1                ; Loop while r3 is less than r1 (10)
             BGT done                  ; If r3 is greater than r1 (10), we're done

             MUL r5, r2, r2            ; r5 = a_i * a_i
             MUL r5, r5, r2            ; r5 = r5 * a_i  --> r5 = a_i*a_i*a_i
             ADD r4, r5                ; Add the cube operation to the summation (r4)

             LDR r2, [r0], #4          ; get the next array element

             ADD r3, #1                ; Increment r3 by 1

             B loop
             ENDP

done         B done                    ; dead loop

             END
```

Solution using ARM Assembly.
The one from the book.

Only works with the Keil uVision IDE!

Note: This IDE is not being used in our labs!

Homework 4 – Chapter 6 – Exercise 4

```
.syntax unified
.cpu cortex-m4
.fpu softvfp
.thumb

.section     .data
array:       .word 2, 4, 7, 3, 1, 2, 10, 11, 5, 13
size:        .word 10

.section     .text
.global      main

main:
        LDR r0, =size
        LDR r1, [r0]      // r1 = 10

        LDR r0, =array
        LDR r2, [r0], #4  // r2 = a_1 --> get the first position of the array

        MOV r3, #1        // We are going to use r3 as our counter in the loop
        MOV r4, #0        // The summation will be stored in r4

loop:
        CMP r3, r1        // Loop while r3 is less than r1 (10)
        BGT done          // If r3 is greater than r1 (10), we are done
        MUL r5, r2, r2    // r5 = a_i * a_i
        MUL r5, r5, r2    // r5 = r5 * a_i
        ADD r4, r5        // Add the cube operation to the summation (r4)
        LDR r2, [r0], #4  // get the next array element
        ADD r3, #1        // Increment r3 by 1

        B loop

done:
        B done            // dead loop
```

Solution using GNU Assembly.

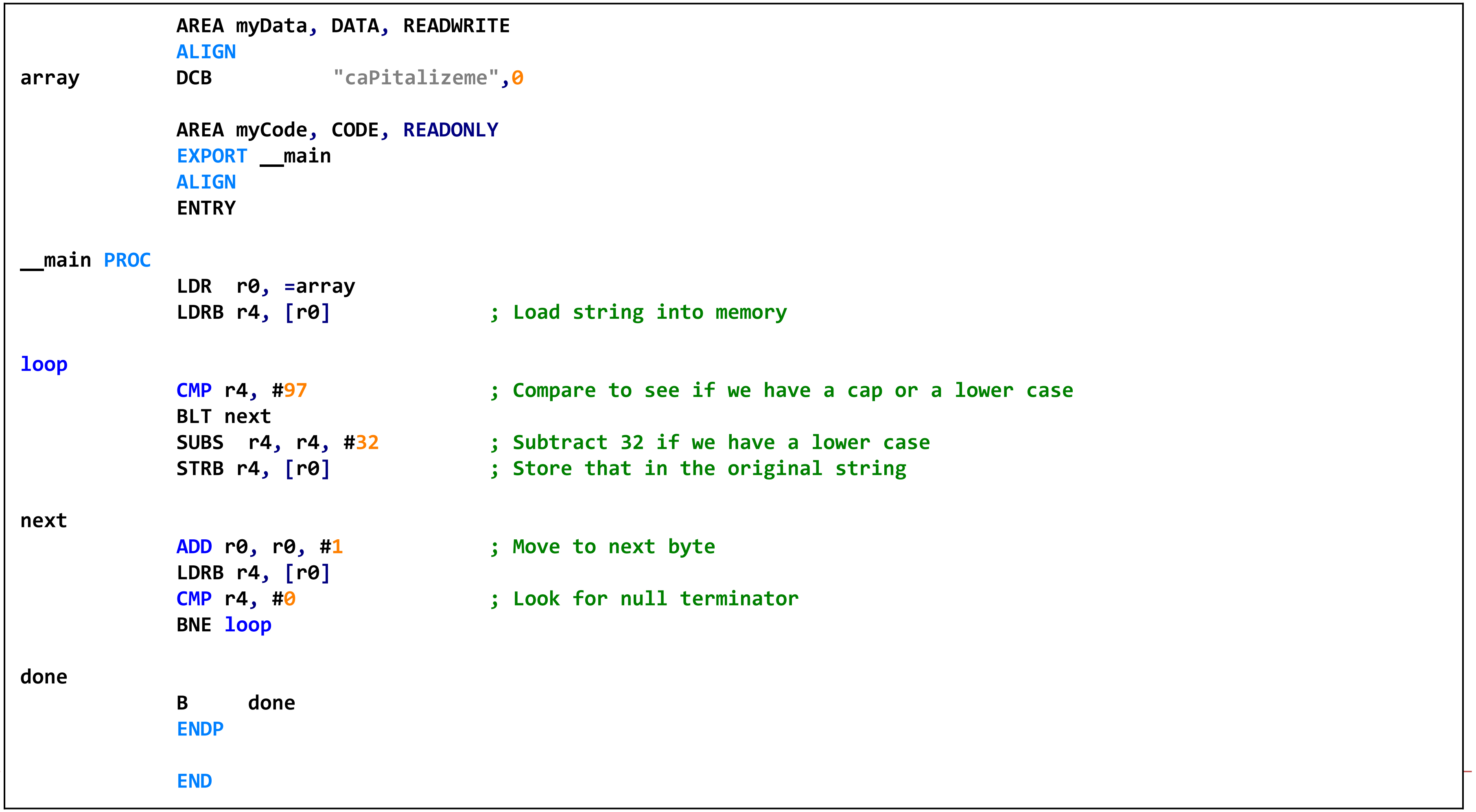
The only kind of assembly that works with the System Workbench for STM32 used in our labs!

Look the textbook Appendix A to learn more how to translate from one to another.

Chapter 7

Chapter 7 – Exercise 1

- Write an assembly program that converts all characters of a string to upper case.

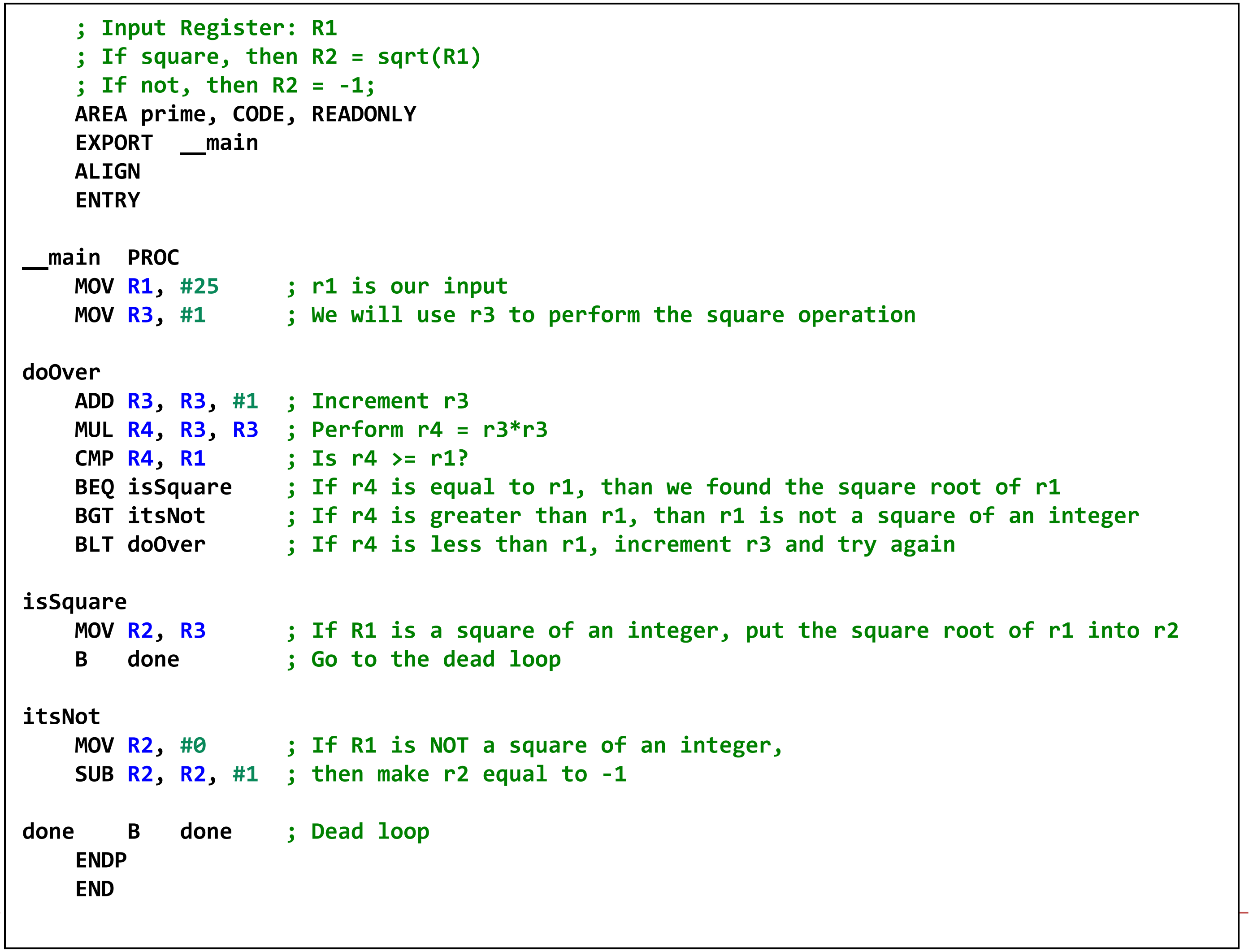


```
             AREA myData, DATA, READWRITE
             ALIGN
array        DCB           "caPitalizeme",0

             AREA myCode, CODE, READONLY
             EXPORT __main
             ALIGN
             ENTRY

__main PROC
             LDR  r0, =array
             LDRB r4, [r0]             ; Load string into memory

loop
             CMP r4, #97               ; Compare to see if we have a cap or a lower case
             BLT next
             SUBS  r4, r4, #32         ; Subtract 32 if we have a lower case
             STRB r4, [r0]             ; Store that in the original string

next
             ADD r0, r0, #1            ; Move to next byte
             LDRB r4, [r0]
             CMP r4, #0                ; Look for null terminator
             BNE loop

done
             B     done
             ENDP

             END
```

Hint:
Use the ASCII table

Chapter 7 – Exercise 9

- Write an assembly program that checks whether an integer is a square of some integer. For example, 25 = 5^2.



```
    ; Input Register: R1
    ; If square, then R2 = sqrt(R1)
    ; If not, then R2 = -1;
    AREA prime, CODE, READONLY
    EXPORT  __main
    ALIGN
    ENTRY

__main  PROC
    MOV R1, #25     ; r1 is our input
    MOV R3, #1      ; We will use r3 to perform the square operation

doOver
    ADD R3, R3, #1  ; Increment r3
    MUL R4, R3, R3  ; Perform r4 = r3*r3
    CMP R4, R1      ; Is r4 >= r1?
    BEQ isSquare    ; If r4 is equal to r1, than we found the square root of r1
    BGT itsNot      ; If r4 is greater than r1, than r1 is not a square of an integer
    BLT doOver      ; If r4 is less than r1, increment r3 and try again

isSquare
    MOV R2, R3      ; If R1 is a square of an integer, put the square root of r1 into r2
    B   done        ; Go to the dead loop

itsNot
    MOV R2, #0      ; If R1 is NOT a square of an integer,
    SUB R2, R2, #1  ; then make r2 equal to -1

done    B   done    ; Dead loop
    ENDP
    END
```

Homework 5

Homework 5 - Chapter 7 – Exercise 5

- Write an assembly program that removes all vowel letters (a, e, i, o, u, A, E, I, O, U) from a string.

```
.syntax unified
.cpu cortex-m4
.fpu softvfp
.thumb

.section .data
input_str:
    .ascii "The quick brown fox jumps over the lazy dog\0"

output_str:
    .ascii "\0"

.section .text
.global main

main:
    LDR r0, =input_str
    LDRB r1, [r0]           // r1 is going to be our original string

    LDR r3, =output_str
    LDRB r2, [r3]           // r2 is going to be our destination string

checkIsLetter:
    CMP r1, #0x00
    BEQ almost_done         // If r1 is equal to 0x00, this is the end of the string

    CMP r1, #0x41
    BLT nextChar_withCopy   // If it is less than 0x41, the char is not a letter

    CMP r1, #0x5B
    BLT capLetter           // If it is greater than or equal to 0x41 AND less than 0x5B, the char is a capitalized letter

    CMP r1, #0x61
    BLT nextChar_withCopy   // If it is greater than or equal to 0x5B AND less than 0x61, the char is not a letter

    CMP r1, #0x7B
    BLT smallLetter         // If it is greater than or equal to 0x61 AND less than 0x7B, the char is a small letter
    BGE nextChar_withCopy  // If it is greater than or equal to 0x7B, the char is not a letter
```

```
nextChar_withCopy:
    MOV r2, r1
    STRB r2, [r3]           // If it is not a vowel, just copy the char to our destination string.
    ADD r3, r3, #1          // Update the memory address of out destination char.

nextChar:
    ADD r0, r0, #1          // Move to the next char in the string
    LDRB r1, [r0]
    B checkIsLetter

capLetter:
    ADD r1, r1, #32         // If the char is a capitalized letter, convert to small letter by adding 32 (decimal)
    B checkIsLetter

smallLetter:
    CMP r1, #0x61           // r1 = 'a'
    BEQ isVowel
    CMP r1, #0x65           // r1 = 'e'
    BEQ isVowel
    CMP r1, #0x69           // r1 = 'i'
    BEQ isVowel
    CMP r1, #0x6F           // r1 = 'o'
    BEQ isVowel
    CMP r1, #0x75           // r1 = 'u'
    BEQ isVowel
    B notVowel

isVowel:
    B nextChar

notVowel:
    B nextChar_withCopy

almost_done:
    ADD r3, r3, #1
    MOV r2, #0x00
    STRB r2, [r3]

done:
    B done              // dead loop

.end
```

Homework 5 - Chapter 7 – Exercise 7

- Write an assembly program that checks whether an unsigned number is a prime number or not.

Homework 5 - Chapter 7 – Exercise 7

```
.syntax unified
.cpu cortex-m4
.fpu softvfp
.thumb

.section .text
.global main

main:
    MOV r1, #31     // r1 will be used as our input
    MOV r2, #1      // r2 will be our result.
                    // At the end of the program:
                    //       If r2 = 1, then the number IS prime.
                    //       If r2 = 0, then the number is NOT prime.

    MOV r3, #2      // r3 will be used as a counter.

testPrime:
    CMP r3, r1
    BEQ done        // If r3 = r1, we done with the program.
    UDIV r4, r1, r3 // r4 = r1 / r3 (only the integer part)
    MUL r4, r3      // r4 = r4*r3
    CMP r4, r1
    BNE notPrimeYet
    MOV r2, #0      // The division and multiplication gave us the original number.
    B done          // Therefore, the number is NOT prime and r2 will be 0,
                    //   and we are done with the program.

notPrimeYet:
    ADD r3, #1      // Test another integer to perform the division and multiplication.
    B testPrime

done:
    B done

.end
```

Homework 5 - Chapter 7 – Exercise 12

```
.syntax unified
.cpu cortex-m4
.fpu softvfp
.thumb

.data
size:
    .word 10
array:
    .word 10,20,30,40,50,60,70,80,90,100

.text
.global main

main:
    LDR r2, =size
    LDR r2, [r2]             // r2 = size of the array
    LDR r3, =array           // r3 = memory address of the array

    // 1st) Let's compute the mean
    MOV r7, #0               // loop index
    MOV r0, #0               // summation
    B check_mean             // Let's loop over the entire array

loop_mean:
    LDR r4, [r3, r7, LSL #2] // r6 = array(i), where r7 = i
    ADD r0, r0, r4           // r0 = r0 + array(i) --> Summation
    ADD r7, r7, #1           // Update the loop index --> r7 = i + 1

check_mean:
    CMP r7, r2               // While i <= array_size,
    BLT loop_mean            //   keep summing the array elements

    // If i > array_size, summation is done.
    // Let's divide by the size of the array to obtain the mean.
    UDIV r0, r0, r2         // r0 = (r0 / array_size) --> mean

    // 2nd) Let's compute the variance
    MOV r7, #0              // loop index
    MOV r1, #0              // sum of squares
    B check_variance
```

```
loop_variance:
    LDR r4, [r3, r7, LSL #2] // r6 = array(i), where r7 = i
    SUB r5, r4, r0           // r5 = array(i) - mean
    MLA r1, r5, r5, r1       // r1 = r1 + (array(i) - mean)^2 --> Multiple and accumulate
    ADD r7, r7, #1           // Update the loop index --> r7 = i + 1

check_variance:
    CMP r7, r2               // While i <= array_size,
    BLT loop_variance        // keep adding (array(i) - mean)^2

    // If i > array_size, summation is done.
    // Let's divide by the size of the array to obtain the variance.
    UDIV r0, r1, r2          // r0 --> Variance

stop:
    B    stop               // Dead Loop

.end
```

Homework 6 (variations)

Homework 6 - Chapter 8 – Exercise 1

- "PUSH {r3}" is equivalent to what?
  - Cortex-M processors uses *full descending* stack.
  - It means, r3 will be pushed in the memory position indicated by the stack pointer, and the stack pointer will be decreased by 4.

Homework 6 - Chapter 8 – Exercise 4

- How many byte does the stack need to pass the arguments when each of the following function is called?
  - `int32_t fun1(uint8_t a, uint16_t b, uint8_t c, int32_t d)`
  - Hint: Page 169

  - In this case, we don’t need to use the stack to pass the arguments:

```
a -> r0
b -> r1
c -> r2
d -> r3
```

Homework 6 - Chapter 8 – Exercise 5

- Which register(s) holds the return value in the following functions?
  - `int16_t fun1()`
  - Hint: Page 169
  - The return only needs 16 bits. So, we only need **r0** to hold the return argument.
  - **Note:** some functions in this question return a pointer to a memory address.

Homework 6 - Chapter 8 – Exercise 17-ish

GNU Assembly

```
.syntax unified
.cpu cortex-m4
.fpu softvfp
.thumb

.data
result:
    .word 0
constants:
    .word 2, 5

.text
.global main
.func computeFunction

main:
    MOV r0, #2  // 1st argument --> x
    MOV r1, #3  // 2nd argument --> y

    BL computeFunction

    // Let's put the result in the memory
    LDR r1, =result
    STR r0, [r1]

stop:
    B    stop              // Dead Loop

computeFunction:
    LDR r2, =constants

    LDR r3, [r2]        // r3 = b --> b = 2
    LDR r4, [r2, #4]    // r4 = c --> c = 5

    // f(x, y) = b*x*y + c
    MUL r5, r0, r1      // r5 = x*y
    MUL r5, r5, r3      // r5 = b*r5
    ADD r5, r5, r4      // r4 = r5 + c

    MOV r0, r5          // return value is stored in r0

    BX LR
.endfunc
.end
```

ARM Assembly

```
    AREA myData, DATA, READWRITE
    ALIGN
result     DCD 0
constants DCD 2, 5

    AREA myCode, CODE, READONLY
    EXPORT __main
    ALIGN
ENTRY

__main PROC
    MOV r0, #2  ; 1st argument --> x
    MOV r1, #3  ; 2nd argument --> y

    BL computeFunction

    ; Let's put the result in the memory
    LDR r1, =result
    STR r0, [r1]

stop
    B    stop          ; Dead Loop

    ENDP

computeFunction PROC
    LDR r2, =constants

    LDR r3, [r2]        ; r3 = b --> b = 2
    LDR r4, [r2, #4]    ; r4 = c --> c = 5

    ; f(x, y) = b*x*y + c
    MUL r5, r0, r1      ; r5 = x*y
    MUL r5, r5, r3      ; r5 = b*r5
    ADD r5, r5, r4      ; r4 = r5 + c

    MOV r0, r5          ; return value is stored in r0

    BX LR
    ENDP

    END
```