CZ1106
CE1106

Chapter 1

Introduction

1

CZ1106
CE1106

Chapter 1

Introduction

Classes of Computers and Early History of Computing

Learning Objectives (1.1)

1. Describe the following classes of computers:
   - Supercomputers
   - Microcomputers
   - Embedded systems
2. Describe two early computer architecture designs.

2

CZ1106
CE1106

What is a Computer?

- From supercomputer → server → PC → tablet → mobile phone → watch.
  - All these devices contain some form of computational elements.
  - No definitive way to classify computers. But we review three broad categories:

    Supercomputer, microcomputers and embedded systems.



Supercomputer → Microcomputer → Embedded

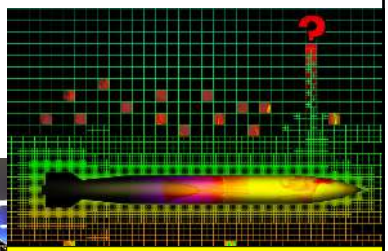
3

CZ1106
CE1106

Classes of computers

Supercomputers

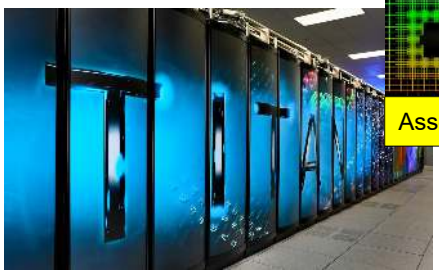
- Very large, powerful and expensive computers.
  - High computational performance and can operate on large data sizes (for high precision calculations).
  - Generally scalable by adding more processors.
  - Applications - weather forecasting, simulation of complex physical systems and sub-atomic structures.



Assess Health of Nuclear Bombs

The Titan Supercomputer
at Oak Ridge National Laboratory, USA

Computational peak performance is around 17-27 petaFLOPS.



Titan consist of
•18,000+ Nvidia Tesla K20 GPUs
•700 terabytes of memory

4

CZ1106
CE1106

Classes of computers

Microcomputers

- Microcomputers contain a microprocessor as a processing unit and external memory and peripheral chip support.
  - More powerful **workstations** are used as servers and the more common variety such as desktop **PC** and **notebooks** are for home-office computing applications.

High-end Server

Personal Computer

Notebook

5

CZ1106
CE1106

Classes of computers

Embedded Systems

- Compact devices that usually employ a single-chip (**microcontroller**) containing the processing unit, memory and relevant peripheral support.
  - They are called **embedded** systems as the presence of the microprocessor is non-obvious. Such devices are all around us.

Examples of embedded systems

6

CZ1106
CE1106

Early Days of the Digital Computer

- Major progress made during World War II (1940's)
- Computer research funded mainly by the War Department
- To solve problems related to ballistics

7

CZ1106
CE1106

Typical Ballistic Computation

Analog gear-based computer

- Knobs input numbers such as target speed and course, range to target, wind speed, wind direction, own speed, own course, etc.

  The outputs controlled the motors of the gun.

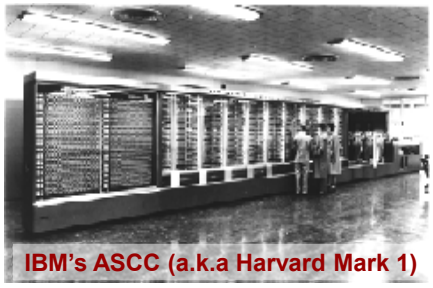
8

CZ1106
CE1106

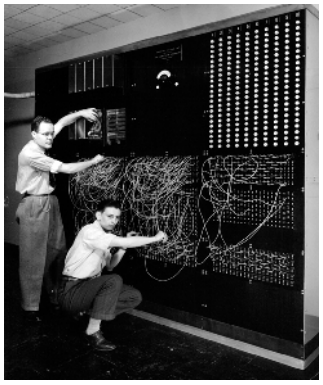
Harvard and Von Neumann

- Two major classes of computer architecture emerged.
- **Harvard architecture**, named after Harvard series of relay calculators developed by Howard Aiken at Harvard Univ.

Howard Aiken



IBM's ASCC (a.k.a Harvard Mark 1)



9

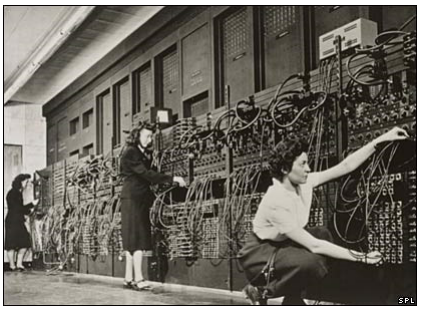
CZ1106
CE1106

Harvard and Von Neumann

- Two major classes of computer architecture emerged.
- **Harvard architecture**, named after Harvard series of relay calculators developed by Howard Aiken at Harvard Univ.
- **Von Neumann architecture**, developed by John Von Neumann at Princeton University. Influenced ENIAC's design.

ENIAC

John von Neumann

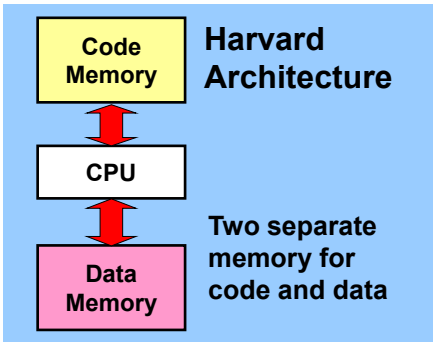


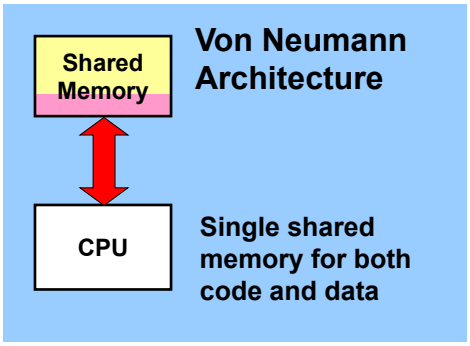
10

CZ1106
CE1106

Harvard and Von Neumann

- Two major classes of computer architecture emerged.
- **Harvard architecture**, named after Harvard series of relay calculators developed by Howard Aiken at Harvard Univ.
- **Von Neumann architecture**, developed by John Von Neumann at Princeton University. Influenced ENIAC's design.

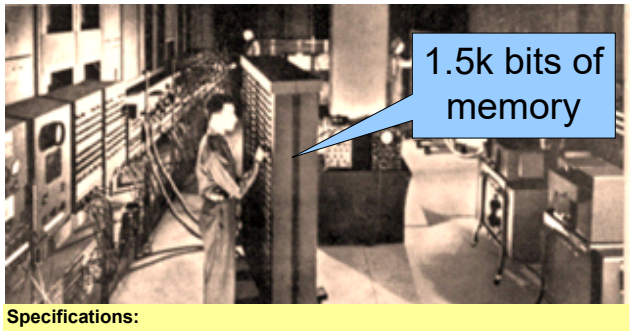




11

CZ1106
CE1106

ENIAC – the first digital computer



Specifications:
Weighed 30 tons, contained 19,000 vacuum tubes, 1,500 relays and consumed 200kW of power.

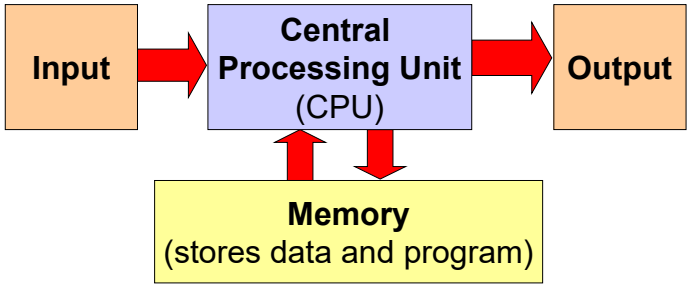
Electronic Numerical Integrator and Calculator

- In 1943, the US army funded Presper Eckert and John Mauchly at Univ. of Pennsylvania to build ENIAC, based on **von Neumann's architecture**.

12

CZ1106
CE1106

The von Neumann Architecture



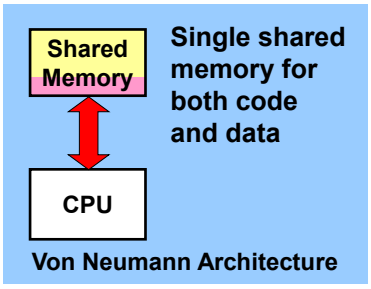
- Many modern day computers are still based on von Neumann's design, which consist of:
  - Central Processing Unit (CPU)
  - Memory
  - Input and Output

13

CZ1106
CE1106

Summary

- Computers can be classified in many ways, e.g. by function, size, general design, etc.
  - We looked at three classes, namely supercomputers, microcomputers and embedded systems.
- Two early rivals in computer architecture designs, the **Harvard** and **von Neumann** architectures.
  - In part, due to the high cost of memory in the early days of computing, the shared memory design of the von Neumann design became the preferred architecture.



14

CZ1106
CE1106

Chapter 1

Introduction

Basic Components of a Microcomputer

Learning Objectives (1.2)

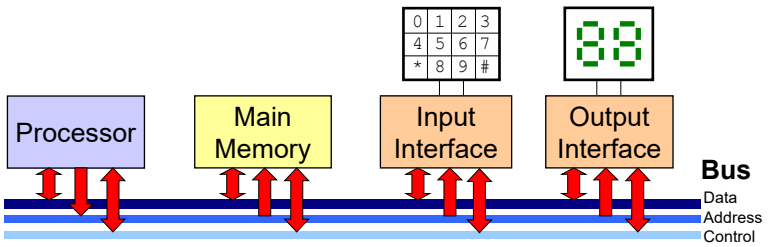
1. Describe the basic components of a microcomputer.
2. Describe the purpose of the CPU clock and reset circuitry.

15

15

CZ1106
CE1106

Components of a Microcomputer

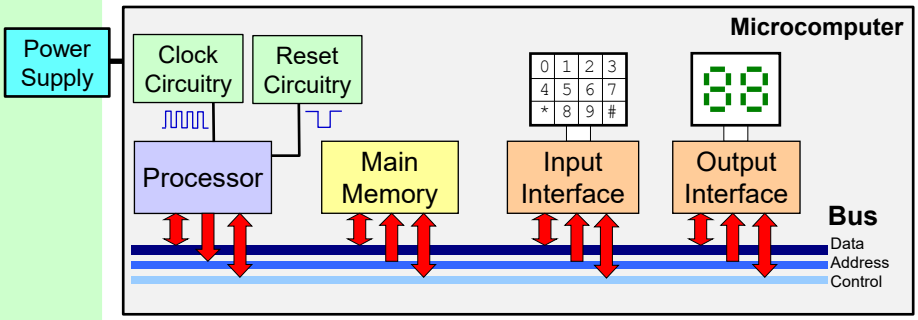


- Consist of three main components: **processor**, main **memory** and **I/O** interfaces.
- They are interconnected by a **bus** structure, which consist of a collection of wires through which binary information can be transferred in parallel.

16

CZ1106
CE1106

Components of a Microcomputer



- Consist of three main components: **processor**, main **memory** and **I/O** interfaces.
  - They are interconnected by a **bus** structure, which consist of a collection of wires through which binary information can be transferred in parallel.
  - Other important components include the **power** supply, CPU **clock** and **reset** circuitries.

17

CZ1106
CE1106

Clock

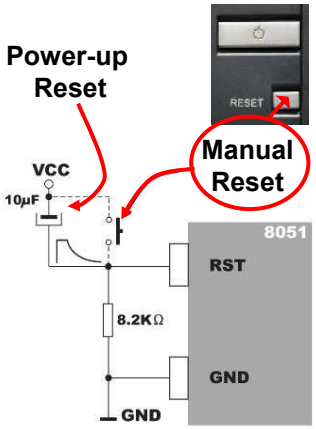
- Most computers are **synchronous** and are driven by a master or system **clock**.
  - The **speed performance** of the computer is governed by the frequency of the clock.
  - The CPU requires a fixed number of clock ticks (**cycles**) to execute each instruction.
  - Many **different clock** frequencies are derived from the one master clock.
  - Operation closer to the CPU core (e.g. registers and arithmetic & logic units) are **clocked faster** and those involving external components (e.g. memory or peripheral access) are **clocked slower**.

18

CZ1106
CE1106

Reset Circuitry

- The CPU is put into a known state on power up. The **reset circuitry** provides an external signal that asserts the Reset pin when power is applied.
  - An active-low signal on the reset pin for a **substantial duration** (several clock cycles) is required to reset the CPU.
  - Most computer system provide an additional **manual reset** button to reset the CPU without switching off the power.
  - On reset, the CPU is put into a known initial state where the boot-up code can then execute.



19

CZ1106
CE1106

Summary

- The basic components within a computer consist of the CPU, memory and I/O interfaces.
- The **memory** is a very critical component in a computer as it stores both data and instructions.
  - The access speed of the memory usually determines the performance of the computer.
  - A fast processor with a fast clock that is coupled with slow memory will still execute instructions slowly.
- Understanding how data and instructions are organised in memory can help programmers write more **efficient programs**.

20

20

CZ1106
CE1106

Chapter 1

Introduction

Desktop PC and Tablet PC Examples

Learning Objectives (1.3)

1. Describe the hardware composition of a desktop PC.
2. Describe the hardware composition of a tablet computer.

21

CZ1106
CE1106

Computer Hardware Decomposition

- What are the major components within the typical computers that we use?

Desktop Personal Computer

Tablet Computer

22

CZ1106
CE1106

Inside a Desktop Personal Computer

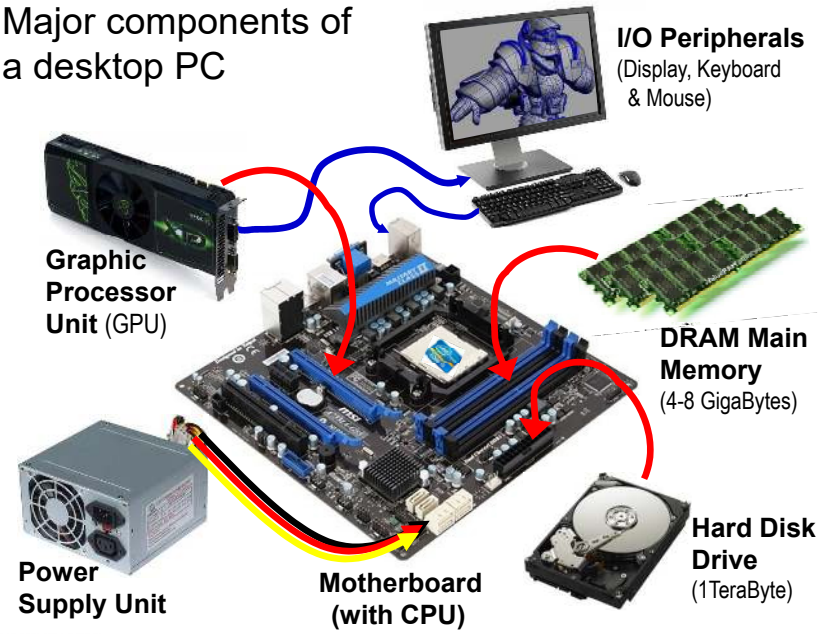
- Major components of a desktop PC

23

CZ1106
CE1106

Inside a Desktop Personal Computer

- Major components of a desktop PC



24

CZ1106
CE1106

Inside a Tablet Computer

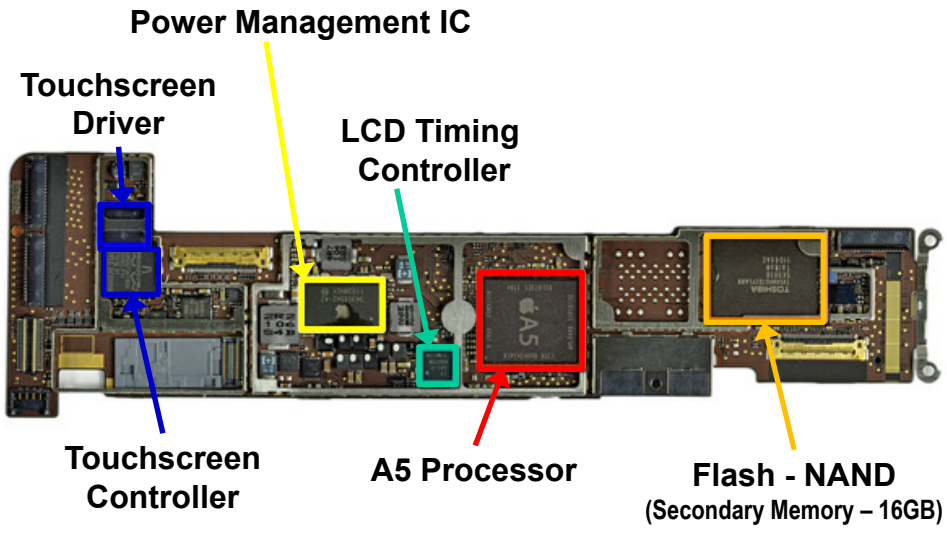
- Major components of the iPad2

Source: http://www.appleinsider.com/articles/11/03/11/live_teardown_of_apples_ipad_2_currently_underway.html

25

CZ1106
CE1106

Inside the iPad 2



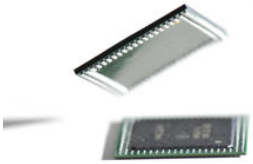
Source: http://www.appleinsider.com/articles/11/03/11/live_teardown_of_apples_ipad_2_currently_underway.html

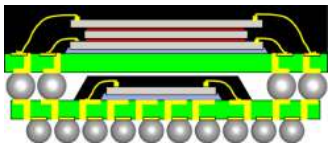
26

CZ1106
CE1106

Apple A5 Processor

- The A5 is a **package on package** (PoP) **system-on-a-chip** (SoC) that was designed by Apple and made by Samsung.





Package on package

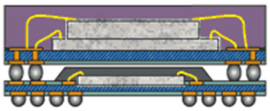
Source:..http://www.appleinsider.com/articles/11/03/15/x_ray_of_apples_a5_cpu_in_ipad_2_confirms_manufacturing_by_samsung.html

27

CZ1106
CE1106

Benefit of PoP Packaging

- Package on package (PoP) is an IC packaging technique that vertically stacks and interconnect separate packages (e.g. CPU and memory) via ball grid array (BGA) connections.
- Some benefits of PoP packaging:



  - **Save space** on motherboard - reduce size of product.
  - **Minimize track length** between CPU and memory - faster signal propagation and reduced electrical noise.
  - Memory units can be **tested separately** before combining with CPU units - improve manufacturing yield and supports multiple memory suppliers.
  - **Different-sized memory** can be coupled with CPU based on user requirements - simplifies inventory control.

Try: Google Search "Benefits of Package on Package"

28

CZ1106
CE1106

A5 Processor (System-on-a Chip)

- The A5 processor is with its built in I/O interfaces and support is considered a system-on-a-chip (SoC).
- A dual-core **ARM Cortex-A9** CPU with 4.5MB cache memory.
- 1GHz CPU clock, can be dynamically reduced to save battery life.
- 512MB low-power DDR SDRAM (@533MHz).
- Dual core PowerVX SGX543MP2 GPU to speed up graphics.

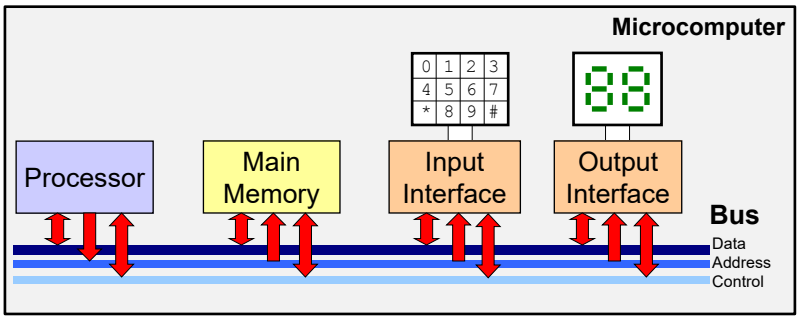
Source:http://www.appleinsider.com/articles/11/03/15/x_ray_of_apples_a5_cpu_in_ipad_2_confirms_manufacturing_by_samsung.html

29

CZ1106
CE1106

Summary

- Whether a desktop or tablet PC, the basic components of a computer remains the same.
- These basic components are essentially the **CPU**, **memory** and the various **I/O interfaces** that permit peripherals to be connected to the computer.



30

CZ1106
CE1106

Chapter 2

Data Organisation in Memory

1

CZ1106
CE1106

Chapter 2

Data Organisation in Memory

Role of Memory in Computing

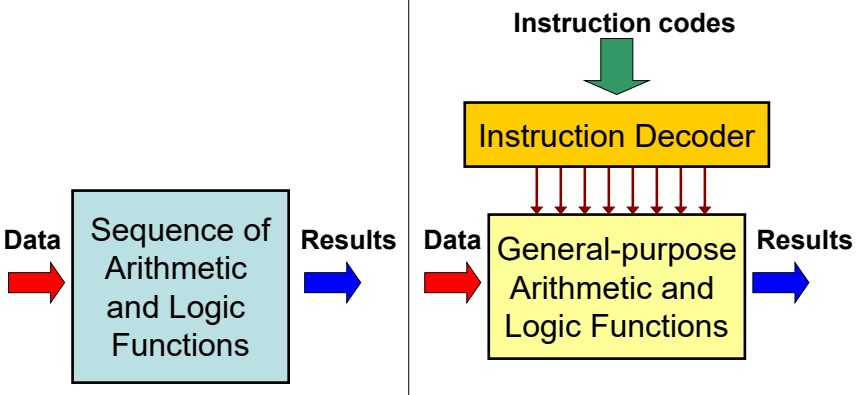
Learning Objectives (2.1)

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.

2

CZ1106
CE1106

Approaches to Computing



Programming in Hardware

Programming in Software

3

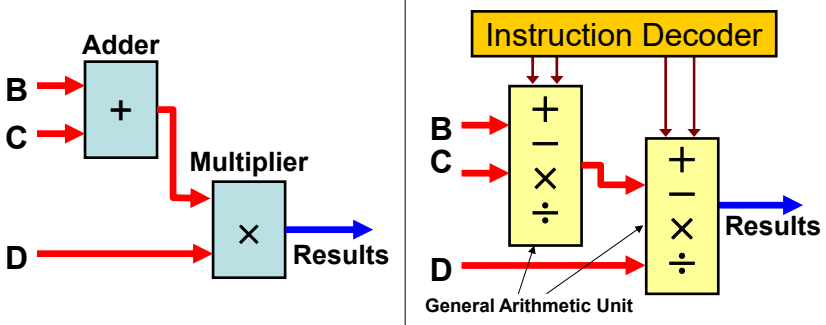
CZ1106
CE1106

Approaches to Computing

Implementing A = (B+C)×D

Fast computation but very inflexible

Slower and more complex but easily programmable



Programming in Hardware

Programming in Software

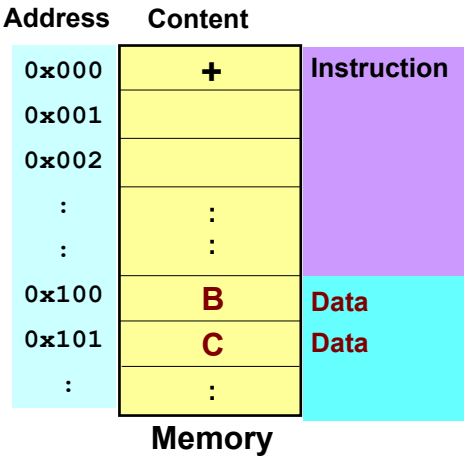
4

CZ1106 CE1106

Code, Data and Memory

- What is code and what is data?
  - **Code** is a sequence of instructions.
  - **Data** are values these instructions operate on.
- What is the memory?
  - Its a sequential list of addressable storage elements for storing both instructions and data.

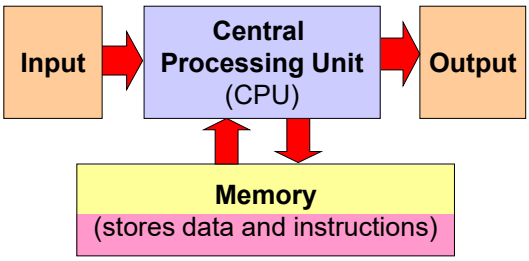
e.g. B+C



5

CZ1106 CE1106

The Stored Program Concept



- Most modern day computer design are based on von Neumann's stored program concept:

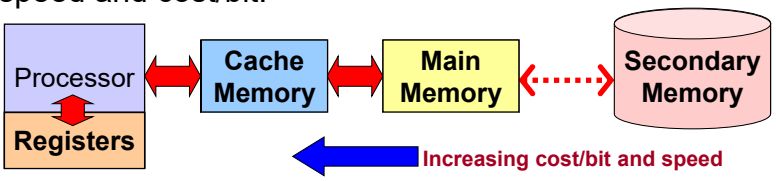
1. Both data & instructions are stored in the same memory
2. Contents of memory are addressable by location, without regard to data type
3. Execution occurs sequentially (unless explicitly modified)

6

CZ1106
CE1106

Memory Hierarchy

- Memories are generally organized in levels of increasing speed and cost/bit.

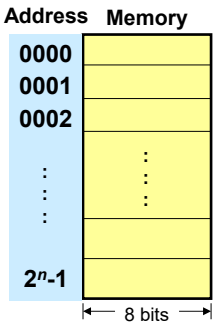


Registers	Very fast access but limited numbers within CPU. Operates at CPU clock rate (size: 2-128 registers)
Cache Memory	Fast access static RAM close to CPU. Typical access time 3-20nS (size: up to 512 kB)
Main memory	Usually dynamic RAM or ROM (for program storage). Typical access time 30-70nS. (size: up to 16GB)
Secondary Memory	Not always random access but non-volatile. Maybe be based on magnetic or flash technology. Typical access time 0.03-100mS. (size: up to 4TB)

CZ1106
CE1106

Characteristics of Main Memory

- **Fix-sized** (typically 8-bit) storage location accessible at high speed and in **any order**.
- Each byte-sized location has a unique **address** that is accessed by specifying its binary pattern on the **address bus**.



- Memory size is dependent on number of lines (*n*) in the address bus. (Memory size = 2^n bytes).
- Memory stores both **data** and **instructions. Consecutive** locations used to store multi-byte data.

CZ1106
CE1106

Summary

- Programming in software provides flexibility.
- Functionality specified by bit patterns in the instructions.
- Instructions are stored sequentially in memory.
- Programs are executed by fetching these instructions one at a time from memory.
- Memory stores both instructions and data.
- Memory contents are addressable by a unique address.
- Each unique address stores a fixed number of bits (i.e. 8 bits or a byte)

9

CZ1106
CE1106

10

10

CZ1106
CE1106

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.

11

CZ1106
CE1106

Data Representation in Memory

- Programming languages like C has many different data types.
  - Numbers
  - Characters
  - Boolean
  - Arrays
  - Structures
  - Pointers
- How are these variables stored in memory?
  - Knowledge of their representation in memory allows us to find efficient ways to access and process them in our program.
  - Note: Lecture notes will use ANSI C programming language as an example.

12

CZ1106 CE1106

Number Representation

- ANSI C data types representing numbers come in various varieties (basic type, sign & size).
- There a two **basic types**. Whole number or **integer** and **floating point** number.
  - Integer, declared as `int` (e.g. 32,676)
  - Floating point, declared as `float` (e.g. 3.2676×10^4)
- Floating point numbers are useful for scientific calculations and has issue of trading off **precision** and **range** for a given size (in bits).
  (to be covered in later lectures in Computer Arithmetic)
- Some CPUs are only integer-based and make use of an additional floating point unit (FPU) to support floating point computations.

13

CZ1106 CE1106

Number Representation (cont)

- Floating point numbers are always **signed**.
- Integers can be either **signed** or **unsigned**.
  - Signed integer, declared as `int` (e.g. -1)
  - Unsigned integer, declared as `unsigned int` (e.g. 255)
- Most processors interpret signed numbers using the **2's complement** representation.

2's complement	Unsigned
`0111 1111`$_2$ = (**127**)	`0111 1111`$_2$ = (**127**)
`1111 1111`$_2$ = (**-1**)	`1111 1111`$_2$ = (**255**)

- Use unsigned numbers where possible to increase the positive range (e.g. counting a population).

Learn More: Google "Signed number representation"

14

CZ1106 CE1106

Number Representation (cont)

- The **range** can be increased by using more bytes to represent the number.
- Use appropriate suffix (`short` and `long`) to specify required size.
- Use appropriate size to reduce memory requirements of your program.

Type	Bytes	Bits	Range	
signed char	1	8	-128 -> +127	
unsigned char	1	8	0 -> +255	
short int	2	16	-32,768 -> +32,767	(+/-32KB)
unsigned short int	2	16	0 -> +65,535	(64KB)
unsigned int	4	32	0 -> +4,294,967,295	(4GB)
int	4	32	-2,147,483,648 -> +2,147,483,647	(+/-2GB)
long int	4	32	-2,147,483,648 -> +2,147,483,647	(+/-2GB)
long long int	8	64	-(2^63) -> (2^63)-1	
float	4	32		
double	8	64		
long double	12	96		

ANSI C numeric data types and their respective size and range

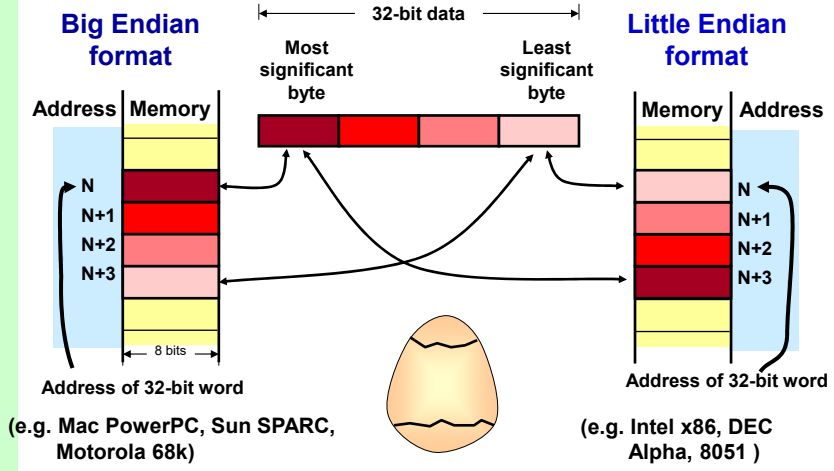
Note: These sizes may change depending of the processor and compiler

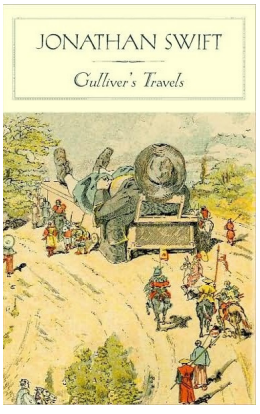
15

CZ1106 CE1106

Data Organization in Memory

- How is a 32-bit number stored in memory?
- There are two ways, depending on the **byte-ordering** of the data in memory





Learn More: Google "Byte ordering"

16

CZ1106
CE1106

Summary

- Numbers are represented in a variety of ways.
  - Number of bits.
  - Integer or Floating Point.
  - Signed or Unsigned.
- For a given data size (i.e. number of bits), range and precision must trade off against each other.
- Multi-byte integers are stored using either Big or Little Endian byte-ordering.

17

CZ1106
CE1106

18

18

CZ1106
CE1106

Chapter 2

Data Organisation in Memory

Character and Boolean Representation

Learning Objectives (2.3)

1. Describe the char data type and its representations.
2. Describe the Boolean data type and its implementation.

19

CZ1106
CE1106

Character Representation

- Computer not only process numbers but handles character data (e.g. text processing, print display).
- In ANSI C, a character type is declared as **`char`**.
- A char variable required **one byte** of memory storage
- Data in a computer is stored in **binary** but they are transformed into representative characters through some **encoding standard**.
- The most common character encoding standard is the 7-bit **ASCII** code.

```
main()
{
 char c;    //declare a character variable

 c = 'a'; //assign character 'a' to variable

}
```

- There are many other character encoding standards - DEC's Sixbit (6-bit), IBM's EBCDIC (8-bit) and Unicode (16-bit), etc.

Learn More: Google "Character Codes"

20

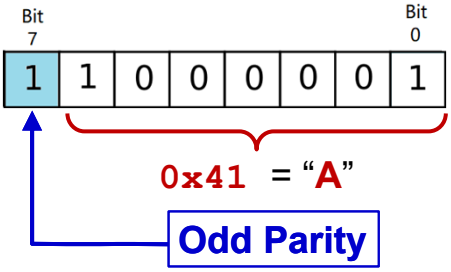
CZ1106 CE1106

ASCII

- American Standard Code for Information Interchange (ASCII) is a 7-bit code for representing characters.
  - Useful properties – lower, upper and digits are **contiguous,** which makes it easy to check character's category and also transpose it from one case to another.
  - A byte is normally used to store an ASCII character and MSB could be used for **parity error checking**.

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

7-bit ASCII Table



Learn More: Google "Parity bit and ASCII"

21

CZ1106 CE1106

SIXBIT

- DEC's SIXBIT character code was popular in the 1960's and 70's.
  - Not used for general text processing as it lacks control characters like CR and LF.
  - Six-bit character format was popular with DEC's PDP-8 and PDP-10 computers, which used 18-bit and 36-bit processors respectively.

LS / MS	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
1	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
2	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_

SIXBIT Table

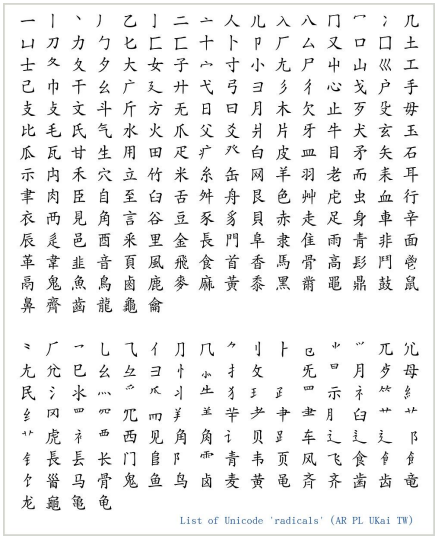
22

CZ1106
CE1106

Unicode

- Developed to handle text expressions for all major living languages in the world.
  - An **8,16 or 32-bit** character encoding standard that is downward compatible with ASCII.
  - Adopted by technologies such as XML, Java programming language, Microsoft .NET Framework and many operating systems.
  - Unicode 13.0 was launched on 20 March 2020. It contains a total of 143,859 characters.

Unicode radicals for KangXi
214 radicals to support Chinese, Japanese and Korean ideographs (U+2F00 – U+2FDF)



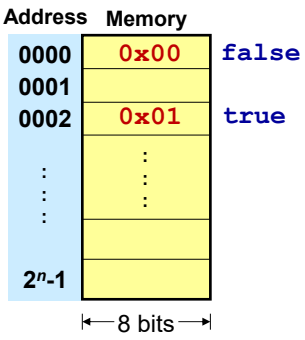
Learn More: Google "Unicode" or "Unicode characters"

23

CZ1106
CE1106

Boolean Representation

- Boolean variables have only 2 states.
  - The Boolean type was made available in ANSI C (after 1999) as `_Bool` with the `stdbool.h` header file.
  - Values assigned in C: **False** = 0, **True** = non-zero (e.g. 1)
  - Memory storage for Boolean variables is **inefficient** as most implementation use a byte (minimum memory unit) to store a 1-bit Boolean value.
  - The 8051 processor has 128 bit-addressable memory locations.

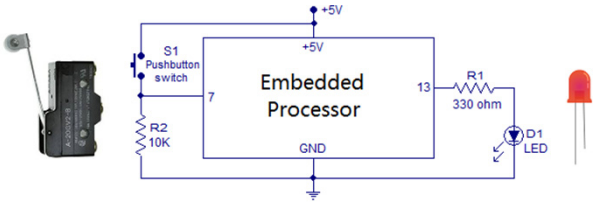


24

CZ1106
CE1106

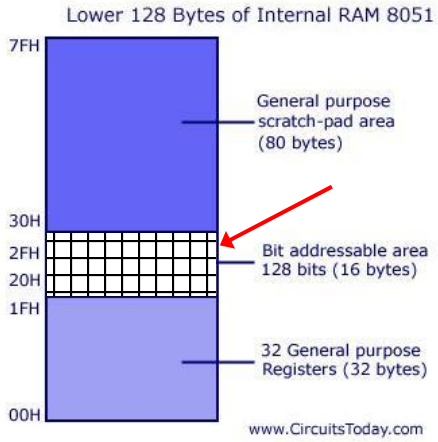
8051 - Bit Addressable Memory

- The 8051 processor support a small portion of bit-addressable memory.
- In embedded applications, data variables are often related to ON-OFF status of discrete sensors and output (e.g. switches or LEDs)



- Bit-addressable memory provide an efficient way to handle such information.

8051 8-bit Processor



25

CZ1106
CE1106

Summary

- Characters data type is used to handle text.
- They are byte-sized.
- They need an encoding standard (ASCII being the most common).
- In the C programming language they are declared using `char`.

- Boolean data types are used to represent true and false states.
- They are usually stored using a whole byte.
- In ANSI C, the `false` value is given by the numeric value 0.
- The `true` value is taken as any non-zero value.

26

CZ1106
CE1106

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

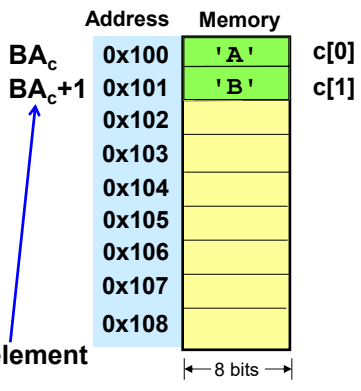
27

CZ1106
CE1106

Array Representation

- A linear array is a consecutive area in memory storing a series of homogenous data type.
- Elements of an array are accessed through appropriate offset from the **base address** (BA) of the array.

```
main()
{
 char c[2];      //2 element char array c
 c[0]='A';       //assign "A" to 1st element
 c[1]='B';       //assign "B" to 2nd element

}
```



Offset from base address (BA_c) to access each element of the array `c`

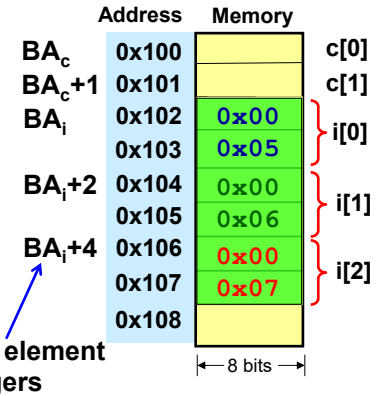
28

CZ1106 CE1106

Array Representation (cont)

- Computation of the address of array element must take into account its data type size.
- Address is computed by multiplying element number by size of data type before addition to the base address (BA).

```
main()
{
 char c[2];          //2 element char array c
 short int i[3];     //3 element integer array i
 i[0] = 5;           //assign values to array i
 i[1] = 6;
 i[2] = 7;

}
```



Offset (n × 2) from base address (BA_i) to access each element n of the array `i` consisting of 2 byte-sized short integers

29

CZ1106 CE1106

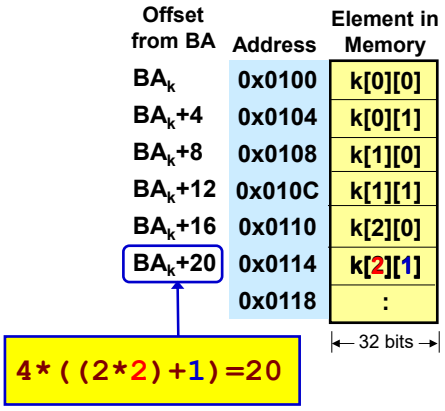
Nested Array

- Each element of an array can itself be an array.
- General principles of array allocation and referencing hold for nested array.
- An array of arrays is then created.

```
main()
{
 int k[3][2];  //a 3x2 integer array
}
```

- The offset from BA for element `k[a][b]`

  `= sizeof(int)*((2*a)+b)`



30

CZ1106 CE1106

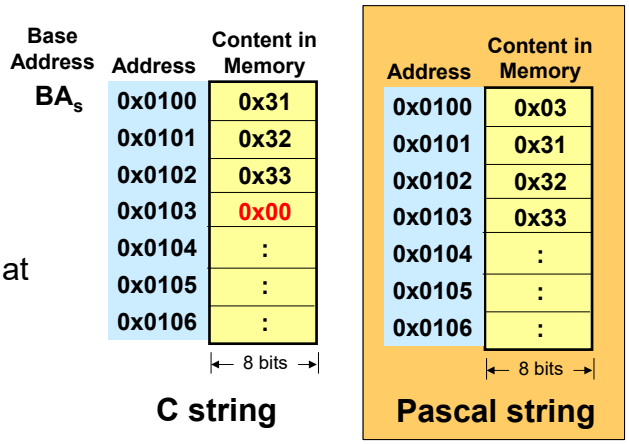
String Representation

- A string in a C program is an array of characters terminated by a null (`0x00`) character.

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

- An alternative is the **Pascal string**, which stores the length of the string at the start of the string.

Which method is better?



Learn more: Google Search "C strings vs Pascal strings"

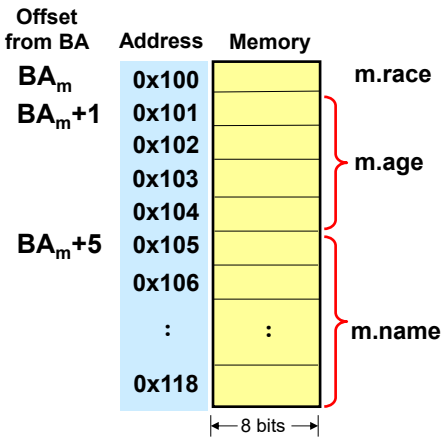
31

CZ1106 CE1106

Structure Representation

- Structure allows new data types to be created by combining objects of different types.
- Each data type in a **declared** structure variable occupies predefined consecutive locations based on data type size.

```
main()
{
 struct Man ; //define structure Man
 {
  char race;
  int age;
  char name[20];
};
 Man m;
}
```



32

CZ1106
CE1106

Summary

- Arrays stores a series of similar data types in consecutive memory locations.
  - Elements are accesses using offsets from a base address.
  - Offset computation must take into account size of data type.
- Structures stores a series of dissimilar data types in consecutive memory locations.
  - Memory space for structure data type is only allocated when structure variables are declared.

33

CZ1106
CE1106

34

34

CZ1106
CE1106

Chapter 2

Data Organisation in Memory

Pointer Representation

Learning Objectives (2.5)

1. Describe the representation of pointers in memory.

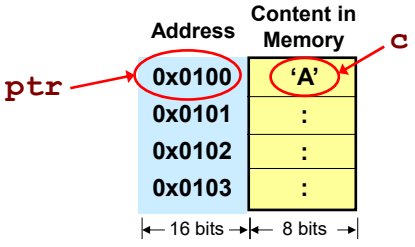
35

CZ1106
CE1106

Pointers Representation

- Pointers in C provides a mechanism for referencing memory variables, elements of structures and arrays.
- C pointers are declared to point to a particular data type.
- The value of a pointer is an **address**. Its **size** is **fixed** (regardless of data type).
- The size of the pointer depends on the processor's address range.

```
main()
{
 char c ;       //char variable c
 char *ptr;     //char pointer ptr
 c = 'A';       //assign value 'A' to c
 ptr = &c;      //ptr gets address of c
}
```



Variable	Size (Bytes)
c	1
ptr	2

Note: 1. Assume address of **c** = 0x0100
2. Assume addresses in CPU are specified using 16 bits (2 bytes).

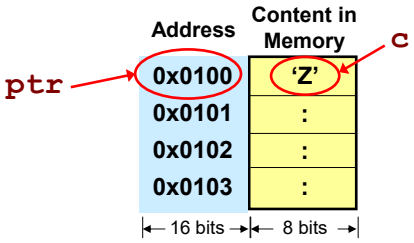
36

CZ1106 CE1106

Dereferencing a Pointer

- When properly initialized, a pointer contains the start address where the memory variable can be found.
- We can use the dereferencing operator (*) to copy a value to the address pointed to by the pointer **ptr**.
- Knowing the start address (base address) of an array or structure makes it easy to access their different elements.

```
main()
{
 char c ;        //char variable c
 char *ptr;      //char pointer ptr
 c = 'A';        //assign value 'A' to c
 ptr = &c;       //ptr gets address of c
 *ptr = 'Z';     //use dereferencing
                 //operator on ptr to give
                 // variable c value 'Z'
```



37

CZ1106 CE1106

Summary

- Pointer is a variable whose value is an address.
- The size of a pointer variable is independent of data type it is pointing to.
- The size of a pointer is dependent on the addressable memory space of the processor.

38

CZ1106
CE1106

Chapter 2

Data Organisation in Memory

Data Alignment

Learning Objectives (2.6)

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

39

CZ1106
CE1106

Data Alignment

- Most computer systems have restrictions on the allowable address for accessing various data types.
  - Multi-byte data (e.g. `int`, `double`) must be aligned to addresses that are multiple of values such as 2, 4, or 8.
  - Programs written with Microsoft (Visual C++) or GNU (gcc) and compiled for a 64-bit Intel processor will use the following data alignment enforcement:

Data Type	Size (Byte)	Example of allowable start addresses due to alignment
char	1	0x..0000, 0x..0001, 0x..0002
short	2	0x..0000, 0x..0002, 0x..0004
int	4	0x..0000, 0x..0004, 0x..0008
float	4	0x..0000, 0x..0004, 0x..0008
double	8	0x..0000, 0x..0008, 0x..0010
pointer	8	0x..0000, 0x..0008, 0x..0010

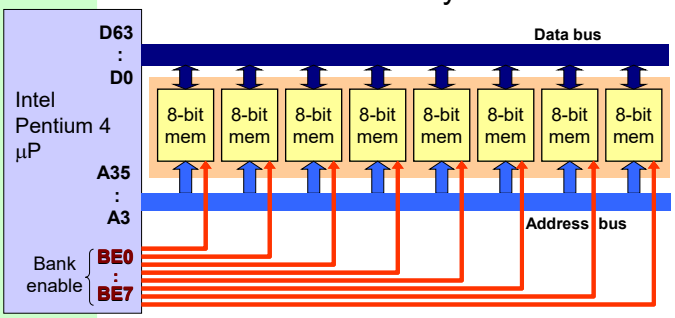
40

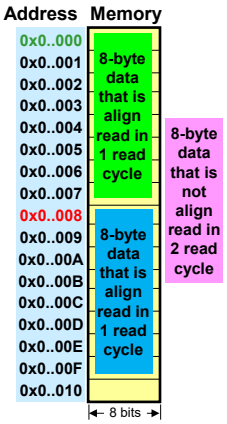
CZ1106
CE1106

Memory Interfacing & Data Alignment

(Case Study: Intel Pentium 4)

- Main memory consist of multiple 8-bit memory modules required to make up 64-bit data word size of processor.
- Data width is selected by additional control lines (**BE0-BE7**).





- Only one address pattern can exist on address bus during data transfer. 8-byte data needs **two memory cycles** if not aligned to 8-byte address boundary.

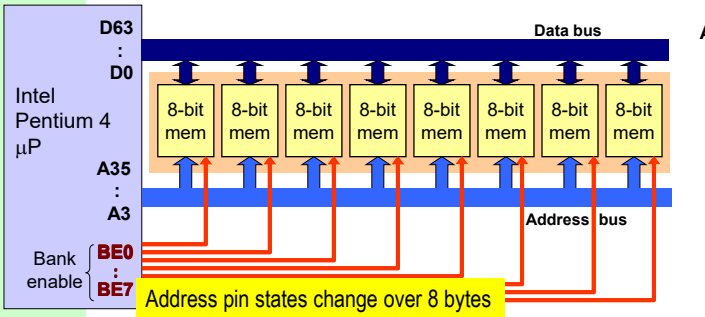
Learn More: Google "memory and data alignment"

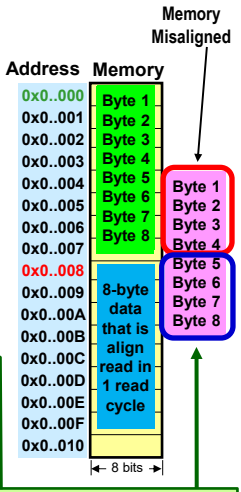
41

CZ1106
CE1106

Address Mapping of 64-bit data

(Case Study: Intel Pentium 4)





Memory Align

Address Pins	A_{35}	A_{34}	.....	A_{06}	A_{05}	A_{04}	A_{03}	A_{02}	A_{01}	A_{00}
Byte 1	0	0	.....	0	0	0	0	1	0	0
Byte 2	0	0	.....	0	0	0	0	1	0	1
Byte 3	0	0	.....	0	0	0	0	1	1	0
Byte 4	0	0	.....	0	0	0	0	1	1	1
Byte 5	0	0	.....	0	0	0	1	0	0	0
Byte 6	0	0	.....	0	0	0	1	0	0	1
Byte 7	0	0	.....	0	0	0	1	0	1	0
Byte 8	0	0	.....	0	0	0	1	0	1	1

Note:
Address pin A_{03} cannot be in states 0 and 1 simultaneously to transfer a mis-aligned word in a single memory cycle.

Note: Lines BE0-BE7 implements address bits A_{02} to A_{00} and determine which byte(s) in the 8 possible addresses are accessed during memory transfer.

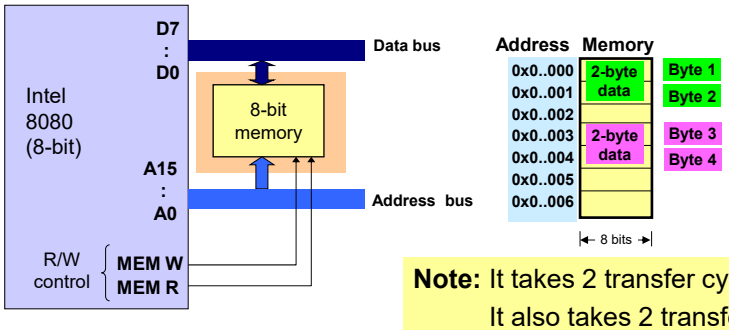
42

CZ1106 CE1106

Data Alignment and Data Bus Width

(Case Study: An 8-bit Processor - Intel 8080)

- The extent of data alignment is dependent on the width of the processor's data bus.
- A processor with an **8-bit** data bus does not have any data alignment issues. Multi-byte data can be fetched from **any address**.
- A processor with a **16-bit** (i.e. 2 bytes) data bus only needs to align 2, 4 or 8 byte-sized data to **even addresses** (e.g. `0x0000`, `0x0002`, `0x0004`, etc).



Note: It takes 2 transfer cycles to read Bytes 1 and 2
It also takes 2 transfer cycles to read Bytes 3 and 4

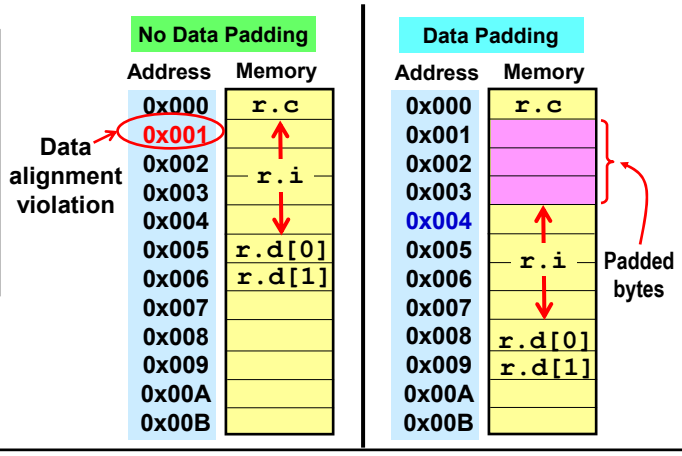
43

CZ1106 CE1106

Data Alignment with Structures

- **Data padding** (addition of meaningless bytes) is used by compilers to ensure alignment of different data types within a structure.

```
struct rec1 {
  char c;
  int i;
  char d[2];
 }
 rec1 r;
```



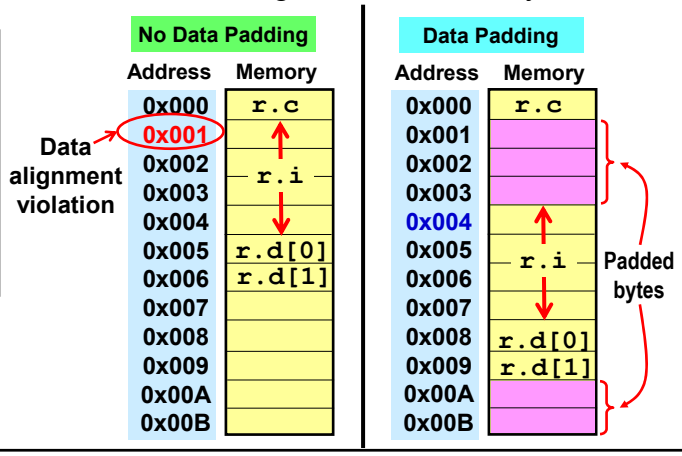
44

CZ1106 CE1106

Data Alignment with Structures

- **Data padding** (addition of meaningless bytes) is used by compilers to ensure alignment of different data types within a structure.
- Padded bytes are added between end of last structure element and the start of the next to maintain alignment in an array of structures.

```
struct rec1 {
  char c;
  int i;
  char d[2];
 }
 rec1 r;
   :
 rec1 t[10];
```



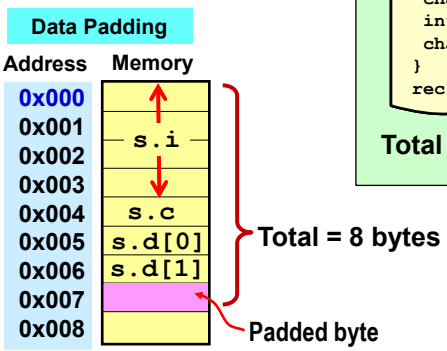
45

CZ1106 CE1106

Data Alignment with Structures (cont)

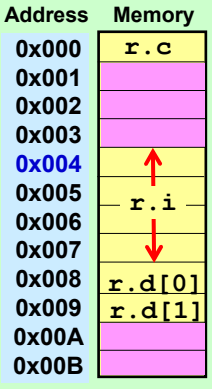
- Structures should be constructed with data alignment constraints in mind.
- **Rearrange the order** of data objects in the structure based on their size to minimize data padding where possible.

```
struct rec2 {
  int i;
  char c;
  char d[2];
 }
 rec2 s;
```




```
struct rec1 {
  char c;
  int i;
  char d[2];
 }
 rec1 r;
```

Total = 12 bytes



Before re-arranging structure (previous)

Learn More: Google "Structure and Padding in C"

46

CZ1106
CE1106

Summary

- Data alignment restricts where in memory multi-byte data can be stored.
  - Data must be aligned to addresses that are multiple of the size of their data type.
  - The extent of the data alignment restriction depends on the size of the processor's data bus.
- Structure data types should consider data alignment issues.
  - Data padding is used to meet alignment restrictions.

47

CZ1106
CE1106

Chapter 3

Instruction Organisation in Memory

1

CZ1106
CE1106

Chapter 3

Instruction Organisation in Memory

Characteristics of Instructions and the ARM Programmer's Model

Learning Objectives (3.1)

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

2

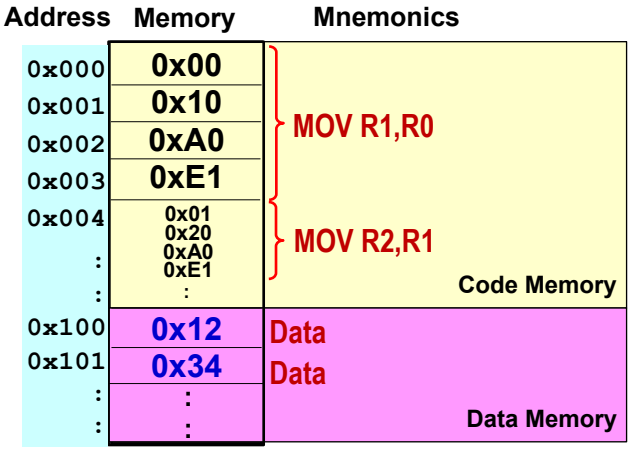
CZ1106 CE1106

Code and Data in Memory

- Instructions are stored in memory as binary patterns called **machine codes**.
- They are also represented in more readable **mnemonics**.
  - ARM (32-bit CPU) example:
    Note: ARM instructions are **32 bits** long

```
MOV R1,R0
MOV R2,R1
:
```

  - Memory is partitioned into separate **code** and **data** segments.

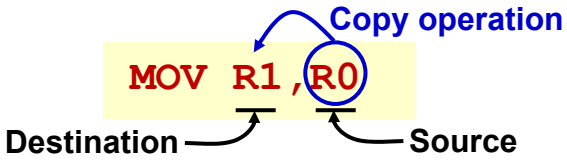


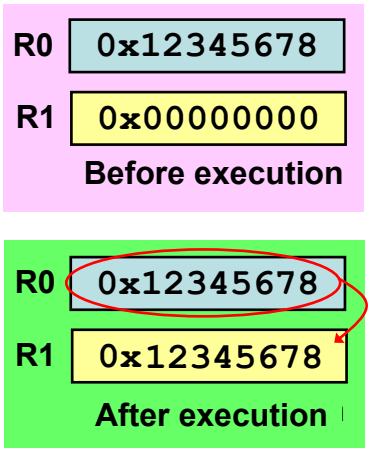
3

CZ1106 CE1106

ARM Instruction Format (Introduction)

- Introduction to a simple ARM mnemonic.
  - The `MOV` operator is a two-operand instruction that copies the source operand to the destination operand.
  - The right operand is the source and left operand is the destination.





  - In this example, the both operands are **registers** in the ARM CPU.

4

CZ1106 CE1106

Instruction Organization in Memory

- Instructions are binary encoded and stored in memory. Unlike data, instruction bytes tell CPU what actions to take (i.e. they are **executable**).
- Most instruction formats consist of **two** parts.

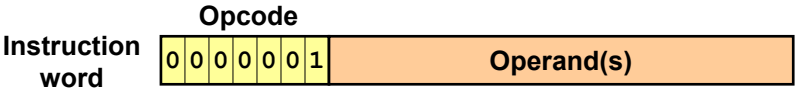
  - First part of instruction (**op-code**) specifies the operation to be carried out (e.g. move, add, subtract).
  - Remaining bits (**operands**) specify the data itself or the location of data and where results is to be stored.
  - Some operations produce no storable result but may alter program sequence or influence status flags (e.g. **N**, **Z**, **V**, **C**).

5

CZ1106 CE1106

Opcode and Operands

- **How are opcode encoded in an instruction?**



  - If there are 80 different operations, (e.g. `add`, `sub`, etc) then at least 7 bits ($2^6 < 80 < 2^7$) of opcode is needed to represent all the unique bit patterns.
  - The **more variety** of operations supported by the instruction set, the **longer the length** of each instruction.
- **Are there many ways to specify the operand(s)?**
  - Numerous. An operand can be stored as part of the **instruction**, stored in a **register** or stored in **memory**.
  - The method by which the operand is specified is known as **addressing mode**.

6

CZ1106 CE1106

Addressing Mode Examples

Addressing Mode	ARM	Intel
Absolute (Direct)	None	`MOV AX,[1000h]`
Register Direct	`MOV R1,R0`	`MOV AX,DX`
Immediate	`MOV R1,#3`	`MOV AX,0003h`
Register Indirect	`LDR R1,[R0]`	`MOV AX,[BX]`
Register Indirect with Offset	`LDR R1,[R0,#4]`	`MOV AX,[BX+4]`
Register Indirect with Index	`LDR R1,[R0,R2]`	`MOV AH,[BX+DI]`
Implied	`BNE LOOP`	`JMP -8`

7

CZ1106 CE1106

Role of Instructions

- The role of an instruction is to make purposeful **changes** to the **current state** of the processor.
  - The visible current state of a processor is defined by the **programmer's model** and contents in **memory**.
  - There are three broad categories of instructions for general programming available in a processor:

Data Processing

ARM examples:

```
ADD R0,R1,R2
SUB R1,R2,#3
EOR R3,R3,R2
```

Data Transfer

ARM examples:

```
MOV R1,R0
STR R0,[R2,#4]
LDR R1,[R2]
```

Program Control

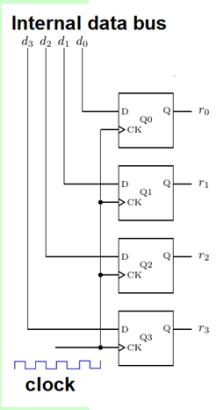
ARM examples:

```
B   Back
BNE Loop
BL  Routine
```


8

CZ1106 CE1106

Typical Programmer's Model & Memory

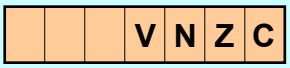


Register circuit

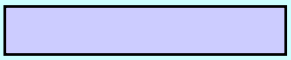
Programmer's model

Registers

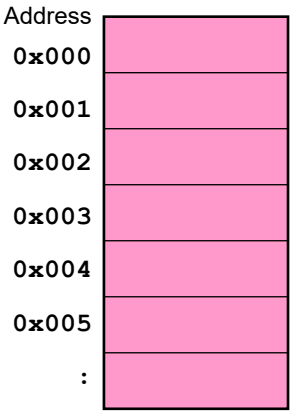
Stack Pointer (SP)



Status Register (SR)



Program Counter (PC)



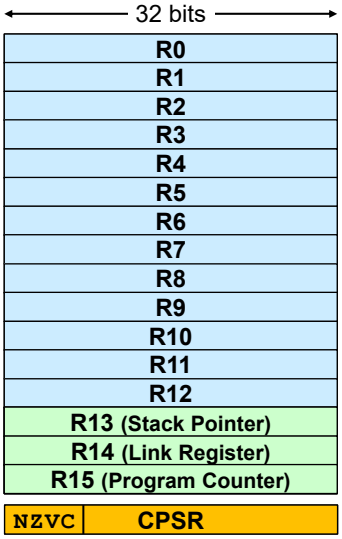
Memory

9

CZ1106 CE1106

The ARM Programmer's Model

- The ARM processor can operate in various modes. The following are visible registers in the **User mode**.
  - There are 16 32-bit registers.
  - R0 – R12 are general purpose registers.
  - R13 is the Stack Pointer (**SP**).
  - R14 is the Link Register (**LR**).
  - R15 is the Program Counter (**PC**)
  - The Current Processor Status Register (**CPSR**) holds the condition code bits (**NZVC**)



10

CZ1106 CE1106

Program Counter

- ARM's register **R15** in the is the Program Counter (**PC**) and it keeps track of **program execution.**
  - The content of the **PC** is an **address**.
  - This address is the start address of the **next instruction** to be fetched.
  - The **PC automatically increments** by the length of the instruction executed (i.e. increment by 4 in the ARM CPU).
  - Sequential execution can be altered by modifying the contents of the **PC** to a new address location (i.e. a jump or a branch operation)
  - Due to **pipeline** architecture of the ARM CPU, the value in the **PC** is the address of the current instruction being executed plus 8 bytes (i.e. 2 instructions).

11

CZ1106 CE1106

Stack Pointer and Link Register

- Stack Pointer (**SP**).
  - By convention, **R13** in the ARM processor is designated the stack pointer.
  - The **SP** is used to maintain a space in memory **(stack)** that is used to **temporarily** stored away register information which will be needed again later.
- Link Register (**LR**).
  - **R14** is used as a Link Register during the calling of subroutines.
  - **R14** gets a copy of the **PC** (R15) when a Branch with Link (`BL`) instruction is executed.
  - At all other times, **R14** can also act as a general purpose register.

Learn more: Google Search "ARM stack pointer" or "Link Register"

12

CZ1106 CE1106

CPSR Condition Code Flags

CPSR bit(s)	Name	Description NZVC CPSR
31	N	Can set if last operation produced a **negative** result
30	Z	Can set if last operation produced a **zero** result
29	C	Can set if last operation produced a **carry out** in the most significant bit
28	V	Can set if the last operation produced an **overflow** for a **signed** arithmetic operation

N – Negative; **Z** – Zero; **C** – Carry ; **V** – Overflow

13

CZ1106 CE1106

Summary

- An instruction usually consist of an **opcode** and an **operand**.
- Instructions in a program change the **states** of a processor as it executes.
- The states of the processor consist of values in:
  - general purpose registers (e.g. **R0 – R12**)
  - specialised function registers (e.g. **CPSR, SP**, **LR** and **PC**)
  - memory contents
- The design of a program is to ensure that these **states changes** in a purposeful manner during the execution of the sequence of instructions.

14

CZ1106
CE1106

Chapter 3

Instruction Organisation in Memory

Basic Execution Cycle

Learning Objectives (3.2)

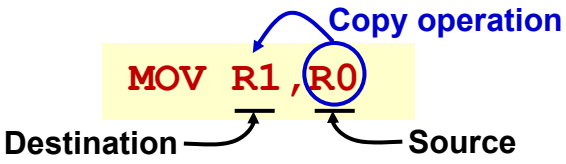
1. Describe the function of a simple **LDR** instruction from the ARM instruction set.
2. Describe the basic execution cycle of a simple **LDR** instruction.
3. Describe the factor that determines the execution speed of an instruction.

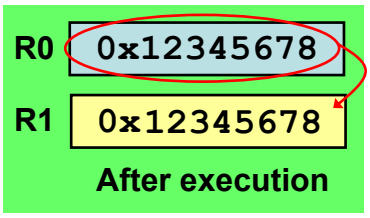
15

CZ1106
CE1106

The MOV Instruction (Review)

- The `MOV` operator copies **register** content to a register.



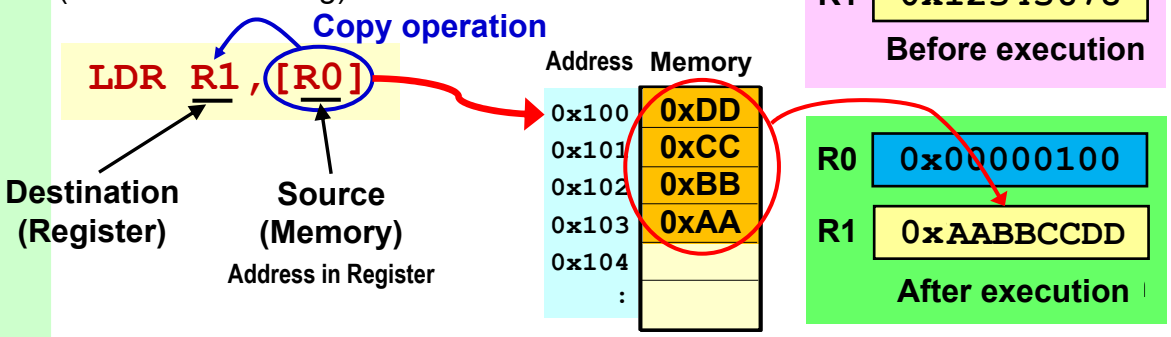


16

CZ1106 CE1106

The LDR Instruction

- The `LDR` operator copies **memory** content to a register.
  - The left operand is always a destination register.
  - The right source operand is a memory location whose address is contained in a register (indirect addressing).



17

CZ1106 CE1106

The Role of the Processor

- What is the role of the processor (CPU)?
  - **Fetch instructions** - CPU reads instructions from memory.
  - **Decode instructions** - Instruction is decoded to determine action required.
  - **Fetch data** – Some data from memory may need to be fetched in order to complete execution of instruction (optional).
  - **Execute** – Instruction execution may require CPU to perform some arithmetic or logical operation on the data, or just move the data into a register.
  - This **fetch-decode-execute** cycle is performed repeatedly by the CPU once powered-on.

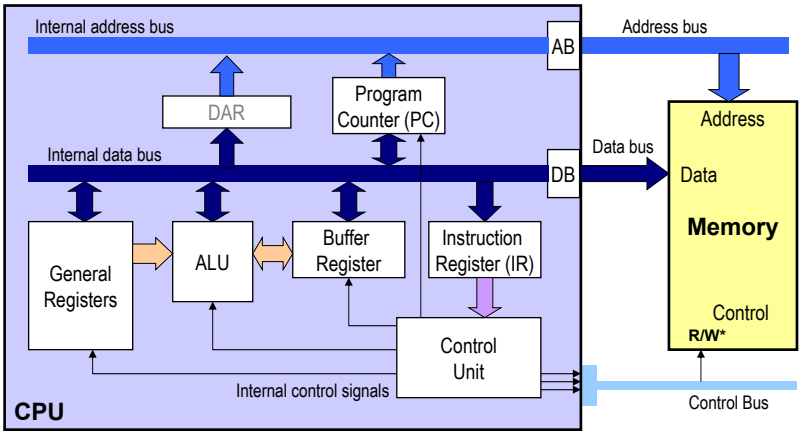
18

CZ1106
CE1106

Basic Execution Cycle

A Simple Processor

- Basic components of a simple processor (CPU) and its interface signals to external main memory.



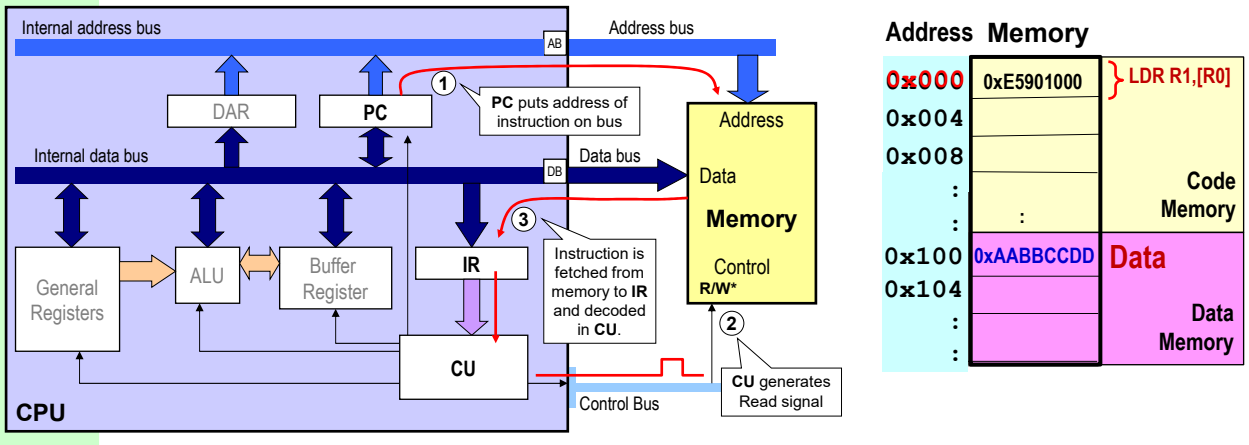
19

CZ1106
CE1106

Basic Execution Cycle

Fetch Cycle - Instruction

- Consider an instruction like: `LDR R1,[R0]`.
  In the fetch cycle, the **PC** points to address of next the instruction and its opcode is fetched into the **IR**.



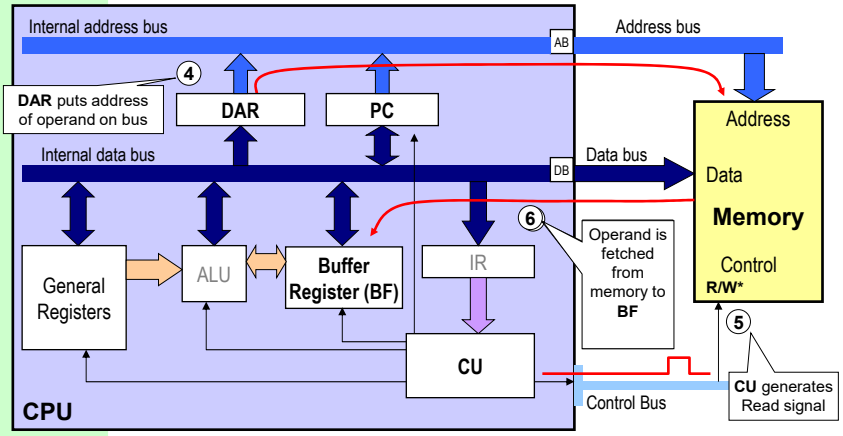
20

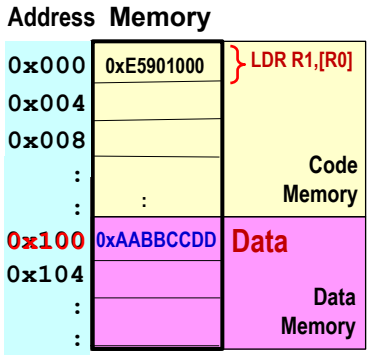
CZ1106
CE1106

Basic Execution Cycle

Fetch Cycle - Operand

- Decoding `LDR R1,[R0]` suggest an operand is needed. Since its is stored in memory, another fetch is required to retrieve operand into CPU.





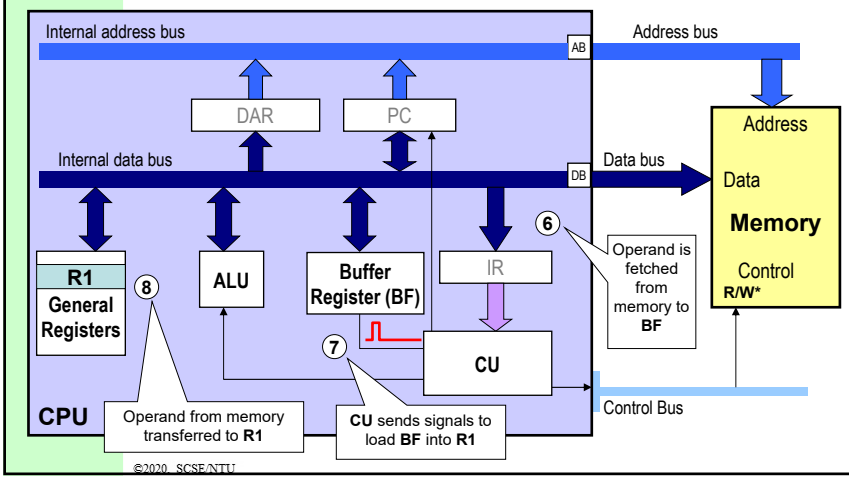
21

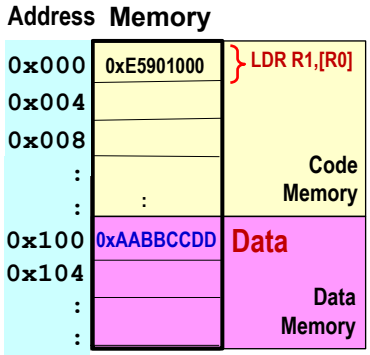
CZ1106
CE1106

Basic Execution Cycle

Execute Cycle - Load

- In the execution cycle of `LDR R1,[R0]`, the operand fetch from memory is loaded into the destination register **R1**.



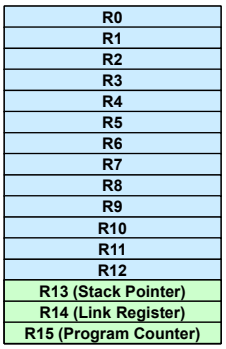


22

CZ1106
CE1106

Program Execution (Summary)

- Execution of a single instruction may involve multiple accesses to memory. In most single memory (von Neumann-type) CPU, different instructions take **different number** of clock **cycles** to execute.
- Data transfer on external bus is slower than within CPU's internal bus. μP system performance is limited by data traffic bandwidth between CPU and memory (also known as the **von Neumann bottleneck**).
- Keeping regularly used operands in CPU **registers** helps reduce memory access.
- Keeping instructions and data in separate memories (**Harvard architecture**) can help make instructions execute in more regular cycles (using parallel fetches) and thus improve performance.



23

CZ1106
CE1106

Chapter 4

Addressing Modes

1

CZ1106
CE1106

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.

2

CZ1106 CE1106

What is an Assembly Program?

- Unlike high-level programming languages, assembly level statements:
  - Are known as **mnemonics**. Each has a one-to-one correspondence with a binary pattern (**machine code**) that is directly understood by CPU.
  - Are **hardware-dependent** and address the architecture of processor directly. (e.g. they are CPU register-aware and reference them by name).
  - Are converted to machine code by an **assembler**.

```
if (a > c)
  b = a;
else
  b = c;
```

C program example

```
      CMP R0,R2
      BLE Else
      MOV R1,R0 ;b=a
      B   Skip
Else  MOV R1,R2 ;b=c
Skip  :
```

ARM assembly program equivalent

3

CZ1106 CE1106

Why Use Assembly Language?

- More efficient codes can be created:
  - Codes with faster execution **speed**.
    e.g. Algorithms for **real-time** signal processing in handheld devices can be **computationally demanding**.
  - More compact program **size**.
    e.g. Low cost embedded devices may have **small memory** capacity but require many functionalities.
  - Exploit **optimized features** of processor's ISA.
    e.g. High-level language compiled codes may not **exploit optimized** instructions, addressing modes and features available in the processor instruction set architecture to produce efficient run-time code.
  - Many cybersecurity jobs needs good knowledge in assembly programming.

4

CZ1106 CE1106

When to Use Assembly Language?

- Critical parts of the operating system's software.
  Especially parts of system kernel that are constantly being executed (e.g. scheduler, interrupt handlers).
- Input/Output intensive codes.
  Device drivers and "loopy" segments of code that processes streaming data (e.g. video decoders, etc).
- Time-critical codes.
  Code that detect incoming sensor signals and respond rapidly, e.g. Anti-lock brake system (ABS) in cars.

Learn More: Google "Is Linux kernel written in assembly"

5

CZ1106 CE1106

Addressing Modes

- Addressing mode (AM) is concerned with how data is **accessed**, not the way data is processed.
  - The correct AM allows the CPU to identify the actual operand or the address location where operand is stored.
- The ARM processor instruction set architecture supports many different addressing modes.
  - Register direct
  - Immediate data
  - Register indirect
  - Register indirect with offset
  - Register indirect with index register
  - Pre and post auto-indexing

Learn More: Google "addressing modes"

6

CZ1106 CE1106

Addressing Mode Examples

Addressing Mode	ARM	Intel
Absolute (Direct)	None	`MOV AX,[1000h]`
Register Direct	`MOV R1,R0`	`MOV AX,DX`
Immediate	`MOV R1,#3`	`MOV AX,0003h`
Register Indirect	`LDR R1,[R0]`	`MOV AX,[BX]`
Register Indirect with Offset	`LDR R1,[R0,#4]`	`MOV AX,[BX+4]`
Register Indirect with Index	`LDR R1,[R0,R2]`	`MOV AH,[BX+DI]`
Implied	`BNE LOOP`	`JMP -8`

7

CZ1106 CE1106

Summary

- Codes written well in assembly language can usually execute **faster** and are smaller in **size**.
- Code for **low-level** OS kernels, **I/O** intensive and **time**-critical operations can benefit significantly from assembly-level coding.
- Understanding the **characteristics** and **application** of different **addressing modes** available in a processor's ISA allows programmers to write efficient codes.

8

CZ1106
CE1106

Chapter 4

Addressing Modes

Register Direct and Immediate Addressing

Learning Objectives (4.2)

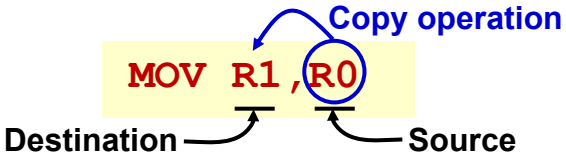
1. Describe what is register direct.
2. Describe what is immediate data and its application.

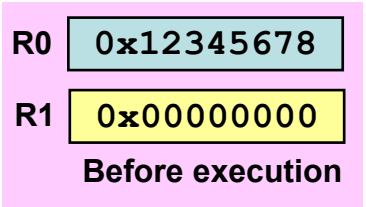
9

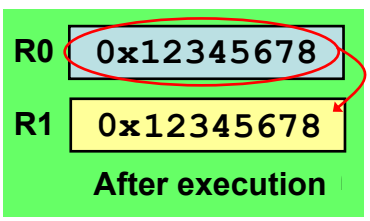
CZ1106
CE1106

Register Direct

- Operand is the content of the specified **register**.
  - Register direct can be used for both destination and source operand.
  - In the `MOV` instruction, the right operand is the source and left operand is the destination.







  - A **fast** addressing mode since no further memory access is involved during execution.
  - Should be used to optimise execution speed.

10

CZ1106 CE1106

Register Direct (cont)

- All ARM's 16 registers can be a register direct operand.
  - These registers can be either a source or destination operand.

```
MOV   R3,LR  ;make copy of LR in R3
MOVS  R0,R0  ;test for N or Z condition in R0
MOV   PC,R1  ;make a jump to address in R1
```

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
SP
LR
PC

11

CZ1106 CE1106

Immediate Addresing

- **Operand** is directly specified **within** the **instruction** itself.
  - "**#**" symbol precedes the immediate value.
  - Example: `MOV R1,#3` — Immediate Value; Copy operation
  - After execution, the **immediate value** is **copied** into the destination register (left operand).
  - Immediate addressing can only be used as a **source operand**.
  - Used for loading **constant values** into registers. Values must be known at the time of coding (e.g. load loop count into a loop counter register).

R1 0x12345678 **Before execution**

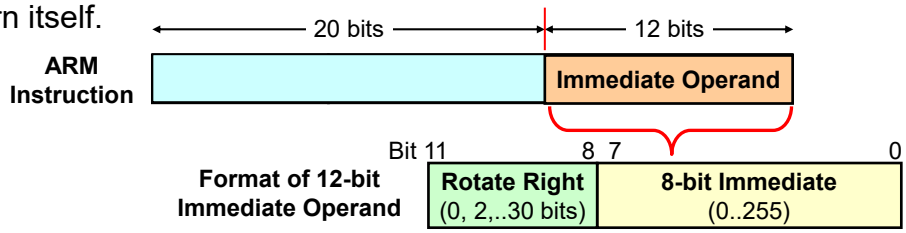
R1 0x00000003 **After execution**

12

CZ1106
CE1106

Immediate Addressing (cont)

- How is the 32-bit immediate value encoded?
  - The immediate value is specified **within the instruction** bit pattern itself.



- How can a 12-bit operand encode a 32-bit immediate value?
  - It can only describe a **subset** of all 2^{32} possible values.
  - Immediate value is a number between (**0..255**) rotated right by **2*n*** bits, where the value of ***n*** is given by 4 bits (**0** ≤ ***n*** ≤ **15**).

13

CZ1106
CE1106

Immediate Addressing (cont)

- Assembler does the necessary calculations and gives warning if requested immediate value cannot be encoded.

```
MOV R3,#0xFF    ;immediate values within 8 bits always valid
MOV R0,#0x100   ;right rotate 8-bit value of 0x01 with n=12
MOV R1,#0x102   ;this is not a valid immediate value
```

- A combination of instructions can be used to achieved the desired immediate values that is not valid.

```
MOV R1,#0x100 ; load 0x100 to R1
ADD R1,R1,#2  ; add 2 to R1
```


14

CZ1106
CE1106

Summary

- Register direct is **efficient** as its execution involves no access to memory.
- Immediate addressing encode the operand **within the instruction**.
  - Like register direct, immediate addressing in the ARM is **efficient** as memory access is not incurred during execution, only when fetching the instruction.
  - Because data is encoded within fixed-length instruction, **only a subset** of immediate values are available.
- Immediate addressing is used when the operand **value is known** during the time of coding (e.g. loading known constants into registers).

15

CZ1106
CE1106

16

CZ1106
CE1106

Chapter 4

Addressing Modes

Register Indirect with Base Register

Learning Objectives (4.3)

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

17

CZ1106
CE1106

Limitation of Register Direct and Immediate Addressing

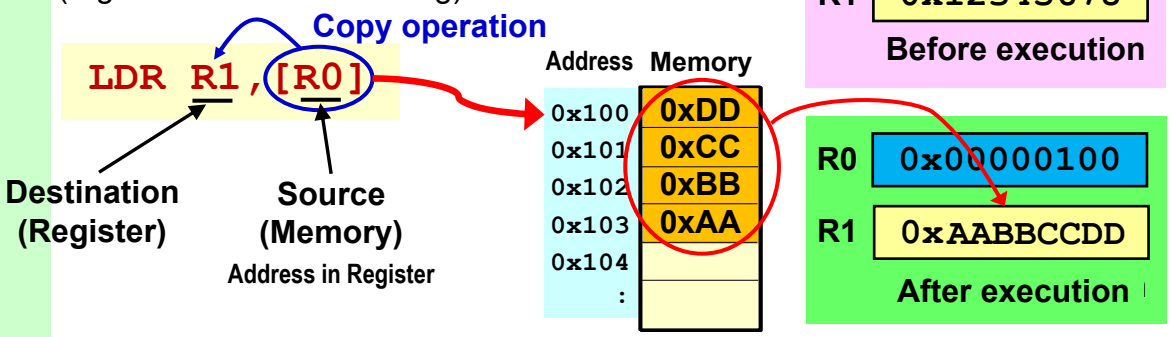
- Register direct and immediate addressing **do not** allow CPU to access operands stored in **memory**.
  - C variables are usually allocated memory for storage (especially large arrays).
- How do you specific a 32-bit address in memory using a 32-bit long instruction?
  - The ARM specifies the 32-bit address of the operand in a 32-bit **register**.
  - The register with the memory address **points to the memory** location where the operand is stored.
  - Memory operand is fetched during instruction execution using **register indirect** addressing.
  - The ARM uses the `LDR` and `STR` mnemonics to access memory operands.

18

CZ1106 CE1106

The LDR Instruction

- The **LDR** operator is used to copy **memory** content to a register.
  - The left operand the destination register.
  - The right source operand is a memory location whose address is contained in a register (register indirect addressing).

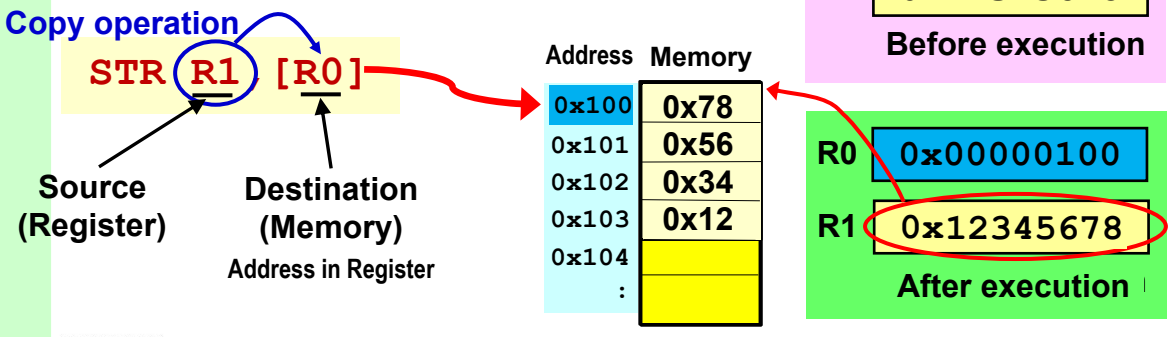


19

CZ1106 CE1106

The STR Instruction

- The **STR** operator is used to copy register content to **memory**.
  - The left operand is always a source register.
  - The right destination operand is a memory location whose address is contained in the indirect register.

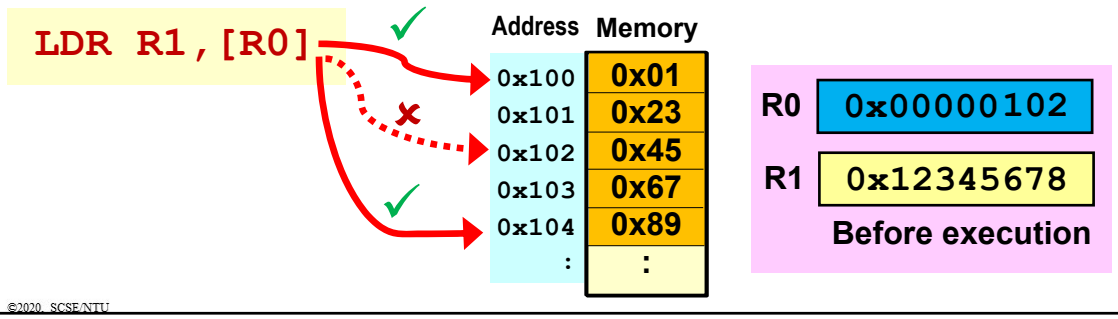


20

CZ1106 CE1106

Data Alignment Constraints

- Access of 32-bit operand from memory must follow data alignment constraints.
  - The **4-byte data** read or written to memory must start at an address that is a **multiple of 4**.
  - The effects of an unaligned memory access depend on the ARM architecture but they invariably result in **performance degradation**.

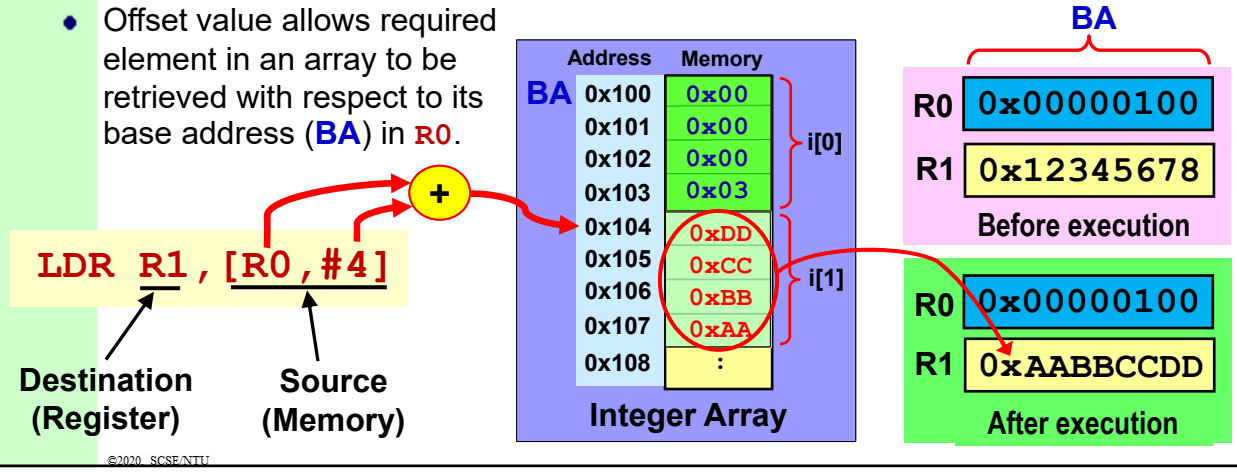


21

CZ1106 CE1106

Register Indirect with Offset

- **Adds** a specific **offset value** to the indirect register to compute the effective address (EA) in memory.
  - **Base Plus Offset** addressing does not change indirect register's content.
  - Offset value allows required element in an array to be retrieved with respect to its base address (**BA**) in `R0`.



22

CZ1106 CE1106

Program Example

Accessing Array Elements

- Use base plus offset to access array element whose index is known during coding time.

```
main()
{
// assume base address
// of array i is 0x100
int i[5];

i[0]=7;
i[4]=7;
}
```

C program example
Assign first & last elements of array `i` with value of 7.

```
MOV   R2,#0x100      1
MOV   R1,#7          2
STR   R1,[R2,#0]     3
STR   R1,[R2,#16]    3
```

Using register indirect with base plus offset

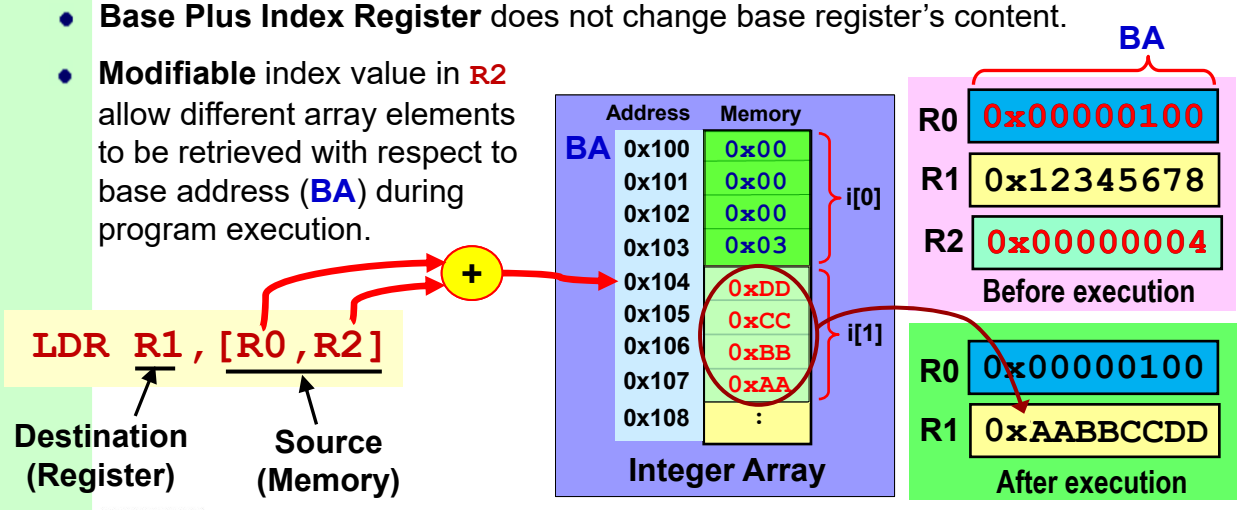
1. Initialize base address of array `i` into register `R2`.
2. Load value of 7 into register `R1`.
3. Store the value of 7 into **i[0]** and **i[4]** using offsets of 0 and 16 of register **R2** respectively .
   - In computing offset, note that each integer element occupies 4 bytes in memory.

23

CZ1106 CE1106

Register Indirect with Index Register

- This variant **adds** the content of the **index register** to the indirect register to compute EA.
  - **Base Plus Index Register** does not change base register's content.
  - **Modifiable** index value in `R2` allow different array elements to be retrieved with respect to base address (**BA**) during program execution.



24

CZ1106 CE1106

Program Example

Clearing All Array Elements

- Use base plus index register to access each array element in turn.

```
main() {
// assume base address
// of array i is 0x100
int i[400];
int n=0;

while (n < 400) {
    i[n] = 0;
    n = n + 1;
    }
}
```

C program example
Initialise all 400 elements in array `i` with zero.

```
MOV   R2,#0x100   (1)
MOV   R0,#0       (2)
MOV   R1,#0
  :
STR   R0,[R2,R1]  (3)
ADD   R1,R1,#4    (4)
```

loop back 399 times

3 ×400 = 1200 cycles

Using register indirect with base plus index

1. Initialize base address of array `i` into register `R2`.
2. Load value of 0 into register `R0` and `R1`(index register).
3. Store 0 in `R0` into `i[n]` using current index value in `R1` plus base address in `R2`.
4. Increment index by 4, the size of each integer element in array.

25

CZ1106 CE1106

Summary

- Register indirect (with the `LDR` and `STR` operators) allows memory operands to be accessed.
- There are two variants of register indirect using base register.
  - Register indirect with **offset** (base plus offset)
  - Register indirect with **index** (base plus index register)
  - Contents in base register **do not change** after execution.
- Given the **base address** of an array, register indirect with base addressing is useful for accessing the contents of the array.
  - Use base plus offset if position of array element is known during coding time
  - Use base plus index if array element position is computed during run time.

26

CZ1106
CE1106

Chapter 4

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.

27

CZ1106
CE1106

Register Indirect with Autoindexing

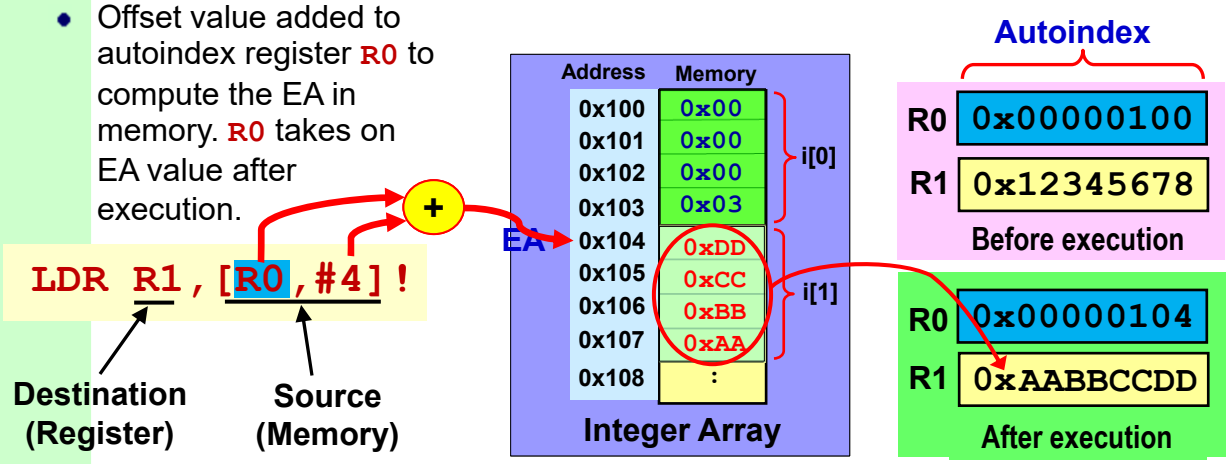
- Recap – Register indirect with base register:
  - The base address in the indirect register can be added with an offset or the contents of index register to compute effective address of operand in memory.
  - Base address is **never modified** after execution as it is assumed to be the sole reference to the start of the array.
- What if we allow the indirect register to be modified before or after computing the effective address?
  - Keep a copy of the array's base address elsewhere.
  - **Autoindexing** allows the indirect register's content to be **modified** during execution.
  - Autoindexing provides an efficient way to access **consecutive** array elements.

28

CZ1106 CE1106

Offset with Autoindexing

- Adds offset value to the autoindex register (AR) to compute effective address (EA) and AR gets modified.
  - "!" in mnemonic causes autoindex register to be modified with the EA.
  - Offset value added to autoindex register R0 to compute the EA in memory. R0 takes on EA value after execution.



29

CZ1106 CE1106

Program Example (Optimised Version)

Clearing All Array Elements

- Use offset with autoindex to efficiently access each array element in turn.

```
main() {
// assume base address
// of array i is 0x100
int i[400];
int n=0;

while (n < 400) {
    i[n] = 0;
    n = n + 1;
    }
}
```

C program example
Initialise all 400 elements in array i with zero.

```
MOV R2,#0x100     1
MOV R1,#0         2
STR R1,[R2]       3
  :
STR R1,[R2,#4]!   4
```

loop back 398 times

2 ×400 = 800 cycles

Using register indirect plus offset with autoindex

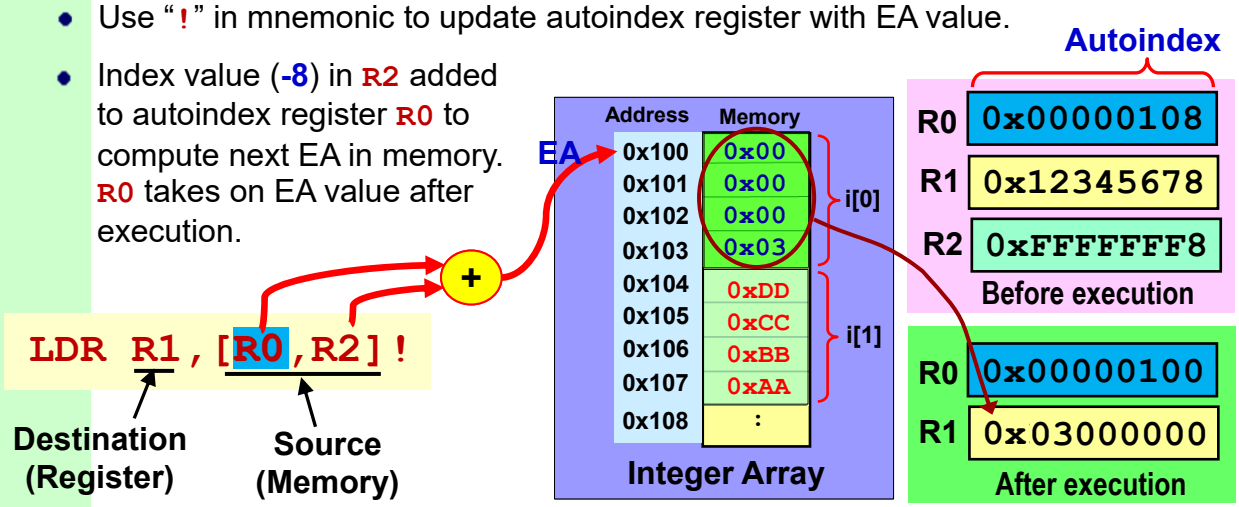
1. Initialize base address of array i into register R2.
2. Load value of 0 into source register R1.
3. Store 0 in R1 into first element of array i[0].
4. Store 0 in R1 into i[n] using current effective address (EA) of autoindex register R2 plus offset 4. Then put this EA into R2.

30

CZ1106 CE1106

Index Register with Autoindex

- Adds index register value to autoindex register (AR) to compute effective address (EA) and AR gets modified.
  - Use "**!**" in mnemonic to update autoindex register with EA value.
  - Index value (**-8**) in `R2` added to autoindex register `R0` to compute next EA in memory. `R0` takes on EA value after execution.



31

CZ1106 CE1106

Pre-index and Post-index

- In **pre-index**, the indirect register is **autoindex before** being used to compute effective address.

```
LDR R1,[R0,#4]!   ; R0 = R0+4
                  ; R1 = mem[R0]
```

Offset with Autoindexing (pre-index)

```
LDR R1,[R0,R2]!   ; R0 = R0+R2
                  ; R1 = mem[R0]
```

Index with Autoindexing (pre-index)

- In **post-index**, the indirect register is used to compute the effective address **before** it is **autoindexed**.

```
LDR R1,[R0],#4    ; R1 = mem[R0]
                  ; R0 = R0+4
```

Offset with Autoindexing (post-index)

```
LDR R1,[R0],R2    ; R1 = mem[R0]
                  ; R0 = R0+R2
```

Index with Autoindexing (post-index)

32

CZ1106 CE1106

Program Example (Alternative Version)

Clearing All Array Elements

- Use offset with **post-index autoindex** to keep all array access within loop.

```
main() {
// assume base address
// of array i is 0x100
int i[400];
int n=0;

while (n < 400) {
    i[n] = 0;
    n = n + 1;
    }
}
```

C program example
Initialise all 400 elements in array `i` with zero.

```
MOV R2,#0x100      1
MOV R1,#0          2
   :
STR R1,[R2],#4     3
```

loop back **399** times

2 ×400 = 800 cycles

Using offset with post-index autoindexing

1. Initialize base address of array `i` into register `R2`.
2. Load value of 0 into source register `R1`.
3. Store 0 in `R1` into **i[n]** using current effective address (EA) in indirect register `R2`. Then add offset `4` to the current EA in `R2` so that `R2` is now pointing to the next array element.

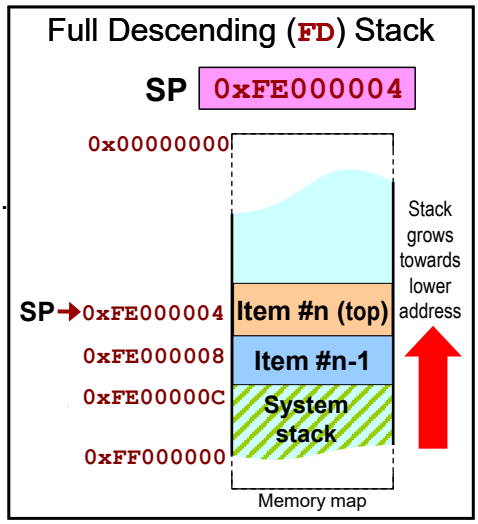
33

CZ1106 CE1106

The System Stack

- A stack is a first-in, last-out linear data structure that is maintained in the memory's data area.
  - The system stack in the ARM is maintained by a dedicated stack pointer (**SP**) or **R13**.
  - The `FD` stack grows towards **lower memory** addresses.
    (e.g. by default, **SP** starts at **0xFF000000** in VisUAL ARM simulator).
  - In the `FD` stack, the **SP** points to the **top item (full)** on the stack (but SP can also point to the next **empty** space on the stack).
  - The 3 basic stack operations are **push**, **pop** and **access** items on the stack.

Learn More: Google "ARM stack implementation"



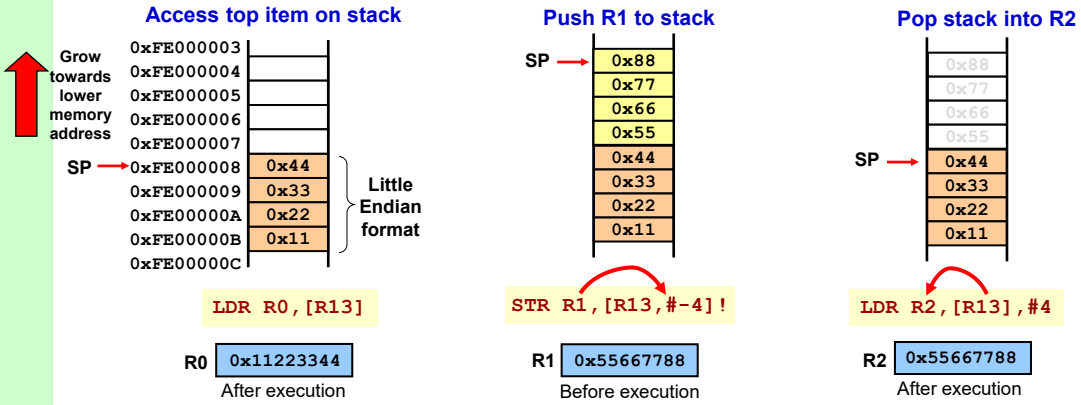
34

CZ1106 CE1106

ARM Stack Implementation (FD)

- The are 4 possible stack implementations supported by the ARM instruction set.
  - **F**ull **D**escending, **F**ull **A**scending, **E**mpty **D**escending and **E**mpty **A**scending

Example of **Full Descending** (FD) stack implementation:

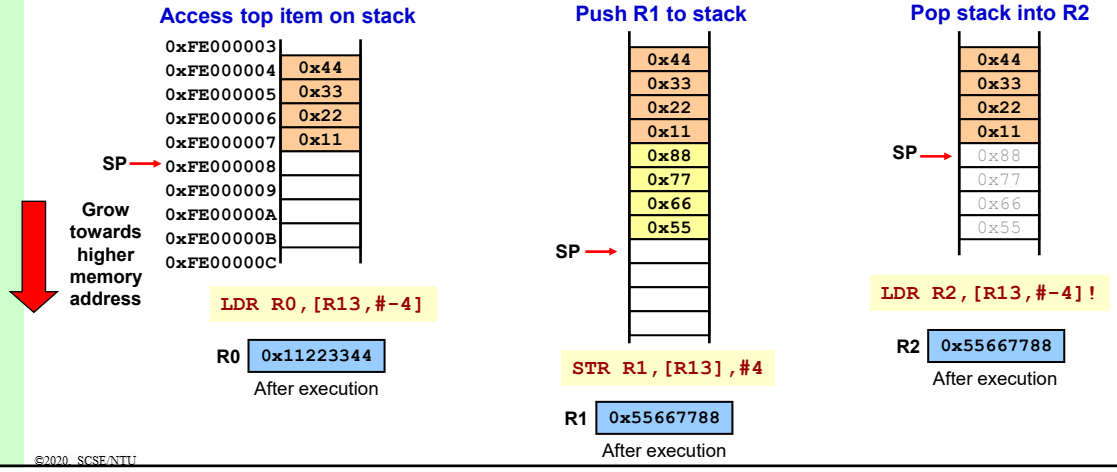


35

CZ1106 CE1106

Empty Ascending Implementation (EA)

- EA is an alternative stack implementation.
  - **Empty** means **SP** points to an available **unoccupied** stack space.
  - **Ascending** means stack grows toward **higher** memory address.



36

CZ1106
CE1106

Summary

- Autoindexing modifies the indirect register besides just computing the effective address.
  - The autoindexing can use either **offset** or **index register**.
  - **Pre-index** does the autoindexing first before computing the effective address.
  - **Post-index** computes the effective address first, then does the autoindexing.
- ARM's autoindexing feature can be used to implement stacks.
  - There are **4** possible stack implementations (FD, FA, ED, EA).
  - For a given stack implementation, the Push and Pop operation must complement each other to ensure the stack grows and collapses correctly.

37

CZ1106
CE1106

38

CZ1106
CE1106

Chapter 4

Addressing Modes

PC-related Addressing Modes

Learning Objectives (4.5)

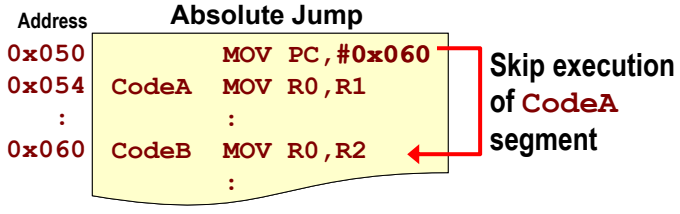
1. Describe difference between absolute & relative jump.
2. Describe the concept of position-independent code and how it is achieved.
3. Describe how data can be accessed using PC relative addressing

39

CZ1106
CE1106

Absolute Jump

- A **new address** can be loaded into the **PC** to alter the sequential order of program execution.
  - An **absolute jump** to a new code position is done by loading the address to jump to into the **PC**.
  - Example: `MOV PC,#0x060 ;Jump to CodeB`



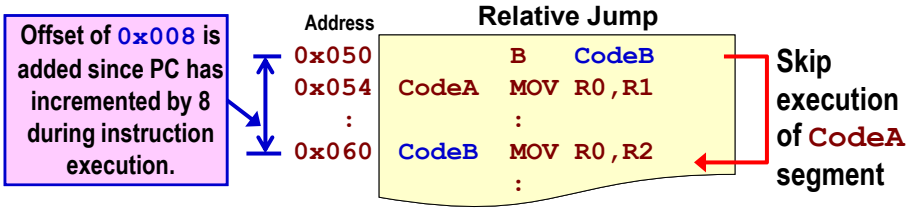
  - Absolute jump is not **position-independent**. This code can only execute correctly in this specific area of code memory.

40

CZ1106 CE1106

Relative Jump

- An **offset** can be added to the **PC** to alter the sequential order of program execution.
  - A **relative jump** is done using the branch instruction (e.g. `B`) with an appropriate **signed offset.** (Note: the range of this offset in ARM is **+/- 32 Mbytes**).
  - Example: `B CodeB ;Jump to CodeB`



  - Relative jump supports **position-independent** code.

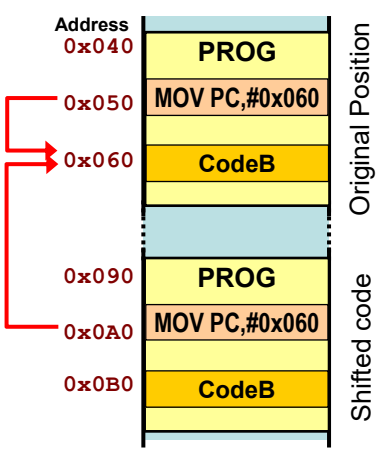
Note: In the ARM processor the PC points 8 bytes ahead of the current executed instruction due to its pipeline architecture.

41

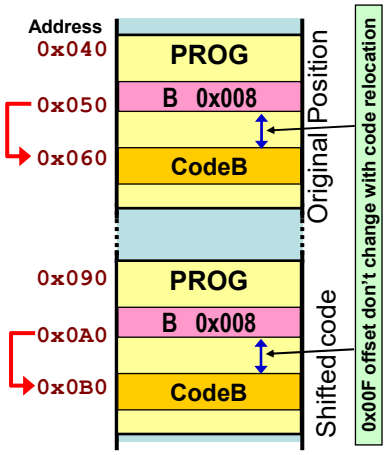
CZ1106 CE1106

Position-Independent Code

- Such programs can be loaded anywhere in memory and still execute correctly (i.e. relocatable).



Does not jump to **CodeB** after **PROG** is shifted to `0x090`



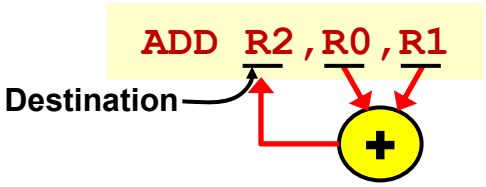
`B` still branch to **CodeB** after **PROG** is shifted to `0x090`

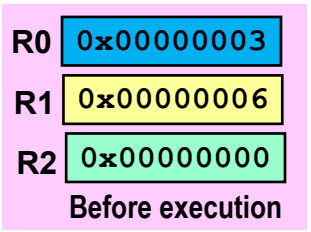
42

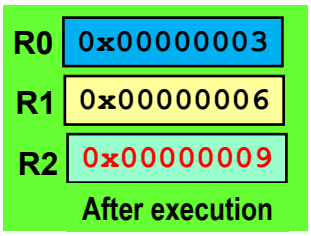
CZ1106 CE1106

ADD Instruction (Introduction)

- This instruction does the **addition** operation.
  - The 3-operand `ADD` instruction adds 2 source operands and puts result into a 3rd destination operand.
  - The right and middle operands are added and the result is placed in the destination register.







  - Destination and middle operands must be registers but rightmost operand can be a **register** or an **immediate** value.

43

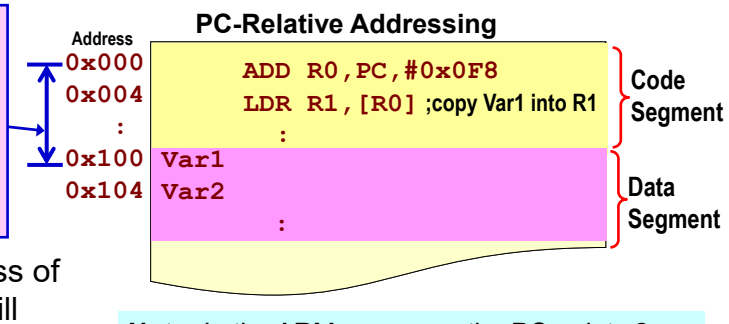
CZ1106 CE1106

Accessing Data

- Position-independent (P-I) programs require data to be accessed relative to the **PC**.
  - **PC-relative addressing** is used to access variables in the data segment of program in memory.
  - E.g.: `ADD R0,PC,#0x0F8` **; Get P-I address of Var1 in Data Seg into R0**

PC-relative offset of `0x0F8` is added since PC has incremented by `8` when executing `ADD` instruction.

PC-relative Offset:
Var address – (PC value + 8)



  - Referencing absolute address of variable in whatever ways will violate **P-I** requirements.

Note: In the ARM processor the **PC** points **8** bytes ahead of the current executed instruction.

44

CZ1106
CE1106

Summary

- Non-sequential execution of code can be achieved by modifying the PC contents directly.
  - The Branch instruction does this by **adding** a **signed offset**.
  - Such **relative** jumps create **position-independent** code.
- **PC-relative addressing** with **appropriate offsets** allows memory data to be accessed in a position-independent manner.

45

CZ1106
CE1106

Chapter 5

Instruction Set

1

CZ1106
CE1106

Chapter 5

Instruction Set

Data Transfer Instructions

Learning Objectives (5.1)

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

2

CZ1106 CE1106

Instruction Set – Basic Categories

- Non system-level instructions in a processor can be typically classified into three basic groups:

Data Transfer	Data Processing	Program Control
ARM examples:	ARM examples:	ARM examples:
`MOV R1,R0`	`ADD R0,R1,R2`	`B Back`
`STR R0,[R2,#4]`	`SUB R1,R2,#3`	`BNE Loop`
`LDR R1,[R2]`	`EOR R3,R3,R2`	`BL Routine`

- **Data transfer** – instructions that move data between registers and/or memory.
- **Data processing** – instructions that modify the data in register through arithmetic or logical operations.
- **Program control** – instructions that alter the normal sequential execution flow of a program.

3

CZ1106 CE1106

Register Data Transfer

- Moves source operand to the destination register.
  - With `MOV`, the source operand can use either **register direct** or **immediate** addressing.

```
MOV   R1,R0   ; make copy of R0 in R1
MOVS  R0,#0   ; move 0 into R0 and set Z flag
```

  - With move complement (NOT) `MVN`, the source operand is bit-wise inverted before moving into the destination register.

```
MVN   R1,R0   ; R1 = NOT (R0)
MVN   R0,#0   ; move 32-bit value of -1 into R0
```

`R0 = 0xFFFFFFFF` after execution

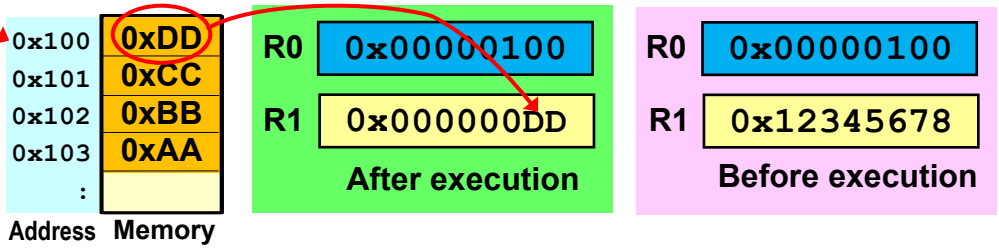
4

CZ1106 CE1106

Memory Data Transfer

- Data in memory can be transferred to or from a register.
  - With **LDR**, the memory data at the effective address is moved to the **destination register** using various **indirect register** addressing modes.

```
LDR   R1,[R0]   ; copy 32-bit value pointed by R0 into R1
LDRB  R2,[R0]   ; copy 8-bit value pointed by R0 into R2 (byte zero-extends to 32 bits)
```



5

CZ1106 CE1106

Memory Data Transfer

- Data in memory can be transferred to or from a register.
  - With **LDR**, the memory data at the effective address is moved to the **destination register** using various **indirect register** addressing modes.

```
LDR   R1,[R0]   ; copy 32-bit value pointed by R0 into R1
LDRB  R2,[R0]   ; copy 8-bit value pointed by R0 into R2 (byte zero-extends to 32 bits)
```

  - With **STR**, the content in **source register** is copied to the effective address in memory using various indirect addressing modes.

```
STR   R1,[R0]   ; copy R1 (4 bytes) starting at address pointed by R0
STRB  R2,[R0,#1]! ; copy byte in R2 to only one address at [R0+1]; then R0=R0+1
```


6

CZ1106 CE1106

Program Example

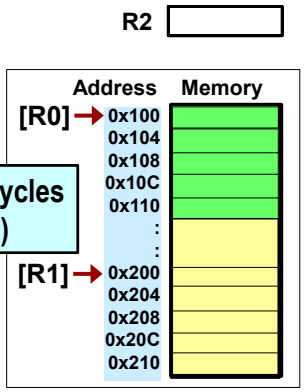
Copying a Block of Memory

- Block copy is used to replicate a contiguous segment of memory from one location to another.
- The **MOV**, **LDR** and **STR** data transfer mnemonics are required to perform the block copy operations.

```
        MOV   R0,#0x100   ; setup source pointer
        MOV   R1,#0x200   ; setup destination pointer
loop    LDR   R2,[R0]     ; memory to register transfer
        STR   R2,[R1]     ; register to memory transfer
        ADD   R0,R0,#4    ; increment source pointer
        ADD   R1,R1,#4    ; increment destination pointer
```

loop back 5 times

30 cycles (6x5)



Block copy 5 words starting at address 0x100 to 0x200

7

CZ1106 CE1106

Program Example (optimised version)

Copying a Block of Memory

- The code can be further optimised for speed and size by using the autoindexing feature.
- The register indirect with **post-index** autoindexing will **automatically** add the 4 offset to the array pointers **after** memory access.

```
        MOV   R0,#0x100    ; setup source pointer
        MOV   R1,#0x200    ; setup destination pointer
loop    LDR   R2,[R0],#4   ; memory to register transfer
                           ; with post index autoindexing
        STR   R2,[R1],#4   ; register to memory transfer
                           ; with post index autoindexing
```

loop back 5 times

20 cycles (4x5)

Block copy 5 words starting at address 0x100 to 0x200

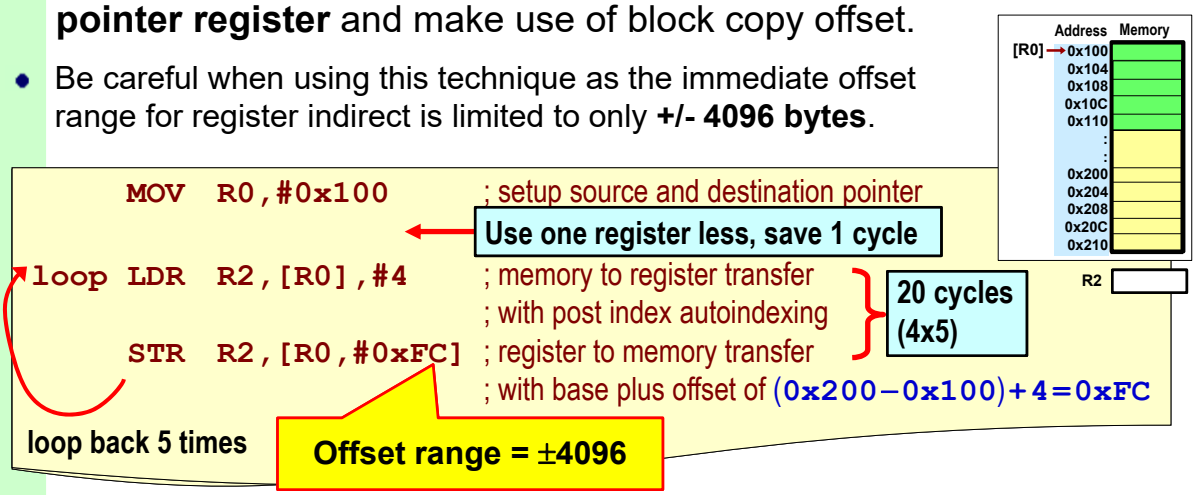
8

CZ1106 CE1106

Program Example (further optimisation version)

Copying a Block of Memory

- Further minor optimisation by using only **one pointer register** and make use of block copy offset.
  - Be careful when using this technique as the immediate offset range for register indirect is limited to only **+/- 4096 bytes**.



Block copy 5 words starting at address 0x100 to 0x200

9

CZ1106 CE1106

Summary

- The **efficient** data transfer instruction **MOV** and **MVN** are probably the most commonly used instructions.
  - Can be used with the **register direct** and **immediate** addressing modes.
- Memory data transfer requires the use of **LDR** and **STR** instructions.
  - Numerous variants of **register indirect** addressing modes can be used.
  - Memory data transfer instructions require **two** clock cycles to execute.
- Byte-sized memory access can be done using **LDRB** and **STRB**.
  - Byte moved into register is zero-extended.
  - Byte access of memory does not have data alignment restrictions.

10

CZ1106
CE1106

Chapter 5

Instruction Set

Arithmetic Instructions

Learning Objectives (5.2)

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

11

CZ1106
CE1106

Instruction Set – Basic Categories

- Non system-level instructions in a processor can be typically classified into three basic groups:

Data Transfer	Data Processing	Program Control
ARM examples:	ARM examples:	ARM examples:
`MOV R1,R0`	`ADD R0,R1,R2`	`B Back`
`STR R0,[R2,#4]`	`SUB R1,R2,#3`	`BNE Loop`
`LDR R1,[R2]`	`EOR R3,R3,R2`	`BL Routine`

- Data transfer – instructions that move data between registers and/or memory.
- **Data processing** – instructions that modify the data in register through **arithmetic**, logical or shift operations.
- Program control – instructions that alter the normal sequential execution flow of a program.

12

CZ1106 CE1106

Arithmetic Instructions

- The basic arithmetic operations are **Add** and **Subtract**.
  - These arithmetic instructions involve **three** operands.

```
ADD R2,R0,R1
```

  - Add and subtract can only involve **registers** or **immediate values** (as source operand).

```
ADD R2,R0,#4
```

  - The ARM instruction set provides some **variants** of the basic add and subtract operations to provide more flexibility during programming.

ADD	(addition)
SUB	(subtraction)
RSB	(reverse subtraction)
ADC	(add with carry)
SBC	(subtract with carry)
RSC	(reverse subtract with carry)

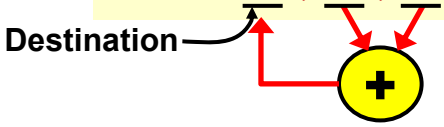
13

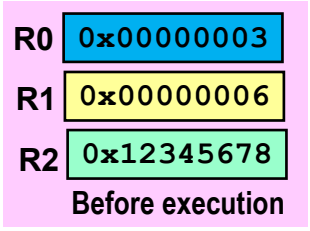
CZ1106 CE1106

ADD Instruction

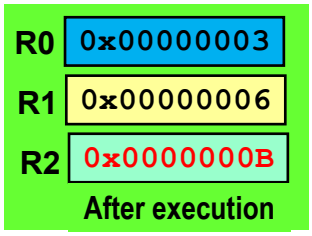
- Addition is a **commutative** operation that adds two source operands (order is immaterial).
  - But only **rightmost** operand can take on an **immediate** value. The other operands must be registers

```
ADDS R2,R0,#8
```





  - In order to influence the condition code flags (**N**,**Z**,**V**,**C**), the "**S**" suffix must be added in the mnemonic.



14

CZ1106 CE1106

Condition Code Flags and ADD

- `ADD` can affect all the `N`, `Z`, `V`, `C` flags.

		Signed Number	Unsigned Number
(+ve)	0000 0001	(1)	(1)
(+ve)	+0111 1111	(127)	(127)
(-ve)	1000 0000	(-128)	(128)

N=1, V=1 ← 2's complement oVerflow

		Signed Number	Unsigned Number
(+ve)	0000 0001	(1)	(1)
(-ve)	+1111 1111	(-1)	(255)
(+ve)	0000 0000	(0)	(0)

Z=1, C=1 ← unsigned overflow

- The `V` flag when set, indicates an **overflow** when adding **signed** numbers.
- Overflow is detected when both **signed numbers** added have the **same sign** but the result has the **opposite sign**.
- The `C` flag when set, indicates an **overflow** when adding **unsigned** numbers.

15

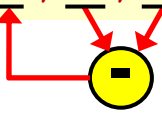
CZ1106 CE1106

SUB Instruction

- Subtraction is a **non-commutative** operation.
- All operands are register but the **rightmost** operand (subtrahend) can take on an **immediate** value.

```
SUBS R2,R0,#4  ; R2=R0-4
```

Destination



- To influence all the condition code flags (`N`,`Z`,`V`,`C`), the "`S`" suffix must be added to the `SUB` mnemonic.
- `RSB` can be used to reverse the subtraction order.

```
RSBS R2,R0,#4  ; R2=4-R0
```

R0	0x00000009
R1	0x00000006
R2	0x12345678

Before execution

R0	0x00000009
R1	0x00000006
R2	0x00000005

After executing `SUB`

16

CZ1106 CE1106

SUB Instruction (cont)

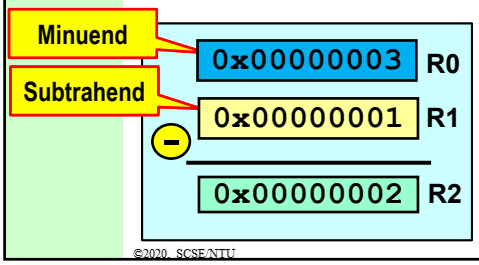
- Subtraction is done by **adding** the minuend to the **negated** subtrahend.

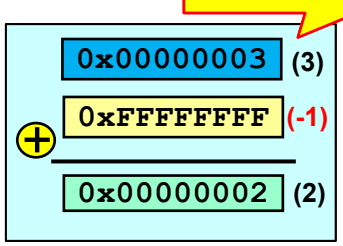
A – B = A + (-B) — 2's complement operation

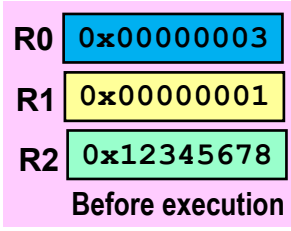
- **2's complement** is used to negate subtrahend.

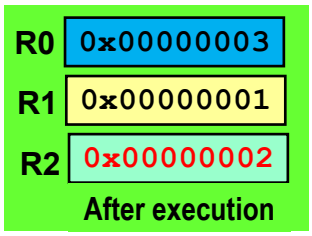
```
SUB R2,R0,R1 ; R2=R0–R1
```

In binary
```
  ...0000 0011
+ ...1111 1111
  ...0000 0010
```







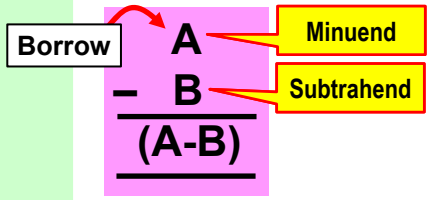


17

CZ1106 CE1106

Condition Code Flags and SUB

- **SUB** can affect all the **N**, **Z**, **V**, **C** flags.
- In the ARM's **SUB** instruction, the **C** flag clears (**C**=0) if the subtraction produce a **borrow** and **C** sets (**C**=1)otherwise.
- A borrow occurs in subtraction when the **unsigned value** of the Minuend is **less** than the unsigned value of the Subtrahend.



Minuend Subtrahend
unsigned (A) < unsigned (B) **C=0**

Minuend Subtrahend
unsigned (A) ≥ unsigned (B) **C=1**

- The **V** flag is set (**V**=1) when the result is out of the signed 32-bit range.

$-2^{31} < (A-B) \geq +2^{31}$ **V=1**

- An **unsigned underflow** is indicated by (**C**=0).

Learn More: Google "ARM subtraction carry flag"

18

CZ1106 CE1106

Program Example

Convert Negative to Positive

- The code converts an integer array of 10 negative values to its equivalent positive values.
- The 2's complement operation on each retrieved word converts it from -ve to +ve. The –ve word in memory is then replace with its +ve equivalent.

```
        MOV   R0,#0x100   ; setup source pointer
Loop    LDR   R1,[R0]     ; get next –ve int in mem to R1
        MVN   R1,R1       ; complement –ve word
        ADD   R1,R1,#1    ; add 1 to do 2's complement
        STR   R1,[R0],#4  ; put result back then increment
                          ; to next array element
```

1st –ve integer converted

R1 0x00000001

60 cycles (6x10)

loop back 9 times

Address	Memory	
[R0] → 0x100	0xFFFFFFFF	(-1)
0x104	0xFFFFFFFE	(-2)
0x108	0xFFFFFFFD	(-3)
0x10C	0xFFFFFFFC	(-4)
0x110	0xFFFFFFFB	(-5)
0x114	0xFFFFFFFA	(-6)
0x118	0xFFFFFFF9	(-7)
0x11C	0xFFFFFFF8	(-8)
0x120	0xFFFFFFF7	(-9)
0x124	0xFFFFFFF6	(-10)

Convert 10 words starting at address 0x100 to its +ve values

19

CZ1106 CE1106

Program Example (optimized version)

Convert Negative to Positive

- The `SUB` instruction can be used to do the negation operation more efficiently.
- By **subtracting** the value **0** with the –ve value retrieved from memory, the result will be its +ve equivalent.

e.g. 0 – (-5) = +5

```
        MOV   R0,#0x100   ; setup source pointer
        MOV   R2,#0       ; load zero into R2
Loop    LDR   R1,[R0]     ; get next –ve int in mem to R1
        SUB   R1,R2,R1    ; do (0 – R1) to get +ve value
        STR   R1,[R0],#4  ; put result back then increment
                          ; to next array element
```

R2 = 0

50 cycles (5x10)

loop back 9 times

R1 = R2 – R1
R1 = 0 – R1

R1 0x00000001

Address	Memory	
[R0] → 0x100	0xFFFFFFFF	(-1)
0x104	0xFFFFFFFE	(-2)
0x108	0xFFFFFFFD	(-3)
0x10C	0xFFFFFFFC	(-4)
0x110	0xFFFFFFFB	(-5)
0x114	0xFFFFFFFA	(-6)
0x118	0xFFFFFFF9	(-7)
0x11C	0xFFFFFFF8	(-8)
0x120	0xFFFFFFF7	(-9)
0x124	0xFFFFFFF6	(-10)

Convert 10 words starting at address 0x100 to its +ve values

20

CZ1106 CE1106

Program Example (further optimisation version)

Convert Negative to Positive

- The **RSB** instruction can be used to do the negation with an additional register.
- The **reverse subtract** instruction allow the immediate value of **0** to be used as the minuend, thereby removing the need for a zeroed register.

```
        MOV   R0,#0x100    ; setup source pointer
                           <- Use one register less, save 1 cycle
Loop    LDR   R1,[R0]      ; get next –ve int in mem to R1
        RSB   R1,R1,#0     ; do (0 – R1) to get +ve value
        STR   R1,[R0],#4   ; put result back then increment
                           ; to next array element
```

loop back 9 times

R1=0 – R1

50 cycles (5x10)

R1 0x00000001

[R0] →

Address	Memory	
0x100	0xFFFFFFFF	(-1)
0x104	0xFFFFFFFE	(-2)
0x108	0xFFFFFFFD	(-3)
0x10C	0xFFFFFFFC	(-4)
0x110	0xFFFFFFFB	(-5)
0x114	0xFFFFFFFA	(-6)
0x118	0xFFFFFFF9	(-7)
0x11C	0xFFFFFFF8	(-8)
0x120	0xFFFFFFF7	(-9)
0x124	0xFFFFFFF6	(-10)

Convert 10 words starting at address 0x100 to its +ve values

21

CZ1106 CE1106

Carry-based Arithmetic Instructions

- ARM provides arithmetic instructions that takes the **carry bit** into consideration.
- These instructions are mainly used to support **multi-precision** arithmetic that involves data size larger than the 32-bit registers in the ARM CPU.

```
ADC R2,R0,R1 ; R2=R0+R1+C
```
ADD with carry

```
SBC R2,R0,R1 ; R2=R0–R1+NOT(C)
```
SUB with carry

```
RSC R2,R0,R1 ; R2=R1–R0+NOT(C)
```
RSB with carry

- Like the other arithmetic instructions, the "**S**" suffix can be added to the mnemonic to influence the condition code flags (**N**,**Z**,**V**,**C**),.

22

CZ1106
CE1106

Summary

- The **ADD** and **SUB** instructions are 3-operand instructions.
  - Supports **register direct** and **immediate** addressing (rightmost operand only).
  - Influence all condition code flags (**N**,**Z**,**V**,**C**) when "**S**" suffix is used.
  - **SUB** is non-commutative. **RSB** allows the minuend to be an **immediate value**.
- Arithmetic instructions that incorporate the carry flag (**C**) can be employed for multi-precision arithmetic.

23

CZ1106
CE1106

24

24

CZ1106
CE1106

Chapter 5

Instruction Set

Logical, Shift and Rotate Instructions

Learning Objectives (5.3)

1. Describe the operation and uses of the various logical instructions.
2. Describe the operation and uses of the various shift and rotate instructions.
3. Describe how multiplication and division can be done using bit shift.

25

CZ1106
CE1106

Instruction Set – Basic Categories

- Non system-level instructions in a processor can be typically classified into three basic groups:

Data Transfer	Data Processing	Program Control
ARM examples:	ARM examples:	ARM examples:
`MOV R1,R0`	`ADD R0,R1,R2`	`B Back`
`STR R0,[R2,#4]`	`SUB R1,R2,#3`	`BNE Loop`
`LDR R1,[R2]`	`EOR R3,R3,R2`	`BL Routine`

- Data transfer – instructions that move data between registers and/or memory.
- **Data processing** – instructions that modify the data in register through arithmetic, **logical or shift** operations.
- Program control – instructions that alter the normal sequential execution flow of a program.

26

CZ1106 CE1106

Logical Instructions

- Logical instructions provide various **Boolean** operators.
  - `MVN` is a **two-operand** instruction that does the **NOT** operation.

Example: `MVNS R2,R2`

R2 `0x00000000` → R2 `0xFFFFFFFF` **N=1** and **Z=0**

Before execution / After execution

  - The `AND`, `ORR` and `EOR` operators are **three-operand** instructions for the **AND**, **OR** and **EX-OR** operations respectively.

Example: `ORRS R1,R1,#0x0000FFFF`

R1 `0x12345678` → R1 `0x1234FFFF` **N=0** and **Z=0**

Before execution / After execution

  - The "`S`" suffix can be used to influence the **N** and **Z** bits in the CC flags.

27

CZ1106 CE1106

Logical Instructions
AND, OR and EOR Applications

- The basic logical instructions can be used to:
  - `AND` – **clear** specific bits in destination operand.
  - `ORR` – **set** specific bits in destination operand.
  - `EOR` – **complement** specific bits in destination operand.

AND truth table

A	B	Z = A.B
0	**0** *	0
0	1	0
1	**0** *	0
1	1	1

* Binary **0 mask** is used to **clear** the bit

OR truth table

A	B	Z = A+B
0	0	0
0	**1***	1
1	0	1
1	**1***	1

* Binary **1** mask is used to **set** the bit

EX-OR truth table

A	B	Z = A⊕B
0	0	0
0	**1***	1
1	0	1
1	**1***	0

* Binary **1** mask is used to **complement** the bit

28

CZ1106 CE1106

Instruction examples

AND, ORR and EOR Applications

Bits 7 6 5 4 3 2 1 0

R0 `..01010101`

Initial condition of least significant 8 bits register **R0**

- e.g. `AND R1,R0,#..11110000` (e.g. `AND R1,R0,#0xF0`)

  R1 `..01010000` Bits 0 to 3 cleared after execution

- e.g. `ORR R0,R0,#..11110000` (e.g... `ORR R0,R0,#0xF0`)

  R0 `..11110101` Bits 4 to 7 set after execution

- e.g. `EOR R2,R0,#..11110000` (e.g. `EOR R2,R0,#0xF0`)

  R2 `..10100101` Bits 4 to 7 inverted after execution

29

CZ1106 CE1106

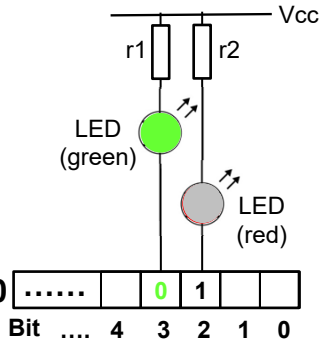
Program Example

Alternate LED Flashing

- Turn **Green** and **Red** LEDs on and off alternately.
- **Green** and **Red** LED states are mapped to **bits 3** and **2** of **R0** respectively. All other bits must not be affected during pattern change.
- `AND` is used to turn on the **active-low** LEDs and `ORR` is used to turn them off. Two patterns per cycle is needed to alternate the ON and OFF between LEDs.

```
Loop  AND   R0,R0,#0xFFFFFFFB   ; turn on Red (bit 2 = 0)
      ORR   R0,R0,#0x00000008   ; turn off Green (bit 3 = 1)
      output pattern in R0 and time delay
      AND   R0,R0,#0xFFFFFFF7   ; turn on Green (bit 3 = 0)
      ORR   R0,R0,#0x00000004   ; turn off Red (bit 2 = 1)
      output pattern in R0 and time delay
```

loop back



Alternating patterns for bits 3 and 2 in R0

30

CZ1106 CE1106

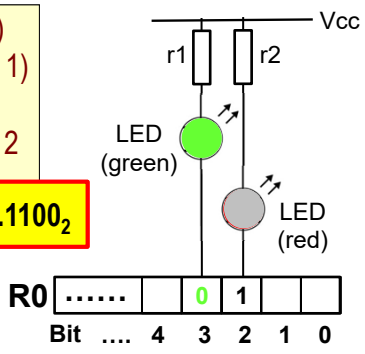
Program Example (Improved version)

Alternate LED Flashing

- Turn **Green** and **Red** LEDs on and off alternately.
- **Green** and **Red** LED states are mapped to **bits 3** and **2** of **R0** respectively. All other bits must not be affected during pattern change.
- Once the alternate pattern for the two LEDs are in place, **EOR** can be used to **flip** or **invert** their state after each cycle.

```
        AND  R0,R0,#0xFFFFFFFB   ; turn on Red (bit 2 = 0)
        ORR  R0,R0,#0x00000008   ; turn off Green (bit 3 = 1)
        output pattern in R0 and time delay
Loop    EOR  R0,R0,#0x0000000C   ; flip state of bits 3 and 2
        output pattern in R0 and time delay
        loop back
```

EOR mask = 00 .....1100_2



Alternating patterns for bits 3 and 2 in R0

31

CZ1106 CE1106

Shift and Rotate Instructions

- ARM has several shift and rotate operations:
  - Logical Shift Left (**LSL**) and Logical Shift Right (**LSR**).

  - Arithmetic Shift Right (**ASR**)

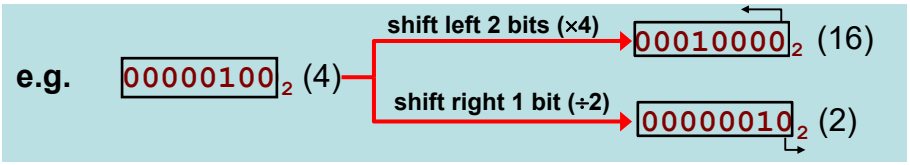
  - Rotate Right (**ROR**) and Rotate Right Extended (**RRX**).

32

CZ1106 CE1106

Doing Arithmetic with Shift

- Shift performs multiply (shift left) or divide (shift right) by a factor of 2^N, where N is the no. of bits shifted.



- In signed or unsigned **multiply**, binary "0"' is shifted into the LSB of the register from the right using Logical Shift Left (**LSL**).
- In **unsigned divide**, binary "0" is shifted into the MSB of the register from the left using Logical Shift Right (**LSR**).
- In **signed divide**, the sign bit is shifted into the MSB from the left using Arithmetic Shift Right (**ASR**).

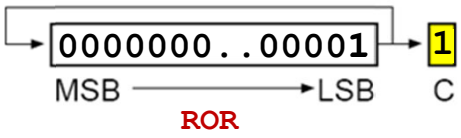
- The "**S**" suffix is used on the data processing operator to influence the **C** flag.

33

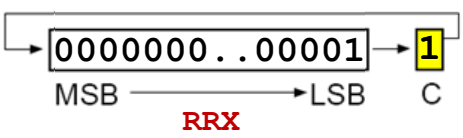
CZ1106 CE1106

Rotate Operations

- Rotate is also called **cyclical shift**, as no bits in the register is lost during the shifting operation.
- In basic rotate right (**ROR**), the bit shifted out of register is returned in at the leftmost end and is also placed into the **C**-flag.



- In rotate right extended (**RRX**), the **C**-flag is shifted into the register at the leftmost end, while the bit shifted out replaces the current **C**-flag.



34

CZ1106
CE1106

Shift and Rotate Mnemonics

- The ARM **efficiently combines** the shift operation with the data transfer or processing instruction.
- Shift operation is applied to the **rightmost** operand (2nd source operand).
- Number of bits to shift is specified as an **immediate** value or a value within a **register** (dynamic shift):

```
MOV  R0,R0,LSL #1  ; R0=R0<<1
```
Shift R0 left by 1 bit

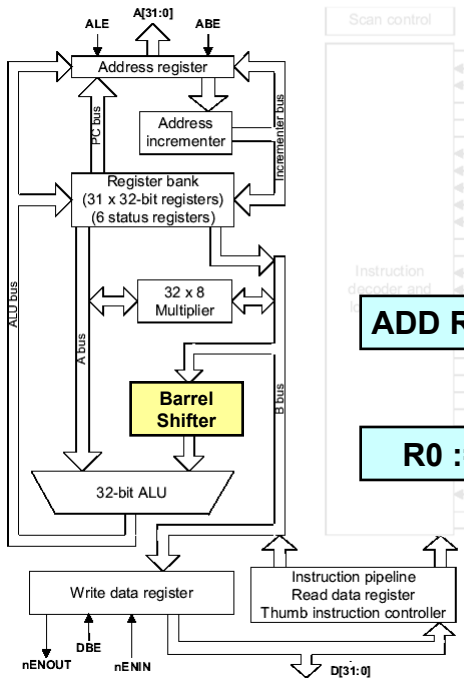
```
ADD  R2,R1,R0,LSR #2  ; R2=R1+R0>>1
```
ADD R1 with 2-bit right shifted R0. Put result in R2.

```
ADDS R2,R1,R0,LSL R4  ; R2=R1+R0<<R4
```
Shift R0 by R4 bits before ADD with R1. Update N,Z,V,C flags and result in R2.

```
ORRS R0,R0,R0,ROR #1  ; R0=R0||R0>>1
```
OR R0 with a 1-bit right rotated version of itself. Set C flag if bit rotated out is 1.

35

CZ1106
CE1106



ADD R0, R1, R2, LSR #2

R0 := R1 + (R2 >> 2)

AMR7TDMI Processor Logic

36

CZ1106 CE1106

Program Example

Count Number of 1's

- Count the number of binary 1s in register **R0**.
- Rotate R0 32 times (ROR), at each rotate use AND to mask all bits except LSB. Then add the LSB. The cumulated total will be the number of 1s in R2.

```
        MOV   R2,#0           ; clear 1-counter for binary 1s
        MOV   R3,#32          ; set loop counter to 32 times
Loop    MOV   R0,R0,ROR #1    ; rotate right 1 bit
        AND   R1,R0,#1        ; clear all bits except LSB
        ADD   R2,R2,R1        ; add LSB to 1-counter
        SUB   R3,R3,#1        ; decrement loop counter
```

loop back 31 times

Initial R0 = 0x22222222

R0	0x11111111
R1	0x00000001
R2	0x00000001
R3	0x0000001F

AND mask of 0x0000001

Changes after one loop

Count the number of binary 1s in R0

37

CZ1106 CE1106

Program Example (optimized version)

Count Number of 1's

- Count the number of binary 1s in register **R0**.
- LSR is used to shift content in R0 1 bit at a time into the C flag. Then ADDC can be used to sum the 1s going into the C flag. Loop ends when R0 has no more 1s and Z flag is set

```
        MOV   R2,#0           ; clear 1-counter for binary 1s
Loop    MOVS  R0,R0,LSR #1    ; 1-bit right shift to move LSB
                              ; into C flag
        ADC   R2,R2,#0        ; add C flag to 1-counter
```

loop back if not zero

R2 = R2 + 0 + C

Initial R0 = 0x22222222

R0	0x11111111	C = 0
R2	0x00000000	
R0	0x08888888	C = 1
R2	0x00000001	

1st loop

2nd loop

Count the number of binary 1s in R0

38

CZ1106
CE1106

Summary

- Logical instructions such as **`AND`**, **`ORR`**, **`EOR`** can be used to **clear**, **set** and **complement** specific bits in a register, respectively.
- Arithmetic shift instruction can be used as a fast way of implementing **multiplication** and **division** by values of 2^N.
- In the ARM, shift and rotate operations are used in **conjunction** with data transfer and data processing operations.

39

CZ1106
CE1106

40

40

CZ1106
CE1106

Chapter 5

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.

41

CZ1106
CE1106

Instruction Set – Basic Categories

- Non system-level instructions in a processor can be typically classified into three basic groups:

Data Transfer	Data Processing	Program Control
ARM examples:	ARM examples:	ARM examples:
`MOV R1,R0`	`ADD R0,R1,R2`	`B Back`
`STR R0,[R2,#4]`	`SUB R1,R2,#3`	`BNE Loop`
`LDR R1,[R2]`	`EOR R3,R3,R2`	`BL Routine`

Covered in Modular Programming

- Data transfer – instructions that move data between regi ory.
- Data processing – instructions that modify the data ugh arithmetic, logical or shift operations.
- **Program control** – instructions that alter the normal sequential execution flow of a program.

42

CZ1106
CE1106

Program Control Instructions

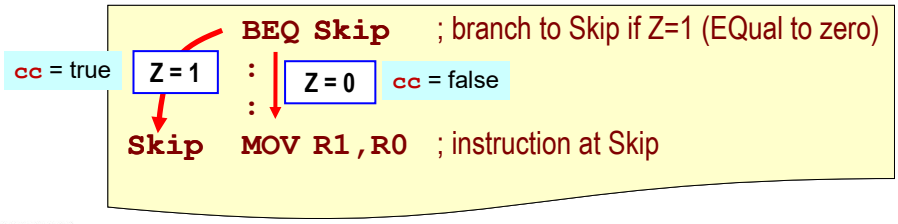
- These instructions facilitate the **disruption** of a program's normal **sequential** flow.
  - The disruption of sequential flow is implemented by modifying the contents of the Program Counter (**PC**).
  - The content of the **PC** can be modified **directly** or by using a **Branch** instruction.
  - A jump can be executed based on a given condition (e.g. if result of previous execution is negative) and this is called a **conditional branch**.
  - Conditional branch is useful for implementing:
    - conditional constructs (e.g. **if** or **if-else**)
    - loop constructs (e.g. **for** or **while** loops)

43

CZ1106
CE1106

Conditional Branch (Bcc)

- ARM provides conditional branch using **`Bcc`**.
  - If the condition specified in the condition field (**`cc`**) is **true**, a displacement is added to the **PC**, otherwise next instruction is executed.
  - **`Bcc`** uses **PC-relative** addressing mode with a displacement range of **±32MB**.
  - The **PC** value used to compute required displacement is **8 bytes** ahead of the current **`Bcc`** being executed.
  - **`Bcc`** is used with **address labels** that allows the assembler to compute the required displacement values.



44

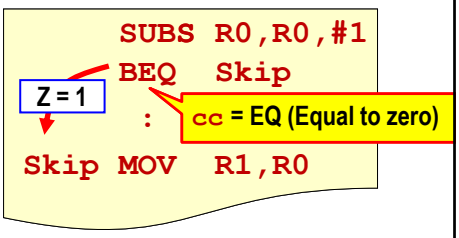
CZ1106 CE1106

Test Conditions for Bcc

- ARM provides **different** conditional branch options.
  - 15 possible conditions is permitted in the condition field (`cc`) using combinations of the `N`, `Z`, `V`, `C` flags.

e.g. `Bcc`	Operation and CC flag conditions	
`B` or `BAL`	PC ← PC ± n	**Branch Always**
`BEQ`	If Z = 1, PC ← PC ± n	**Branch Equal**
`BVS`	If V = 1, PC ← PC ± n	**Branch Overflow Set**

  - **Flexible** conditional branch can be programmed based on outcome of instructions **prior** to `Bcc`.
  - The choice of condition (`cc`) is dependent on whether the test is for a **signed** or **unsigned** computation.



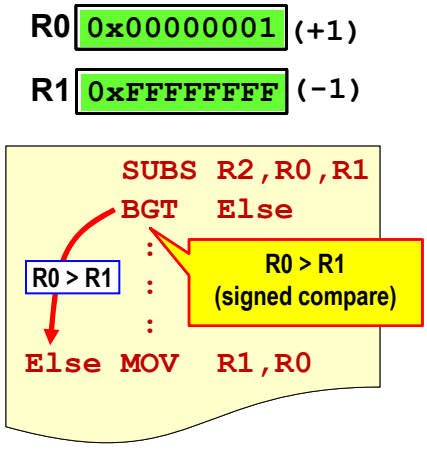
45

CZ1106 CE1106

Different Bcc Conditions

- There are 15 possible conditional tests for `Bcc`.

Suffix	Flags	Meaning	
`EQ`	Z = 1	Equal	
`NE`	Z = 0	Not equal	
`CS or HS`	C = 1	Higher or same, **unsigned**	Unsigned comparison
`CC or LO`	C = 0	Lower, **unsigned**	Unsigned comparison
`MI`	N = 1	Negative	
`PL`	N = 0	Positive or zero	
`VS`	V = 1	Overflow	
`VC`	V = 0	No overflow	
`HI`	C = 1 and Z = 0	Higher, **unsigned**	Unsigned comparison
`LS`	C = 0 or Z = 1	Lower or same, **unsigned**	Unsigned comparison
`GE`	N = V	Greater than or equal, **signed**	Signed comparison
`LT`	N != V	Less than, **signed**	Signed comparison
`GT`	Z = 0 and N = V	Greater than, **signed**	Signed comparison
`LE`	Z = 1 and N != V	Less than or equal, **signed**	Signed comparison
`AL`	Can have any value	Always. This is the default when no suffix is specified.	



46

CZ1106 CE1106

Program Example

Count Number of 1's

- Count the number of binary 1s in register **R0**.
- Loop counter `R3` is initialized to 32 at the start.
- `SUBS` is used to decrement `R3` to zero and set the `Z` flag when that happens.
- `BNE` is used to test for `Z`=0, until `Z`=1 (i.e. `R3`=0), it will keep looping back.

```
        MOV   R2,#0           ; clear 1-counter for binary 1s
        MOV   R3,#32          ; set loop counter to 32 times
Loop    MOV   R0,R0,ROR #1    ; rotate right 1 bit
        AND   R1,R0,#1        ; clear all bits except LSB
        ADD   R2,R2,R1        ; add LSB to 1-counter
        SUBS  R3,R3,#1        ; decrement loop counter
        BNE   Loop            ; loop back until R3=0
```

loop back 31 times

Z = 0

Z = 1

Count the number of binary 1s in R0

47

CZ1106 CE1106

Program Example (Alternative Count Loop)

Count Number of 1's

- Count the number of binary 1s in register **R0**.
- Loop counter `R3` is initialized to **31** at the start.
- `SUBS` decrements `R3` to negative and sets the `N` flag when that happens.
- `BPL` is used to test for `N`=0, until `N`=1 (i.e. `R3`=-1), it will keep looping back.

```
        MOV   R2,#0           ; clear 1-counter for binary 1s
        MOV   R3,#31          ; set loop counter to 31
Loop    MOV   R0,R0,ROR #1    ; rotate right 1 bit
        AND   R1,R0,#1        ; clear all bits except LSB
        ADD   R2,R2,R1        ; add LSB to 1-counter
        SUBS  R3,R3,#1        ; decrement loop counter
        BPL   Loop            ; loop back until R3=-1
```

loop back 31 times

N = 0

N = 1

Count the number of binary 1s in R0

48

CZ1106 CE1106

Comparing Signed & Unsigned Values

- Appropriate conditional test must be selected based on the number representation used.
- For testing **signed** values, use **GT**, **LT**, **GE**, **LE**.
- e.g.

```
        SUBS  R1,R1,R2 ;R1 = R1 - R2
        BGE   R1≥R2    ;jump to R1≥R2 if result is positive
          :
R1≥R2   :
```

- For testing **unsigned** values, use **HI**, **LO**, **HS**, **LS**.
- e.g.

```
        SUBS  R1,R1,R2 ;R1 = R1 - R2
        BHS   R1≥R2    ;jump to R1≥R2 if R1 higher or equal to R2
          :
R1≥R2   :
```

Destination register gets modified when we try to find out if R1 ≥ R2

Note: `R1≥R2` label is only for illustration. The "≥" is not a valid label symbol in the VisUAL ARM simulator

49

CZ1106 CE1106

Conditional Test using CMP

- Use (`CMP`) instead of (`SUBS`) to compare values of two operands without affecting the operands.
- Comparing a register value (signed) to an immediate value.

```
      CMP   R1,#4        ; test (R1 – 4), where R1 is a signed no.
      BGE   R1≥4         ; branch to R1≥4 if result is positive (i.e. R1 ≥ 4)
        :
R1≥4    :
```

- Finding C string terminator (0) in memory pointed to by `R0`.

```
Loop  LDRB  R1,[R0],#1   ; read mem byte using post-index autoindex
      CMP   R1,#0        ; test ( R1 – 0 )
      BEQ   Found        ; branch to Found if value is 0
      B     Loop         ; keep branching back to start of Loop
Found   :
```


50

CZ1106 CE1106

CMP

- **CMP subtracts** the source operand from the destination register and sets the **CC** flags according to the results.
  - Destination register remain **unmodified** after CMP.
  - CC flags affected in the same manner as the **subtract** instruction (SUBS).

```
SUBS   R1,R1,R2

       BGE    R1≥R2
         :
R1≥R2    :
```

R1 modified to achieve desired flow control

```
CMP    R1,R2

       BGE    R1≥R2
         :
R1≥R2    :
```

Same flow control but R1 unchanged

51

CZ1106 CE1106

Other Conditional Test Instructions

- ARM provides several other operators that can be used to influence the conditional test flags.
  - These conditional test instructions do not modify the destination operand.
  - They do not need the "S" suffix to influence the condition code flags (N,Z,V,C).

```
CMN R0,R1  ; set (N,Z,C,V) based on R0 + R1
```
Compare Negative

```
TST R0,R1  ; set (N,Z,C) based on R0 AND R1
```
Test Bits

```
TEQ R0,R1  ; set (N,Z,C) based on R0 EOR R1
```
Test Equivalence

  - The C flag for TST and TEQ can be influence by applying the shift and rotate operations on the source operand (rightmost).

52

CZ1106
CE1106

Summary

- Conditional branch (**Bcc**) allows us to implement conditional and loop constructs.
- Appropriate (**Bcc**) conditions must be selected for the conditional test used.
  - The (**cc**) choice needs to take into account of data type being used (i.e. signed or unsigned numbers).
- Appropriate operations (e.g. **CMP** or **SUBS**) are used to set the **N**, **Z**, **V**, **C** flags before conditional test can be done.

53

CZ1106
CE1106

54

54

CZ1106
CE1106

Chapter 5

Program Example

Finding the Largest Number

Learning Objectives (5.5)

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

55

CZ1106
CE1106

Program Example

Find Largest Number (FindMax)

- Write an assembly language program to :
  - Find the **largest value** in an integer array and store the result in register **R3**.
  - The array consists of **10 unsigned** numbers stored starting at address `0x100`.
  - **Things to note:**
  - Use correct conditional test for comparing unsigned number.
  - Use appropriate register indirect to access each array element efficiently.
  - Set up appropriate count loop to access all 10 numbers

Largest Value

R3 `0x00000007`

Number Array

	Address	Memory
[R0] →	0x100	0x00000003
	0x104	0x00000007
	0x108	0x00000004
	0x10C	0x00000002
	:	:
	:	:
	0x120	0x00000005
	0x124	0x00000001

56

CZ1106 CE1106

Possible Solution

Find Largest Number (FindMax)

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

Initialisation of registers

Loop Count	R1	0x0000009
Temp Reg	R2	
Current Max	R3	0x00000003

Number Array

Address	Memory
[R0] → 0x100	0x00000003
0x104	0x00000007
0x108	0x00000004
0x10C	0x00000002
:	:

R0 = Address pointer for current array element.
R1 = Loop counter register
R2 = Temporary register holding current no.
R3 = Current maximum value (i.e. the result).

57

CZ1106 CE1106

Possible Solution

Find Largest Number (FindMax)

```
        MOV   R0,#0x100     ;setup pointer to first array element
        MOV   R1,#9         ;load 9 into counter register
        LDR   R3,[R0]       ;assume 1st no. in array is current max
Loop    ADD   R0,R0,#4      ;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          ;branch if R2 ≤ current max (i.e. R3)
        MOV   R3,R2         ;update current max. in R3 with R2
Skip    SUBS  R1,R1,#1      ;decrement 1 from counter register
        BNE   Loop          ;jump back to Loop if not zero
```

R0 = Address pointer for current array element.
R1 = Loop counter register
R2 = Temporary register holding current no.
R3 = Current maximum value (i.e. the result).

58

CZ1106 CE1106

Conditional Execution

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

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

ARM code example

```
      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)
Executes if Z = 1  MOVEQ  R1,#3       ; then { r1 := 3}
Executes if Z = 0  MOVNE  R1,#5       ; else  { r1 := 5}
SKIP      .....              ;
```


59

CZ1106 CE1106

Summary

- **Register indirect** addressing modes (with and without autoindexing) can be used to access array elements in memory.
- Conditional branch (**Bcc**) allows us to implement conditional and loop constructs.
- Appropriate conditions (e.g. **LS** and **NE**) must be selected to implement the required test.
- Appropriate operations (e.g. **CMP** or **SUBS**) are used to set the (**N**, **Z**, **V**, **C**) flags before conditional test can be done.
- Conditional execution (e.g. **MOVHI** or **ADDEQ**) can be used to avoid doing conditional branching.

60

Chapter 6: Modular Programming

Mohamed M. Sabry Aly

N4-02c-92

Learning Objectives

- Describe ARM instructions in implement subroutines

- Describe passing parameters to subroutines using:
  - Registers
  - Memory

Modular Program Design

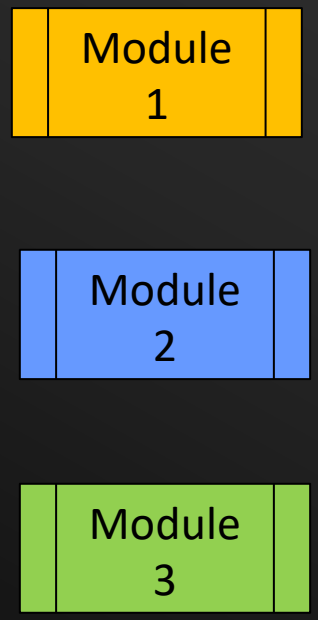
- Real-world applications are very large and complex
  - Google Chrome: ~6.7 Millions line of code
  - Android OS: 12-15 Millions line of code
  - Boeing 787: ~6.5 Millions line of code

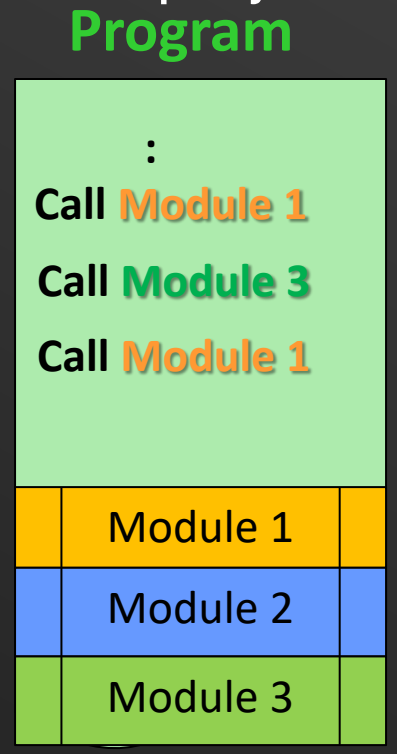
- Large software cannot be a single function

- It is decomposed into several less complex **modules**

Modular Program Design

- Software decomposed to several less complex **modules**
  - Modules can be designed and tested **independently**
  - Modules can reduce overall **program size**
    - Same module may be required in several places
  - Modules that are general can be **re-used** in other projects





Characteristics of a Good SW Module

- Loose coupling—**data** within module is entirely **independent** of other modules (local variables)

- Strong modularity—should perform a **single** logically coherent **task**

Example: Standard Deviation

Case 1: all in one c main()

```
int main() {
    .
    float avg=0.0; //initialization
    for(int i=0;i<N;i++){
       avg +=x[i];
    }
    avg/=N;
    float sigma = 0.0;
    for(int i=0;i<N;i++){
       sigma +=pow(x[i]-avg),2);
    }
    sigma/=N;
    sigma=sqrt(sigma);
    return 0;
}
```

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

Example: Standard Deviation

Case 2: Modular

```
int main() {
    …
    float avg = sum(x,N)/N;
    float sigma = Sigma_f(x,avg,N);
    return 0;
}
```

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

```
float sum( int* x, int N){
    float tot=0.0; //initialization
    for(int i=0;i<N;i++){
       tot +=x[i];
    }
    return tot;
}
```

```
float Sigma_f( int* x, float avg, int N){
    float sigma = 0.0;
    for(int i=0;i<N;i++){
       sigma +=pow(x[i]-avg),2);
    }
    sigma/=N;
    sigma=sqrt(sigma);
    return sigma;
}
```

Example: Standard Deviation

Case 2: Modular

```
int main() {
    …
    float avg = sum(x,N)/N;
    float sigma = Sigma_f(x,avg,N);
    return 0;
}
```

How will this be translated to assembly instructions?

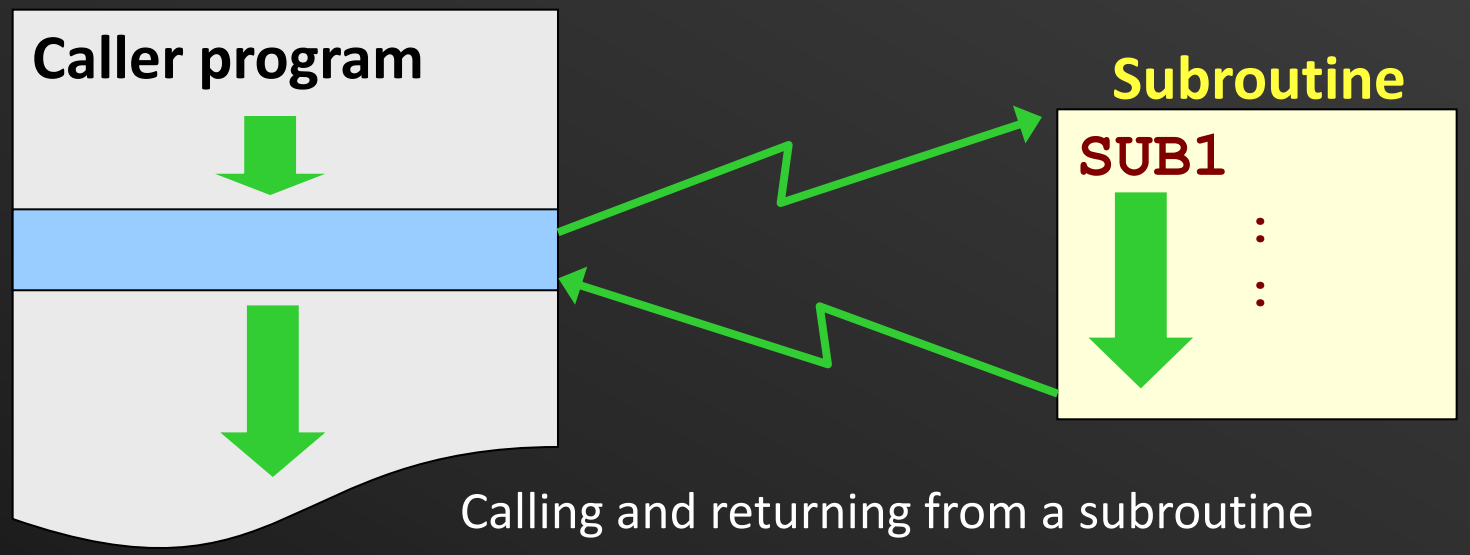
How will ARM go to these two functions and return?

```
float sum( int* x, int N){
    float tot=0.0; //initialization
    for(int i=0;i<N;i++){
       tot +=x[i];
    }
    return tot;
}
```

```
float Sigma_f( int* x, float avg, int N){
    float sigma = 0.0;
    for(int i=0;i<N;i++){
       sigma +=pow(x[i]-avg),2);
    }
    sigma/=N;
    sigma=sqrt(sigma);
    return sigma;
}
```

Subroutines

- Modules (e.g., functions in C) are implemented as **subroutines**
- Subroutine can be called from various parts of the program
- Caller and callee
  - Caller: the program that calls subroutine (SUB1)
  - Callee: subroutine (SUB1)



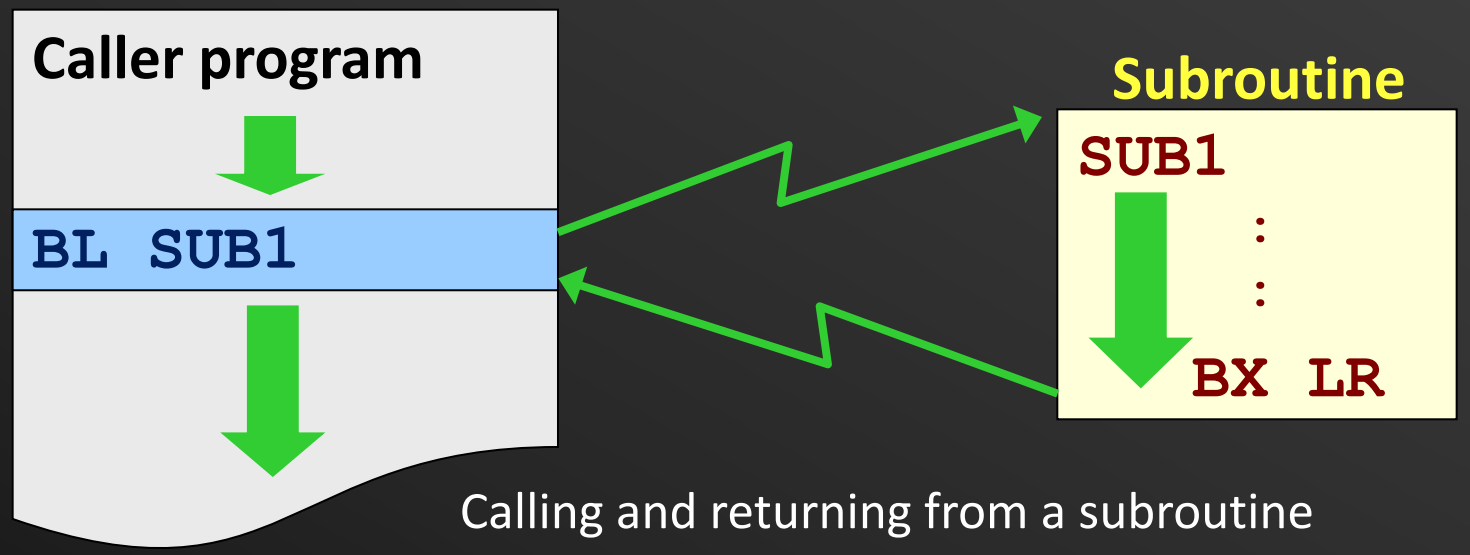
Calling and returning from a subroutine

Subroutines

- On completion, subroutine returns control to the caller
  - Exactly after the subroutine was called
- Calling and returning from a subroutine
  - To go to subroutine (SUB1): **`BL SUB1`**
  - To return to caller program: **`BX LR`**

`BL: branch with link`

`BX: branch and exchange`



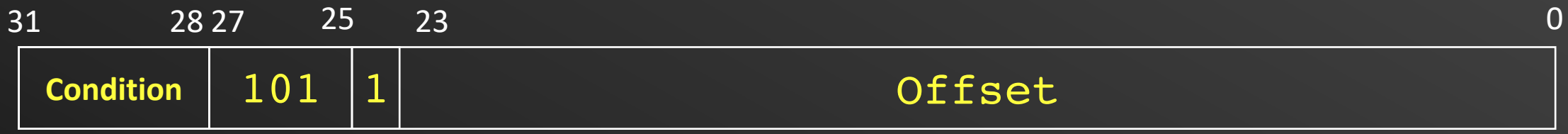
Calling and returning from a subroutine

Why `BL` and `BX` and not `B`?

- Main program branches to subroutine:
  - Can be done with B ➔ what is the draw back?
  - B overwrites value in PC ➔ oldPC value in main program is LOST
  - Maybe add another branch at the end of subroutine ➔ need to know the exact mem location during compilation, not an effective approach

Branch with Link (`BL`)

- **`BL`** used to make subroutine call
- Return address (PC contents + 4) is stored in the link register (R14)



- We can also conditionally make a functional call (more of this later)

Branch with Link (`BL`)

- **`BL`** used to make subroutine call

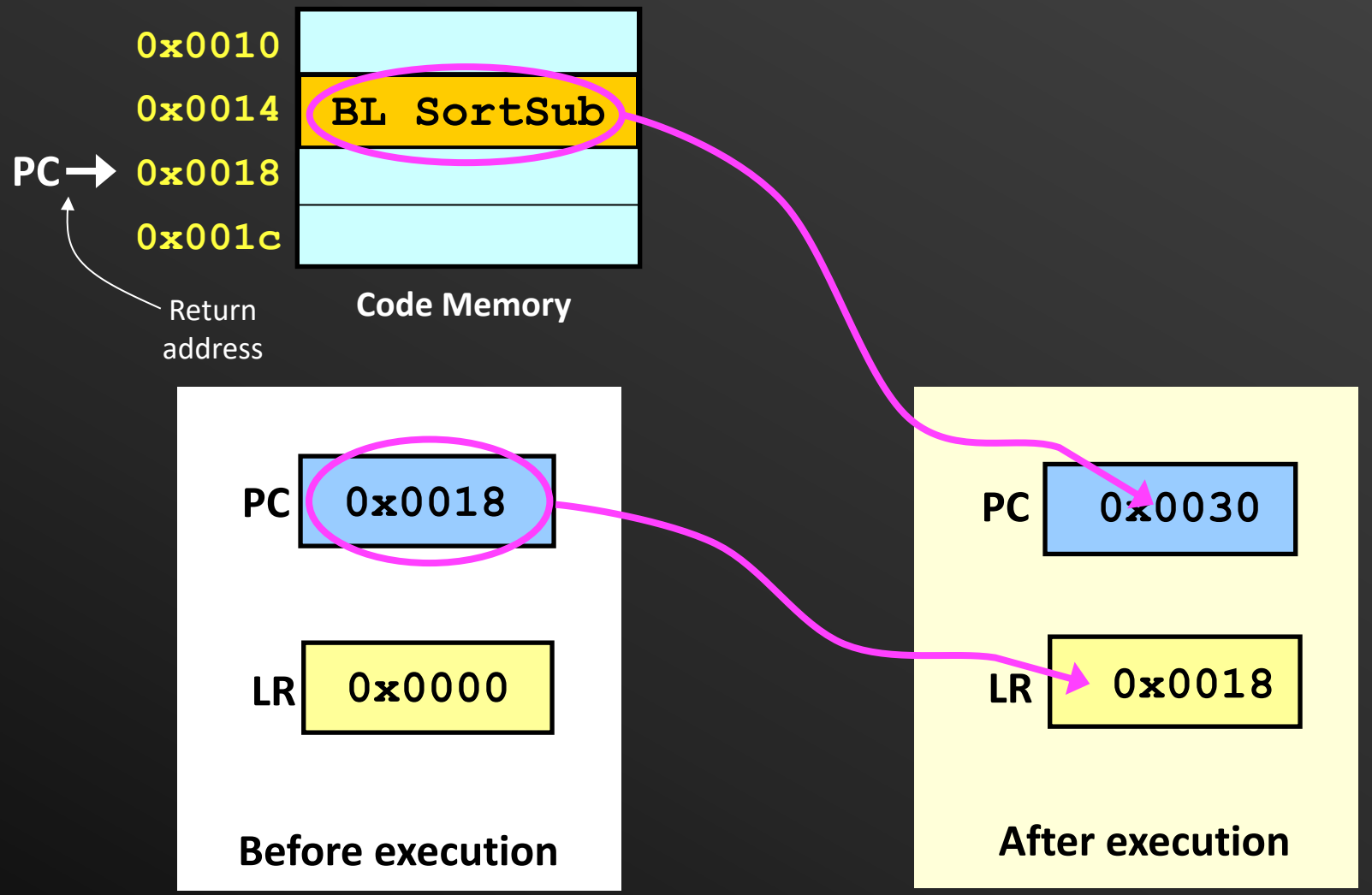
Subroutine Call

```
BL SortSub
```

- Execution sequence:
- Return address (PC contents + 4) is stored in the link register (R14)
- The subroutine address is stored to the PC

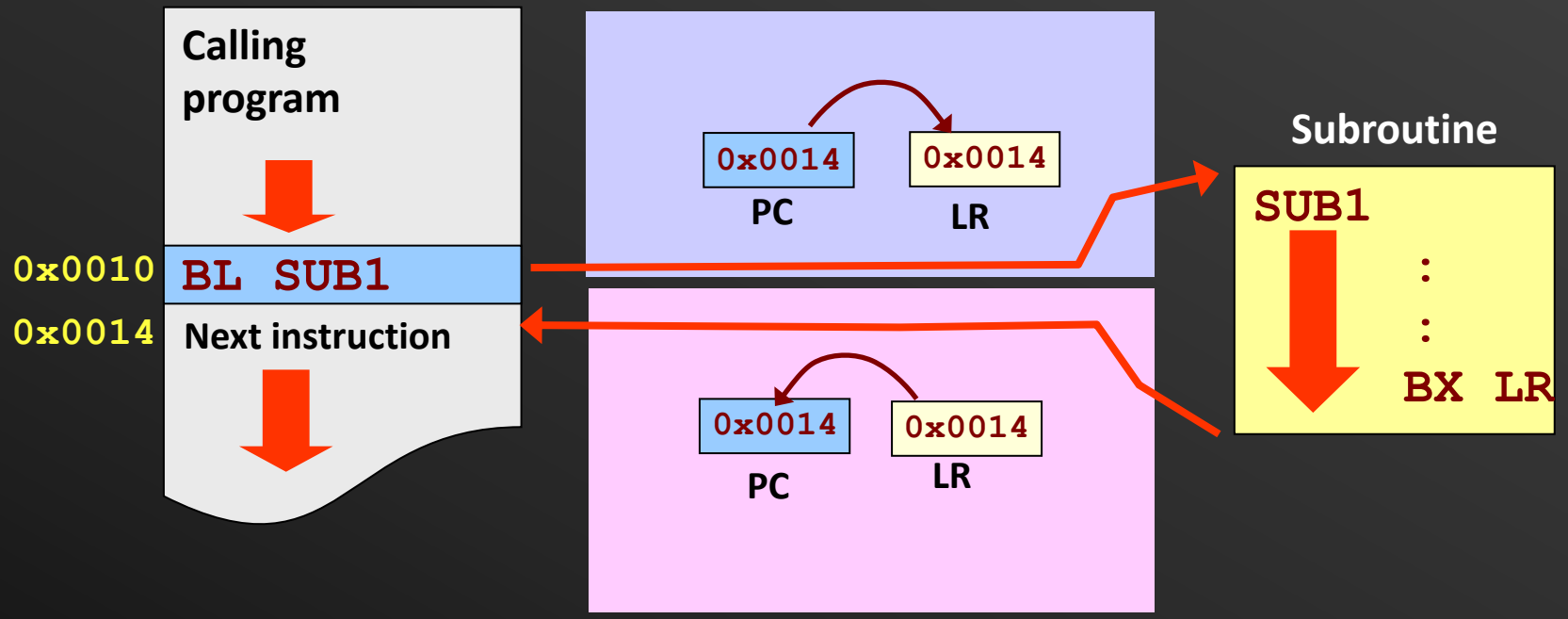
BL (Execution example)

- BL SortSub (assume SortSub is in 0x0030)



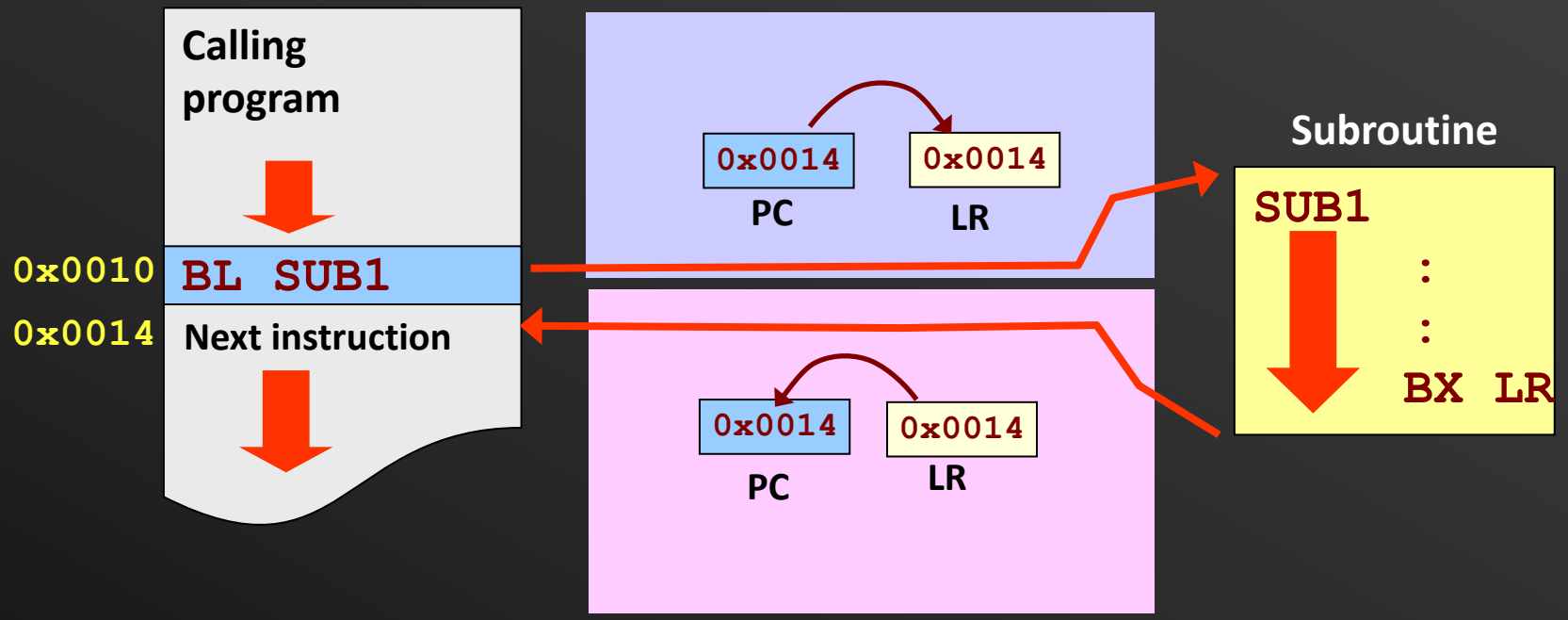
BX instruction

- `BX lr` returns from subroutine
- `lr` contains the return address, the instruction copies the value over to `PC`



BX instruction

- `BX lr` returns from subroutine
- `lr` contains the return address, the instruction copies the value over to `PC`



- Note: VisUAL does not support `bx lr`
  - Use `MOV PC, LR`

Reset to continue editing code

```
1  Num1    DCD    0x20
2  Num2    DCD    0x14
3  Result  DCD    0
4          ADR    SP, 0xFFFFFFFC
5          ADR    r0, Num1
6          ADR    r1, Num2
7          ADR    r2, Result
8          bl     Mean
9          END
10
11
12 Mean    LDR    r4,[r0]
13         LDR    r5,[r1]
14         add    r4,r4,r5
15         lsr    r4,r4,#1
16         STR    r4,[r2]
17         bx     lr
18
```

Syntax Error

Unsupported instruction.

Press F1 for a list of supported instructions.

New | Open | Save | Settings | Tools

Emulation Complete — Line 8 Issues 0

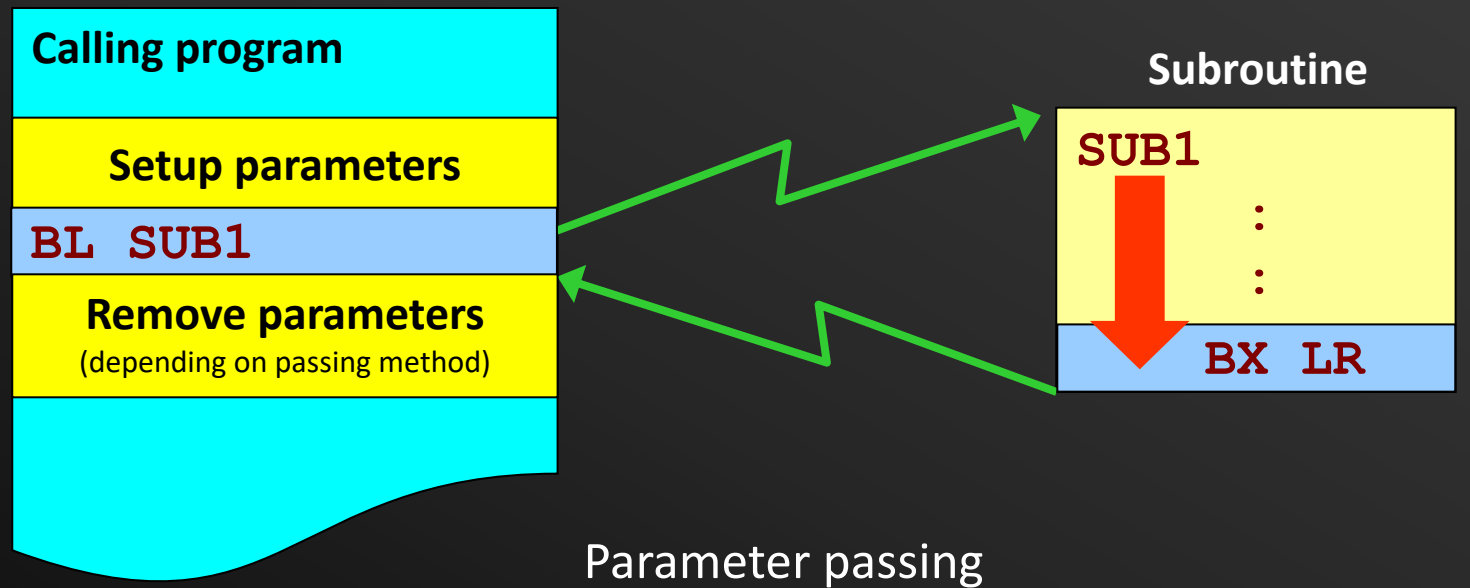
Reset to continue editing code

```
1  Num1    DCD    0x20
2  Num2    DCD    0x14
3  Result  DCD    0
4          ADR    SP, 0xFFFFFFFC
5          ADR    r0, Num1
6          ADR    r1, Num2
7          ADR    r2, Result
8          bl     Mean
9          END
10
11
12 Mean    LDR    r4,[r0]
13         LDR    r5,[r1]
14         add    r4,r4,r5
15         lsr    r4,r4,#1
16         STR    r4,[r2]
17         mov    pc,lr
18
```

R0 R1 R2 R3 R4 R5 R6 R7 R8 R9

Parameter Passing

- Calling programs need to pass parameters to influence a subroutine's execution.
- Parameters must be setup properly before the subroutine is called and appropriately removed after returning.
- There are three basic methods to pass parameters, via **registers**, **memory block** or the **system stack**.



Parameter passing

Parameter Passing using Registers

- Parameters are placed into the register before calling the subroutine
- **Number** of parameters passed are **limited** to the **available registers**
  - Useful when number of parameters are **small**
  - Not all **R0-R12** are preferred to pass parameters

Parameter Passing using Registers

- Parameters are placed into the register before calling the subroutine
- **Number** of parameters passed are **limited** to the **available registers**
  - Useful when number of parameters are **small**
  - Not all **R0-R12** are preferred to pass parameters

Calling convention

R0-R3	R4-R11
Can be used to pass argument values Also used to return values from subroutine Subroutine can modify values	Used to hold local variables Not for passing arguments Must be preserved in the subroutine

Parameter Passing using Registers

- Parameters are placed into the register before calling the subroutine
- **Number** of parameters passed are **limited** to the **available registers**
  - Useful when number of parameters are **small**
  - Not all **R0-R12** are preferred to pass parameters

Calling convention

R0-R3
Can be used to pass argument values
to return values from subroutine
Subroutine can modify values

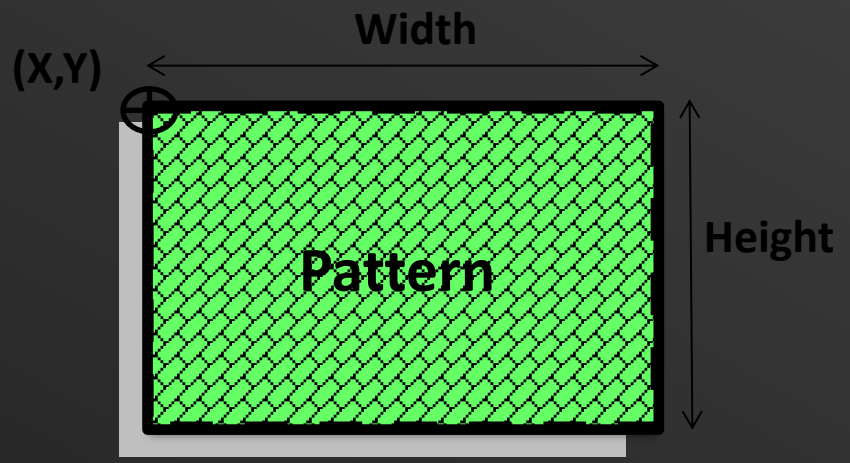
R4-R11
Used to hold local variables
Not for passing arguments
Must be preserved in the subroutine

R12: Scratchpad register, does not need to be preserved
Can be used sometimes as return register

Parameter Passing using Registers

- Parameters are placed into the register before calling the subroutine
- **Number** of parameters passed are **limited** to the **available registers**
  - Useful when number of parameters are **small**
  - Not all **R0-R12** are preferred to pass parameters

```
Draw_rectangle(
X, Y,
Height, Width,
Border, Line_style
Fill, Color, Pattern,
Shadow);
```



Parameter Passing using Registers

- Parameters are placed into the register before calling the subroutine
- **Number** of parameters passed are **limited** to the **available registers**
  - Useful when number of parameters are **small**
  - Not all **R0-R12** are preferred to pass parameters

- **Pro – efficient** as parameters are already in register within the subroutine and can be used immediately.

- **Con – lacks generality** due to the limited number of registers.

Example: Bit Counting Subroutine

- Write a subroutine to:
  - Count the number of “1” bits in a word.
  - Return result in register **R0**.

- **Design considerations:**
- How do we transfer the word into the subroutine?
  - Put the word into a **register**, which can then be accessed within the subroutine (e.g. register **R1**).
- How do we check if each individual bit is a “1” or a “0”?
  - Rotate **R1** right 32 times with the carry bit. After each rotate, test carry bit to check if (**C**=1). If yes, increment bit counter register **R0**.

Solution #1


```
Count1s
```

Start by labeling the subroutine
and placing the return instruction

```
        MOV     PC, LR          ; same as bx lr
```

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

Solution #1


```
Count1s  MOV       R0, #0    ;Clear R0
         MOV       R2, #32   ; Set counter R2 with 32
```

Initialize R0 with 0 and
the counter register R2 with 32

```
         MOV       PC, LR          ; same as bx lr
```

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

Solution #1


```
Count1s  MOV     R0, #0  ;Clear R0
         MOV     R2, #32 ; Set counter R2 with 32
Loop     RRXS    R1,R1     ; Rotate right and extend R1
```

Rotate to the right (arithmetic), and use the carry

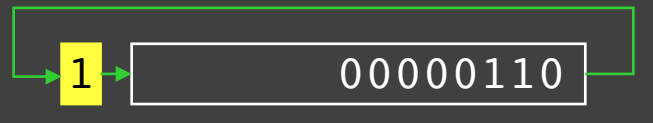
S: set the status registers after rotation

```
         MOV     PC, LR          ; same as bx lr
```

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

Solution #1




```
Count1s  MOV     R0, #0          ;Clear R0
         MOV     R2, #32         ; Set counter R2 with 32
Loop     RRXS    R1,R1           ; Rotate right and extend R1
         ADC     R0,R0,#0        ; Add the carry to R0
```

Add the carry value to R0

```
         MOV     PC, LR          ; same as bx lr
```

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

Solution #1


```
Count1s  MOV     R0, #0          ; Clear R0
         MOV     R2, #32         ; Set counter R2 with 32
Loop     RRXS    R1,R1           ; Rotate right and extend R1
         ADC     R0,R0,#0        ; Add the carry to R0
         SUBS    R2,R2,#1        ; decrement counter by 1
         BNE     Loop            ; loop if not zero
         MOV     PC, LR          ; same as bx lr
```

Decrement and set Status registers

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

Solution #1


```
Count1s  MOV     R0, #0          ;Clear R0
         MOV     R2, #32  ; Set counterR2 with 32
Loop     RRXS    R1,R1           ; Rotate right and extend R1
         ADC     R0,R0,#0        ; Add the carry to R0
         SUBS    R2,R2,#1        ; decrement counter by 1
         BNE     Loop            ; loop if not zero
         MOV     PC, LR          ; same as bx lr
```

Issue, the carry of SUBS
Will be used in RRXS ➔ R1 value is destroyed

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

Solution #2

```
Count1s  EOR     R0,R0,R0      ;Clear R0
         ADD     R2, R0, #32   ; Set counterR2 with 32
         ADD     R3, R0, #1    ; Set R3 with 1
Loop     AND     R4, R3, R1, ROR R2          ; ?
         ADD     R0,R0,R4      ; Add the lsb of R0
         SUBS    R2,R2,#1      ; decrement counter by 1
         BNE     Loop          ; loop if not zero
         MOV     PC, LR        ; same as bx lr
```

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

Solution #2

```
Count1s  EOR     R0,R0,R0      ;Clear R0
         ADD     R2, R0, #32   ; Set counterR2 with 32
         ADD     R3, R0, #1    ; Set R3 with 1
Loop     AND     R4, R3, R1, ROR R2          ; ?
         ADD     R0,R0,R4      ; Add the lsb of R0
         SUBS    R2,R2,#1      ; decrement counter by 1
         BNE     Loop          ; loop if not zero
         MOV     PC, LR        ; same as bx lr
```

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

Solution #2

```
Count1s  EOR      R0,R0,R0      ;Clear R0
         ADD      R2, R0, #32   ; Set counterR2 with 32
         ADD      R3, R0, #1    ; Set R3 with 1
Loop     AND      R4, R3, R1, ROR R2          ; ?
```

Set R3 as a mask with a value of 1
Then apply this mask with a rotated value of R1

```
         MOV      PC, LR        ; same as bx lr
```

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

Solution #2

```
Count1s  EOR     R0,R0,R0       ;Clear R0
         ADD     R2, R0, #32    ; Set counterR2 with 32
         ADD     R3, R0, #1     ; Set R3 with 1
Loop     AND     R4, R3, R1, ROR R2          ; ?
```

Set R3 as a mask with a value of 1
Then apply this mask with a rotated value of R1
The **rotated R1 is not stored anywhere**

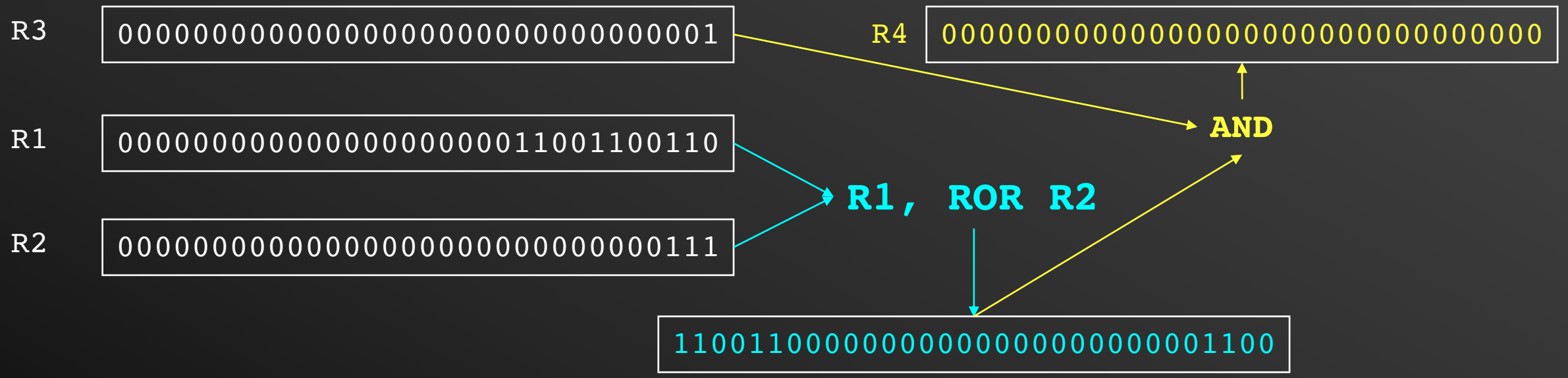
```
         MOV     PC, LR         ; same as bx lr
```

Value passed in R1, return value in R0

VisUAL does not support `bx lr`

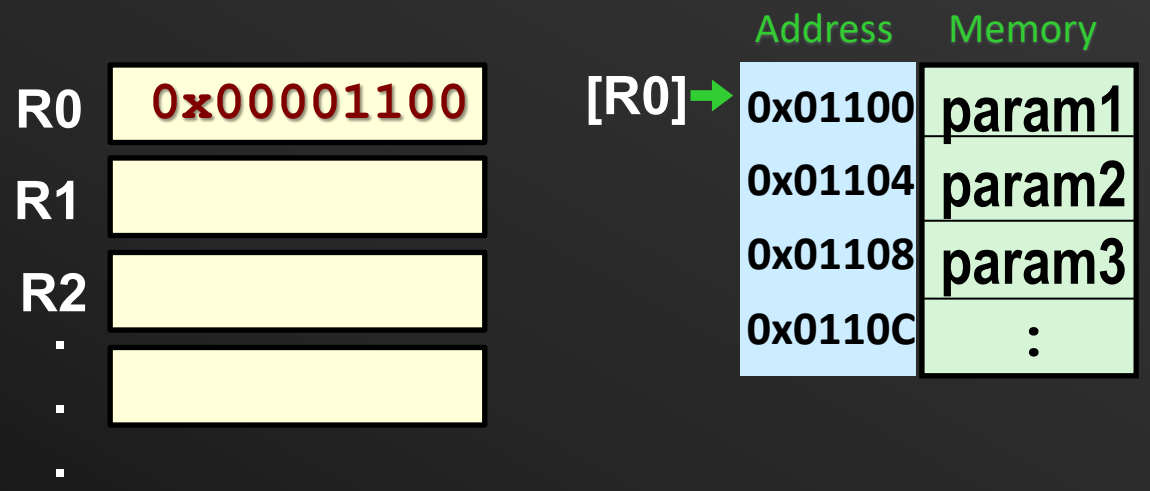
A Deep Dive

```
AND     R4, R3, R1, ROR R2
```



Parameter Passing using Memory

- A region in memory is treated like a mailbox and is used by both the calling program and subroutine.
  - Parameters to be passed are gathered into a **block** at a predefined memory location
  - The start address of the memory block is passed to the subroutine via an **address register**.
  - Useful for passing **large number of parameters**.



Example: Lower to Upper Case Subroutine

- Write a subroutine to:
  - To convert an ASCII string from lower to upper case.
  - The string is terminated by a NULL character (**0x00000000**).
  - The start address of the string is passed via **R0**.
  - A segment of the calling program shows how the parameter is setup and the subroutine called:

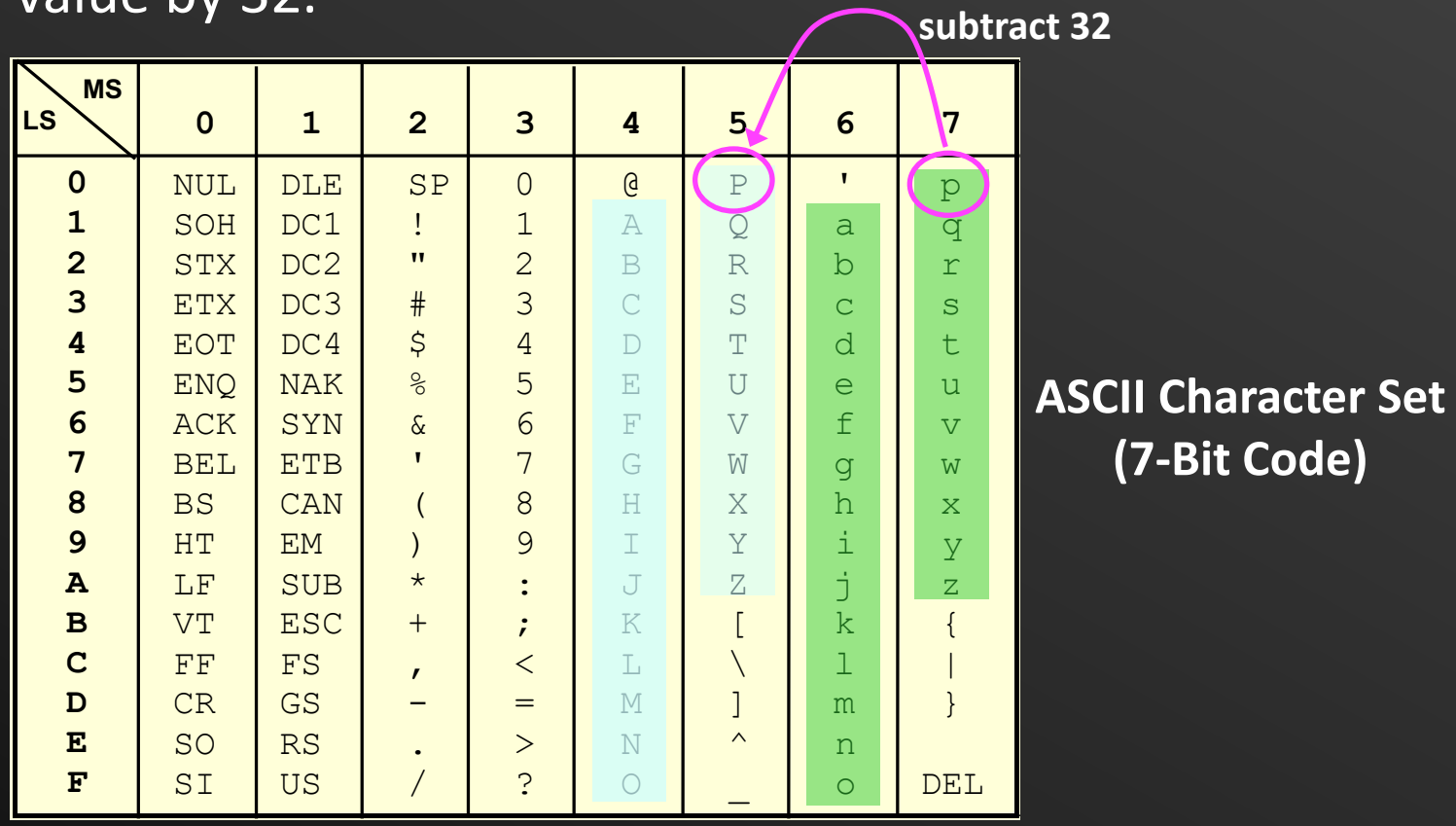
```
;Calling program
    :
MOV    R0,#0x100   ;move start addr. of string to R1
BL     Lo2Up       ;branch to Lo2Up subroutine
    :
```

Address	Memory
0x100	"a"
0x104	"p"
0x108	"p"
0x10C	"l"
0x110	"e"
0x114	0x000
0x118	:

Algorithm Design

- How to convert an ASCII character from lower to upper case?

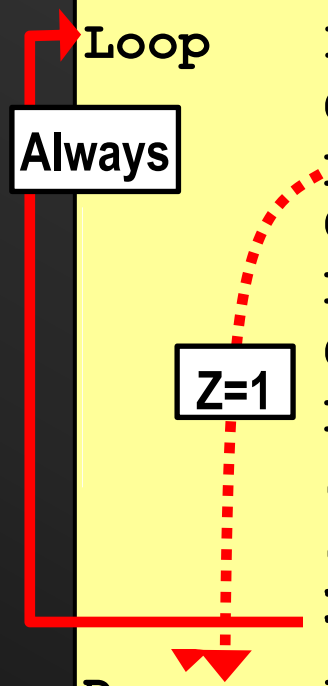
  - Check that the character's value is between 'a' and 'z'.
  - If so, subtract its value by 32.



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

ASCII Character Set (7-Bit Code)

Possible Solution




```
Lo2Up   STMFD   SP!,{r0,r1}   ;save registers used within subroutine
Loop    LDR     R1,[R0],#4    ;get current char from string in memory
        CMP     R1,#0         ;Compare with NULL
        BEQ     Done          ;if NULL char, branch to Done
        CMP     R1,#0x061     ;compare with lower limit "a"
        BLT     Loop          ;if smaller than "a", do not convert
        CMP     R1,#0x07A     ;compare with upper limit "z"
        BGT     Loop          ;if greater than "z" do not convert
        SUB     R1,R1,#32     ;convert to upper case by subtracting 32
        STR     R1,[R0,#-4]   ;write modified char back to string in memory
        B       Loop          ;branch back to Loop
Done    LDMFD   SP!,{r0,r1}   ;restored saved registers before returning
        MOV     PC,LR         ;return from subroutine
```

Note: Subroutine modifies **R0** & **R1** but restores their original contents before returning. This produces a **transparent subroutine** that does not effect the proper operation of calling program

Possible Solution

```
Lo2Up   STMFD   SP!,{r0,r1}    ;save registers used within subroutine
Loop    LDR     R1,[R0],#4     ;get current char from string in memory
```

Increment R0 by 4 and update R0
(to point to next memory location)

Refer to the location to R0-4 (the original R0)

```
        STR     R1,[R0,#-4]    ;write modified char back to string in memory
        B       Loop           ;branch back to Loop
Done    LDMFD   SP!,{r0,r1}    ;restored saved registers before returning
        MOV     PC,LR          ;return from subroutine
```

Note: Subroutine modifies **R0** & **R1** but restores their original contents before returning. This produces a **transparent subroutine** that does not effect the proper operation of calling program

Possible Solution

```
Lo2Up   STMFD   SP!,{r0,r1}   ;save registers used within subroutine
Loop    LDR     R1,[R0],#4    ;get current char from string in memory
        Increment R0 by 4 and update R0
        (to point to next memory location)
        Refer to the location to R0-4 (the original R0)
        STR     R1,[R0,#-4]   ;write modified char back to string in memory
        B       Loop          ;branch back to Loop
Done    LDMFD   SP!,{r0,r1}   ;restored saved registers before returning
        MOV     PC,LR         ;return from subroutine
```

R0=0x100

R0=0x104

R0-4=0x100

Note: Subroutine modifies **R0** & **R1** but restores their original contents before returning. This produces a **transparent subroutine** that does not effect the proper operation of calling program

Optimizing Memory Usage

- Example assumes each character in 32 bits
  - But each character requires 8 bits only

- Can we access each Byte separately?

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

Optimizing Memory Usage

- Example assumes each character in 32 bits
  - But each character requires 8 bits only

- Can we access each Byte separately?

Use the {B} option in Access

LDRB
STRB

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

Efficient Memory Solution

```
Lo2Up   STMFD   SP!,{r0,r1}     ;save registers used within subroutine
Loop    LDRB    R1,[R0],#1      ;get current char from string in memory
        CMP     R1,#0           ;Compare with NULL
        BEQ     Done            ;if NULL char, branch to Done
        CMP     R1,#0x061       ;compare with lower limit "a"
        BLT     Loop            ;if smaller than "a", do not convert
        CMP     R1,#0x07A       ;compare with upper limit "z"
        BGT     Loop            ;if greater than "z" do not convert
        SUB     R1,R1,#32       ;convert to upper case by subtracting 32
        STRB    R1,[R0,#-1]     ;write modified char back to string in memory
        B       Loop            ;branch back to Loop
Done    LDMFD   SP!,{r0,r1}     ;restored saved registers before returning
        MOV     PC,LR           ;return from subroutine
```

Note: Subroutine modifies **R0** & **R1** but restores their original contents before returning. This produces a **transparent subroutine** that does not effect the proper operation of calling program

Summary

- `BL` is required to call a subroutine, return address is in `LR` register
- A return from subroutine is done with `BX LR` or `MOV PC,LR`
- Passing parameters **using registers** is the simplest and fastest.
  - Number of parameters that can be passed is **limited** by the **available registers**.
- Passing parameters using a **memory block** can support a **large number** of parameters or data types like arrays.

Chapter 6: Modular Programming-Cont

Mohamed M. Sabry Aly

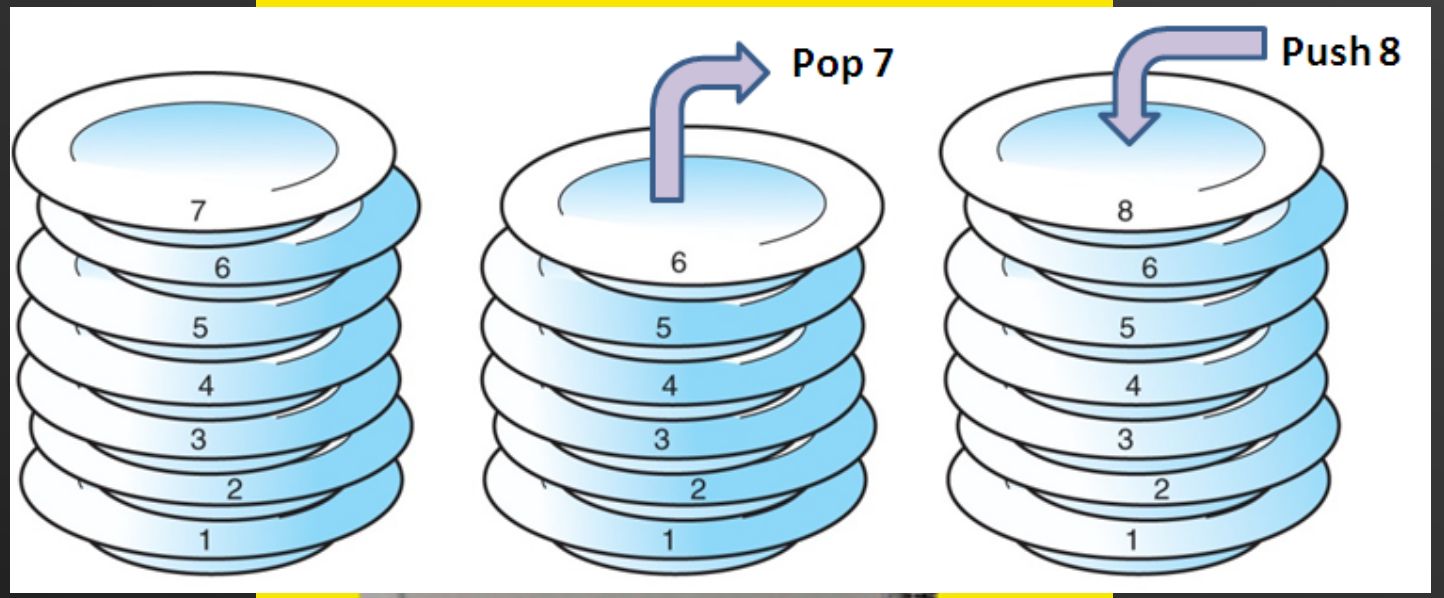
N4-02c-92

Learning Objectives

- List the stack manipulation operations & its implementation
- Describe passing parameters to subroutines using the stack
- Identify the difference between passing by value and by reference.
- Describe how a transparent subroutine can be implemented.

System Stack

- A stack is a first-in, last-out linear data structure that is maintained in the memory's data area.

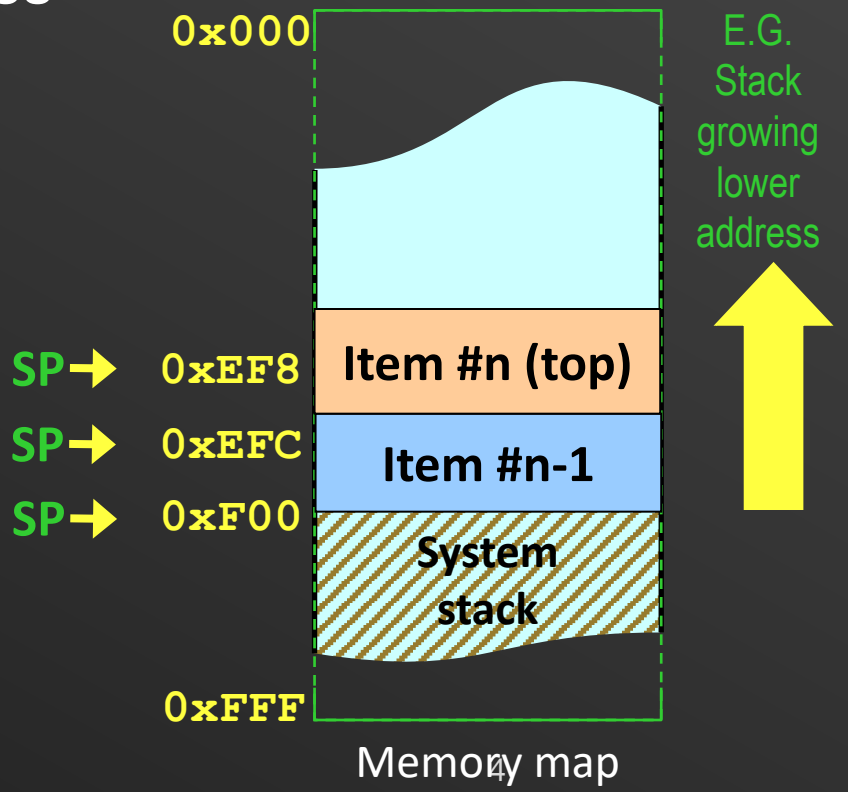


Stack of warm plates on dispenser

System Stack

- A stack is a first-in, last-out linear data structure that is maintained in the memory's data area.

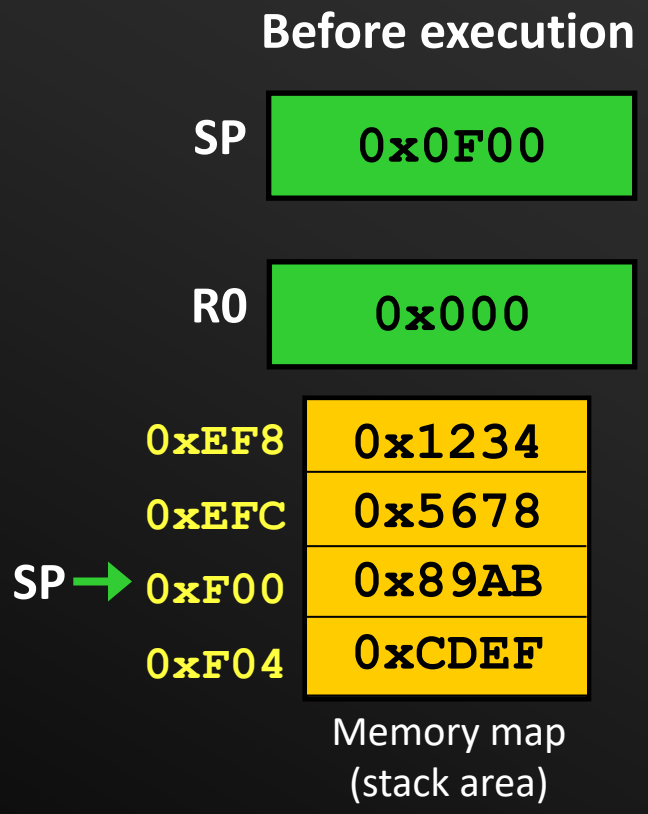
  - The system stack in the ARM processor is maintained by a dedicated stack pointer (**SP, R13**).
  - Stack can grow towards lower or higher memory address
    - Preferred direction is lower, start from max address ➔ Why?
  - The **SP** points to the top item on the system stack.
  - The 3 basic stack operations are **push**, **pop** and **access** items on the stack.

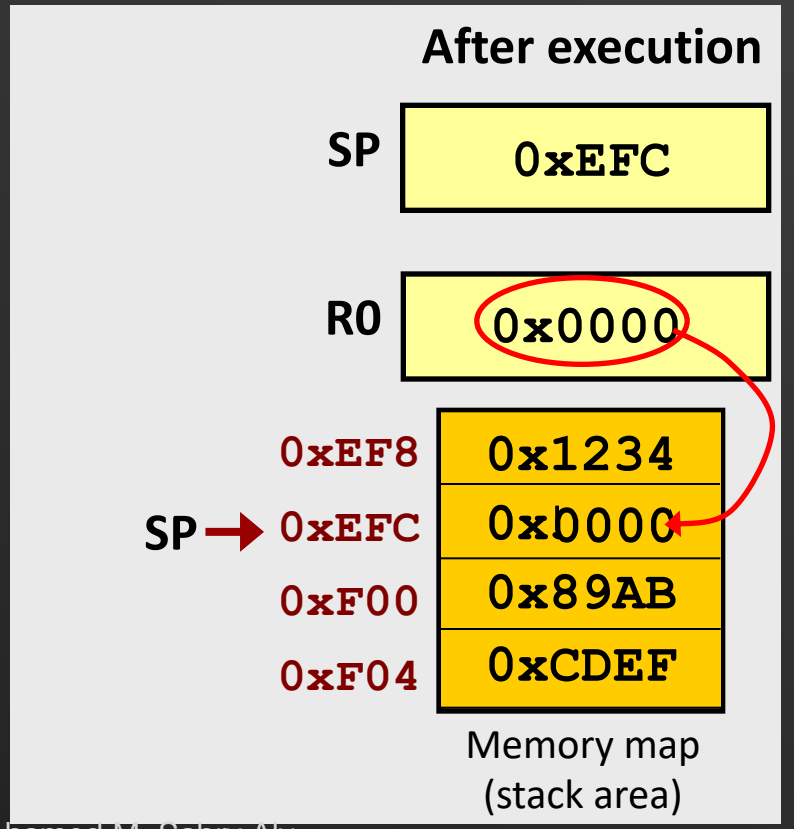


Ref: Null & Lobur (3rd Ed) section A.2.3 – Stacks

Push Data to the Stack

- Writing to stack can be done using **`STR`** instruction
  - Need to also increase the stack pointer before storing
- Syntax: `STR R0, [SP,#-4]!`





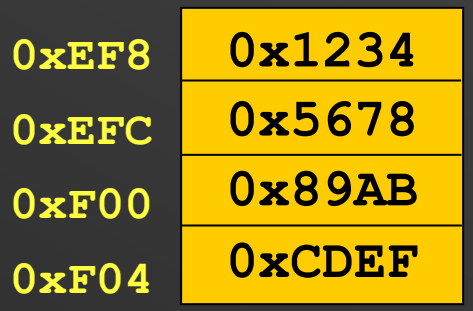
Pop Data off the Stack

- Popping from the stack can be done with **LDR** instruction
  - Need to update pointer AFTER read
- Syntax: `LDR R0, [SP],#4`

Removing from stack

- After popping, is the data **erased** from the stack?

NO



- Data still resides in memory and can be read

```
LDR R0, [SP, #-4]
```

SP is not changed

Pushing Registers to Stack

- E.g., want to push R0 and R1

Method 1:

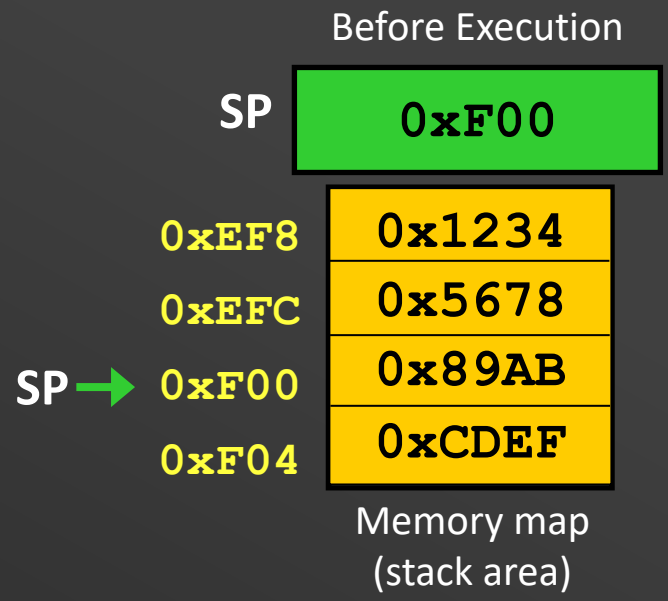
```
STR R0, [SP, #-4]!
STR R1, [SP, #-4]!
```

Pushing Registers to Stack

- E.g., want to push R0 and R1

Method 1:

```
STR  R0, [SP, #-4]!
STR  R1, [SP, #-4]!
```



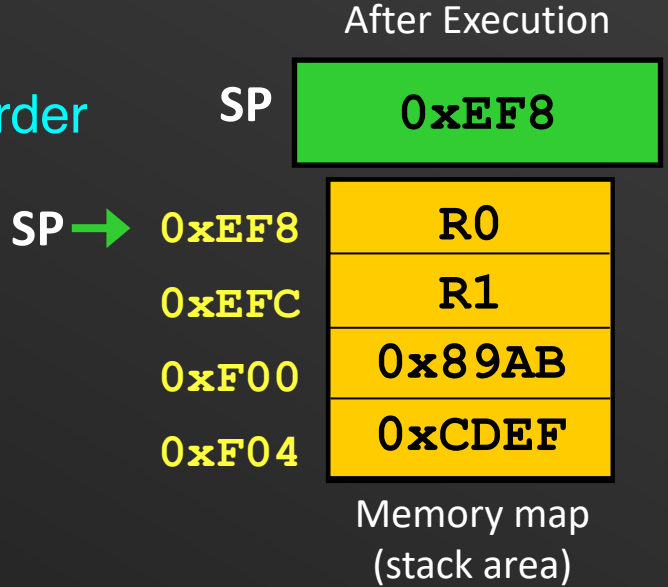
Method 2:

Use Stack pointer and write back updated value after operation

```
STMFD SP!,{R1,R0}
```

Write registers in descending order

Store multiple registers fully descending



Pushing and popping to stack

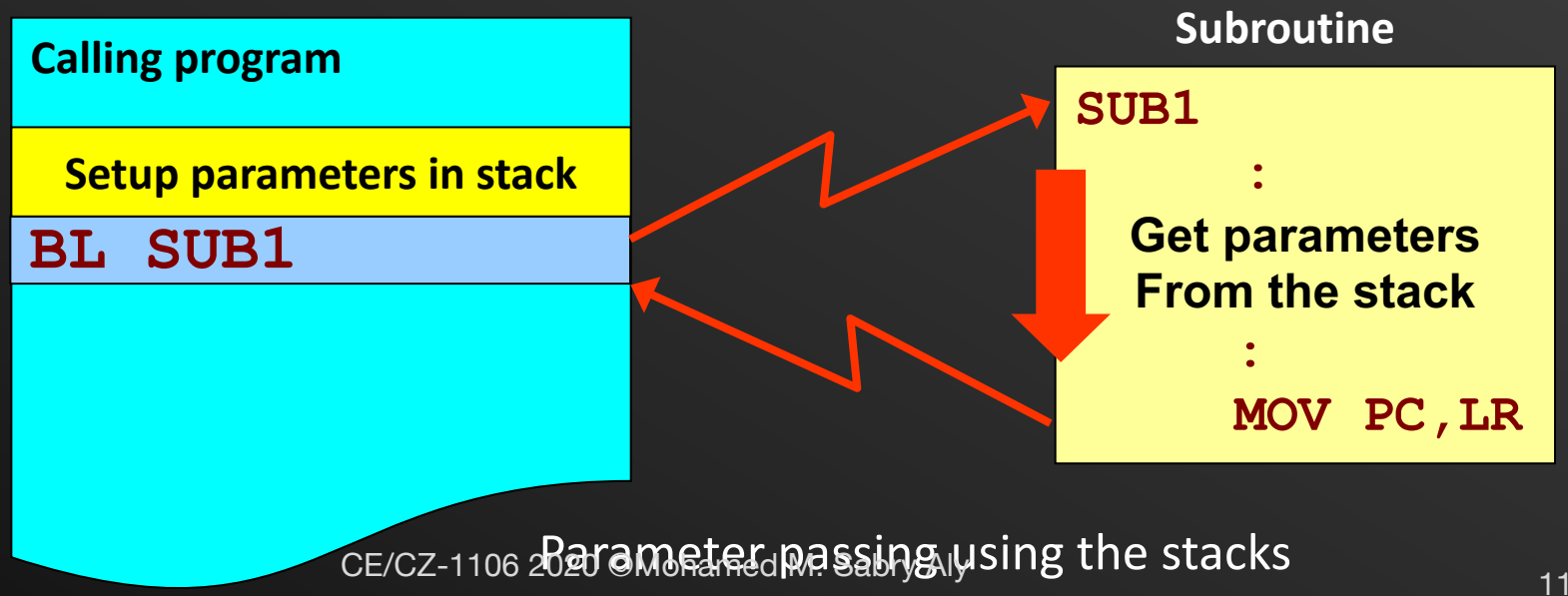
```
STMFD SP!, {list of registers}

LDMFD SP!, {list of registers}
```

```
STMFD SP!,{R1,R0}

LDMFD SP!,{R1,R0}
```

Parameter Passing using Stack

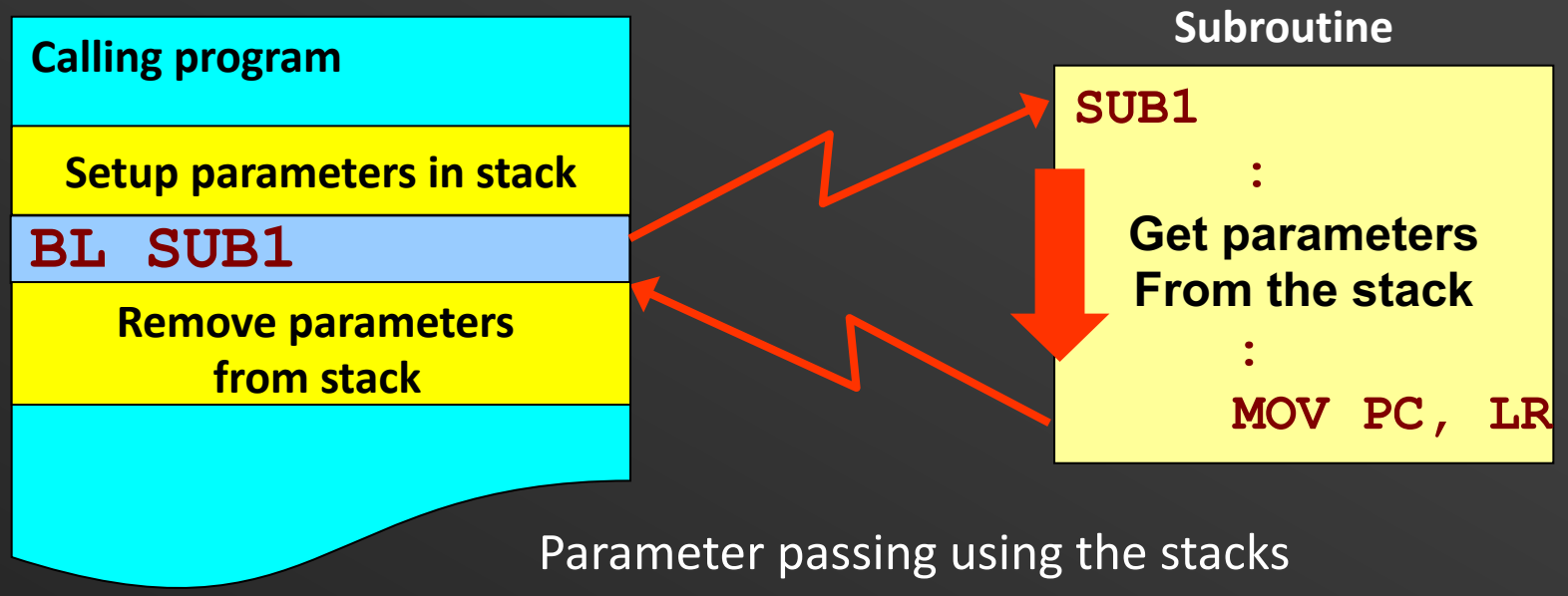
- Parameters are pushed onto the stack before calling the subroutine and retrieved from the stack within the subroutine.

- Most **general** method of parameter passing – no registers needed, supports recursive programming.
- **Large** numbers of parameters can be passed as long as stack does not overflow.



Parameter passing using the stacks

Parameter Passing using Stack



Parameter passing using the stacks

- Parameters pushed to the stack must be **removed** by the calling program immediately after returning from subroutine.

- If not, repeated pushing of parameters to the stack will lead to a stack overflow.

Sum from 1 to N

- Write a subroutine to:
  - Sum the positive numbers from 1 to **N**, where **N** is a value passed to the subroutine.
  - The computed sum should be directly updated to a memory variable **Answer**, whose address is **0x100**.
  - All parameters are to be passed via the **stack**.

Solution:

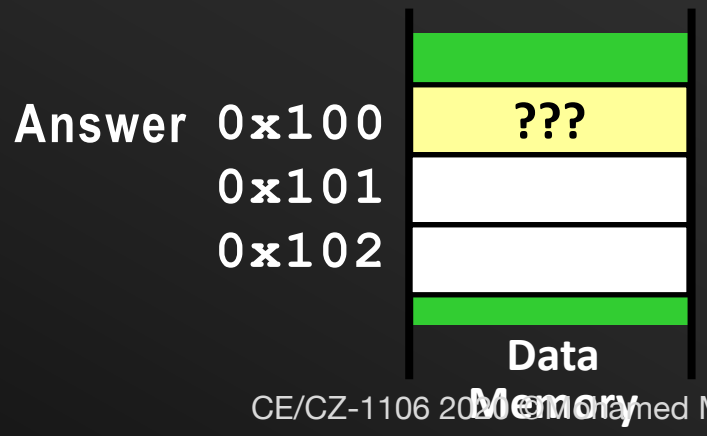
- Push two parameters on stack, the value of **N** and **address** of memory variable **Answer**.

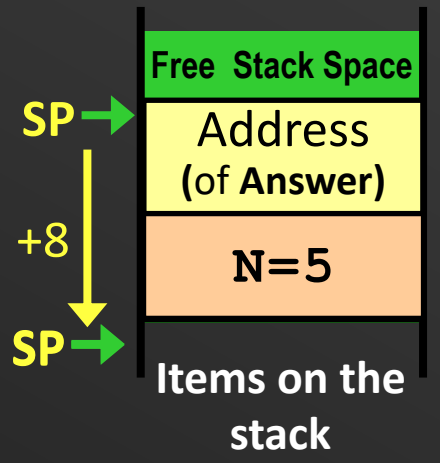
Calling the Subroutine

```
        :
Parameters setup
        MOV   R1, #5          ;Set R0 with #5
        MOV   R0, #0x100      ;Set R1 with the address
        STMFD SP!, {r1,r0}    ;push R0&R1 to stack
        BL    Sum1N           ;call subroutine Sum1N
        ADD   SP,SP,#8        ;add 8 to pop the two parameters from stack
        :
```

;find the sum of (1+2+3+4+5), where N=5

Remove parameters from the Stack





Calling the Subroutine

```
        :                                ;find the sum of (1+2+3+4+5), where N=5
Parameters setup
        MOV   R1, #5                     ;Set R0 with #5
        MOV   R0, #0x100                 ;Set R1 with the address
        STMFD SP!, {R1,R0}               ;push R0&R1 to stack
        BL    Sum1N                      ;call subroutine Sum1N
        ADD   SP,SP,#8                   ;add 8 to pop the two parameters from stack
        :
```

Note:

1. Parameters set up before calling the subroutine are **removed** immediately after returning from subroutine.
2. Removal is done by returning the contents of the stack pointer to its **original value** before the parameters were pushed to the stack.

Sum from 1 to N Subroutine
Possible Solution

R6 Answer Addr
R5 N

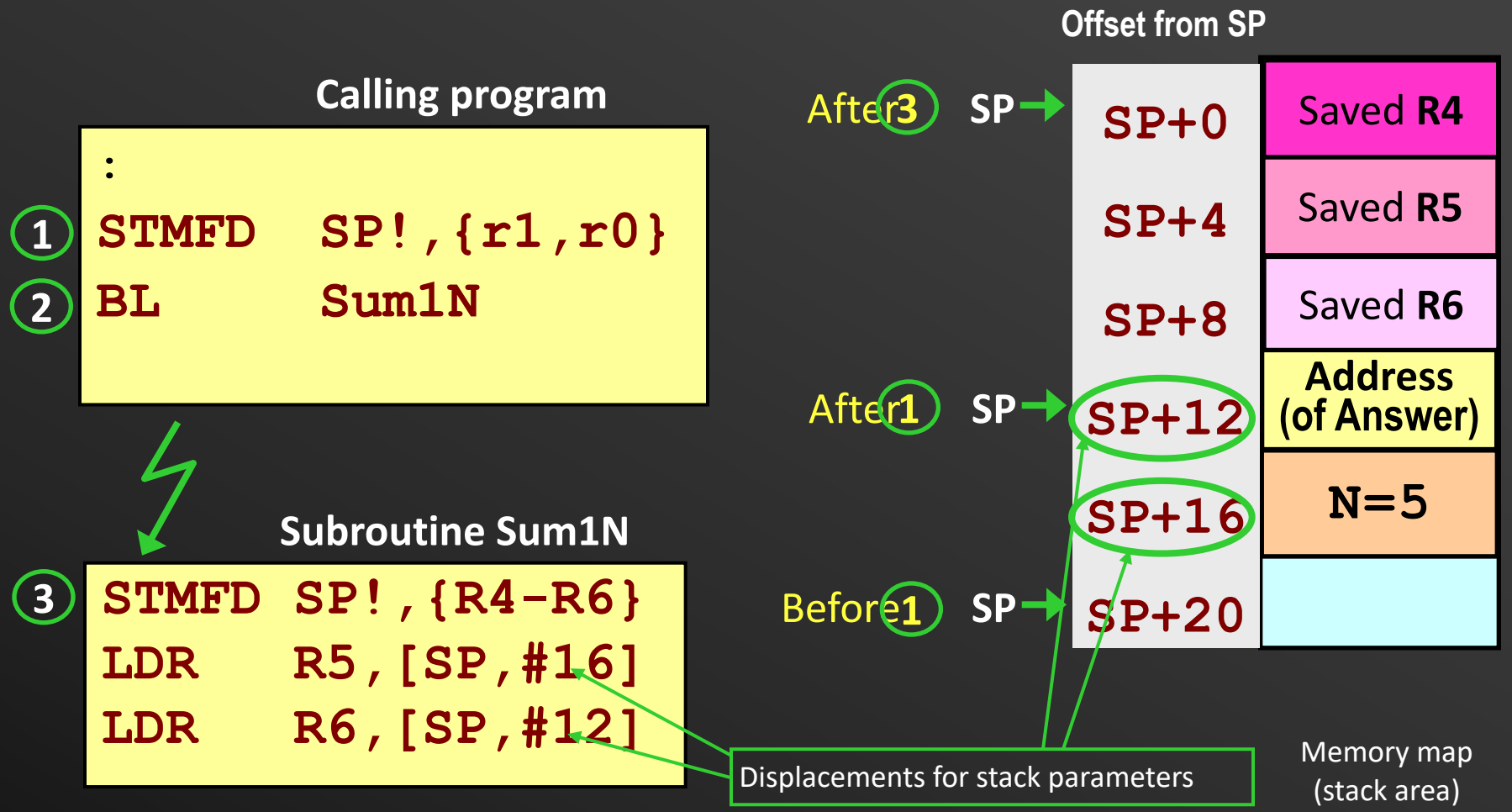
```
Sum1N   STMFD  SP!,{R4,R5,R6}     ;save registers to stack
Retrieve stack parameters
        LDR    R5,[SP,#16]        ;Load N from stack to R5
        LDR    R6,[SP,#12]        ;Load Answer's From stack
        MOV    R4,#0              ;clear summation register R0 to 0
Loop    ADD    R4,R4,R5           ;add current value in R5 to R4
        SUBS   R5,R5,#1           ;decrement current value in R4 by 1
        BNE    Loop               ;jump back to Loop if R4 not yet zero
        STR    R4,[R6]            ;write sum to Answer
        LDMFD  SP!,{R4,R5,R6}     ;restored saved registers
        MOV PC,LR                 ;return from subroutine
```

Z=0

Z=1

Note: The subroutine needs three registers. **R4** to compute the sum from 1 to N. **R6** to be an address pointer to the memory variable **Answer** where the results will written to. **R5** holds the value of N, which is decremented by 1 after each loop till it reaches 0.

Sum from 1 to N Subroutine
Accessing Stack Parameters

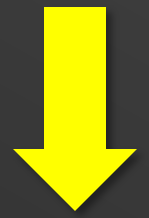


Note: **SP indirect with offset** can be used to access parameters on the stack but knowledge of all items on the stack is needed to compute the correct offset from the current **SP** position.

Transparent Subroutines

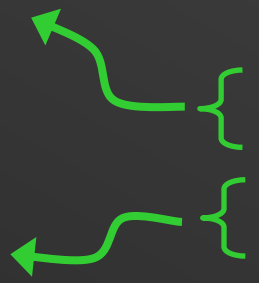
- A transparent subroutine will **not affect any CPU resources** used by the program calling it.
- To achieve this, all local registers used by the subroutine (R4-R11) must be **saved on the stack** on entry and **restored from stack** before returning.

```
SUB1 STMFD  SP!,{R4-R7}     ; save R4 to R7 to stack
      :
      :                     ; registers R4 to R7 are
      :                     ; used in subroutine
     LDMFD  SP!,{R4-R7}     ; restore R4 to R7 from stack
     MOV  PC,LR             ; return to calling program
```



Passing by value and by reference

- Parameters are passed to subroutines in 2 ways:
  - **Pass by value** – the value of the data (or variable) is passed to the subroutine.
  - **Pass by reference** – the address of the variable is passed to the subroutine .



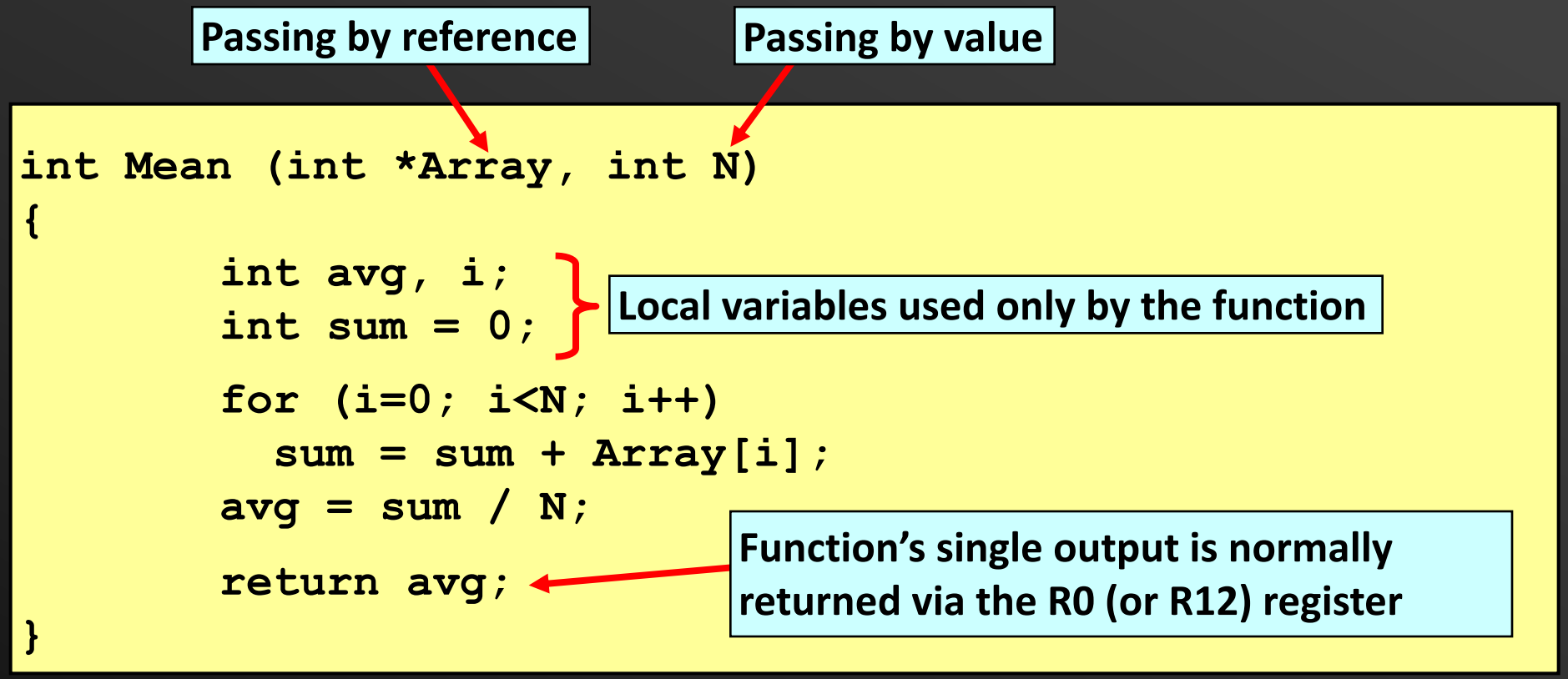
Calling program

```
MOV     R1,#5
MOV     R0,#0x100
STMFD   SP!,{R1,R0}
BL      Sum1N
```

- When is passing by reference used?
  - When the **parameter** passed is to be **modified** by the subroutine.
  - **Large quantity** of data (e.g. array) have to be passed between subroutine and calling program.

C Function Example

- **C** function to compute the mean value of **N** elements in an integer **Array**.



Note: Observe that **local variables** are required by the function to compute the result.

C Function Example

- **C** function to compute the mean value of **N** elements in an integer **Array**.
- Pass both parameters **by reference**.

```
int Mean (int *Array, int *N)
{
        int avg, i;
        int sum = 0;

        for (i=0; i<*N; i++)
          sum = sum + Array[i];
        avg = sum / *N;

        return avg;
}
```

Note: Observe that **local variables** are required by the function to compute the result.

Local Variables

- Subroutines often use local variables whose scope and **life span** exist only during the execution of the subroutine.

Review

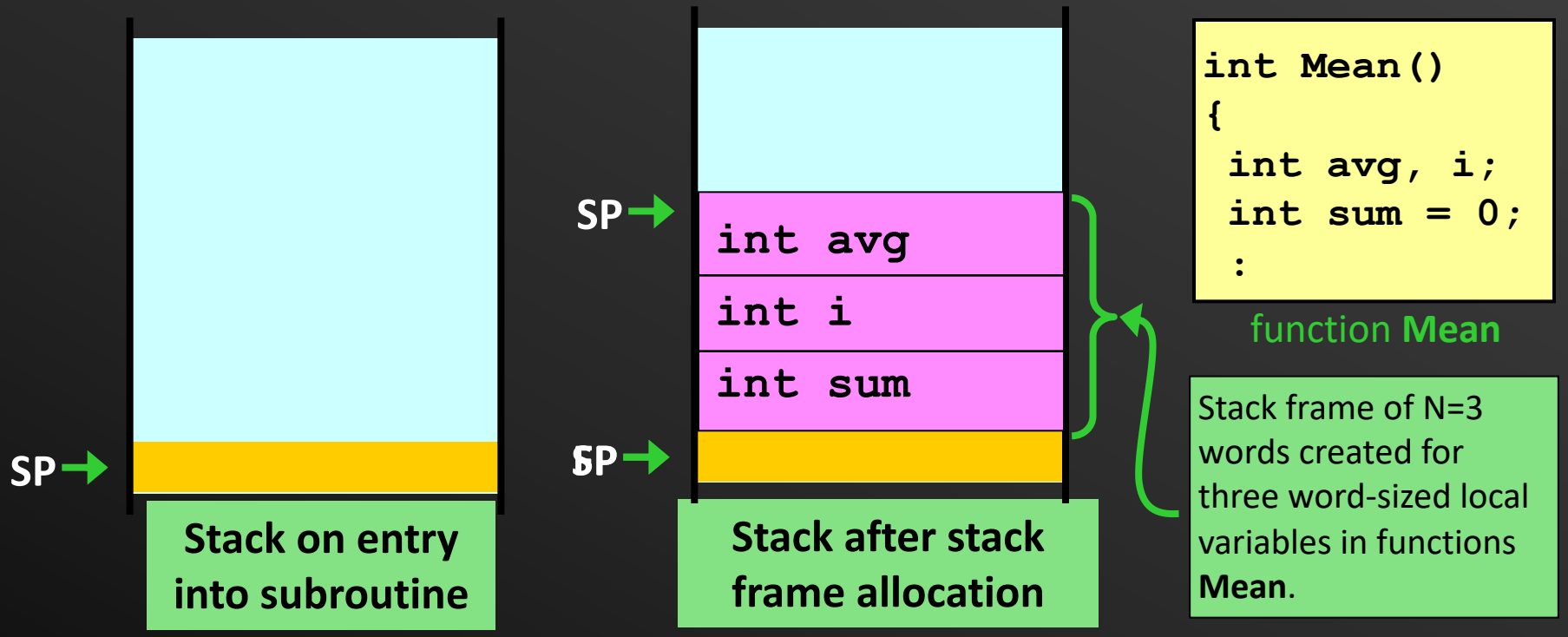
- **Characteristics of a good software module.**
- Loose coupling – **data** within module is entirely **independent** of other modules (local variables).

Local Variables

- Subroutines often use local variables whose scope and **life span** exist only during the execution of the subroutine.
- Memory storage for these variables is **created on entry** into the subroutine and **released on exit** from subroutine.
- The **system stack** is the ideal place to create memory space for temporary variables.
- This temporary memory space is called the **stack frame**.

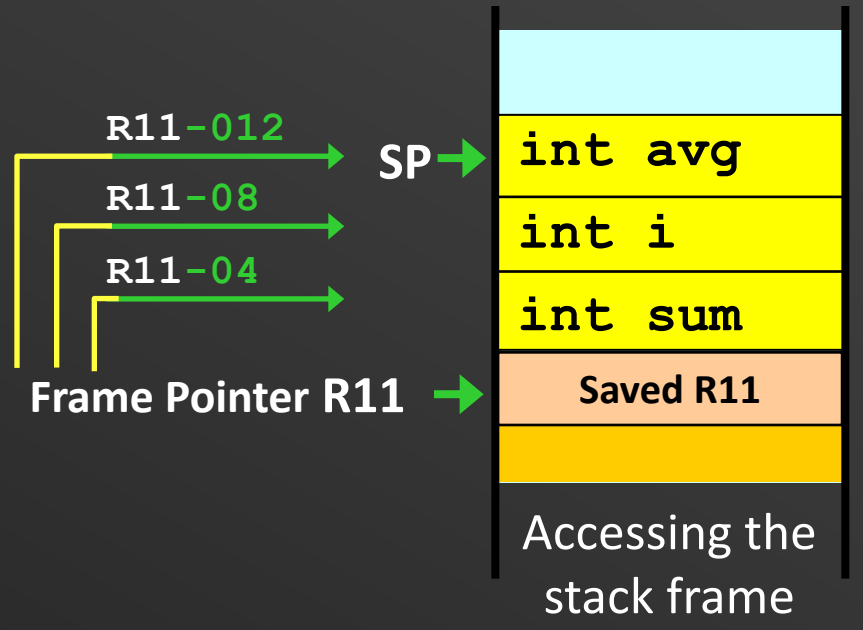
Stack Frame

- Stack frame of **N** words is created on the system stack immediately on entry into a subroutine.
- Memory space within the stack frame can be accessed using with a **frame pointer (FP)** or the stack pointer **(SP)**.
- Frame pointer in ARM can be any register
  - General convention **FP** is **R11**



Using the Frame Pointer

- Consider the use of a frame pointer register **R11**.
  - Original contents of **R11** is saved on the stack before it is used as the frame pointer.
  - Frame pointer (**R11**) now points to the saved **R11** and a stack frame is created by adding frame size **4N** to **SP**.
  - **R11** is the frame reference and an appropriate negative displacement from **R11** can be used to access any stack frame variable.



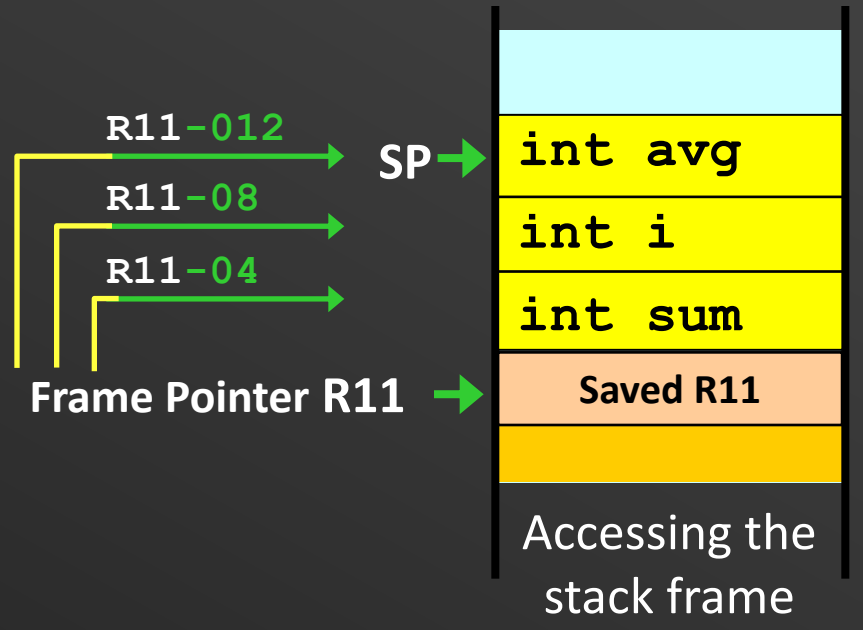
  - When exiting the subroutine, the stack frame can be destroyed by adding **4N+4** to **SP** and copying the saved **R11** value on the stack back into **R11**.

Using the Frame Pointer

- When exiting the subroutine, the stack frame can be destroyed by adding **4N+4** to **SP** and copying the saved **R11** value on the stack back into **R11**.

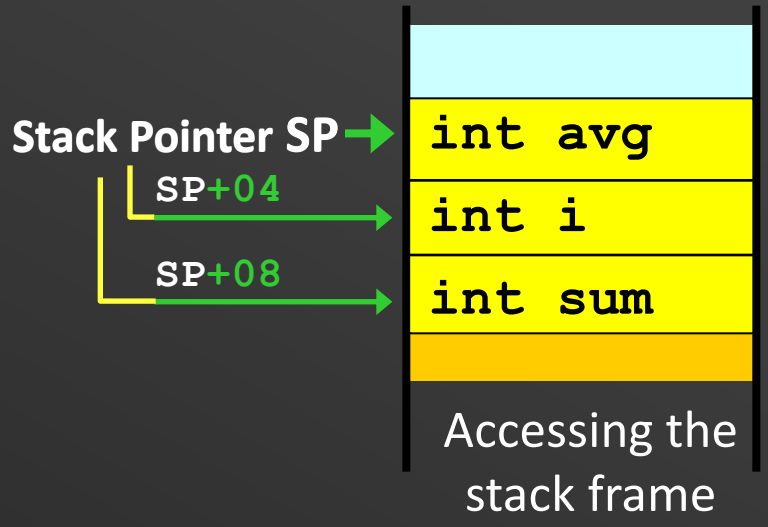
For N=3

```
ADD SP, SP, #16

LDR R11,[R11]
```



Using the Stack Pointer

- A more efficient approach is to use the stack pointer (**SP**) itself.
  - A stack frame is created by adding frame size **4N** to **SP**.
  - **SP** is used as a reference to access all local variables.
  - Appropriate **positive** displacements from **SP** is used to access any of the stack frame variables.
  - **Pro** - This method is more efficient because there is no need to setup a frame pointer.
  - **Con** – More restrictive as system stack cannot be used within subroutine without changing the reference **SP**.



Summary

- Using the **stack** is the most favored means of passing parameters.
  - A combination of methods can be used to implement **Functions**, e.g. parameters passed in via stack and a single result value passed out via a register (e.g. **R0**).
  - Stack-based parameter passing supports **recursion**.
- Parameters is passed by **value** or **reference**.
  - Passing by reference allows the subroutine to directly access memory variables within the calling program .
- A **transparent** subroutine requires registers **used within** the routine to be saved on the **stack** on entry and restore before returning.
- Local variables within a subroutine are usually maintained on the system stack using a **stack frame**.

Chapter 6: Modular Programming-Cont

Mohamed M. Sabry Aly

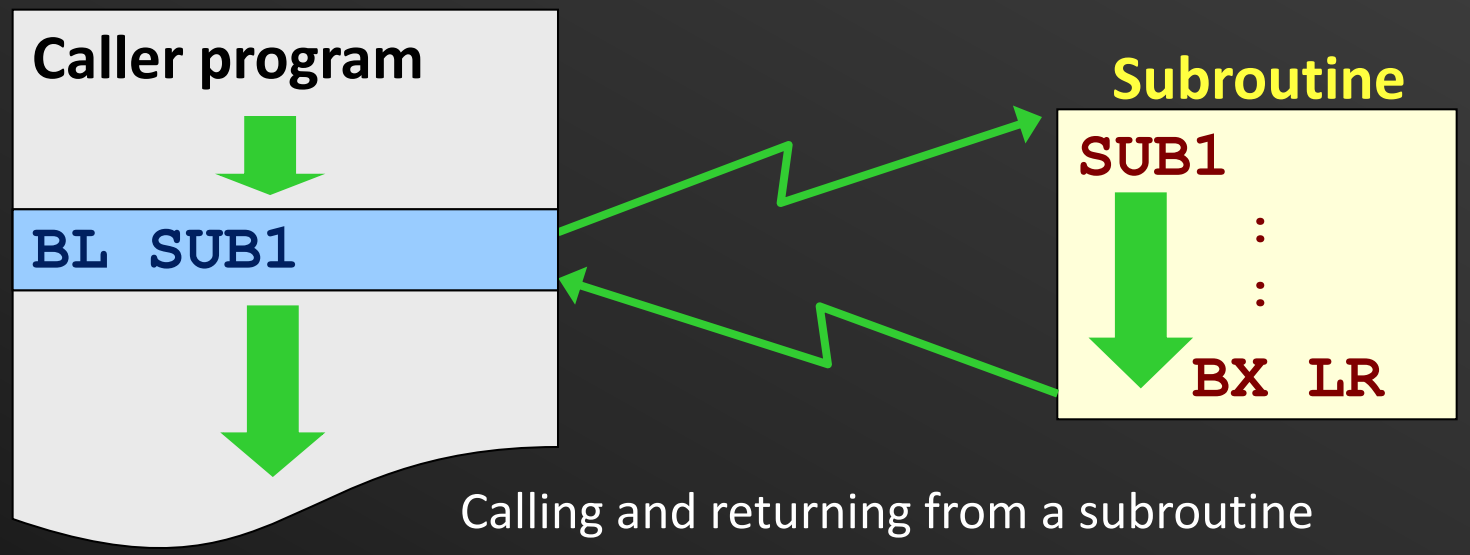
N4-02c-92

Learning Objectives

- Understand the concept of nested subroutine
- Describe how nested subroutines are implemented in ARM
- Describe recursive functions and their implementation in ARM

Subroutines (Recap)

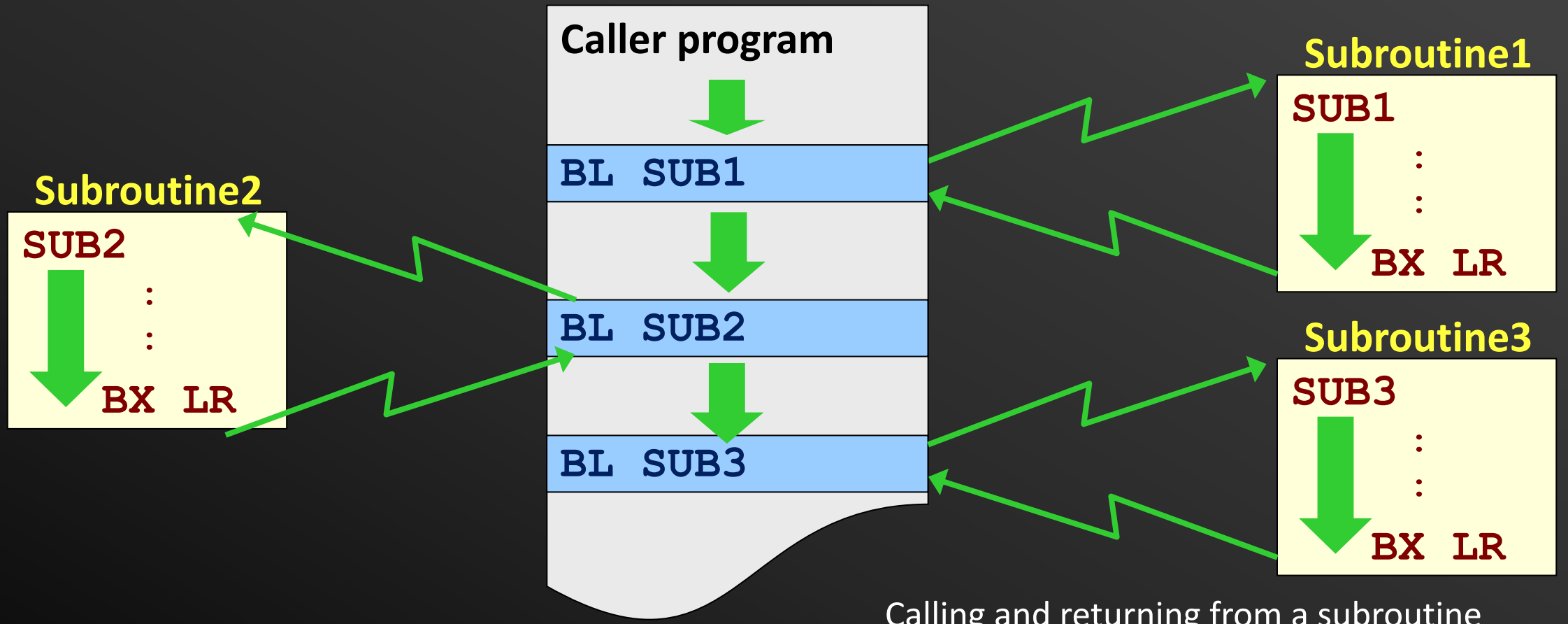
- Modules (e.g., functions in C) are implemented as **subroutines**
- Subroutine can be called from various parts of the program
- Caller and callee
  - Caller: the program that calls subroutine (SUB1)
  - Callee: subroutine (SUB1)



Calling and returning from a subroutine

Multiple Subroutines

- Caller program can call multiple subroutines
- Support of sequential subroutine execution

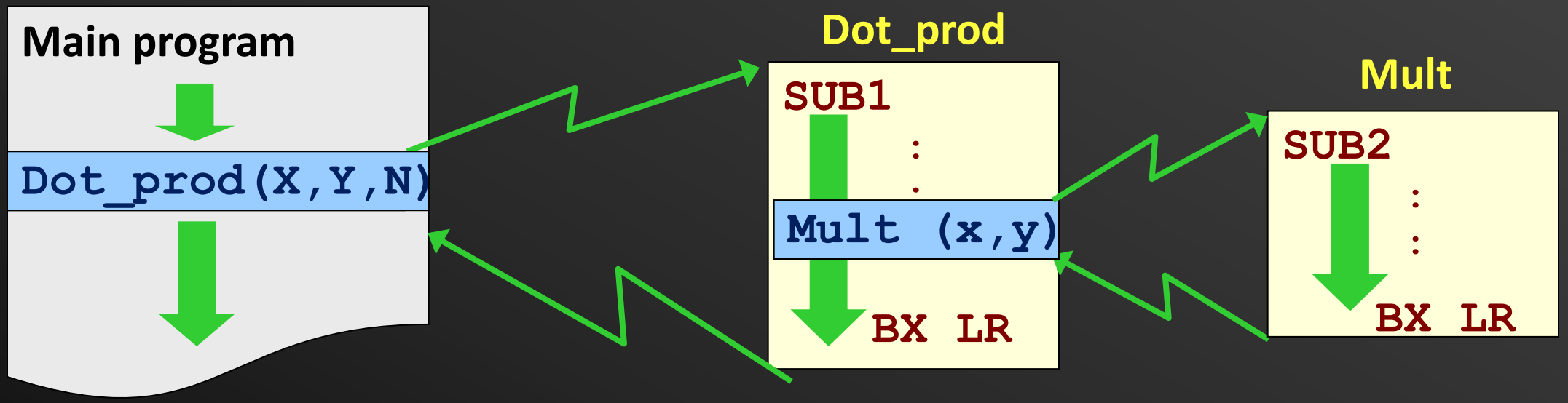


Calling and returning from a subroutine

Do all applications have just 1-level functional call?

Example: Dot product of two arrays

- A subroutine will call another subroutine



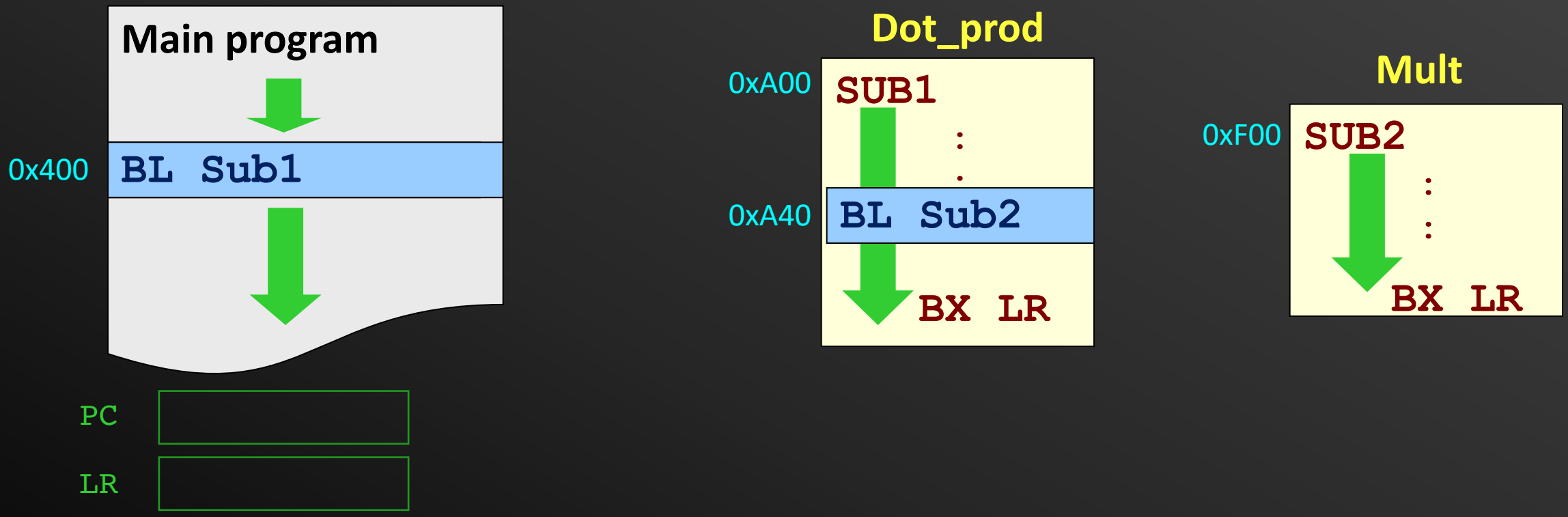
Example: Dot product of two arrays

- A subroutine will call another subroutine

How to ensure right subroutine branch and return?

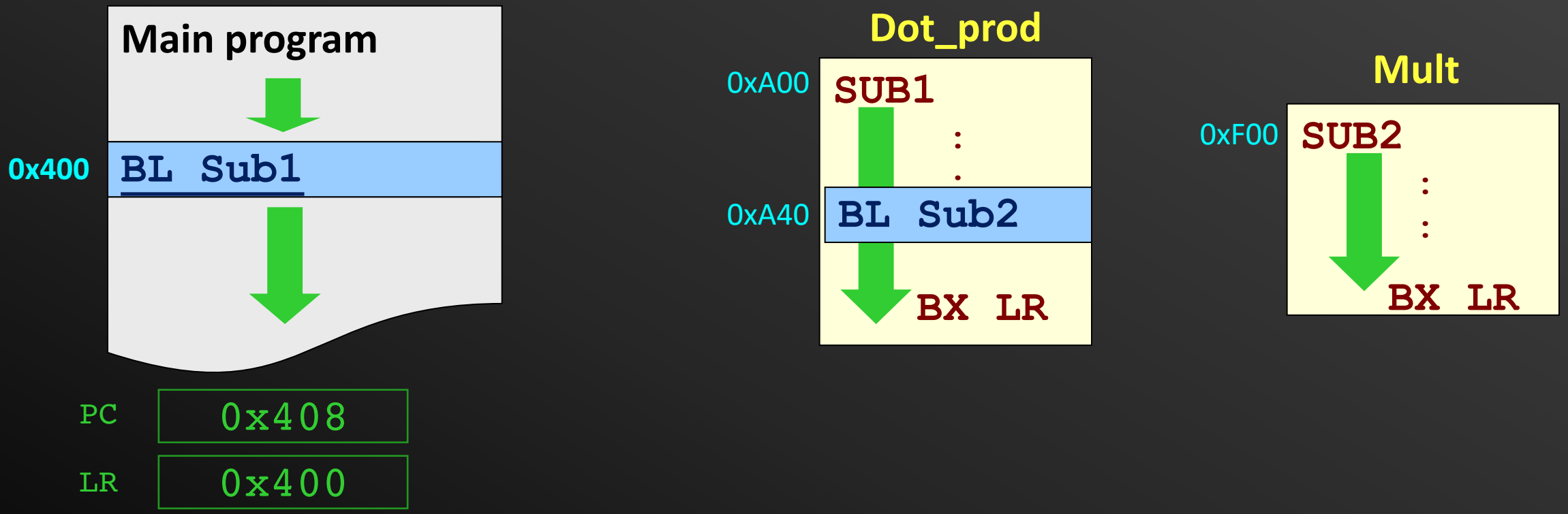
Example: Dot product of two arrays

- A subroutine will call another subroutine



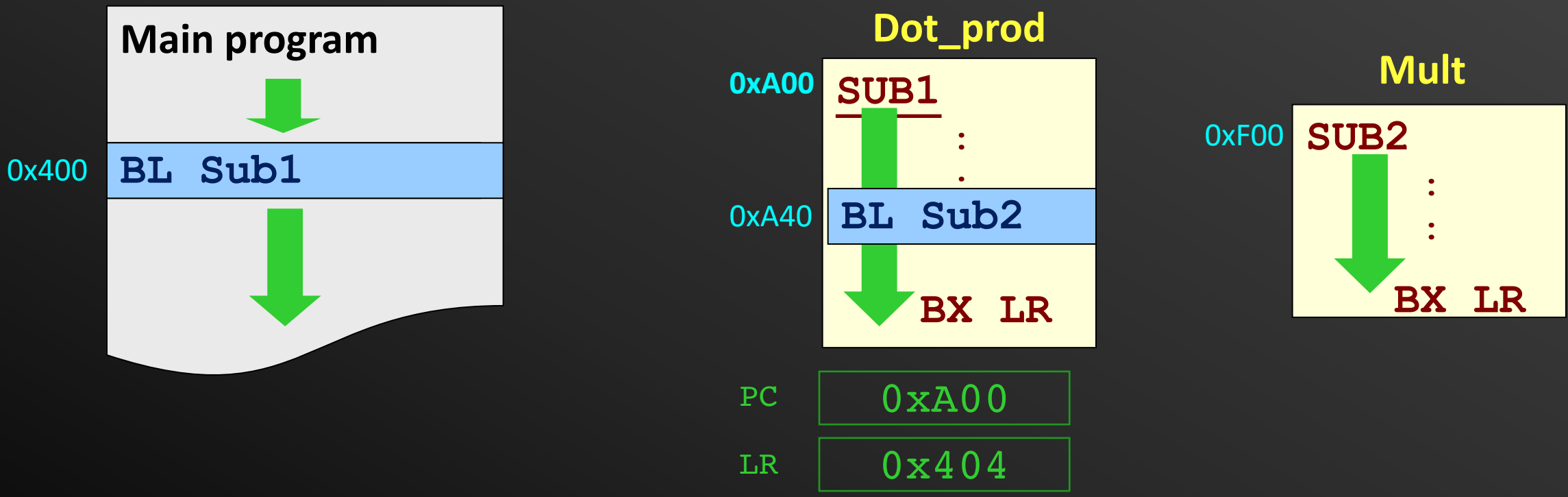
Example: Dot product of two arrays

- A subroutine will call another subroutine



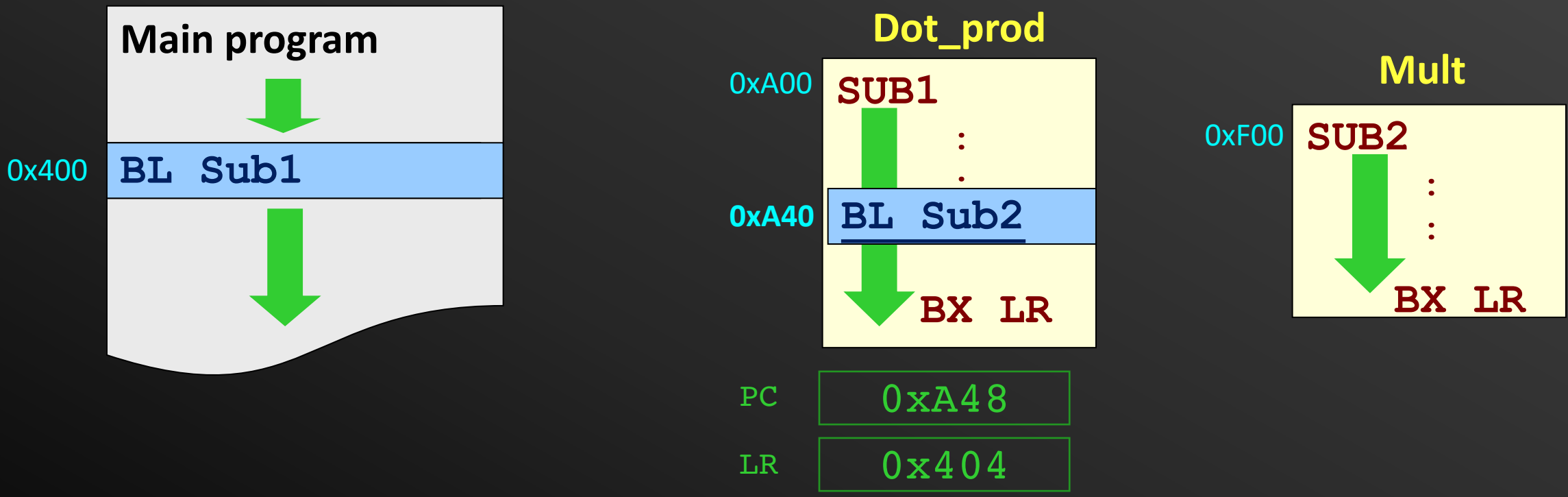
Example: Dot product of two arrays

- A subroutine will call another subroutine



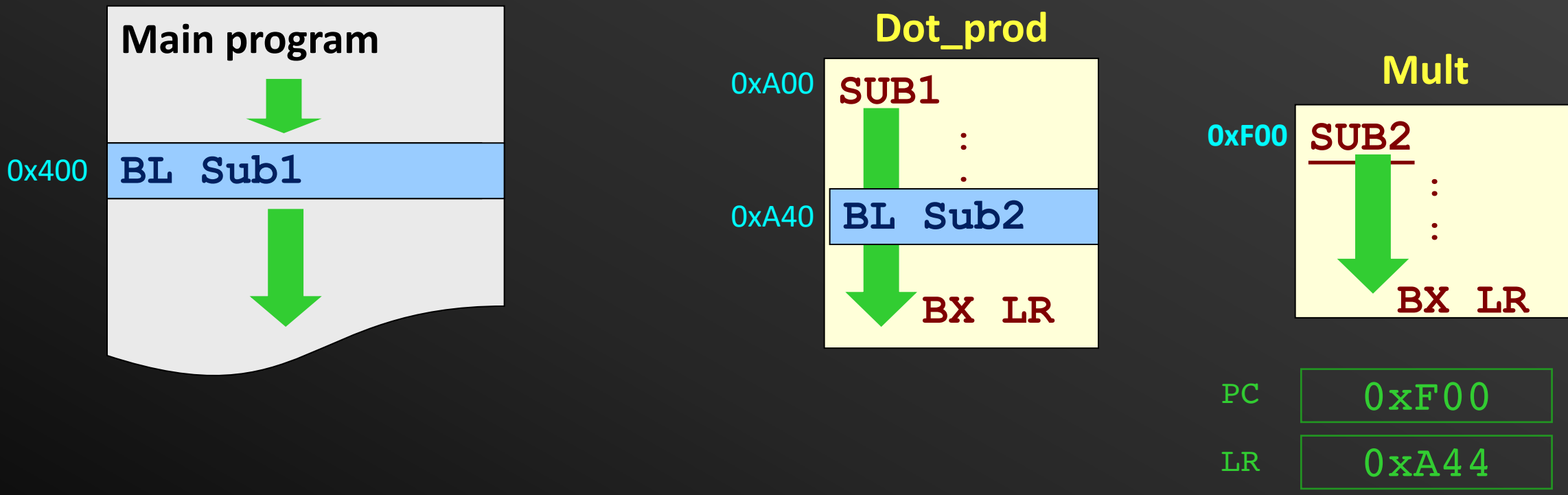
Example: Dot product of two arrays

- A subroutine will call another subroutine



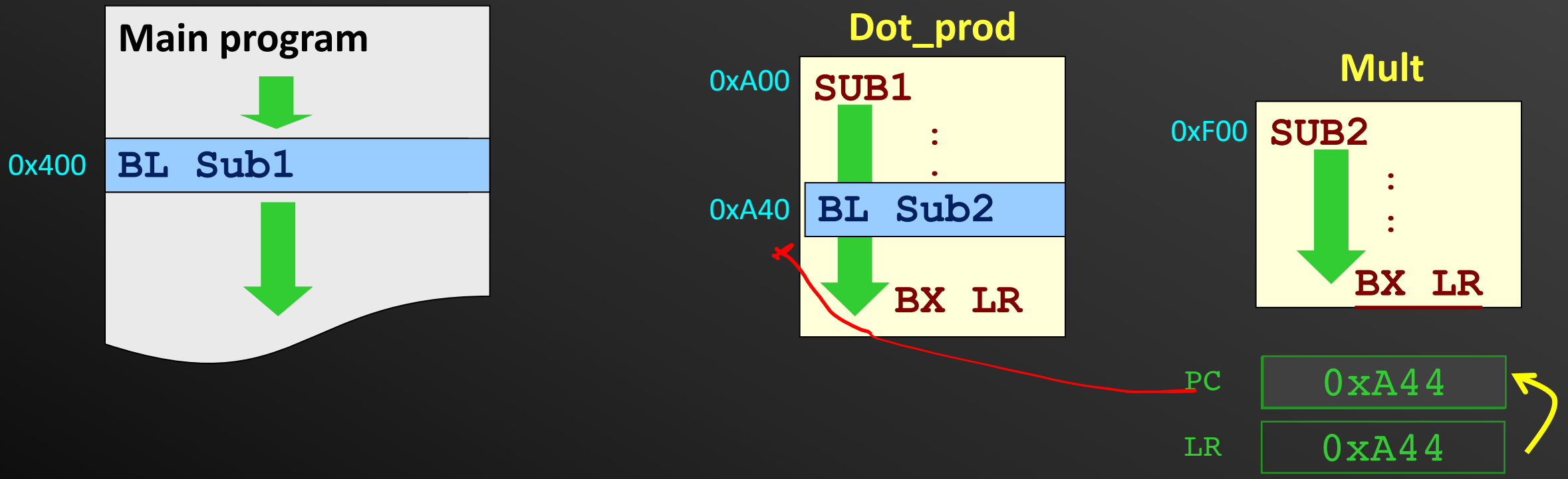
Example: Dot product of two arrays

- A subroutine will call another subroutine



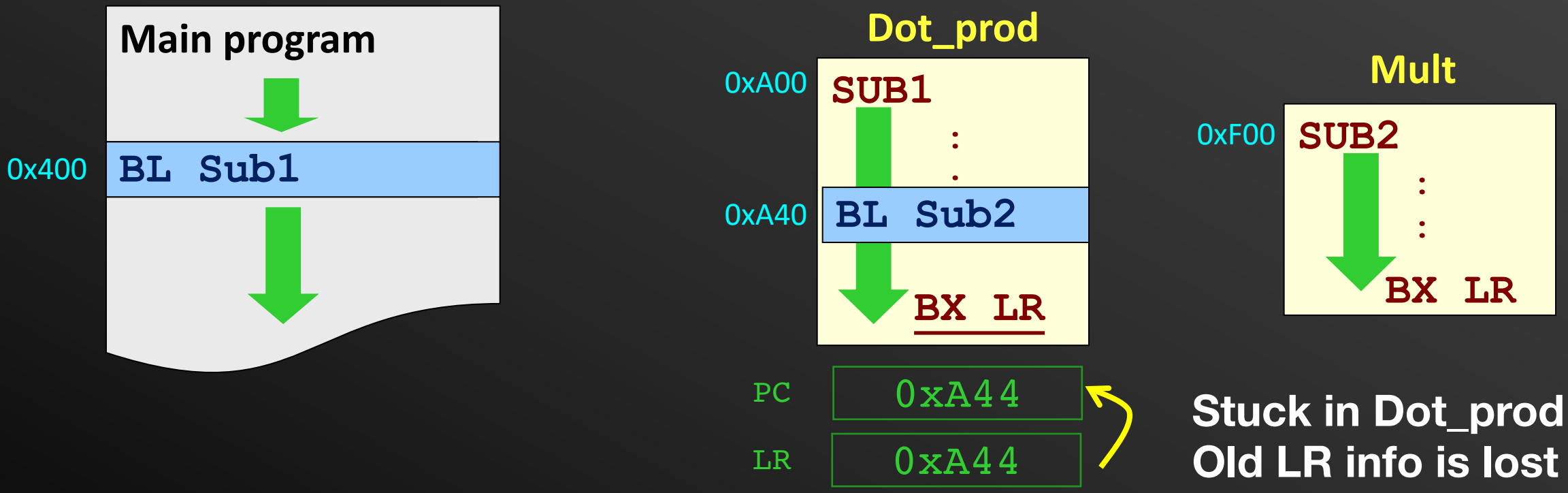
Example: Dot product of two arrays

- A subroutine will call another subroutine



Example: Dot product of two arrays

- A subroutine will call another subroutine



Support for nested Subroutine

- LR needs to be saved somewhere safe → **System stack**

- When so save it? Two options:
  - Just before any BL instruction
  - At the beginning of each subroutine

Example: Dot-product

- Write a subroutine that calculates the dot product of two vectors of **positive integers** using this equation:

$$Z = \sum_{i=0}^{N-1} X_i \times Y_i$$

- Bass addresses of X (0x100), Y(0x200), the array size N(20) and Z(0x300) are passed through the stack

- The subroutine will call another subroutine to calculate the product of two numbers:
  - Numbers are passed in R0,R1, return value in R12

Multiplication subroutine

```
Mult    MOV     R12,#0          ;Clear R12
```

Start by labeling the subroutine and placing the return instruction
Clearing (R2)

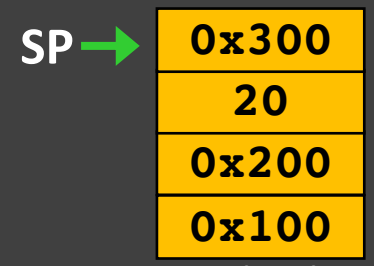
```
        MOV     PC, LR          ; same as bx lr
```

Multiplication subroutine

Iteratively add R0 to R12 for R1 times

```
Mult    MOV     R12,#0          ;Clear R12
Loop    ADD     R12,R12,R0      ;Add R0 to R12
        SUBS    R1,R1,#1        ;Decrement R1 with 1
        BNE     Loop
        MOV     PC, LR          ; same as bx lr
```

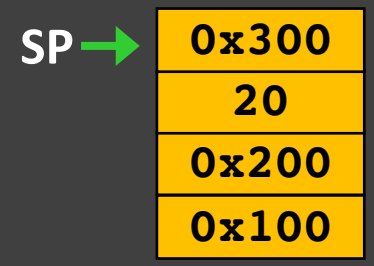
The Dot product Subroutine



System stack when entering DotProd

```
DotProd  STMFD    SP!,{R4-R7}  ;Store regs to stack
         LDR      R4 ,[SP,#28] ;Read location of X
         LDR      R5 ,[SP,#24] ;Read location of Y
         LDR      R6 ,[SP,#20] ;Read arrays length


         LDMFD    SP!,{R4-R7}  ; Restore registers
         MOV      PC, LR       ; same as bx lr
```

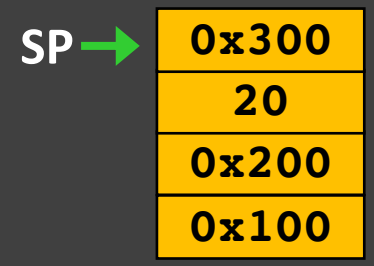
The Dot product Subroutine



System stack when entering DotProd

```
DotProd  STMFD    SP!,{R4-R7}  ;Store regs to stack
         LDR      R4 ,[SP,#28] ;Read location of X
         LDR      R5 ,[SP,#24] ;Read location of Y
         LDR      R6 ,[SP,#20] ;Read arrays length
         MOV      R7, #0       ; Clear R7
Loop1    LDR      R0 ,[R4],#4  ; Get X[i]
         LDR      R1 ,[R5],#4  ; Get Y[i]

         BL       Mult         ; Call Mult Subroutine

         LDMFD    SP!,{R4-R7}  ; Restore registers
         MOV      PC, LR       ; same as bx lr
```

The Dot product Subroutine

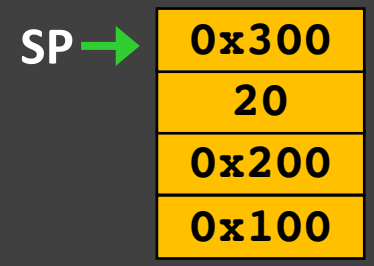


System stack when entering DotProd

```
DotProd     STMFD       SP!,{R4-R7}       ;Store regs to stack
            LDR         R4 ,[SP,#28]      ;Read location of X
            LDR         R5 ,[SP,#24]      ;Read location of Y
            LDR         R6 ,[SP,#20]      ;Read arrays length
            MOV         R7, #0            ; Clear R7 (Sum)
Loop1       LDR         R0 ,[R4],#4       ; Get X[i]
            LDR         R1 ,[R5],#4       ; Get Y[i]

            BL          Mult

            ADD         R7,R7,R12

            LDMFD       SP!,{R4-R7}       ; Restore registers
            MOV         PC, LR            ; same as bx lr
```

```
Mult        MOV         R12,#0            ;Clear R12
Loop        ADD         R12,R12,R0        ;Add R0 to R12
            SUBS        R1,R1,#1          ;Decrement R1 with 1
            BNE         Loop
            MOV         PC, LR            ; same as bx lr
```

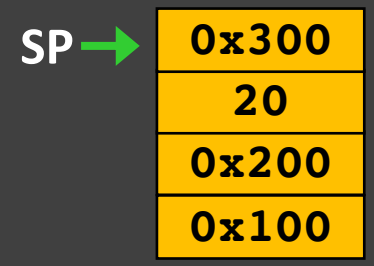
The Dot product Subroutine



System stack when entering DotProd

```
DotProd     STMFD       SP!,{R4-R7}       ;Store regs to stack
            LDR         R4 ,[SP,#28]      ;Read location of X
            LDR         R5 ,[SP,#24]      ;Read location of Y
            LDR         R6 ,[SP,#20]      ;Read arrays length
            MOV         R7, #0            ; Clear R7 (Sum)
Loop1       LDR         R0 ,[R4],#4       ; Get X[i]
            LDR         R1 ,[R5],#4       ; Get Y[i]

            BL          Mult              ; Call Mult Subroutine

            ADD         R7,R7,R12         ; Add the product to R7
            SUBS        R6,R6,#1          ; Reduce the counter by 1
            BNE         Loop1             ; not 0 then repeat

            LDMFD       SP!,{R4-R7}       ; Restore registers
            MOV         PC, LR            ; same as bx lr
```

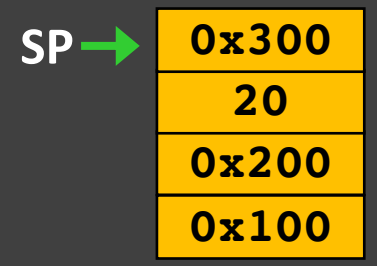
The Dot product Subroutine



System stack when entering DotProd

```
DotProd     STMFD       SP!,{R4-R7}         ;Store regs to stack
            LDR         R4 ,[SP,#28]              ;Read location of X
            LDR         R5 ,[SP,#24]              ;Read location of Y
            LDR         R6 ,[SP,#20]              ;Read arrays length
            MOV         R7, #0                    ; Clear R7 (Sum)
Loop1       LDR         R0 ,[R4],#4               ; Get X[i]
            LDR         R1 ,[R5],#4         ; Get Y[i]

            BL          Mult                ; Call Mult Subroutine

            ADD         R7,R7,R12           ; Add the product to R7
            SUBS        R6,R6,#1            ; Reduce the counter by 1
            BNE         Loop1               ; not 0 then repeat
            LDR         R4, [SP,#16]              ; read destination address
            STR         R7, [R4]            ; Store in destination address
            LDMFD       SP!,{R4-R7}         ; Restore registers
            MOV         PC, LR                    ; same as bx lr
```

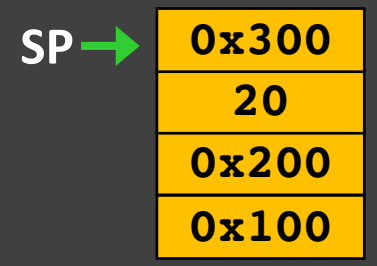
The Dot product Subroutine



System stack when entering DotProd

```
DotProd     STMFD       SP!,{R4-R7}       ;Store regs to stack
            LDR         R4 ,[SP,#28]      ;Read location of X
            LDR         R5 ,[SP,#24]      ;Read location of Y
            LDR         R6 ,[SP,#20]      ;Read arrays length
            MOV         R7, #0            ; Clear R7 (Sum)
Loop1       LDR         R0 ,[R4],#4       ; Get X[i]
            LDR         R1 ,[R5],#4       ; Get Y[i]
            STR         LR ,[SP,#-4]!     ; Push Link register to SP
            BL          Mult              ; Call Mult Subroutine
            LDR         LR ,[SP],#4       ; Pop Link Register from SP
            ADD         R7,R7,R12         ; Add the product to R7
            SUBS        R6,R6,#1          ; Reduce the counter by 1
            BNE         Loop1             ; not 0 then repeat
            LDR         R4, [SP,#16]      ; read destination address
            STR         R7, [R4]          ; Store in destination address
            LDMFD       SP!,{R4-R7}       ; Restore registers
            MOV         PC, LR            ; same as bx lr
```

The Dot product Subroutine



System stack when entering `DotProd`

✓ Straight forward, no other impact on the code

```
STR     LR ,[SP,#-4]!     ; Push Link register to SP
BL      Mult              ; Call Mult Subroutine
LDR     LR ,[SP],#4       ; Pop Link Register from SP
```

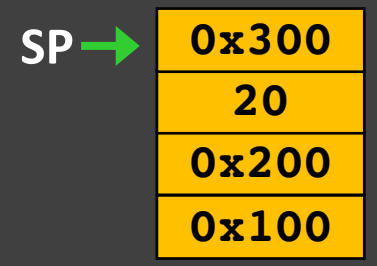
✗ Different instruction structure

Support for nested Subroutine

- LR needs to be saved somewhere safe → **System stack**

- When so save it? Two options:
  - Just before any BL instruction
  - At the beginning of each subroutine

The Dot product Subroutine



System stack when entering DotProd

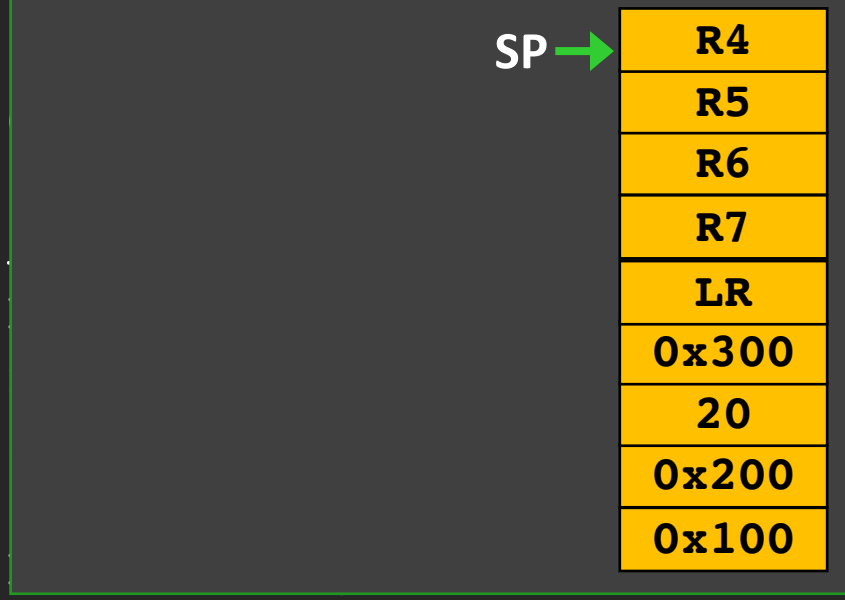
```
DotProd     STMFD       SP!,{R4-R7,LR}    ;Store regs to stack
            LDR         R4 ,[SP,#28]      ;Read location of X
            LDR         R5 ,[SP,#24]      ;Read location of Y
            LDR         R6 ,[SP,#20]      ;Read arrays length
            MOV         R7, #0            ; Clear R7 (Sum)
Loop1       LDR         R0 ,[R4],#4       ; Get X[i]
            LDR         R1 ,[R5],#4       ; Get Y[i]

            BL          Mult              ; Call Mult Subroutine

            ADD         R7,R7,R12         ; Add the product to R7
            SUBS        R6,R6,#1          ; Reduce the counter by 1
            BNE         Loop1             ; not 0 then repeat
            LDR         R4, [SP,#16]      ; read destination address
            STR         R7, [R4]          ; Store in destination address
            LDMFD       SP!,{R4-R7,LR}    ; Restore registers
            MOV         PC, LR            ; same as bx lr
```

The Dot product Subroutine

```
DotProd     STMFD       SP!,{R4-R7,LR}     ;Store regs to stack
            LDR         R4 ,[SP,#28]       ;R
            LDR         R5 ,[SP,#24]       ;R
            LDR         R6 ,[SP,#20]       ;R
            MOV         R7, #0             ;
Loop1       LDR         R0 ,[R4],#4        ;
            LDR         R1 ,[R5],#4        ;

            BL          Mult               ;

            ADD         R7,R7,R12          ;
            SUBS        R6,R6,#1           ;
            BNE         Loop1              ;
            LDR         R4, [SP,#16]       ;
            STR         R7, [R4]           ;                                    ss
            LDMFD       SP!,{R4-R7,LR}     ;
            MOV         PC, LR             ; same as bx lr
```

System stack when entering DotProd

We now push 5 registers to stack
Need to update the offsets

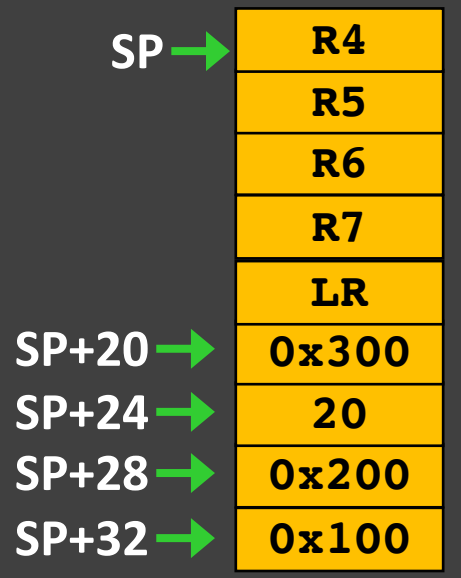


The Dot product Subroutine

```
DotProd     STMFD       SP!,{R4-R7,LR}     ;Store regs to stack
            LDR         R4 ,[SP,#32]       ;R
            LDR         R5 ,[SP,#28]       ;R
            LDR         R6 ,[SP,#24]       ;R
            MOV         R7, #0             ;
Loop1       LDR         R0 ,[R4],#4        ;
            LDR         R1 ,[R5],#4        ;

            BL          Mult               ;

            ADD         R7,R7,R12          ;
            SUBS        R6,R6,#1           ;
            BNE         Loop1              ;
            LDR         R4, [SP,#20]       ;
            STR         R7, [R4]           ;
            LDMFD       SP!,{R4-R7,LR}     ;
            MOV         PC, LR             ; same as bx lr
```

System stack when entering DotProd

We now push 5 registers to stack
Need to update the offsets



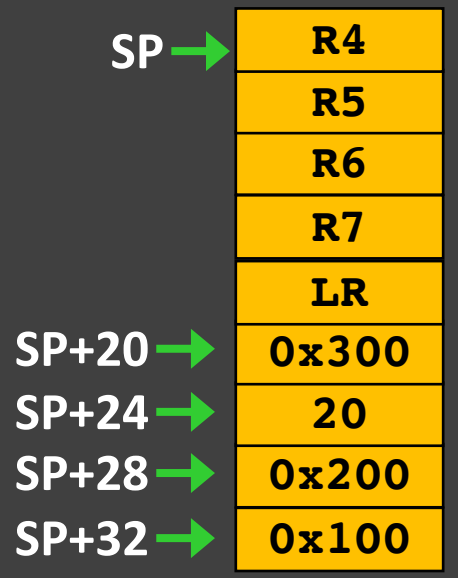
The Dot product Subroutine

```
DotProd     STMFD       SP!,{R4-R7,LR}      ;Store regs to stack
            LDR         R4 ,[SP,#32]        ;R
            LDR         R5 ,[SP,#28]        ;R
            LDR         R6 ,[SP,#24]        ;R
            MOV         R7, #0              ;
Loop1       LDR         R0 ,[R4],#4         ;
            LDR         R1 ,[R5],#4         ;

            BL          Mult                ;

            ADD         R7,R7,R12           ;
            SUBS        R6,R6,#1            ;
            BNE         Loop1               ;
            LDR         R4, [SP,#20]        ;
            STR         R7, [R4]            ;
            LDMFD       SP!,{R4-R7,PC}      ;
```

Why is this correct?

System stack when entering DotProd

We now push 5 registers to stack
Need to update the offsets



Recursive Subroutine

- A recursive routine calls itself within its own body.
- Recursion is an elegant way to solve algorithms or mathematical expressions that have **systematic repetitions**. (e.g. factorial **n! = n×(n-1)×(n-2)….2×1**)

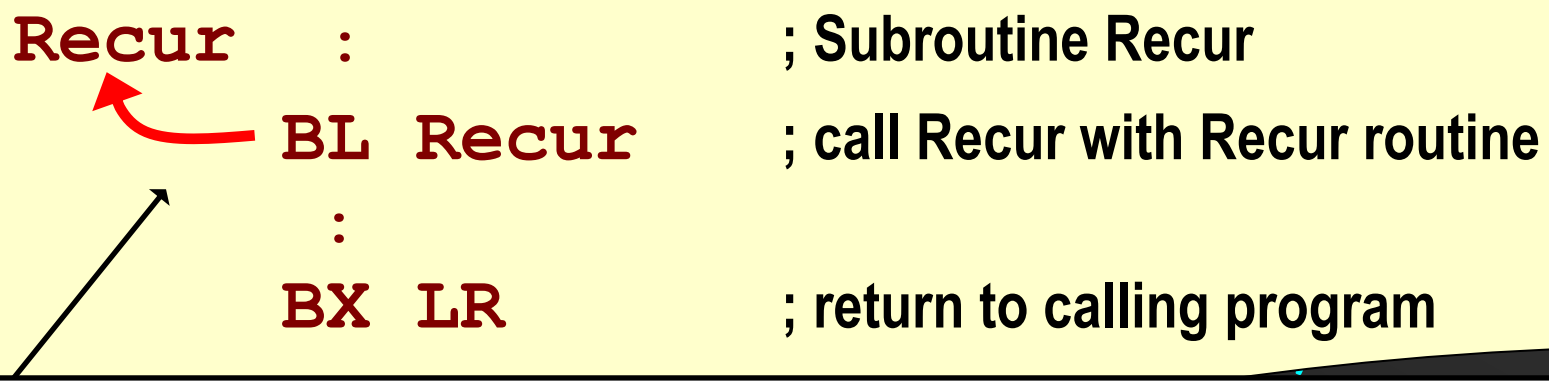
```
int factorial(int n){
    if (n<0)
     return -1; //sanity check
    if (n==0)
     return 1;
    else
     return n * factorial(n-1);
}
```

Recursive Subroutine

- A recursive routine calls itself within its own body.
- Recursion is an elegant way to solve algorithms or mathematical expressions that have **systematic repetitions**. (e.g. factorial **n! = n×(n-1)×(n-2)….2×1**)

Main problem: Without saving LR, execution will be stuck

Recursive Code Example



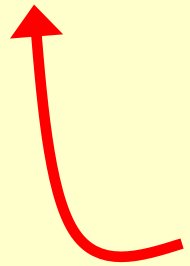
```
Recur   :                ; Subroutine Recur
        BL  Recur        ; call Recur with Recur routine
        :
        BX  LR           ; return to calling program
```

Another problem: Some condition must be reached that allows `BL Recur` to be skipped in order to avoid infinite recursion.

Recursive Subroutine

- A recursive routine calls itself within its own body.
- Recursion is an elegant way to solve algorithms or mathematical expressions that have **systematic repetitions**. (e.g. factorial **n! = n×(n-1)×(n-2)….2×1**)

Recursive Code Example



```
Recur   STMFD SP!,{…,LR}        ; Subroutine Recur
```

Condition to terminate the subroutine **(branch to Done)**

```
        :
        BL Recur                ; call Recur with Recur routine
        :
Done    LDMFD SP!,{…,LR}
        BX LR                   ; return to calling program
```

Example: Fibonacci Sequence

- Number sequence

  0,1,1,2,3,5,8,13,21,34,55

- Starting from index 1

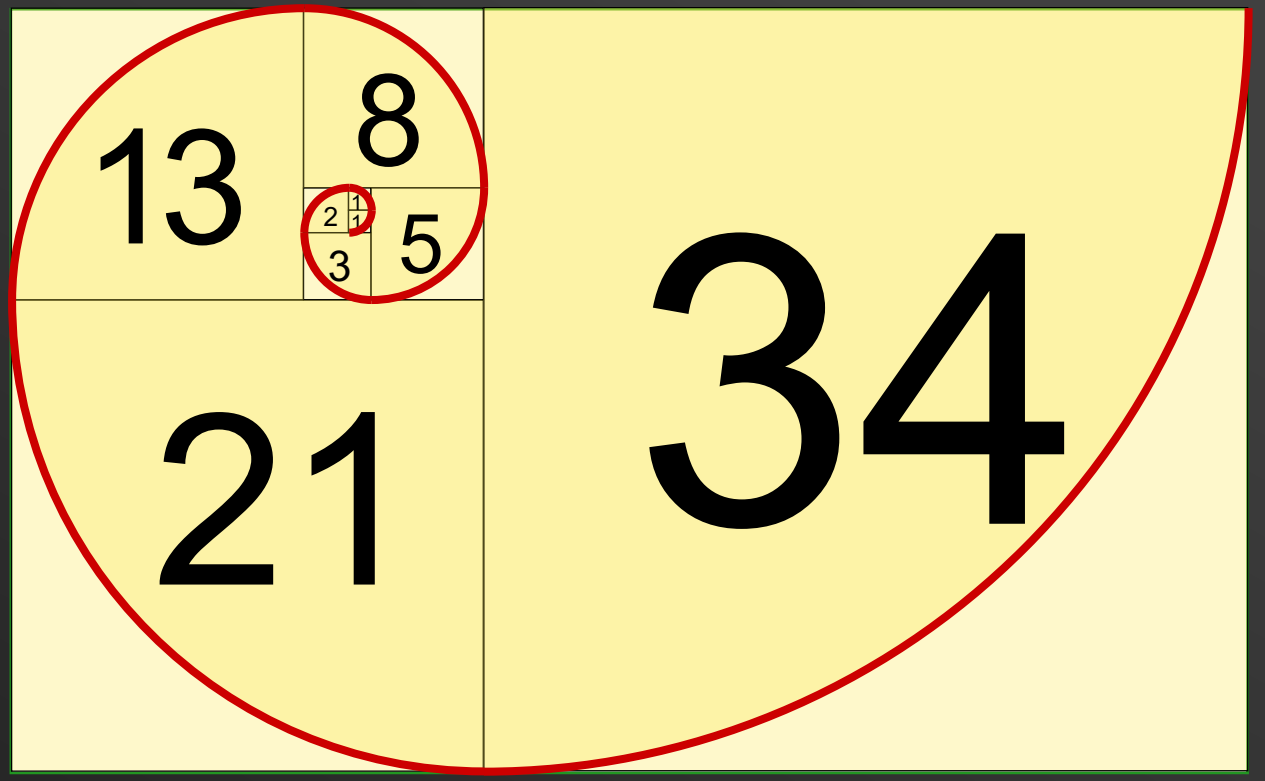
  $F(n) = F(n-1) + F(n-2)$

  $F(2) = 1$

  $F(1) = 0$

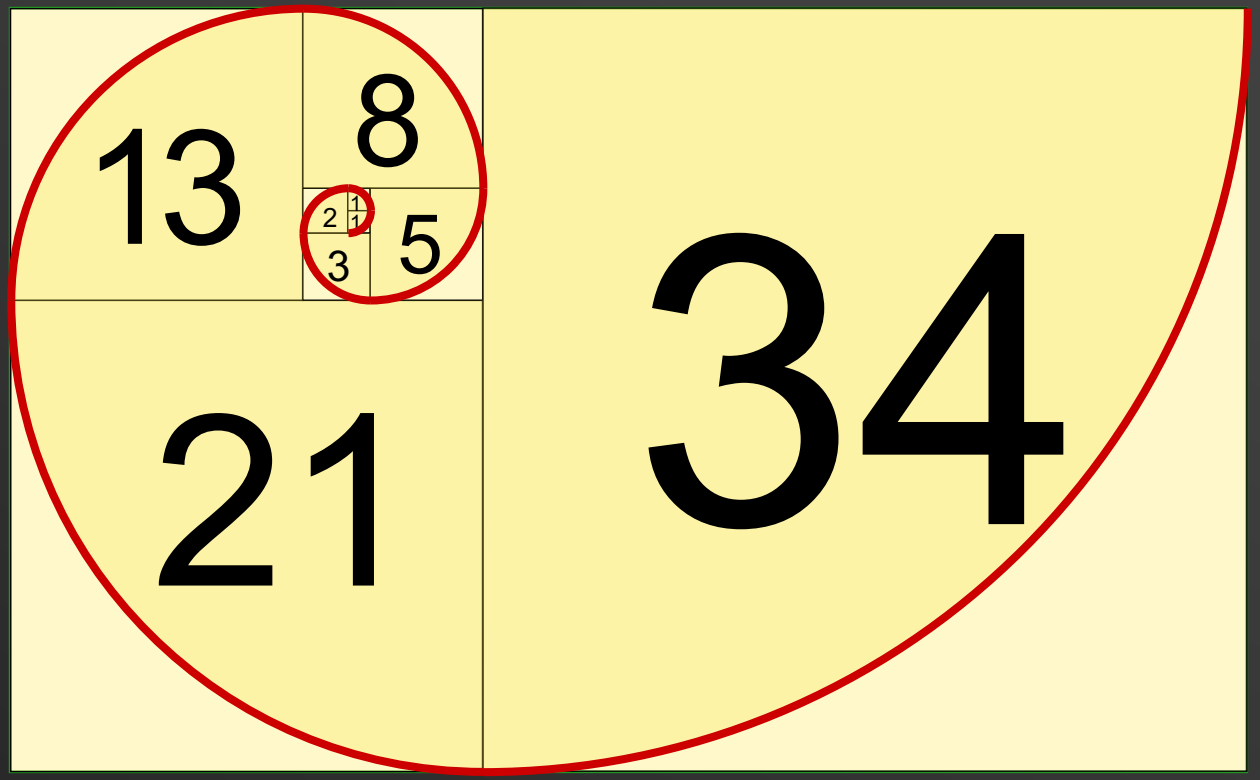
  Pass parameter in stack

  Return result in R12



Example: Fibonacci Sequence

- Number sequence
  0,1,1,2,3,5,8,13,21,34,55




```
int Fibonacci(int n){
     int result;
    if (n==1)
     result=0;
    elseif (n==2)
     result=1;
    else
     result = Fibonacci(n-1) + Fibonacci(n-2);
    return result;
}
```

Example Solution

System stack when entering `Fib`

```
Fib             STMFD           SP!,{R4-R6,LR}      ;Store regs to stack
```

```
Done            LDMFD           SP!,{R4-R6,LR}      ; Restore registers
                MOV             PC, LR              ; same as bx lr
```

Example Solution

System stack when entering `Fib`

```
Fib     STMFD   SP!,{R4-R6,LR}   ;Store regs to stack
        LDR     R4 ,[SP,#16]     ;Read Value of n
        CMP     R4 ,#1           ;Compare with 1 (if n==1)
        MOVEQ   R12,#0           ;Assign result with 0
        CMP     R4 ,#2           ;Compare with 2 (if n==2)
        MOVEQ   R12,#1           ; Assign result with 1
        BLE     Done             ; if n less than equal 2 finish

Done    LDMFD   SP!,{R4-R6,LR}   ; Restore registers
        MOV     PC, LR           ; same as bx lr
```

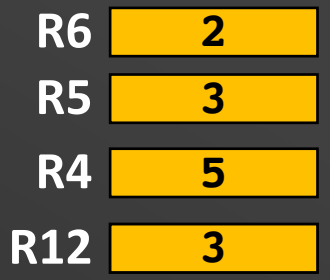
Example Solution

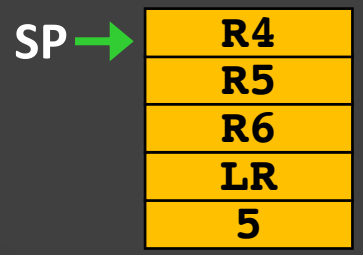

```
Fib     STMFD   SP!,{R4-R6,LR}   ;Store regs to stack
        LDR     R4 ,[SP,#16]     ;Read Value of n
        CMP     R4 ,#1           ;Compare with 1 (if n==1)
        MOVEQ   R12,#0           ;Assign result with 0
        CMP     R4 ,#2           ;Compare with 2 (if n==2)
        MOVEQ   R12,#1           ; Assign result with 1
        BLE     Done             ; if n less than equal 2 finish
        SUB     R5,R4,#1         ; Get n-1
        STMFD   SP!,{R5}         ;Push n-1 to stack
        BL      Fib              ; calculate Fib(n-1)
        LDMFD   SP!,{R5}         ; Pop from stack

Done    LDMFD   SP!,{R4-R6,LR}   ; Restore registers
        MOV     PC, LR           ; same as bx lr
```

Example Solution


```
Fib     STMFD   SP!,{R4-R6,LR}   ;Store regs to stack
        LDR     R4 ,[SP,#16]     ;Read Value of n
        CMP     R4 ,#1           ;Compare with 1 (if n==1)
        MOVEQ   R12,#0           ;Assign result with 0
        CMP     R4 ,#2           ;Compare with 2 (if n==2)
        MOVEQ   R12,#1           ; Assign result with 1
        BLE     Done             ; if n less than equal 2 finish
        SUB     R5,R4,#1         ; Get n-1
        STMFD   SP!,{R5}         ;Push n-1 to stack
        BL      Fib              ; calculate Fib(n-1)
        LDMFD   SP!,{R5}         ; Pop from stack
        MOV     R6,R12           ;Save result in temp register
        SUB     R5,R4,#2         ; Get n-2
        STMFD   SP!,{R5}         ;Push n-2 to stack
        BL      Fib              ; calculate Fib (n-2)
        LDMFD   SP!,{R5}         ; Pop from stack

Done    LDMFD   SP!,{R4-R6,LR}   ; Restore registers
        MOV     PC, LR           ; same as bx lr
```

Example Solution






```
Fib     STMFD   SP!,{R4-R6,LR}   ;Store regs to stack
        LDR     R4 ,[SP,#16]     ;Read Value of n
        CMP     R4 ,#1           ;Compare with 1 (if n==1)
        MOVEQ   R12,#0           ;Assign result with 0
        CMP     R4 ,#2           ;Compare with 2 (if n==2)
        MOVEQ   R12,#1           ; Assign result with 1
        BLE     Done             ; if n less than equal 2 finish
        SUB     R5,R4,#1         ; Get n-1
        STMFD   SP!,{R5}         ;Push n-1 to stack
        BL      Fib              ; calculate Fib(n-1)
        LDMFD   SP!,{R5}         ; Pop from stack
        MOV     R6,R12           ;Save result in temp register
        SUB     R5,R4,#2         ; Get n-2
        STMFD   SP!,{R5}         ;Push n-2 to stack
        BL      Fib              ; calculate Fib (n-2)
        LDMFD   SP!,{R5}         ; Pop from stack
        ADD     R12,R12,R6       ; Calculate Fib(n-1) + Fib(n-2)
Done    LDMFD   SP!,{R4-R6,LR}   ; Restore registers
        MOV     PC, LR           ; same as bx lr
```

Example Solution

```
Fib         STMFD       SP!,{R4-R6,LR}      ;Store regs to stack
            LDR         R4 ,[SP,#16]        ;Read Value of n
            CMP         R4 ,#1              ;Compare with 1 (if n==1)
            MOVEQ       R12,#0              ;Assign result with 0
            CMP         R4 ,#2              ;Compare with 2 (if n==2)
            MOVEQ       R12,#1              ; Assign result with 1
            BLE         Done                ; if n less than equal 2 finish
            SUB         R5,R4,#1            ; Get n-1
            STMFD       SP!,{R5}            ;Push n-1 to stack
            BL          Fib                 ; calculate Fib(n-1)
            LDMFD       SP!,{R5}            ; Pop from stack
            MOV         R6,R12              ;Save result in temp register
            SUB         R5,R4,#2            ; Get n-2
            STMFD       SP!,{R5}            ;Push n-2 to stack
            BL          Fib                 ; calculate Fib (n-2)
            LDMFD       SP!,{R5}            ; Pop from stack
            ADD         R12,R12,R6          ; Calculate Fib(n-1) + Fib(n-2)
Done        LDMFD       SP!,{R4-R6,LR}      ; Restore registers
            MOV         PC, LR              ; same as bx lr
```

Summary

- Nested subroutines are key for truly modular designs
  - Need to ensure that return address is not overwritten.
- Recursive subroutine is very efficient implementation of subroutines
  - Ensure: 1) stopping condition and 2) return address stored

Chapter 7: Flow-control Constructs

Mohamed M. Sabry Aly

N4-02c-92

Learning Objectives

- Be able to convert a high level IF and IF-ELSE construct to its low-level equivalent.

- Describe compute-efficient considerations for compound AND and OR constructs.

- Describe and analyze simple branchless logic constructs.

IF Statements

- How is the **IF** construct implemented?
- **Imitating** the high-level test condition does not result in very efficient assembly-level implementation.

```
if (a > b)
  {S1}
```

Convert to assembly code

```
        CMP a,b
        BGT DoIF
        B Skip
DoIF    {S1}
Skip
```

- High-level test condition is **reversed** in assembly-level to avoid the need for an additional unconditional jump.

```
if (a > b)
  {S1}
```

Convert to assembly code

```
        CMP a,b
        BLE Skip
        {S1}
Skip
```

Note: The reverse condition test of **HS** is **LO**, reverse of **LT** is **GE**

Find Largest Number

```
        MOV   R0,#0x100      ;setup pointer to first array element
        MOV   R1,#9
        LDR   R3,[R0]
Loop    LDR   R2,[R0,#4
        CMP   R2,R3          ;compare R3 and R2 (i.e. R2-R3)
        BLO   Skip           ;branch if R2 <  current max (i.e. R3)
        MOV   R3,R2          ;update current max. in R3 with R2
Skip    SUBS  R1,R1,#1       ;decrement 1 from counter register
        BNE   Loop           ;jump back to Loop if not zero
```

```
if R2 >= R3   // R2 larger or equal to current max
{
   R3 = R2;   // then update R3 with new max R2
}
```

R0 = Address pointer for current array element.
R1 = Loop counter register
R2 = Temporary register holding current no.
R3 = Current maximum value (i.e. the result).

Can We Avoid Reversion?

```
        MOV   R0,#0x100     ;setup pointer to first array element
        MOV   R1,#9
        LDR   R3,[R0]
Loop    LDR   R2,[R0,#4
        CMP   R2,R3
        BLO   Skip
        MOV   R3,R2
Skip    SUBS  R1,R1,#1
        JNE   Loop
```

```
if R2 >= R3   // R2 larger or equal to current max
{
   R3 = R2;   // then update R3 with new max R2
}
```

```
        CMP     R2,R3
        BHS     Assign
        SUBS    R1,R1,#1
        BNE     Loop
Assign  MOV     R3,R2
        SUBS    R1,R1,#1
        BNE     Loop
```

Redundant

Conditional Execution

- In the 32-bit ARM ISA, instructions can be conditionally executed based on the CC flags.
- Consider the following 32-bit ARM code segment.

```
; C code
if (r0 == 1)
 r1 = 3;
```

```
        CMP   r0, #1        ; set CC based on r0 –1
        BNE   Skip          ; if (r0 == 1)
        MOV   r1, #3        ; then { r1 := 3}
SKIP      ......            ;
```

- It can be replaced using conditional execution instructions.

```
        CMP    r0, #1       ; if (r0 == 1)
        MOVEQ  r1, #3       ; then { r1 := 3}
SKIP    ......              ;
```

Largest with Conditional Execution

- Significant reduction in Code size

```
        MOV   R0,#0x100      ;setup pointer to first array element
        MOV   R1,#9
        LDR   R3,[R0]
Loop    LDR    R2,[R0,#
        CMP    R2,R3         ;compare R3 and R2 (i.e. R2-R3)
        MOVGE  R3,R2         ;update current max. in R3 with R2
        SUBS   R1,R1,#1      ;decrement 1 from counter register
        JNE    Loop          ;jump back to Loop if not zero
```

```
if R2 >= R3   // R2 larger or equal to current max
{
   R3 = R2;   // then update R3 with new max R2
}
```

Does Conditional Execution work with Just 1 Instruction?

No, as long as the status registers are not altered from the comparison instruction

Find Largest Number, and Store Its Index

```
R3= X[0];
R4=0;
R0= &X[0]
For (R1=9;R1>0;R1--){
        R0+=4;R2=X[R0];
       if(R2>=R3){
              R3=R2;
              R4=10-R1;
       }
}
```

No conditional execution

```
        MOV   R0,#0x100      ;setup pointer to first array element
        MOV   R1,#9          ;load 9 into counter register
        MOV   R4,#0          ;load 0 into index_max register
        LDR   R3,[R0]        ; 1st no. in array is current max
Loop    LDR   R2,[R0,#4]!    ;get next no. in array
        CMP   R2,R3          ;compare R3 and R2 (i.e. R2-R3)
        BLO   Skip           ;branch if R2 <  current max (i.e.
        MOV   R3,R2          ;update current max. in R3 with R2
        RSUB  R4,R1,#10      ;Store the index in R4
Skip    SUBS  R1,R1,#1       ;decrement 1 from counter register
        BNE   Loop           ;jump back to Loop if not zero
```

Find Largest Number, and Store Its Index

```
R3= X[0];
R4=0;
R0= &X[0]
For (R1=9;R1>0;R1--){
        R0+=4;R2=X[R0];
       if(R2>=R3){
              R3=R2;
              R4=10-R1;
       }
}
```

With conditional execution

```
        MOV   R0,#0x100      ;setup pointer to first array element
        MOV   R1,#9          ;load 9 into counter register
        MOV   R4,#0          ;load 0 into index_max register
        LDR   R3,[R0]        ; 1st no. in array is current max
Loop    LDR     R2,[R0,#4]!  ;get next no. in array
        CMP     R2,R3        ;compare R3 and R2 (i.e. R2-R3)
        MOVGE   R3,R2        ;update current max. in R3 with R2
        RSUBGE  R4,R1,#10    ;Store the index in R4
        SUBS    R1,R1,#1     ;decrement 1 from counter register
        BNE   Loop           ;jump back to Loop if not zero
```

IF-ELSE Statements

- How is the **IF-ELSE** construct implemented?
- Combination of conditional and unconditional jumps used for the **IF-ELSE** construct.
- Reversing high-level test condition does not improve efficiency unless
- the ELSE code segment {S2} is more likely to execute.

```
        CMP a,#3
        BNE DoElse
        {S1}
        B Skip
DoElse  {S2}
Skip    :
```

Reversing test condition

```
if (a == 3)
   {S1}
else
   {S2}
```

Convert to assembly code

```
        CMP a,#3
        BEQ DoIf
        {S2}
        B Skip
DoIf    {S1}
Skip    :
```

IF-ELSE implementation in low-level assembly

Conditional Execution for IF-ELSE

- Higher efficient coding
- In the shown example, Skip will always be the 4th instruction

```
; 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      ......            ;
```

- With conditional execution

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

Compound AND Conditions

- How are compound AND conditions handled?
- Logical AND can bind multiple basic relational conditions.

e.g.

```
if ((a==b) && (b>0)) {S1}
```

order of compound AND test

- Compilers resolve compound conditions into simpler ones.

```
    if (a!=b) then Skip
    if (b<=0) then Skip
       {S1}
Skip    :
```

- Elementary conditions bound by the logical AND are tested from **left-to-right**, in the order given in the C program.
- The first false condition means the remaining conditions are not computed. This is called the **fast Boolean operation**.
- Keep the **least likely** condition **leftmost** in your program for more efficient execution.

Compound Condition Example

- Not all cases will be executed correctly

```
if ((a==b) && (b>0)) {S1}
```

What if a>b and b>0

```
CMP     R1,R2  ;R1<-a, R2<-b
CMPEQ   R2,#0
XXXGT   XXXX   ; S1
```

Compound OR Conditions

- How are compound OR conditions handled?

e.g.

```
if ((a == 1) || (a == 2)) {S1}
```

- Most compilers eliminate an unconditional jump at the end of the compound OR series by **reversing** the **last conditional** test.

```
        if (a == 1) then DoIf
        if (a != 2) then Skip
DoIf    {S1}
Skip      :
```

- The conditional test that is **most likely** to be **true** should be kept leftmost.

Compound Condition Example

```
if ((a == 1) || (a == 2)) {S1}
```

With conditional assignment
S1 needs to be replicated.
Not suitable for multiple instructions

```
CMP     R1,#1  ;R1<-a
XXXEQ   XXXX   ;  S1
CMPNE   R1,#2
XXXEQ   XXXX   ;  S1
```

A more holistic approach

```
        CMP     R1,#1
        BEQ     doIF
        CMPNE   R1,#2
doIF    XXXEQ   XXXX   ;  S1
```

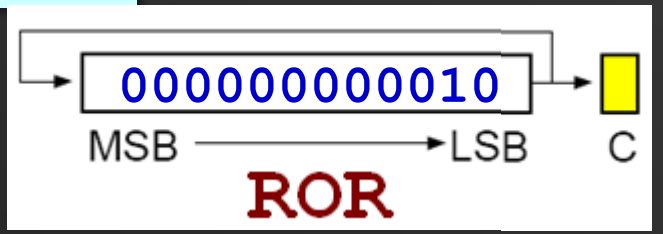
Branchless Logic

- Branchless logic avoid using conditional jump instructions when implementing logical constructs.
- `Bcc` instructions may result in costly **flushing** operations when the wrong next instruction is pre-fetched into the CPU's pipeline.
- How is branchless logic implemented?
- **Exploit arithmetic relationship** to transform the test condition into the corresponding desired outcome. Can only be applied in special cases and desired outcomes are usually Boolean values.

e.g.

```
if ((X & 2) == 2)
  X = true;
else
  X = false;
```

```
AND R1,R1,#0x02
ROR R1,R1,#1
```



- **Conditional execution** can be used to avoid branching.

Summary

- The **IF** and **ELSE** constructs are implemented using one or more conditional jump (**`Bcc`**).
  - Using the **reverse** conditional test can help the IF construct execute more efficiently.
- **Conditional execution in ARM** greatly improves flow-control efficiency.

- Sequencing the **least likely** or **most likely** conditional test can help improve execution speed of compound **AND** and **OR** respectively .
- **Branchless logic** and **conditional execution** techniques can help keep the CPU pipeline efficient by maintaining **sequential** execution.

Chapter 7: Flow-control Constructs

Mohamed M. Sabry Aly

N4-02c-92

Learning Objectives

- Describe how SWITCH constructs can be implemented efficiently for consecutive narrow and random wide cases.

- Contrast the implementations of pre and post-test loop constructs like WHILE, DO-WHILE and FOR.

Switch Statement

- How is the **SWITCH** construct implemented?
- The assembly code produced varies between compilers and depends on the nature and range of the case values.
- Two different SWITCH scenarios are examined:

```
switch(x)  {
 case 0 : {S0};
          break;

 case 1 : {S1};
          break

 case 2 : {S2};
          break;

 case 3 : {S3};
}
```

Running values narrow range

```
switch(x)  {
 case 1    : {S0};
             break;

 case 10   : {S1};
             break

 case 100  : {S2};
             break;

 case 1000 : {S3};
}
```

Random values wide range

Switch – Running & Narrow Values

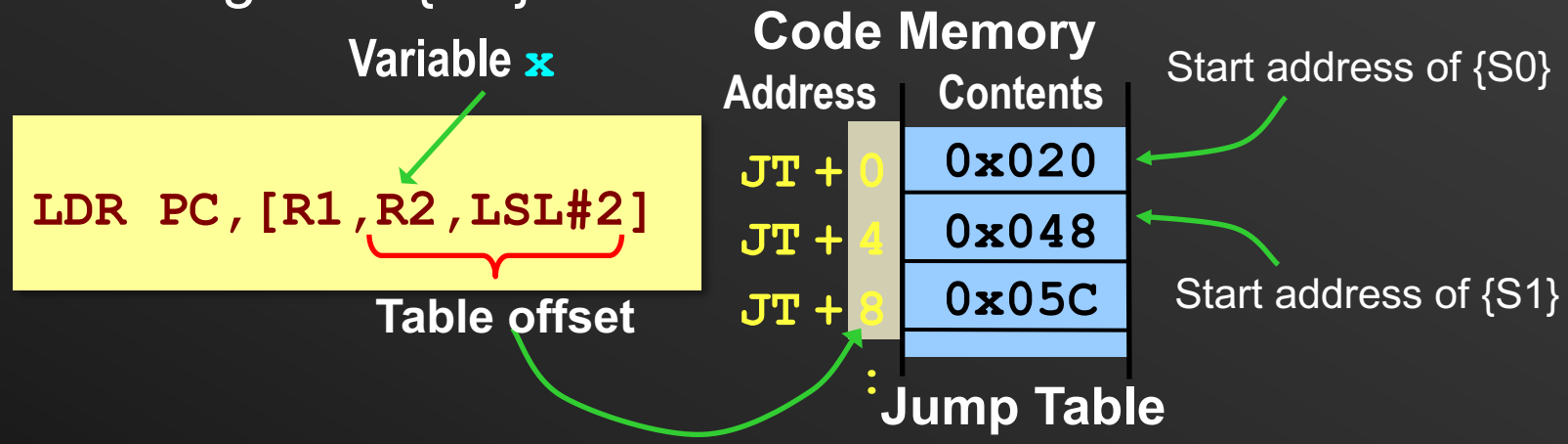
```
switch(x)
{
case 0:
 {S0};
 break;

case 1:
 {S1};
 break;

case 2:
 {S2};
 break;

case 3:
 {S3};
}
```

Switch – Running & Narrow Values

- If cases are consecutive narrow range values, a **Jump Table** is used to avoid testing each case in turn.

```
switch(x)
{
case 0:
  {S0};
 break;

case 1:
  {S1};
 break;

case 2:
 {S2};
 break;

case 3:
 {S3};
}
```

- Jump table contains list of **start addresses** for the code segments that is associated with each case values.
- Var **x** on which the switch is decided, acts as an **offset** into the table to access the corresponding start address.
- Start address is loaded into **PC** to execute the required code segment {S**x**}.



Note: Time taken is on average less than the equivalent if-else-if cascade and is independent of number of cases in the switch construct.

Jump Table Example

```
switch(x)
{
case 0:
 G='F';
 break;

case 1:
 G='D';
 break;

case 2:
 G='C';
 break;

case 3:
 G='B';
}
```

Cascaded if-else

```
      LDR R1, [R2]
      CMP R1, #0
      BEQ C1
      CMP R1, #1
      BEQ C2
      CMP R1, #2
      BEQ C3
      MOV R0, #66 ;'B'
RES   STR R0, [R2,#4]
      .
      .
      .
C1    MOV R0, #70 ;'F'
      B RES
C2    MOV R0, #68 ;'D'
      B RES
C3    MOV R0, #67 ;'C'
      B RES
```

Jump Table

```
      LDR R1, [R2]
      ADR R3, TBL
      LDR PC, [R3,R1, LSL #2]
RES   STR R0, [R2,#4]
      .
      .
0xA00 C1   MOV R0, #70 ;'F'
      B RES
C2    MOV R0, #68 ;'D'
      B RES
C3    MOV R0, #67 ;'C'
      B RES
```

Address	Contents
TBL + 0	0xA00
TBL + 4	0xA08
TBL + 8	0xA10
:	

Jump Table

Jump Table Example (Conditional Execution)

```
switch(x)
{
case 0:
 G='F';
 break;

case 1:
 G='D';
 break;

case 2:
 G='C';
 break;

case 3:
 G='B';
}
```

Cascaded if-else

```
      LDR R1, [R2]
      CMP R1, #0
      MOVEQ R0,#70
      CMP R1, #1
      MOVEQ R0,#68
      CMP R1, #2
      MOVEQ R0,#67
      CMP R1, #3
      MOVEQ R0, #66 ;'B'
RES   STR R0, [R2,#4]
      .
      .
      .
```

Jump Table

```
      LDR R1, [R2]
      ADR R3, TBL
      LDR PC, [R3,R1, LSL #2]
RES   STR R0, [R2,#4]
      .
      .
0xA00 C1   MOV R0, #70 ;'F'
      B RES
C2    MOV R0, #68 ;'D'
      B RES
C3    MOV R0, #67 ;'C'
      B RES
```

Address	Contents
TBL + 0	0xA00
TBL + 4	0xA08
TBL + 8	0xA10
:	

Jump Table

Jump Table Example (Default)

```
switch(x)
{
case 0:
 G='F';
 break;

case 1:
 G='D';
 break;

case 2:
 G='C';
 break;

case 3:
 G='B';
Default:
G='X';
}
```

Cascaded if-else

```
        LDR R1, [R2]
        CMP R1, #0
        BEQ C1
        CMP R1, #1
        BEQ C2
        CMP R1, #2
        BEQ C3
        CMP R1,#3
        BEQ C4
        MOV R0, #88 ;'X'
RES     STR R0,[R2,#4]
        .
        .
C1      MOV R0, #70 ;'F'
        B RES
C2      MOV R0, #68 ;'D'
        B RES
C3      MOV R0, #67 ;'C'
        B RES
C4      MOV R0, #66; 'B'
        B RES
```

Jump Table

```
        LDR R1, [R2]
        MOV R0, #88; 'X'
        ADR R3, TBL
        LDR PC, [R3,R1, LSL #2]
RES     STR R0, [R2,#4]
        .
0xA00 C1  MOV R0, #70 ;'F'
        B RES
C2      MOV R0, #68 ;'D'
        B RES
C3      MOV R0, #67 ;'C'
        B RES
```

Address	Contents
TBL + 0	0xA00
TBL + 4	0xA08
TBL + 8	0xA10
:	

Jump Table

Jump Table Example (Default)

```
switch(x)
{
case 0:
 G='F';
 break;

case 1:
 G='D';
 break;

case 2:
 G='C';
 break;

case 3:
 G='B';
Default:
G='X';
}
```

Cascaded if-else

```
        LDR R1, [R2]
        CMP R1, #0
        BEQ C1
        CMP R1, #1
        BEQ C2
        CMP R1, #2
        BEQ C3
        CMP R1,#3
        BEQ C4
        MOV R0, #88 ;'X'
RES     STR R0,[R2,#4]
        .
        .
C1      MOV R0, #70 ;'F'
        B RES
C2      MOV R0, #68 ;'D'
        B RES
C3      MOV R0, #67 ;'C'
        B RES
C4      MOV R0, #66; 'B'
        B RES
```

Jump Table

```
        LDR R1, [R2]
        MOV R0, #88; 'X'
        CMP R1, #3
        BGT RES
        ADR R3, TBL
        LDR PC, [R3,R1, LSL #2]
0xA00 RES  STR R0, [R2,#4]
        .
C1      MOV R0, #70 ;'F'
        B RES
C2      MOV R0, #68 ;'D'
        B RES
C3      MOV R0, #67 ;'C'
        B RES
```

Address	Contents
TBL + 0	0xA00
TBL + 4	0xA08
TBL + 8	0xA10
:	

Jump Table

Example

- Create a subroutine that estimates the grading statistics of grades
  - Input: array of grades: ‘A’,’B’,’C’,’D’, passes in characters
  - The array terminates with a grade of 0
- Calculate statistics
  - Number of top grading marks
  - Number of B-grading marks
  - Other passing grades
- Pass address of grade array in stack, return three parameters also in stack

Example

- Create a subroutine
  grades
  - Input: array of grade
  - The array terminates
- Calculate statistics
  - Number of top gradi
  - Number of B-gradin
  - Other passing grade

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                         *top++;
                         break;
                   case ('B'):
                         *Bgrading++;
                         break;
                   case ('C'):
                         *other++;
                         break;
                   case ('D'):
                         *top++;
                         break;
                   default:
                        done=1;
        }while (done==0);
}
```

```
Cal_grades      STMFD SP!,{R4-R9}
                ADR   R9 ,Base
                MOV   R4 ,#0
                MOV   R5 ,#0
                MOV   R6 ,#0




                LDMFD SP!,{R4-R9}
                MOV   PC, LR
```

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                         *top++;
                         break;
                   case ('B'):
                         *Bgrading++;
                         break;
                   case ('C'):
                         *other++;
                         break;
                   case ('D'):
                         *top++;
                         break;
                   default:
                        done=1;
        }while (done==0);
}
```

```
Cal_grades      STMFD SP!,{R4-R9}
                ADR   R9 ,Base
                MOV   R4 ,#0
                MOV   R5 ,#0
                MOV   R6 ,#0
                LDR   R8,[SP,#36]
Loop            LDRB  R7,[R8],#1
                CMP   R7,#0
                BEQ   Done


Done            LDR   R8,[SP,#32]
                STR   R4,[R8]
                LDR   R8,[SP,#28]
                STR   R5,[R8]
                LDR   R8,[SP,#24]
                STR   R6,[R8]
                LDMFD SP!,{R4-R9}
                MOV   PC, LR
```

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                         *top++;
                         break;
                   case ('B'):
                         *Bgrading++;
                         break;
                   case ('C'):
                         *other++;
                         break;
                   case ('D'):
                         *top++;
                         break;
                   default:
                              done=1;
        }while (done==0);
}
```

```
Cal_grades      STMFD SP!,{R4-R9}
                ADR   R9 ,Base
                MOV   R4 ,#0
                MOV   R5 ,#0
                MOV   R6 ,#0
                LDR   R8,[SP,#36]
Loop            LDRB  R7,[R8],#1
                CMP   R7,#0
                BEQ   Done
                SUB   R7, R7, #65
                LDR   PC,[R9,R7,LSL #2]
Return          B     Loop
Done            LDR   R8,[SP,#32]
                STR   R4,[R8]
                LDR   R8,[SP,#28]
                STR   R5,[R8]
                LDR   R8,[SP,#24]
                STR   R6,[R8]
                LDMFD SP!,{R4-R9}
                MOV   PC, LR

Cond1           ADD   R4,R4,#1
                B     Return
Cond2           ADD   R5,R5,#1
                B     Return
Cond3           ADD   R6,R6,#1
                B     Return
Cond4           ADD   R6,R6,#1
                B     Return
```

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                          *top++;
                          break;
                   case ('B'):
                          *Bgrading++;
                          break;
                   case ('C'):
                          *other++;
                          break;
                   case ('D'):
                          *top++;
                          break;
                   default:
                                done=1;
        }while (done==0);
}
```

```
Cal_grades      STMFD SP!,{R4-R9}
                ADR   R9 ,Base
                MOV   R4 ,#0
                MOV   R5 ,#0
                MOV   R6 ,#0
                LDR   R8,[SP,#36]
Loop            LDRB  R7,[R8],#1
                CMP   R7,#0
                BEQ   Done
                SUB   R7, R7, #65
                LDR   PC,[R9,R7,LSL #2]
Return          B     Loop
Done            LDR   R8,[SP,#32]
                STR   R4,[R8]
                LDR   R8,[SP,#28]
                STR   R5,[R8]
                LDR   R8,[SP,#24]
                STR   R6,[R8]
                LDMFD SP!,{R4-R9}
                MOV   PC, LR

Cond1           ADD   R4,R4,#1
                B     Return
Cond2           ADD   R5,R5,#1
                B     Return
Cond3           ADD   R6,R6,#1
                B     Return
Cond4           ADD   R6,R6,#1
                B     Return
```

```
Cal_grades (char* arr, int* top, int* Bgrading, int *other)
{
        *top=0;
        *Bgrading=0;
        *other=0;
        int done =0;
        int index=0;
        do{
                char val=arr[index++];
                switch (val):
                {
                   case ('A'):
                         *top++;
                         break;
                   case ('B'):
                         *Bgrading++;
                         break;
                   case ('C'):
                         *other++;
                         break;
                   case ('D'):
                         *top++;
                         break;
                   default:
                        done=1;
        }while (done==0);
}
```

Switch – Random & Wide Values

- If cases are random wide range values, a **fork algorithm** is used to speed up the average search time and avoid testing every case (e.g. when **x** = 1000).
- Due to the wide value spread, the **jump table size** will be **too large**. A cascade of if-else-if comparisons is more efficient.

```
switch(x)
{
case 1:
  {S0};
 break;

case 10:
  {S1};
 break;

case 100:
 {S2};
 break;

case 1000:
 {S3};
 break;
}
```

```
if(x == 1)
  {S0};
else if(x == 10)
  {S1};
else if(x == 100)
  {S2};
else if(x == 1000)
  {S3};
```

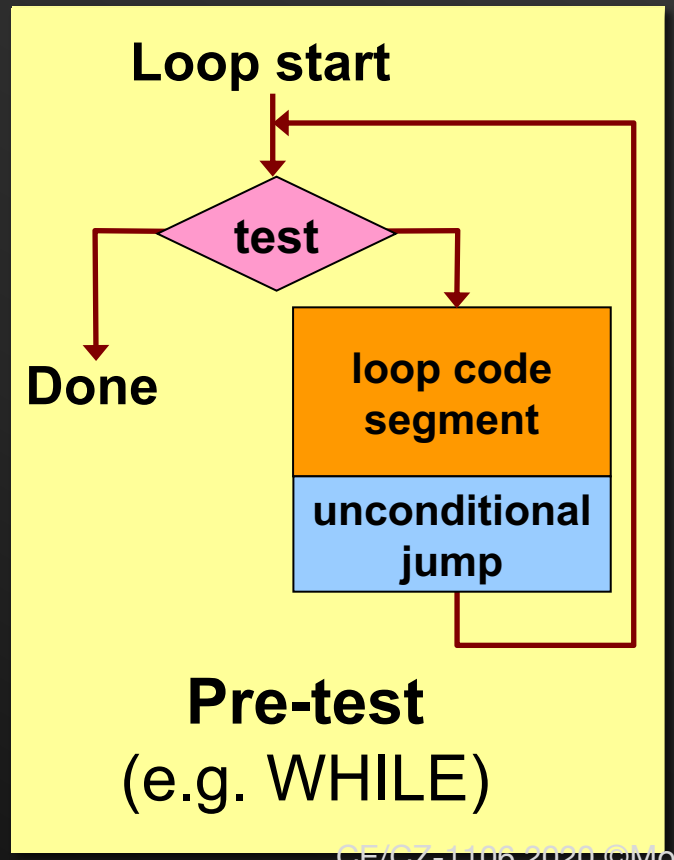
standard if-else-if implementation

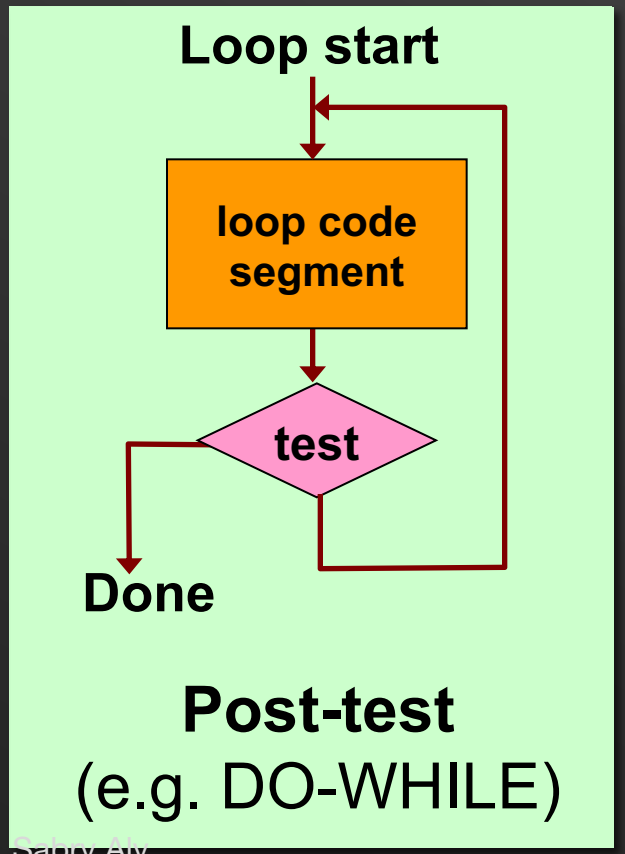
```
if(x <= 10) {
    if(x == 1)
      {S0};
    else if(x == 10)
      {S1};
}
else {
    if(x == 100)
      {S2};
    else if(x == 1000)
      {S3};
}
```

forked if-else-if implementation

Loops

- Loop constructs are distinguished by the position of their conditional test.
- Pre-test loop may **never execute** its loop code segment.
- Post-test loop executes the loop segment **at least once**.





WHILE Implementation

- Implementation of the **WHILE** loop constructs:
- This is an example of a **pre-test** loop.
- If the condition **(VarX > 0)** is false, the loop segment is not executed at all.

```
WHILE (VarX > 0)
{
    Loop segment
}
```

```
Back   CMP "VarX",#0
       BLE Exit
        :
        :   } Loop segment
        :
       B Back
Exit    :
```

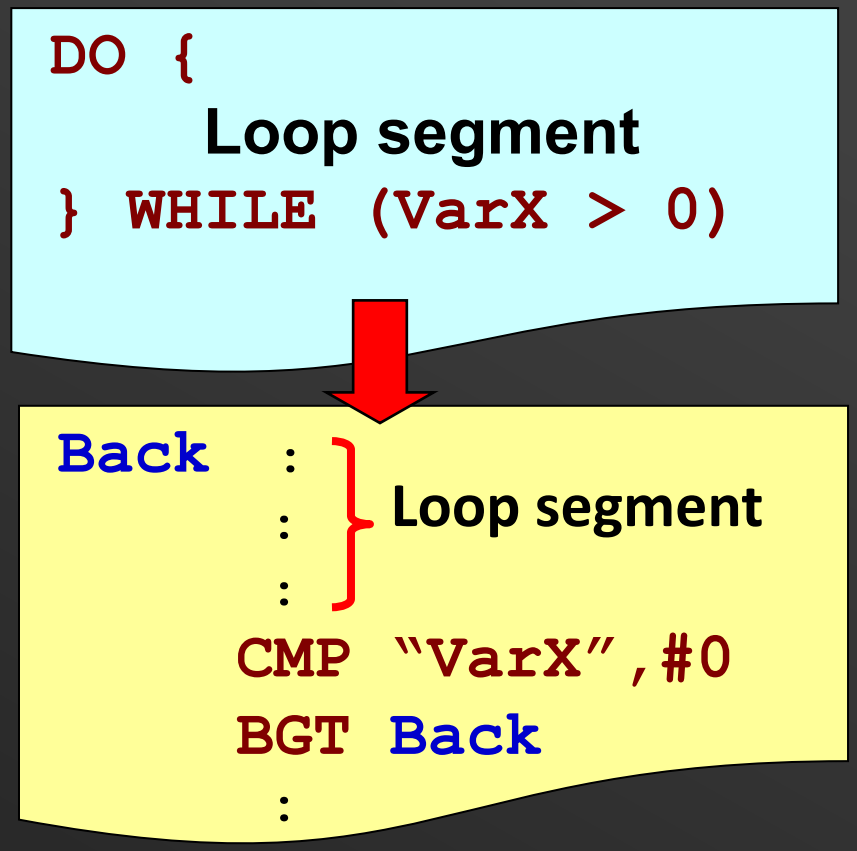
Implementation of WHILE in ARM assembly language

Note: **VarX** is variable. you will need to load it to a register.

DO-WHILE Implementation

- Implementation of the **DO-WHILE** loop constructs
- This is an example of a **post-test** loop.
- The loop segment is executed at least once before condition is tested.
- Post-test loop construct is **more efficient** than the pre-test as there is no need for an additional unconditional jump.

Note: `VarX` is variable. you will need to load it to a register.



Implementation of DO- WHILE in ARM assembly language

FOR Implementation

- Implementation of **FOR** loop constructs:

- The FOR loop is a **pre-test** loop that evaluates the condition first before executing loop segment.

- If loop segment is executed and count **N** is not used in loop segment, some optimizing compilers implement the **FOR** loop using a **post-test** with **decrement** & **test for zero**.

```
FOR (N=0; N<5; N++)
{
    Loop segment (x5)
}
```

```
        MOV R0,#0
Back    CMP R0,#5
        BGE Exit
         :
         :   } Loop segment
        ADD R0,R0,#1
        B   Back
Exit     :
```

Implementation of FOR in ARM assembly language

Example: Summation 1 to N

```
Sum =0;
for(i=0;i<=N;i++)
     Sum=Sum+i;
```

Pre-test

```
        MOV     R2,#0
        MOV     R0,#0
Loop    CMP     R2,#N
        BGT     END
        ADD     R0,R0,R2
        ADD     R2,R2,#1
        B       Loop
END     STR     R0,[“sum”]
```

Post-test

```
        MOV     R2, #N
        MOV     R0,#0
Loop    ADD     R0,R0,R2
        SUBS    R2,R2,#1
        BNE     Loop
        STR     R0,[“sum”]
```

Example: Summation 1 to N

```
Sum =0;
for(i=0;i<=N;i++)
     Sum=Sum+i;
```

Pre-test

```
        MOV     R2,#0
        MOV     R0,#0
Loop    CMP     R2,#N
        BGT     END
        ADD     R0,R0,R2
        ADD     R2,R2,#1
        B       Loop
END     STR     R0,[“sum”]
```

Post-test

```
        MOV     R2, 0
        MOV     R0,#0
Loop    ADD     R0,R0,R2
        ADD     R2,R2,#1
        CMP     R2,#5
        BLE     Loop
        STR     R0,[“sum”]
```

Summary

- **SWITCH** constructs can be implemented efficiently depending nature of the case values.
  - Narrow consecutive values can benefit from a jump table.
  - Forked if-else-if can be used with wide ranged values.
- **Post-test** loops are **more efficient** that pre-test loops for the same loop segment.
- With optimised compilers, the low-level code produced may not tally directly with the high-level operations. (e.g. loop increments may be implemented as decrements for better code efficient).

Chapter 8: Instruction Encoding and ISA

Mohamed M. Sabry Aly

N4-02c-92

Learning Objectives

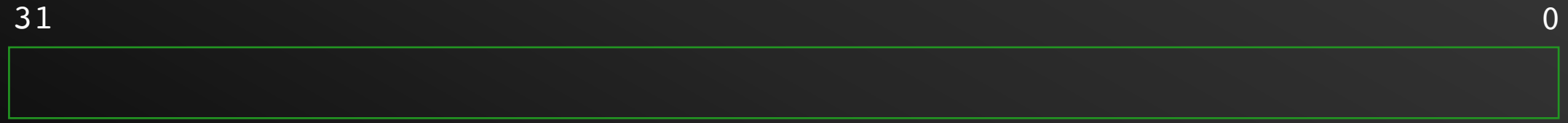
- Describe the instruction format of ARM instruction-set architecture (ISA)

- Describe differences between fixed and variable-length ISA

Instruction Encoding

- For CPU to identify each unique assembly instruction, they must be encoded with unique binary patterns.
  - ARM 32-bit machine encode instructions in a 32-bit word

- What is included?
  - Instruction type ➔ encoded
  - Operand(s) ➔ encoded
  - Condition for conditional execution ➔ encoded
  - Other flags?

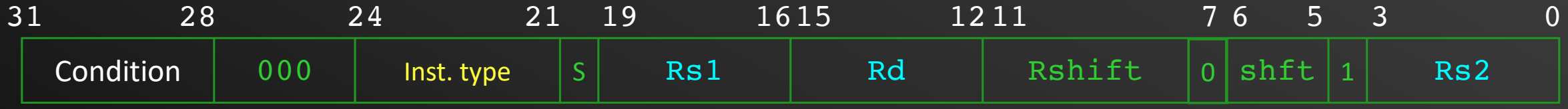


Overall Instruction Format

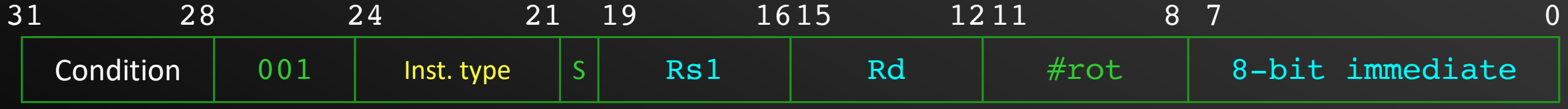
- Arithmetic & Logic instructions
  - ADD, ADC, SUB, SBC, RSB, AND, EOR, RSC, ORR, BIC

XXXCCS Rd, Rs1, Rs2 , {shft #}

XXXCCS Rd, Rs1, Rs2 , {shft Rshift}



XXXCCS Rd, Rs1, #immediate (rotated)

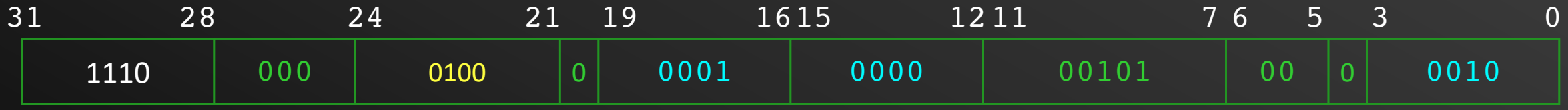


Overall Instruction Format

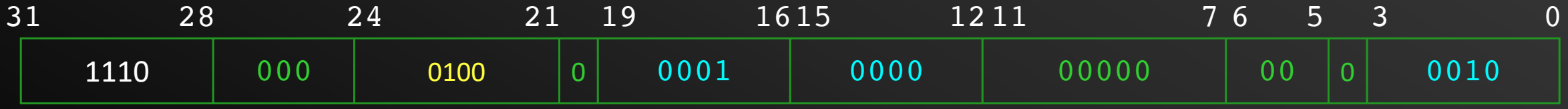
- Arithmetic & Logic instructions
  - ADD, ADC, SUB, SBC, RSB, AND, EOR, RSC, ORR, BIC

XXXCCS Rd, Rs1, Rs2 , {shft #}

ADD R0,R1,R2,LSL #5



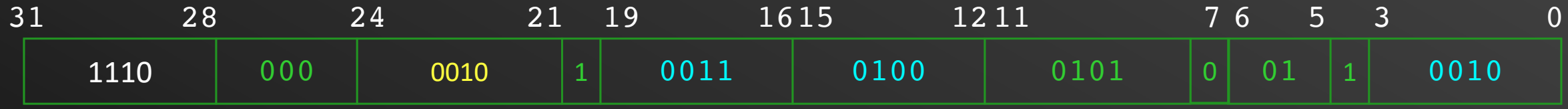
ADD R0,R1,R2



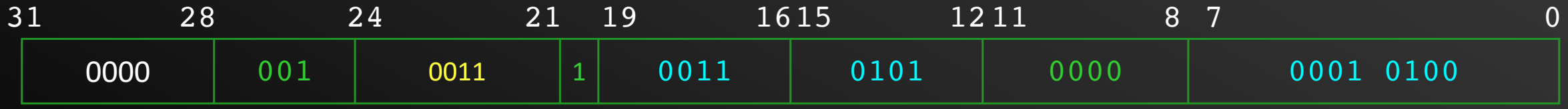
Overall Instruction Format

- Arithmetic & Logic instructions
  - `ADD, ADC, SUB, SBC, RSB, AND, EOR, RSC, ORR, BIC`

`SUBS R4, R3, R2, LSR R5`



`RSBEQS R5,R3,#20`

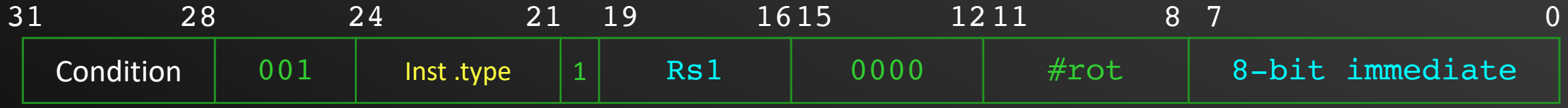


Overall instruction format

- Comparison instructions
  - `TST, TEQ, CMP, CMN`

`XXXCC Rs1, Rs2 , {shft #}`

`XXXCC Rs1, #immediate (rotated)`

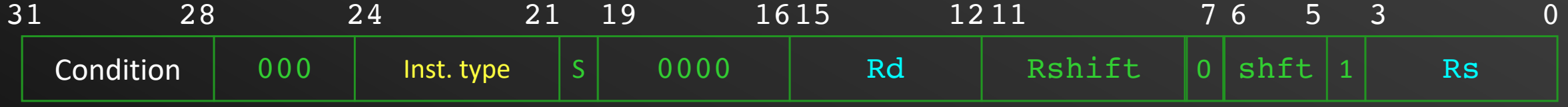


Overall instruction format

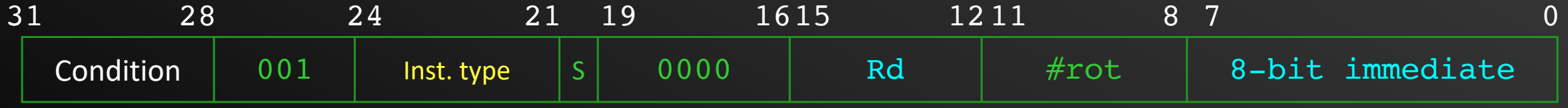
- MOV instructions
  - MOV, MOVN

XXXCCS Rd, Rs , {shft #}

XXXCCS Rd, Rs , {shft #}



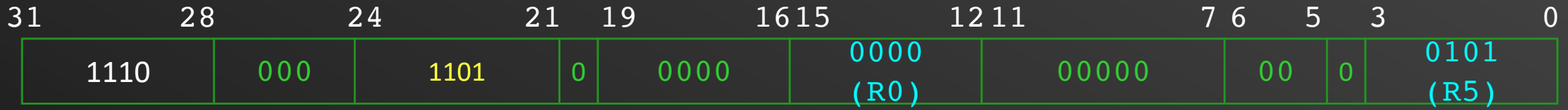
XXXCC Rd, #immediate (rotated)



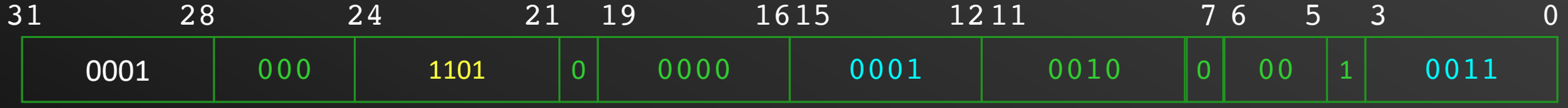
Overall instruction format

- MOV instructions
  - MOV, MOVN

MOV R0,R5



MOVNE R1, R3, LSL R2



MOV R1,#0x40000000

Inst. Type field Field

- 4 bits for instructions ➔ up to 16 combinations
- 4 bits for source and destination registers ➔ 16 registers

Code	Instruction
0000	AND
0001	EOR
0010	SUB
0011	RSB
0100	ADD
0101	ADC
0110	SBC
0111	RSC

Code	Instruction
1000	TST
1001	TEQ
1010	CMP
1011	CMN
1100	ORR
1101	MOV
1110	BIC
1111	MVN

Condition Field

- 4 bits for condition → up to 16 difference combinations

Code	Condition	Flags	Meaning
0000	EQ	Z = 1	Equal
0001	NE	Z = 0	Not equal
0010	CS or HS	C = 1	Higher or same, unsigned
0011	CC or LO	C = 0	Lower, unsigned
0100	MI	N = 1	Negative
0101	PL	N = 0	Positive or zero
0110	VS	V = 1	Overflow
0111	VC	V = 0	No overflow
1000	HI	C = 1 and Z = 0	Higher, unsigned
1001	LS	C = 0 or Z = 1	Lower 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	NV		Reserved (unused)

What about shift instructions, what's there instruction format?

LSL, LSR, ASR, ROR, and RRX has no dedicated instructions

Shift/Rotate instructions

- Instructions are encoded as MOV instructions

  E.g. LSL R1, R2, #5 ≡ MOV R1, R2, LSL #5

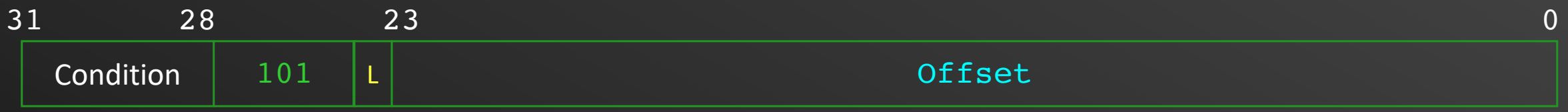
- Specify the shift type in bits 5 and 6

Code	Shift type
00	LSL
01	LSR
10	ASR
11	ROR/RRX

Branch Instructions

- For conditional branch and branch with link

```
B(L)CC   Offset
```



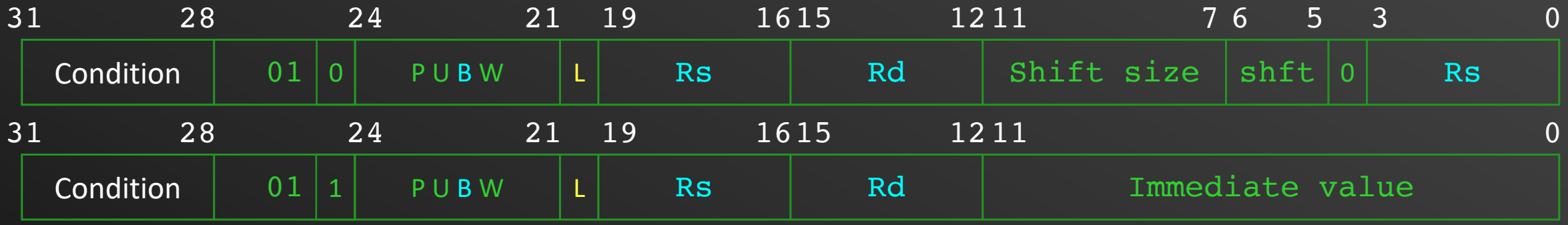
- 26 bits offset shifted to the right by 2 (branching to word addresses) ➔ calculated by the assembler

- Branch range: ±32 MBytes

Load/Store Instructions

- Load/Store word bye

```
LDR/STR{B}CC   Rs,[RD,offset pre or post]W
```

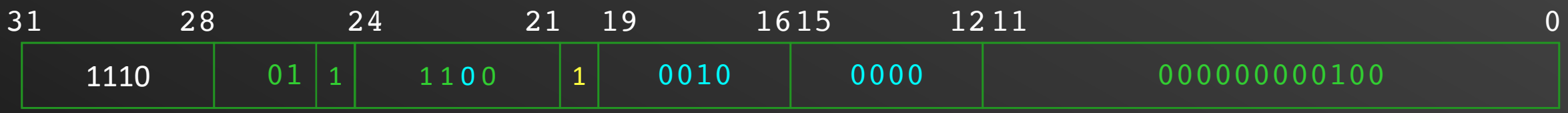


- L bit determines load (1) or store (0)
- P bit determines indexing pre (1) or post (0)
- U determines offset direction add (1) or subtract (0)offset
- B determines byte (1) or word (0) access
- W is for write back (!) of the offset

Load/Store Instructions

- Load/Store word bye

```
LDR   R0,[R2,#4]
```

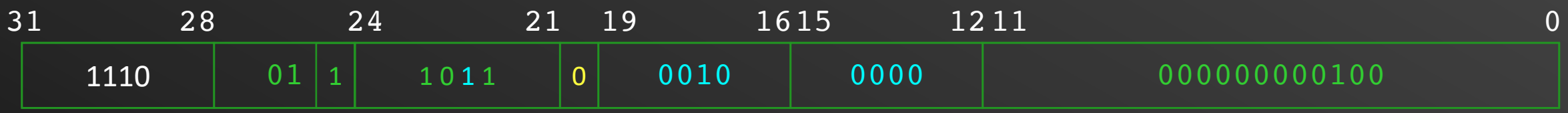


- L bit determines load (1) or store (0)
- P bit determines indexing pre (1) or post (0)
- U determines offset direction add (1) or subtract (0)offset
- B determines byte (1) or word (0) access
- W is for write back (!) of the offset

Load/Store Instructions

- Load/Store word bye

```
STRB R0,[R2,#-4]!
```

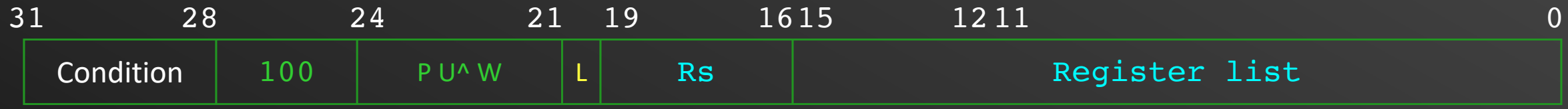


- L bit determines load (1) or store (0)
- P bit determines indexing pre (1) or post (0)
- U determines offset direction add (1) or subtract (0)offset
- B determines byte (1) or word (0) access
- W is for write back (!) of the offset

Load/Store Instructions

- Load/Store Multiple instructions

```
LDM/STMFXCC   RsW,{register list}
```

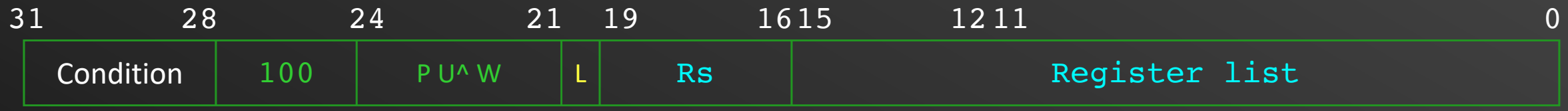


- L bit determines load (1) or store (0)
- P bit determines indexing before (1) or after (0)
- U determines offset direction add (1) or subtract (0)
- ^ is “don’t care”
- W is for write back (!) after operation

Load/Store Instructions

- Load/Store Multiple instructions

```
LDM/STMFXCC   RsW,{register list}
```



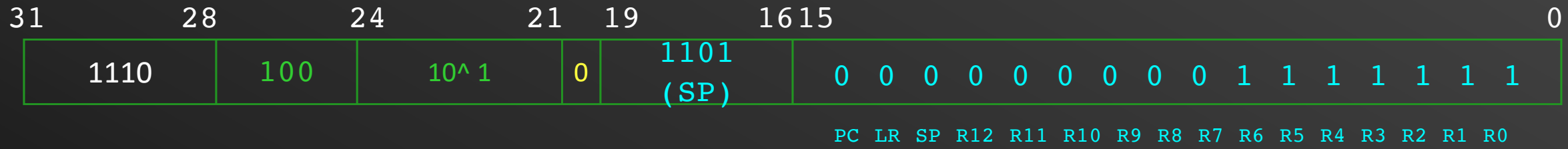
- L bit determines load (1) or store (0)
- P bit determines indexing before (1) or after ((
- U determines offset direction add (1) or subtr
- ^ is “don’t care”
- W is for write back (!) after operation

PU	Instruction
10	STMFD
01	LDMFD
00	STMFA
11	LDMFA

Load/Store Instructions

- Load/Store Multiple instructions

```
STMFD SP!,{R0-R6}
```



- L bit determines load (1) or store (0)
- P bit determines indexing before (1) or after (0)
- U determines offset direction add (1) or subtract (0)
- ^ is “don’t care”
- W is for write back (!) after operation

Fixed vs. Variable-Length instruction

- ARM is fixed-length
  - Operands had to be registers or immediate values

- Variable length instructions
  - Intel 80x86: Instructions vary from 1 to 17 bytes long.
  - Digital VAX: Instructions vary from 1 to 54 bytes long.
  - Require multi-step fetch and decode.
  - Allow for a more flexible (but complex) and compact instruction set.

Variable-Length: Intel Instruction Format

- Complex instructions = complex hardware
- Higher variety for smaller code and faster execution

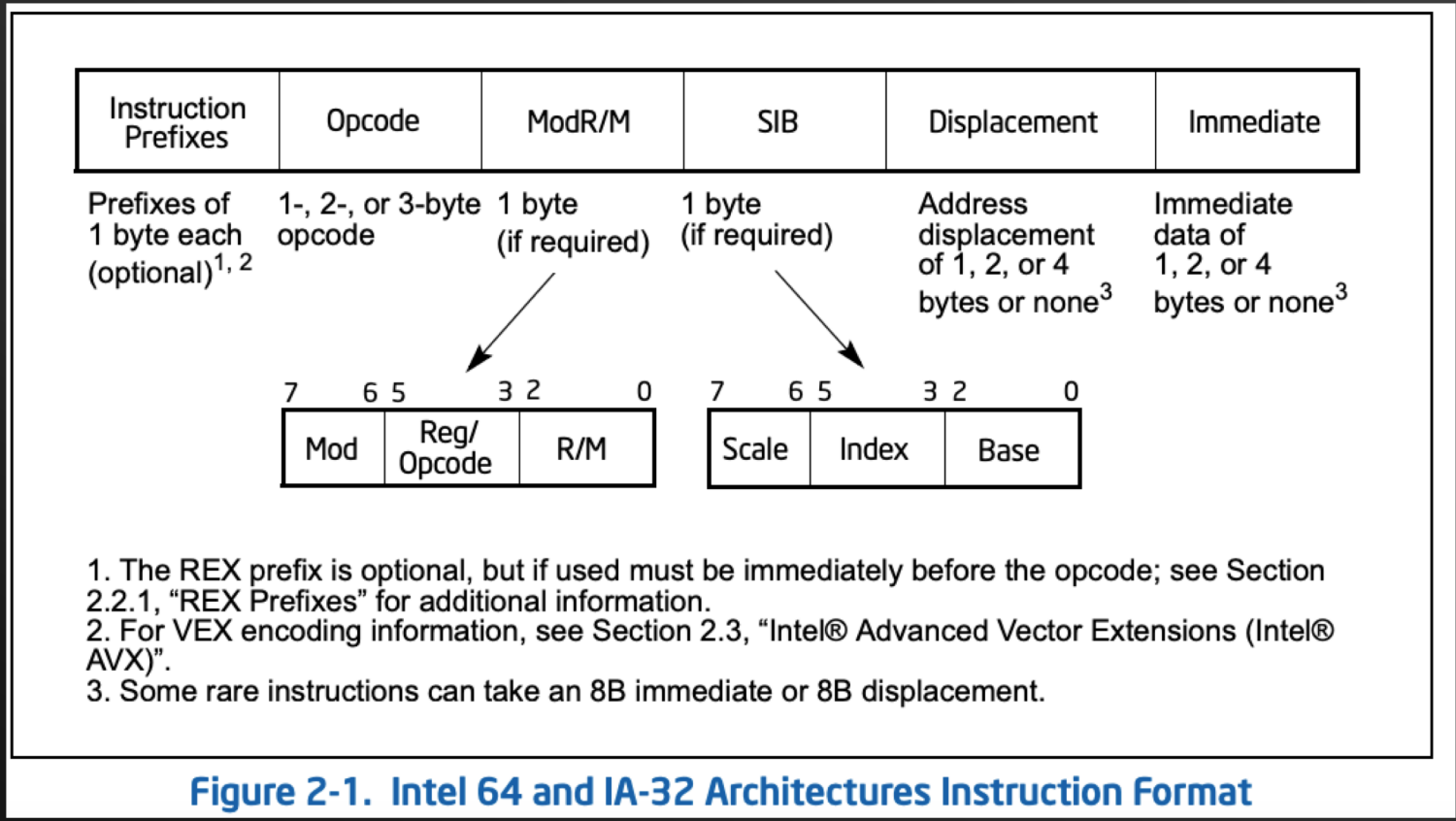
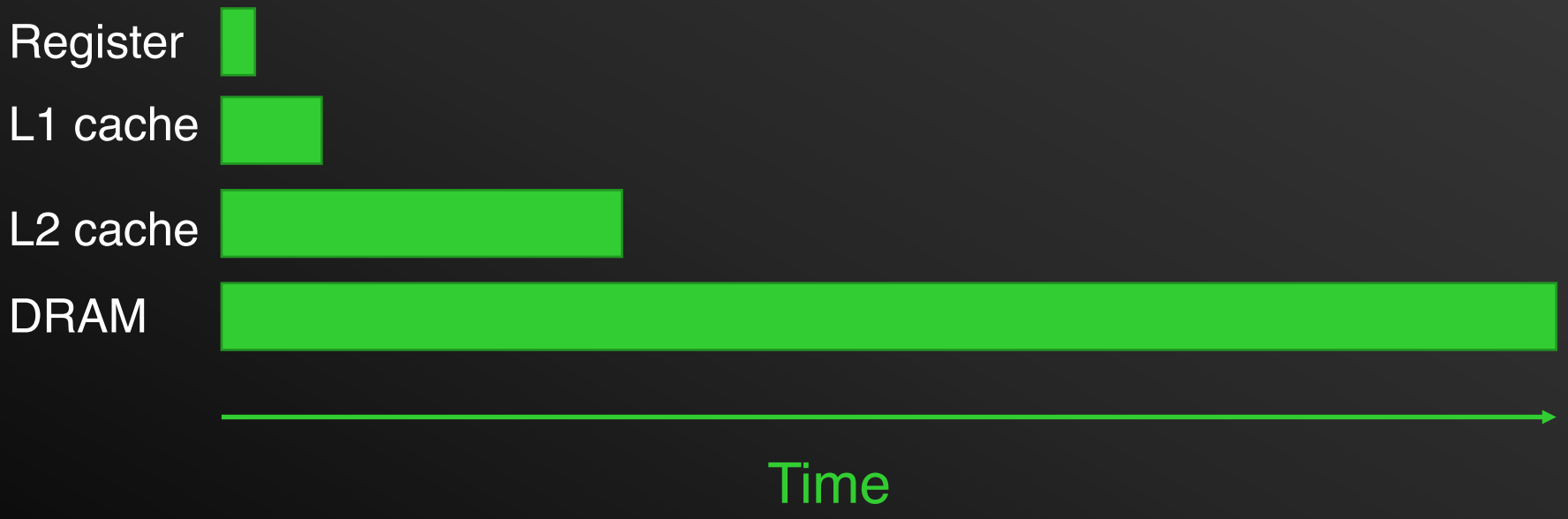


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Using Registers

- Operations using registers are **faster** than those involving memory.
  - Data transfer between registers and ALU is faster due to it physically proximity.
  - To speed up code execution, keep as much of your computations within **registers**.



Implication of Register Count

- Implementing many registers within the CPU will incurs increasing overheads.
  - They take up precious **space** in the silicon die.
  - Connections to various busses increase the **routing** and **multiplexing complexity**.
  - More registers increases the **operand size** during instruction encoding.

ADD (3)

15	14	13	12	11	10	9	8 – 6	5 – 3	2 – 0
0	0	0	1	1	0	0	Rm	Rn	Rd

This form of ADD adds the value of one register to the value of a second register, and stores the result in a third register. The condition code flags are updated, based on the result.

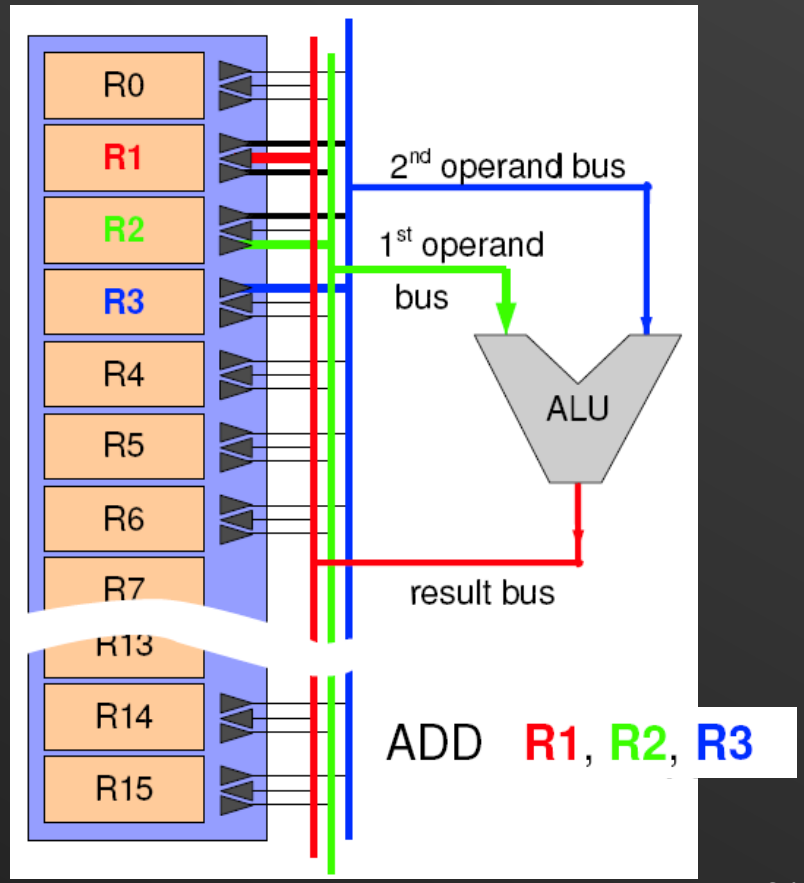
Syntax

`ADD <Rd>, <Rn>, <Rm>`

where:

`<Rd>`	Is the destination register for the completed operation.
`<Rn>`	Specifies the register containing the first value for the addition.
`<Rm>`	Specifies the register containing the second value for the addition.

Instruction Encoding for ADD in Thumb-2



Orthogonality of ISA

- In a truly orthogonal ISA, every instruction is able to use every available addressing modes.
- A truly orthogonal ISA does not restrict certain instruction from using only specific registers.
- Difficult to achieve complete orthogonality due to the **large number of bit patterns** required to express all combinations of operations and addressing modes.
- Increase instruction length will increases the memory size required by the program.

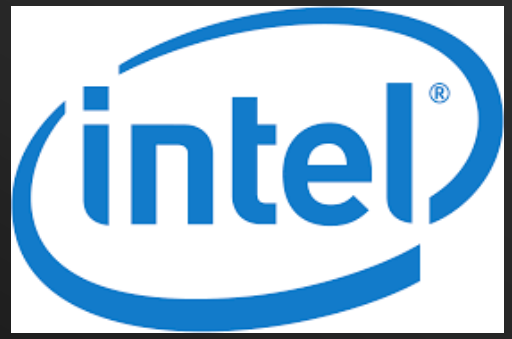
Summary - Why is Good Instruction Set Architecture (ISA) Design Important?

- Good ISA design can yield benefits such as:

  - Increases the execution performance of the processor.
  - Increases the code density (i.e. how much memory a piece of code will occupy).
  - Reduces the power consumption of the CPU.
  - Reduces manufacturing cost of processor.
  - Easy for programmers to write efficient programs.
  - Efficient mapping of high-level programming requirements into low-level instruction sets.

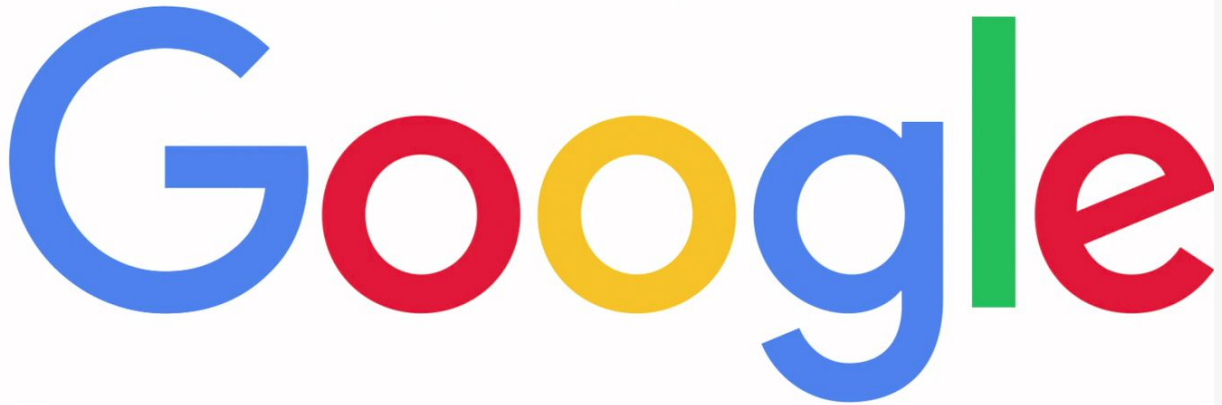
Why Should I Care About Processor Designs and ISA?

- Designing a processor (or a co-processor) is no longer a hardware company job



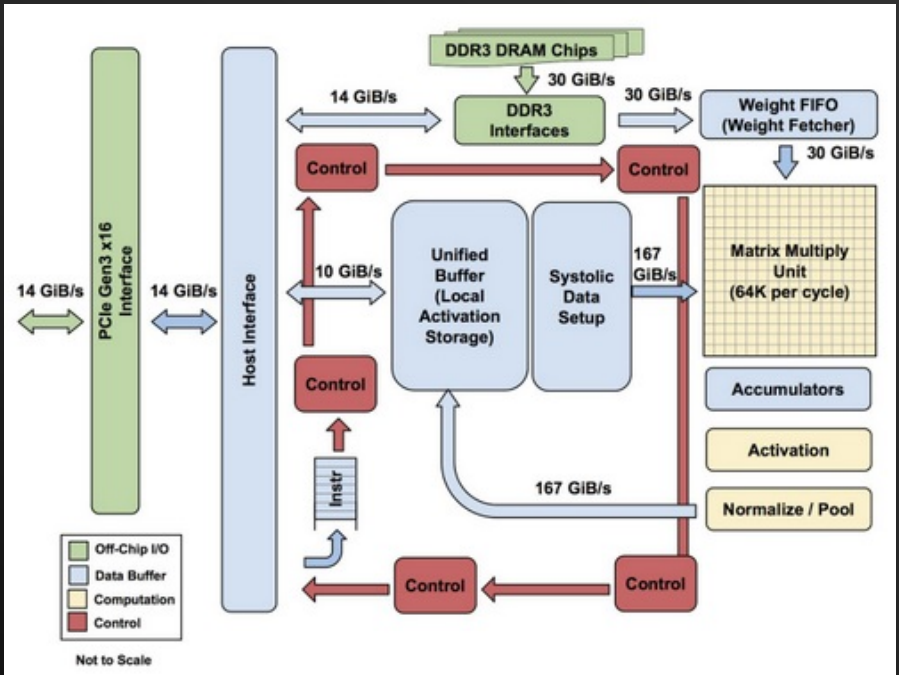
Why Should I Care About Processor Designs and ISA?

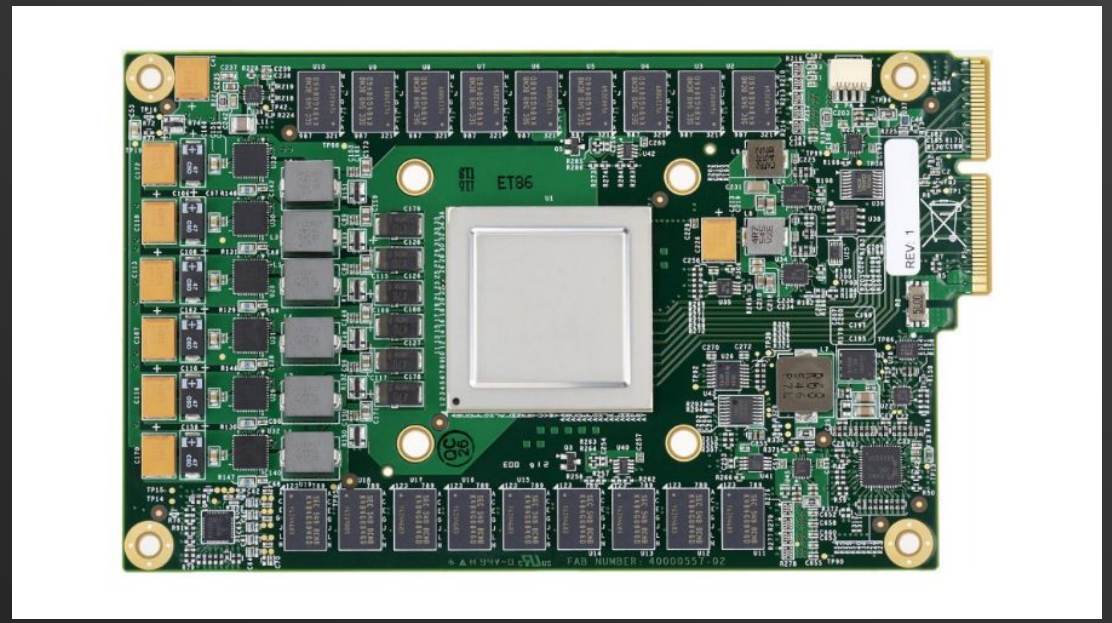
- Designing a processor (or a co-processor) is no longer a hardware company job



The Google TPU

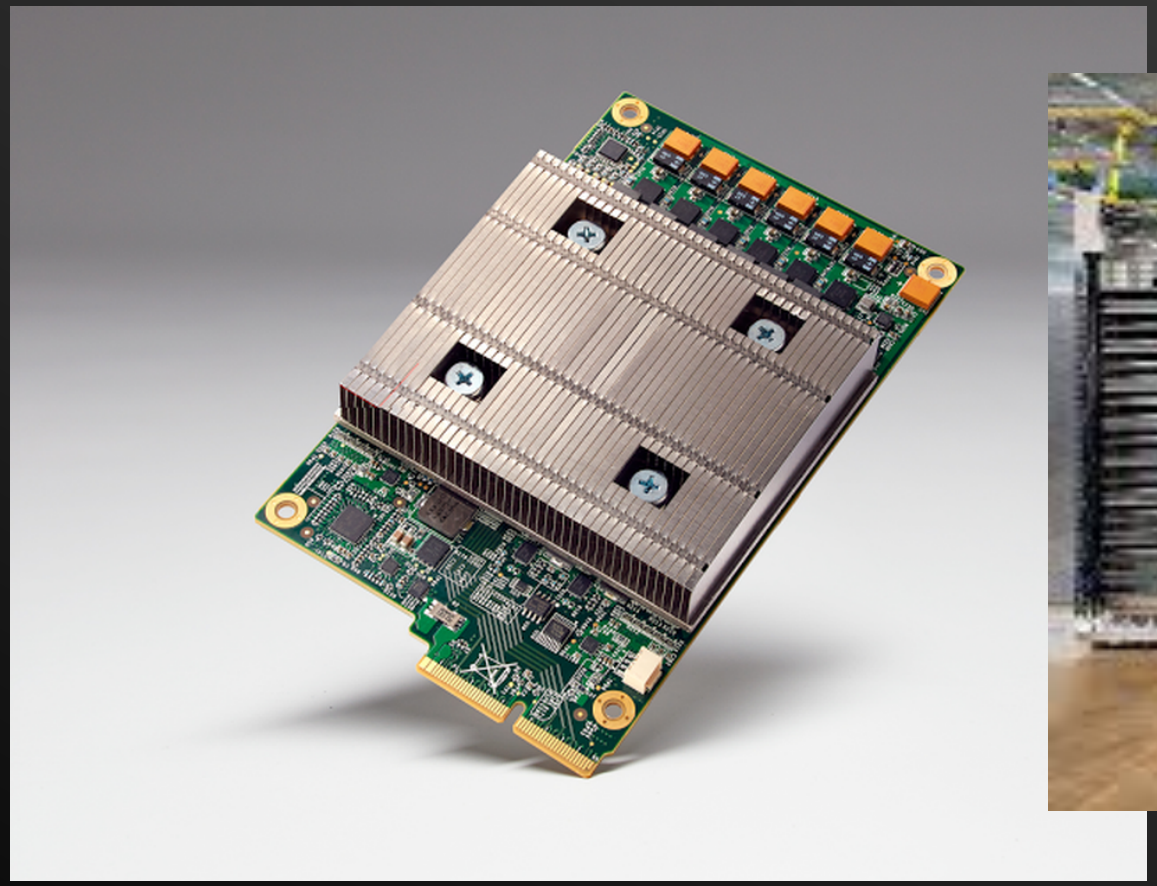
- Tensor Processing Unit
- Designed by Engineers in Google
- Tailored to make AI applications run faster





The Google TPU

- Design by Engineers in Google to make AI applications run faster



Summary

- ARM is fixed-length load/store architecture
  - Operands reside in registers
- Other architectures can use variable-length format
  - E.g, Intel architecture
- Use of registers is key towards higher performance