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

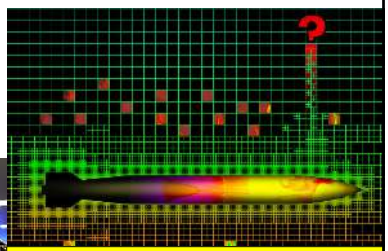
3

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.

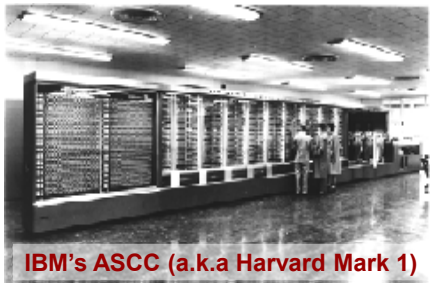
8

CZ1106
CE1106

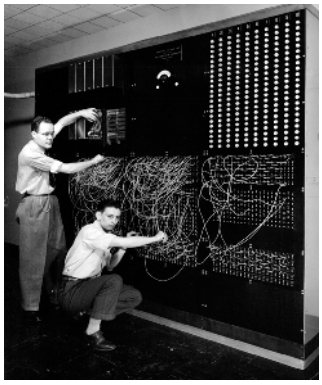
Harvard and Von Neumann

- 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

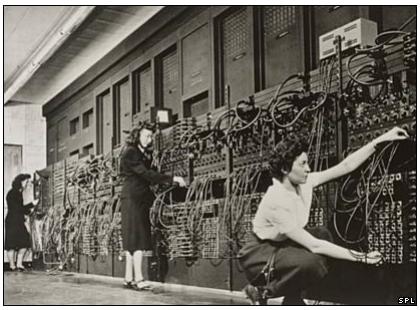
CZ1106
CE1106

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

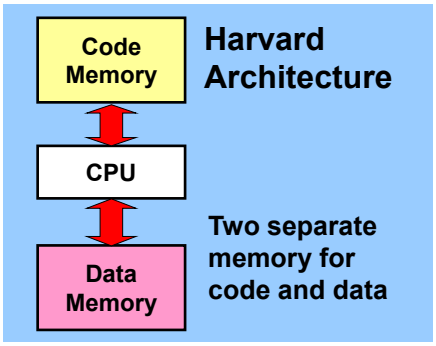


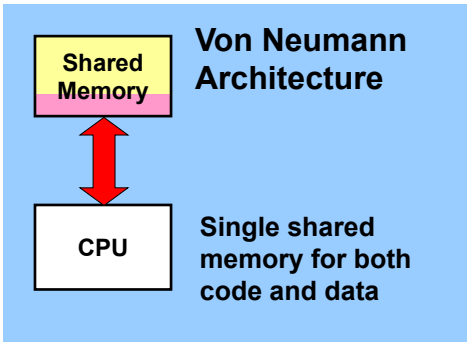
10

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.

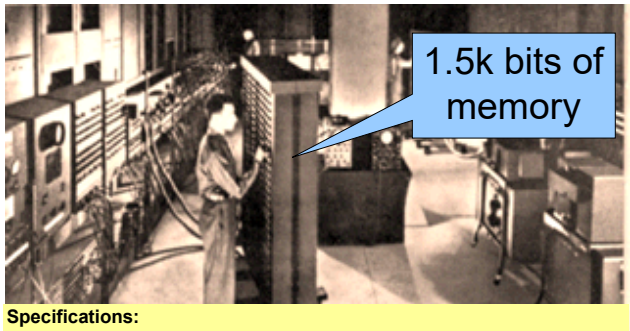




11

CZ1106
CE1106

ENIAC – the first digital computer



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

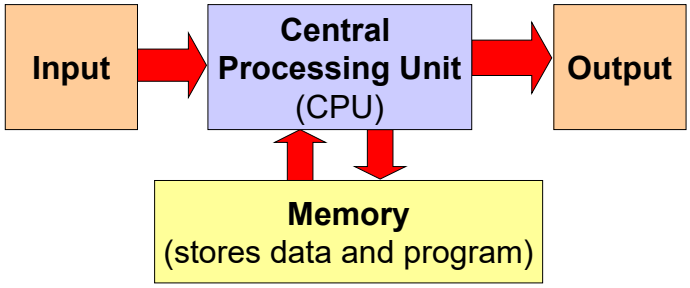
Electronic Numerical Integrator and Calculator

- 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



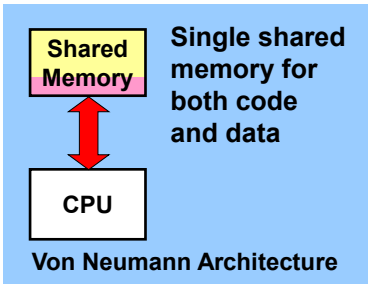
- Many modern day computers are still based on von Neumann's design, which consist of:
  - Central Processing Unit (CPU)
  - Memory
  - Input and Output

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

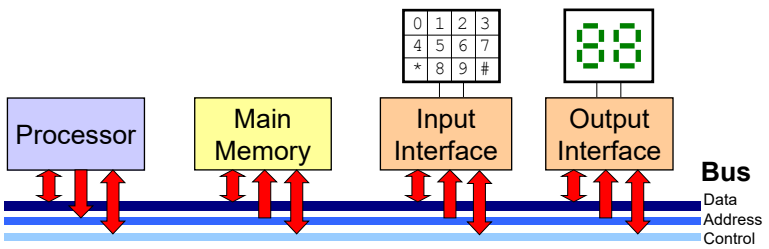
Basic Components of a Microcomputer

Learning Objectives (1.2)

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

CZ1106
CE1106

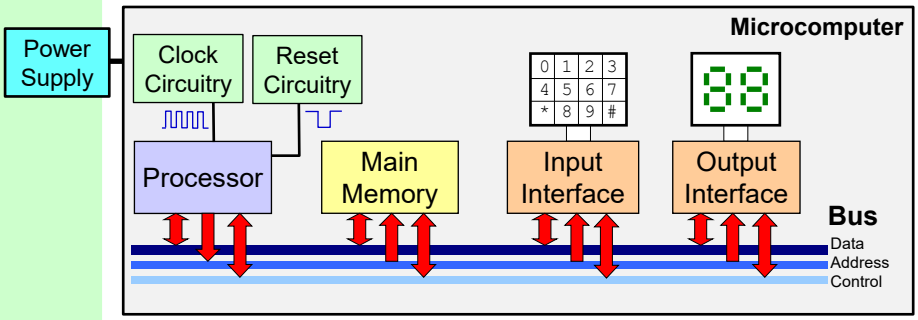
Components of a Microcomputer



- 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.

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

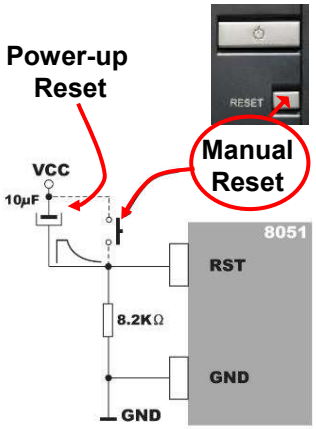
- Most computers are **synchronous** and are driven by a master or system **clock**.
  - The **speed performance** of the computer is governed by the frequency of the clock.
  - The CPU requires a fixed number of clock ticks (**cycles**) to execute each instruction.
  - Many **different clock** frequencies 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

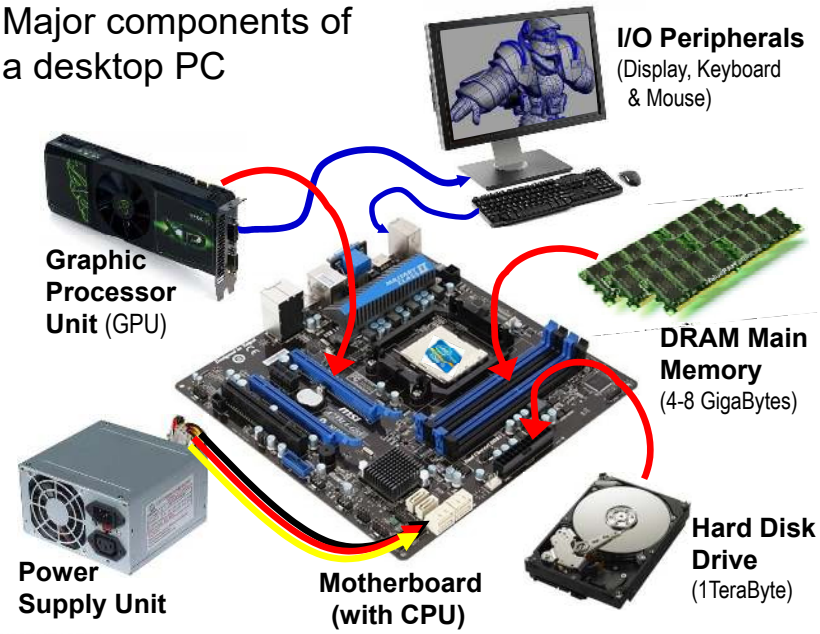
A Gamer's PC

23

CZ1106
CE1106

Inside a Desktop Personal Computer

- Major components of a desktop PC



24

CZ1106
CE1106

Inside a Tablet Computer

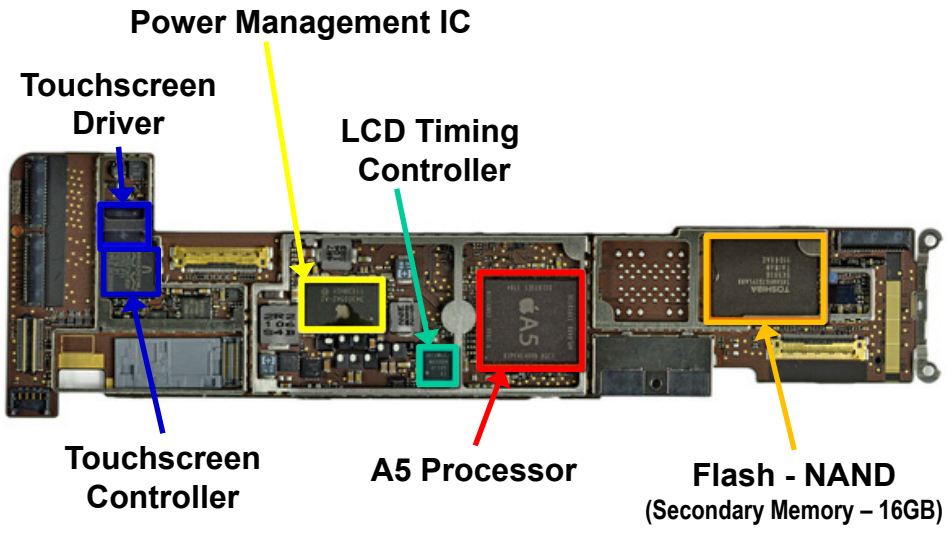
- Major components of the iPad2

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



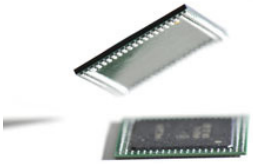
Source: http://www.appleinsider.com/articles/11/03/11/live_teardown_of_apples_ipad_2_currently_underway.html

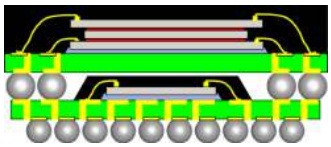
26

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

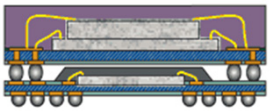
Source:..http://www.appleinsider.com/articles/11/03/15/x_ray_of_apples_a5_cpu_in_ipad_2_confirms_manufacturing_by_samsung.html

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.

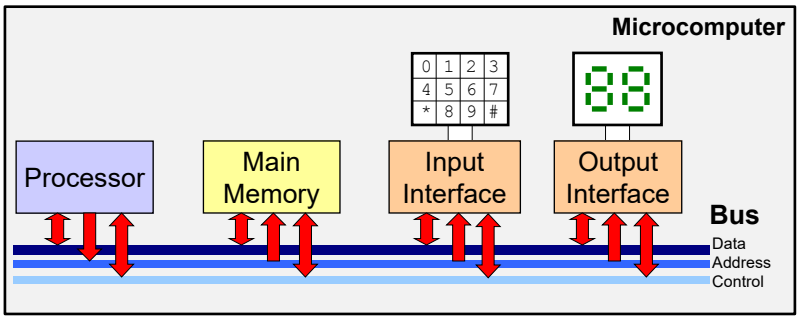
Source:http://www.appleinsider.com/articles/11/03/15/x_ray_of_apples_a5_cpu_in_ipad_2_confirms_manufacturing_by_samsung.html

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

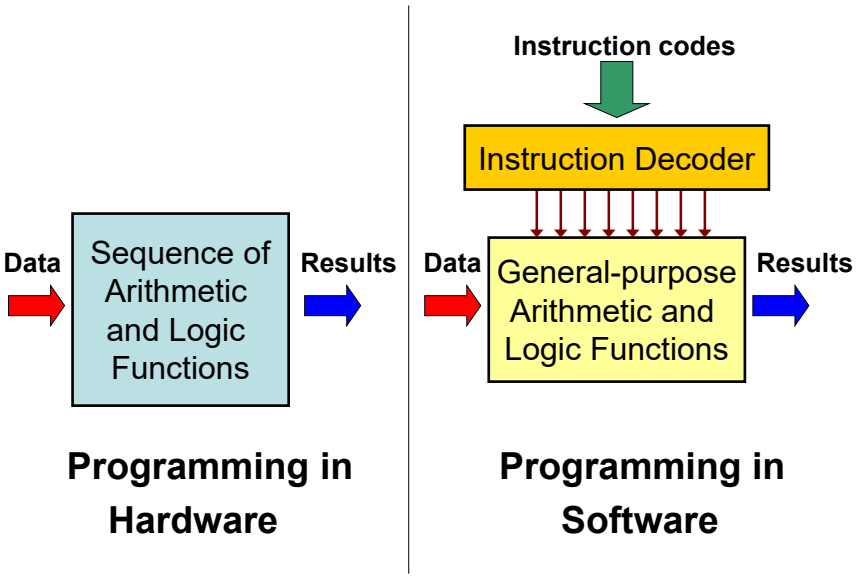
Learning Objectives (2.1)

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

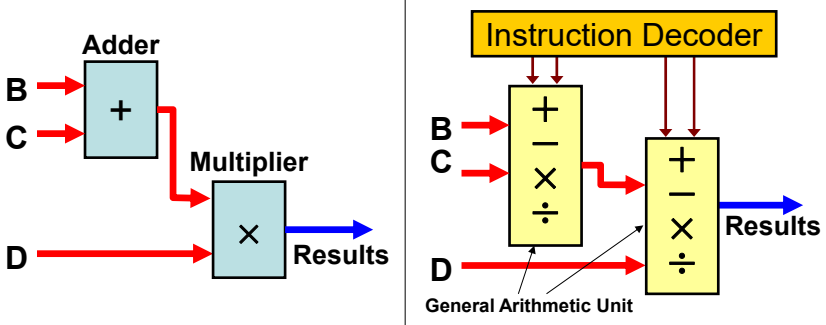
CZ1106
CE1106

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

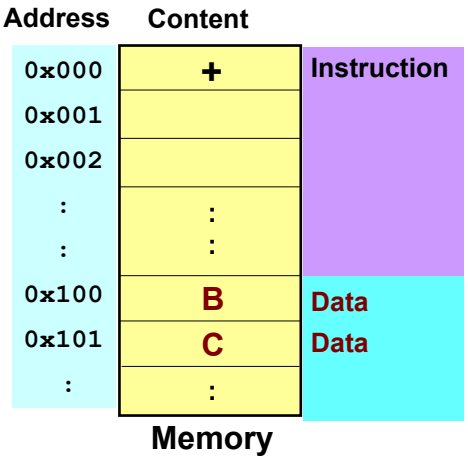
4

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.

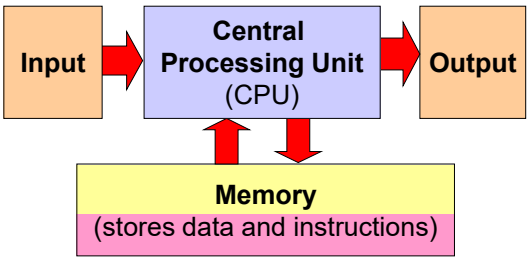
e.g. B+C



5

CZ1106 CE1106

The Stored Program Concept



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

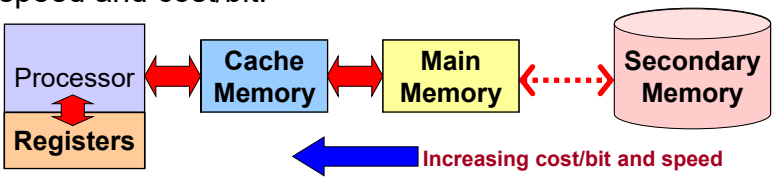
1. Both data & instructions are stored in the same memory
2. Contents of memory are addressable by location, without regard to data type
3. Execution occurs sequentially (unless explicitly modified)

6

CZ1106
CE1106

Memory Hierarchy

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

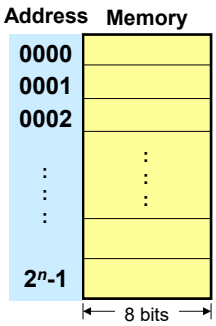


Registers	Very fast access but limited numbers within CPU. Operates at CPU clock rate (size: 2-128 registers)
Cache Memory	Fast access static RAM close to CPU. Typical access time 3-20nS (size: up to 512 kB)
Main memory	Usually dynamic RAM or ROM (for program storage). Typical access time 30-70nS. (size: 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	-(2^32) -> (2^32)-1	
float	4	32		
double	8	64		
long double	12	96		

ANSI C numeric data types and their respective size and range

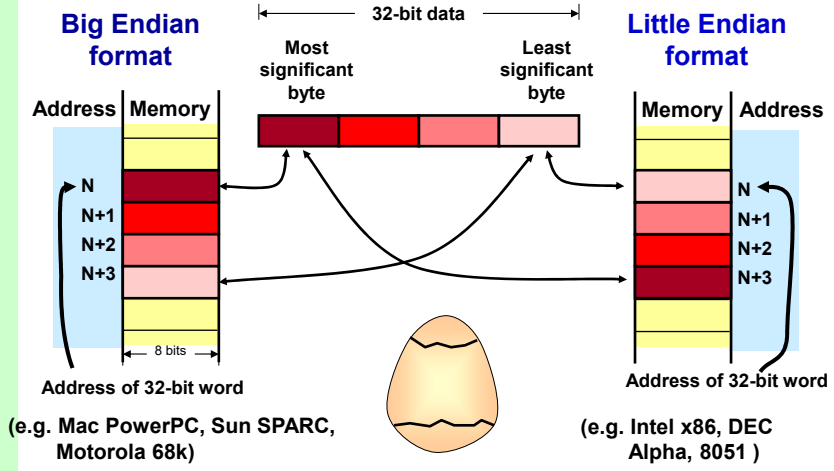
Note: These sizes may change depending of the processor and compiler

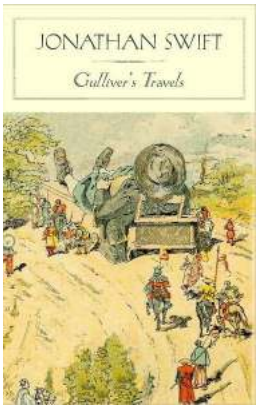
15

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

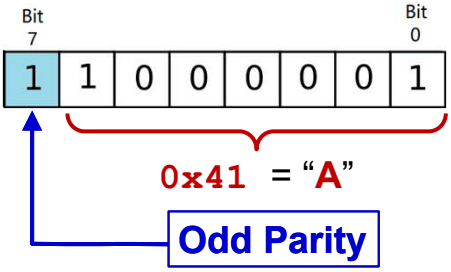
CZ1106 CE1106

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)

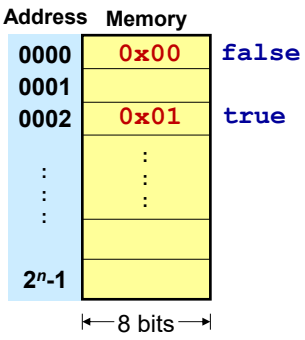
Learn More: Google "Unicode" or "Unicode characters"

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.

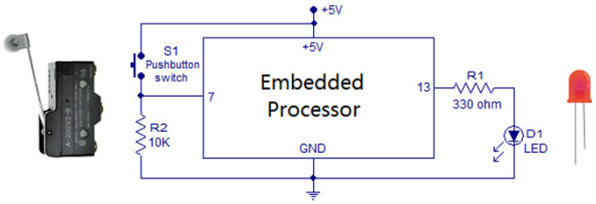


24

CZ1106
CE1106

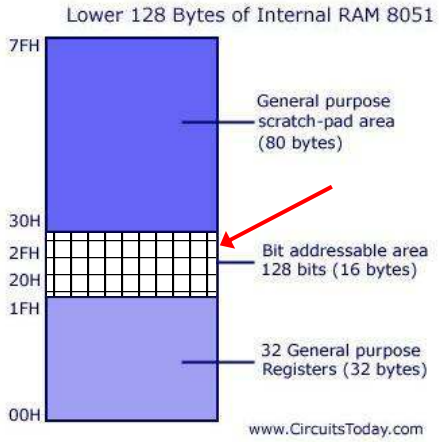
8051 - Bit Addressable Memory

- 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

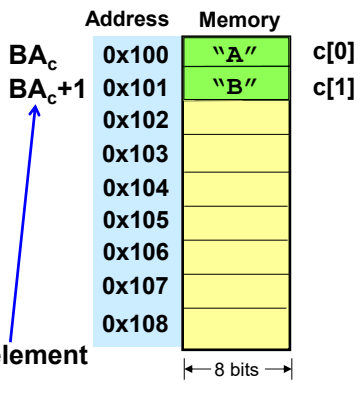
CZ1106
CE1106

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`

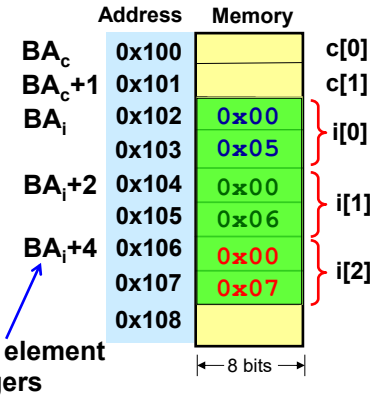
28

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

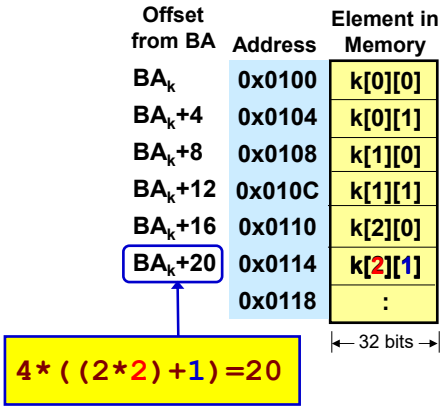
Nested Array

- 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

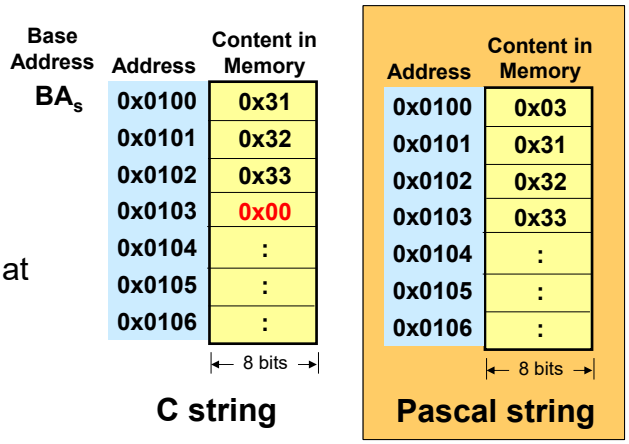
String Representation

- 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"

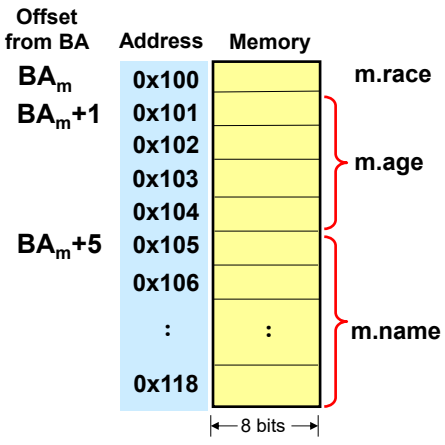
31

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.

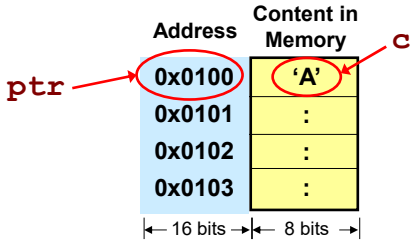
35

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).

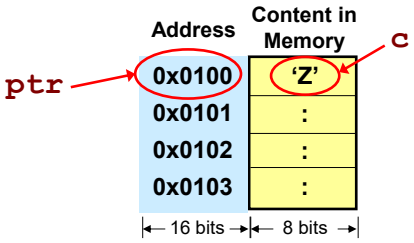
36

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

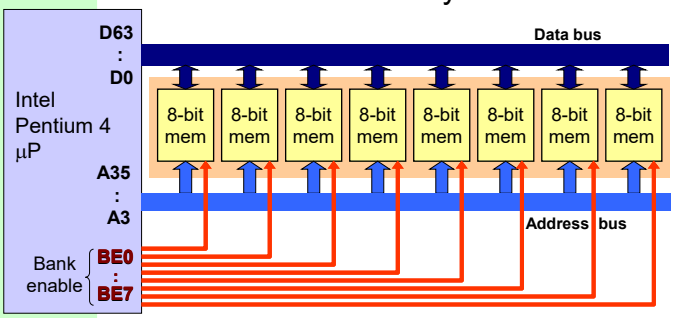
40

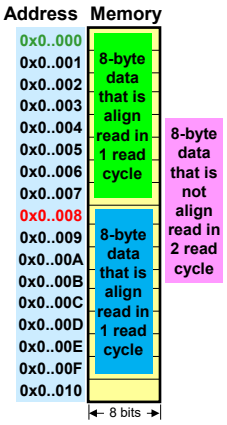
CZ1106
CE1106

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.

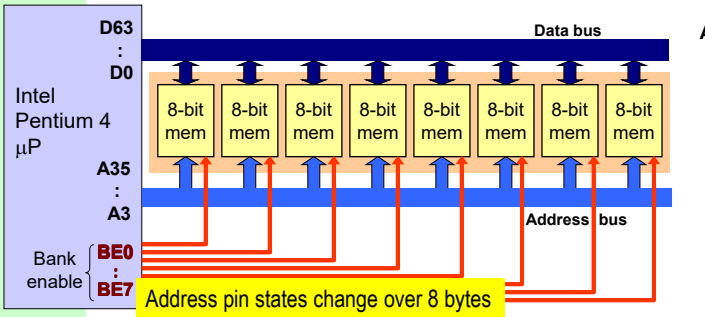
Learn More: Google "memory and data alignment"

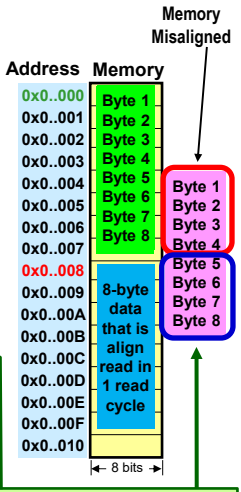
41

CZ1106
CE1106

Address Mapping of 64-bit data

(Case Study: Intel Pentium 4)





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.

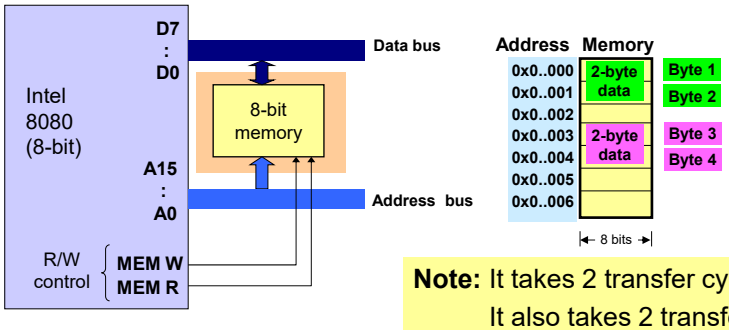
42

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

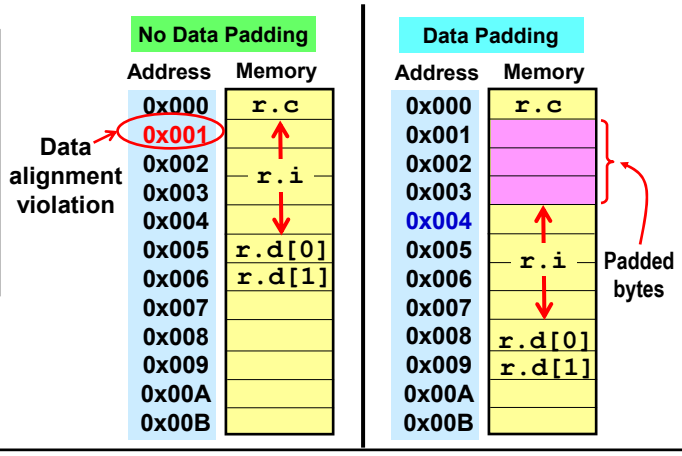
43

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;
```



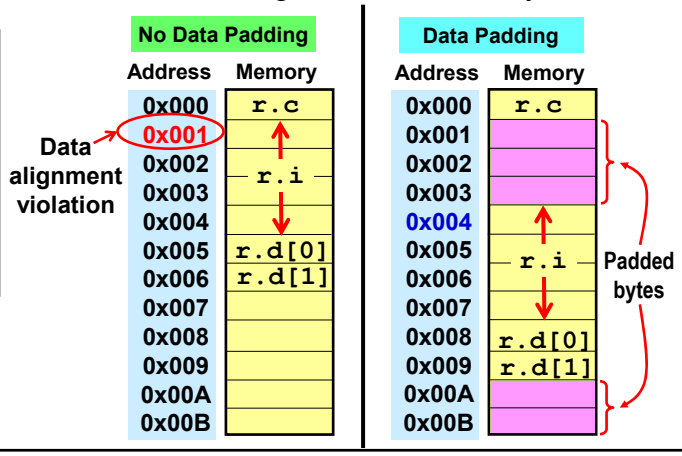
44

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];
```



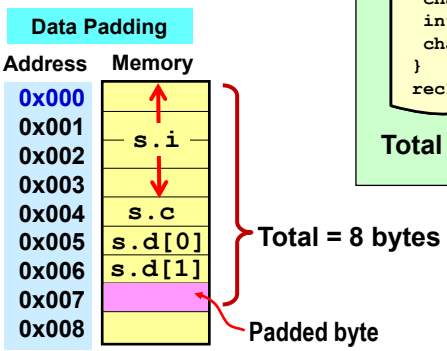
45

CZ1106 CE1106

Data Alignment with Structures (cont)

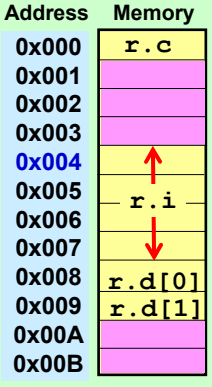
- Structures should be constructed with data alignment constraints in mind.
- **Rearrange the order** of data objects in the structure based on their size to minimize data padding where possible.

```
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

Chapter 3

Instruction Organization in Memory

1. **Role and Characteristics of Instructions**
   - What is an instruction?
2. **Programmer's Model**
   - The programmer's model
   - Describe function of VIP registers
3. **Basic Execution Cycle**
   - Execution of a simple LDR instruction

3

CZ1106
CE1106

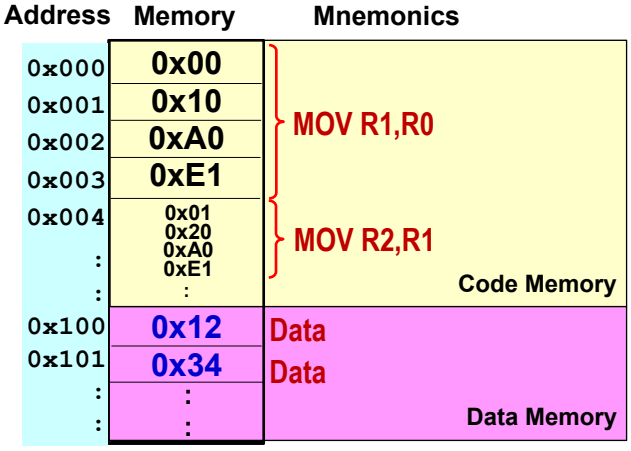
Code and Data in Memory

- 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.

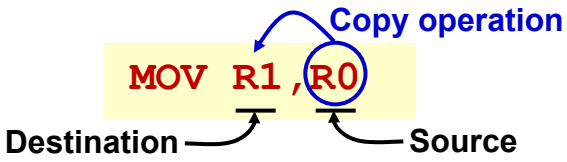


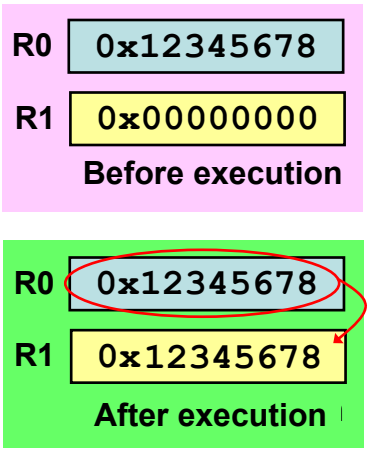
4

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.

5

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.

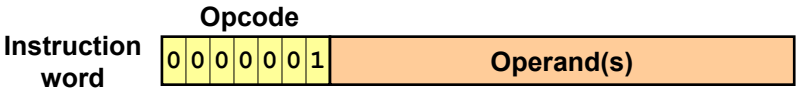
  - First part of instruction (**op-code**) specifies the operation to be carried out (e.g. move, add, subtract).
  - Remaining bits (**operands**) specify the data itself or the location of data and where results is to be stored.
  - Some operations produce no storable result but may alter program sequence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

6

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**.

7

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`

8

CZ1106 CE1106

Role of Instructions

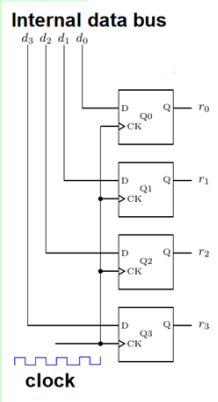
- The role of an instruction is to make purposeful **changes** to the **current state** of the processor.
  - The visible current state of a processor is defined by the **programmer's model** and contents in **memory**.
  - There are three broad categories of instructions for general programming available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

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

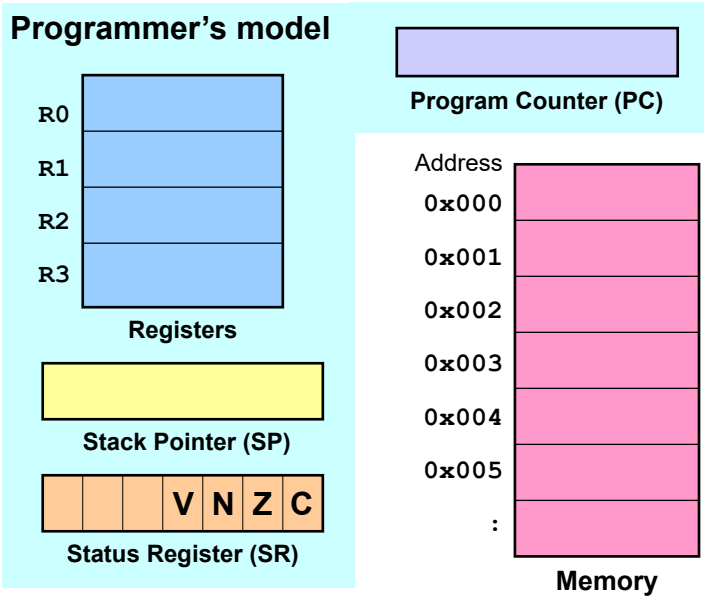
9

CZ1106 CE1106

Typical Programmer's Model & Memory



Register circuit

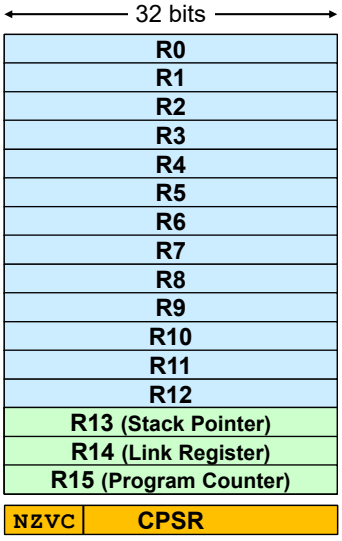


10

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**)



11

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).

12

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"

13

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

14

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.

15

CZ1106
CE1106

16

16

CZ1106
CE1106

Chapter 3

Instruction Organisation in Memory

Basic Execution Cycle

Learning Objectives (3.2)

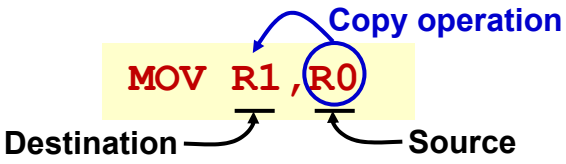
1. Describe the function of a simple **LDR** instruction from the ARM instruction set.
2. Describe the basic execution cycle of a simple **LDR** instruction.
3. Describe the factor that determines the execution speed of an instruction.

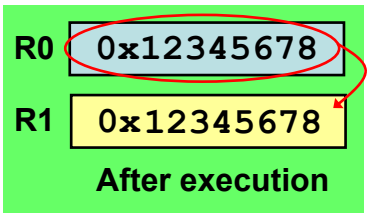
17

CZ1106
CE1106

The MOV Instruction (Review)

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



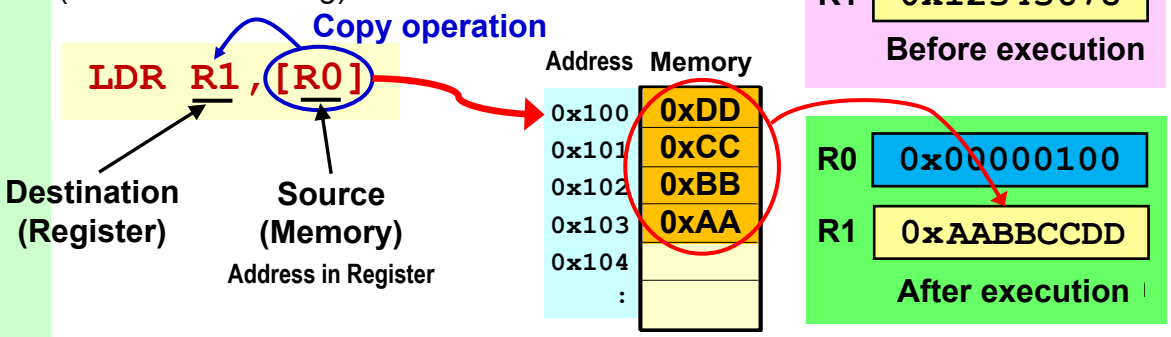


18

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).



19

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.

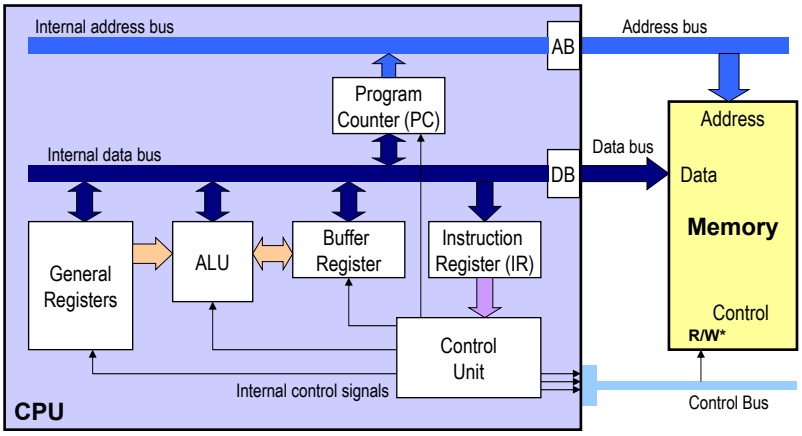
20

CZ1106
CE1106

Basic Execution Cycle

A Simple Processor

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



AB = Address buffer/latch
DB = Data buffer

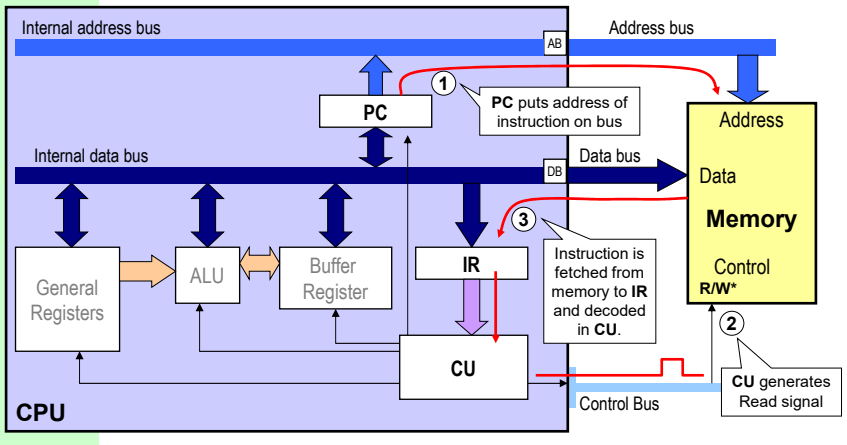
21

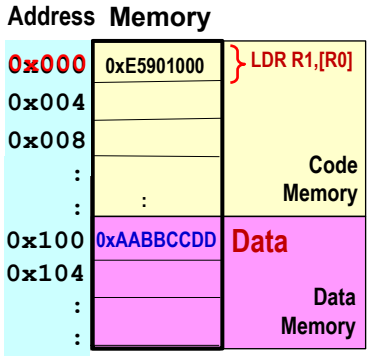
CZ1106
CE1106

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**.





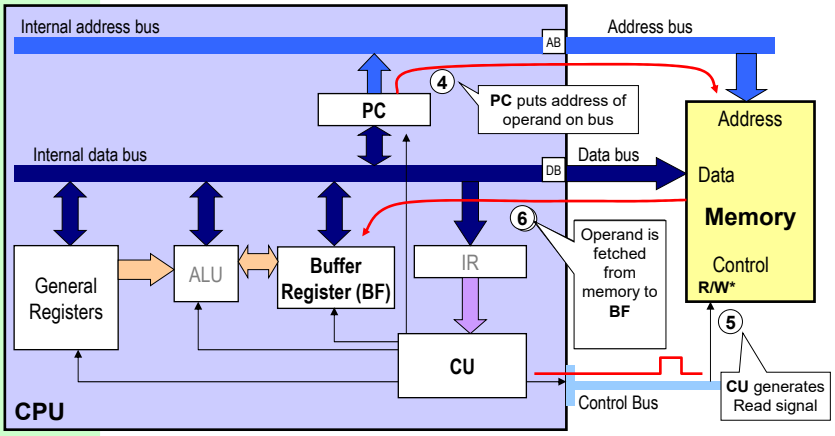
22

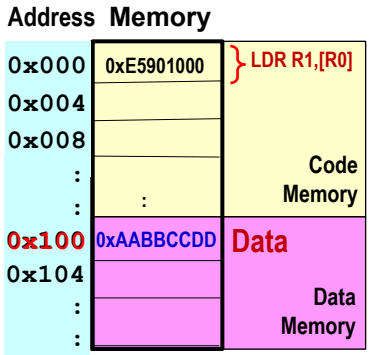
CZ1106
CE1106

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.





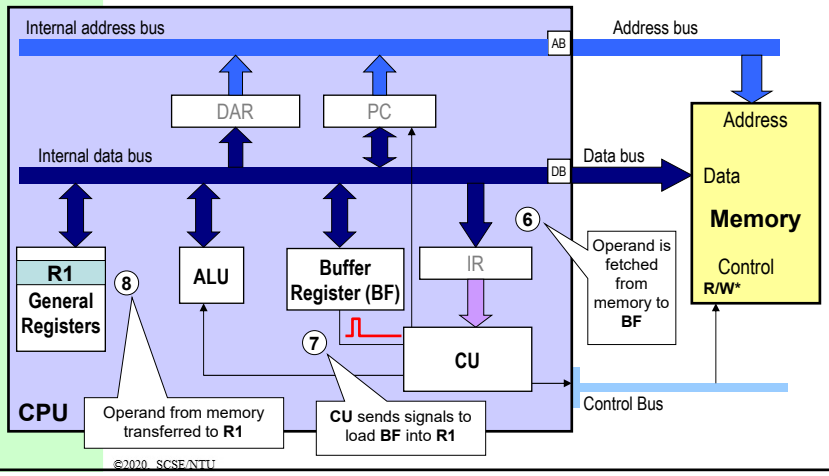
23

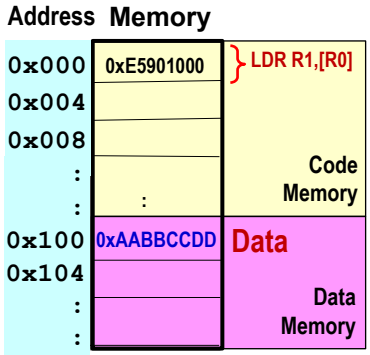
CZ1106
CE1106

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**.



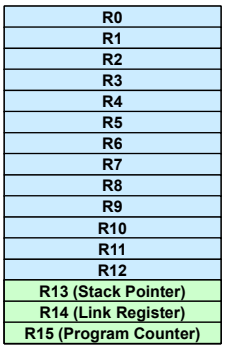


24

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.



25

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)

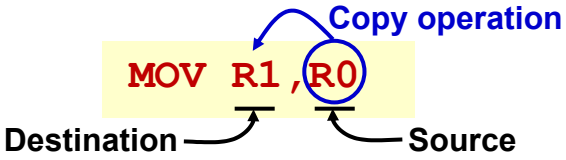
1. Describe what is register direct.
2. Describe what is immediate data and its application.

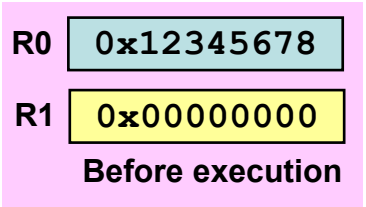
9

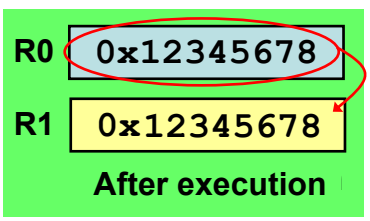
CZ1106
CE1106

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**

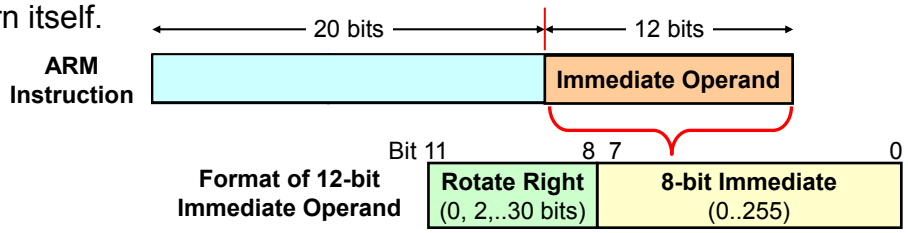
R1 0x00000003 **After execution**

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

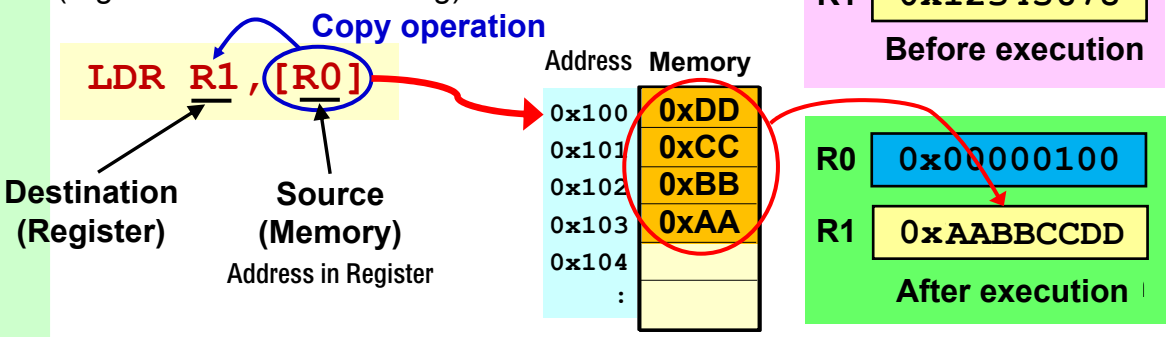
- Register direct and immediate addressing **do not** allow CPU to access operands stored in **memory**.
  - C variables are usually allocated memory for storage (especially large arrays).
- How do you specific a 32-bit address in memory using a 32-bit long instruction?
  - The ARM specifies the 32-bit 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).

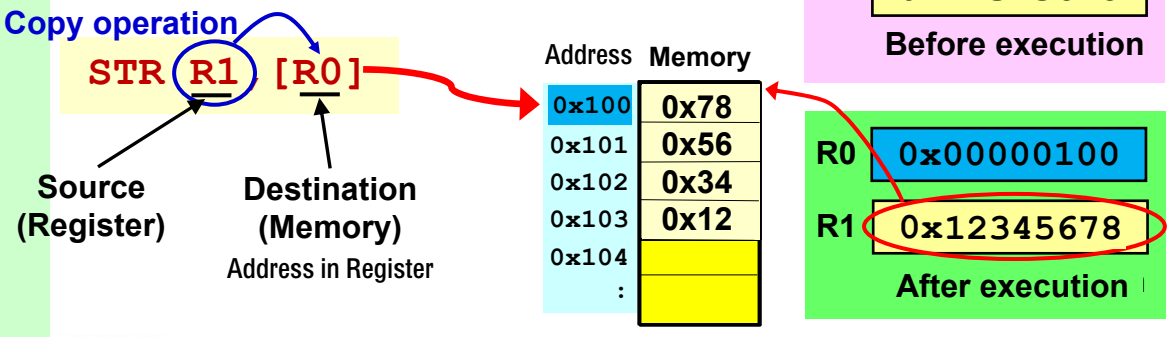


19

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.

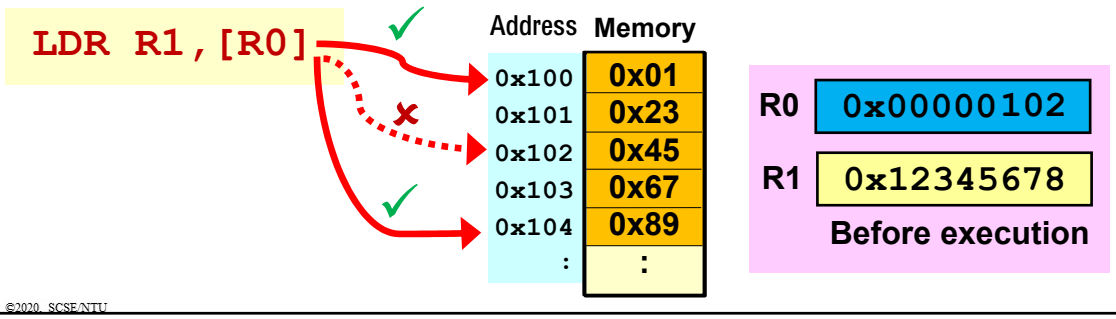


20

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**.

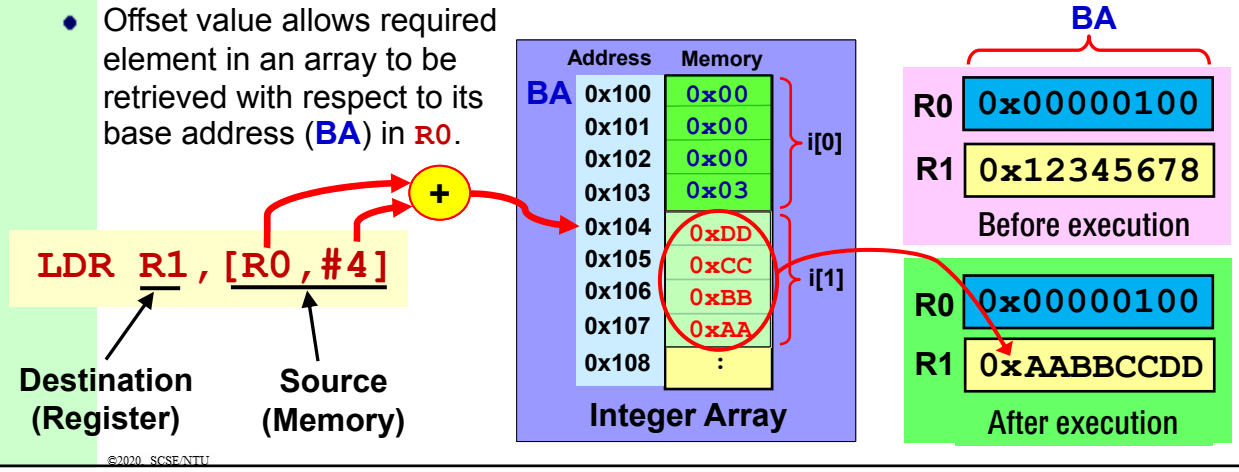


21

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

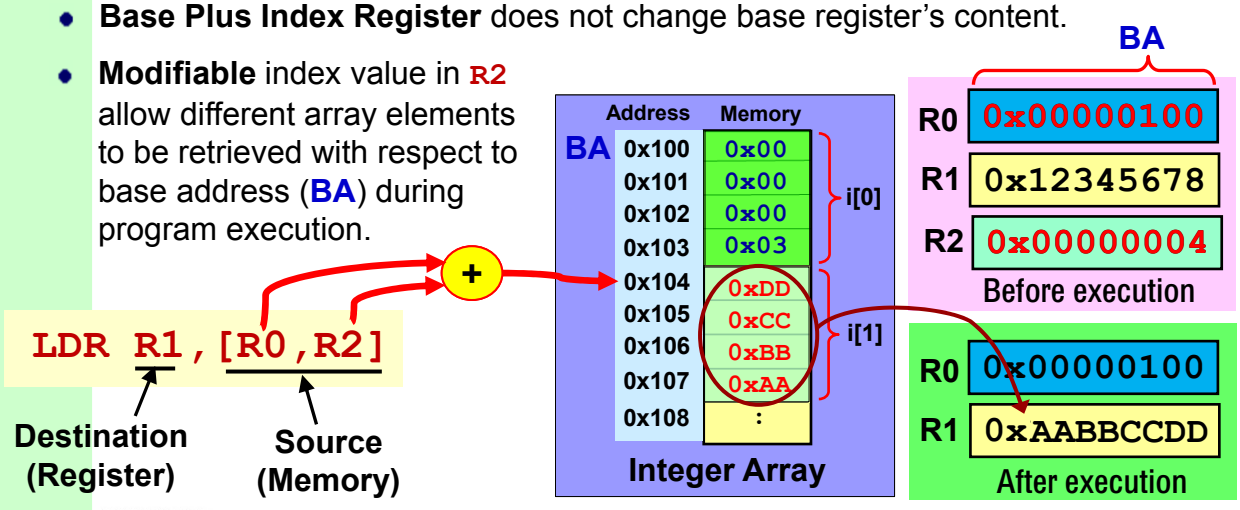
1. Initialize base address of array `i` into register `R2`.
2. Load value of 7 into register `R1`.
3. Store the value of 7 into i[0] and i[4] using offsets of 0 and 16 of register R2 respectively .
   - In computing offset, note that each integer element occupies 4 bytes in memory.

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

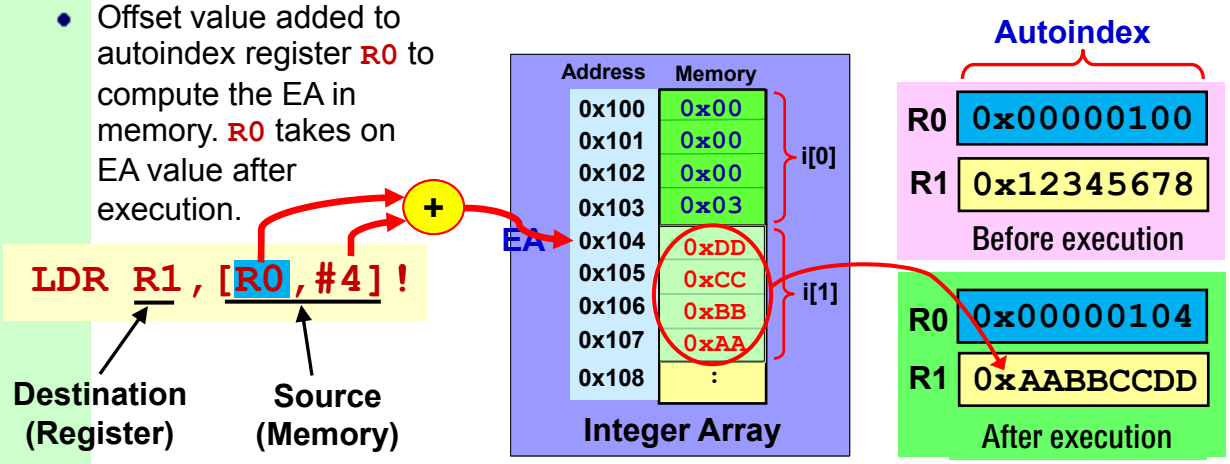
- Recap – Register indirect with base register:
  - The base address in the indirect register can be added with an offset or the contents of index register to compute effective address of operand in memory.
  - Base address is **never modified** after execution as it is assumed to be the sole reference 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

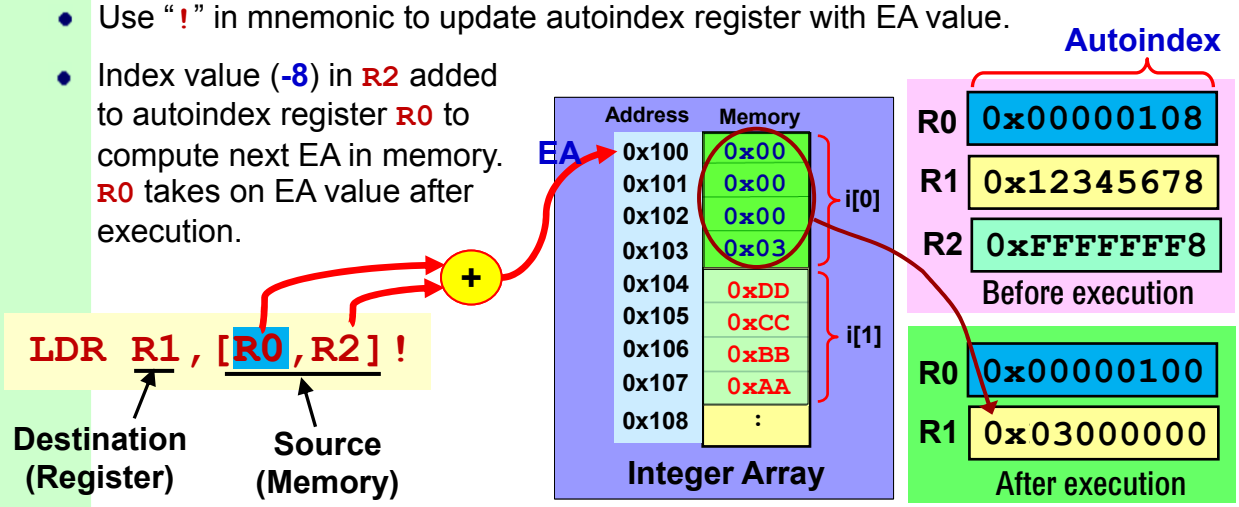
1. Initialize base address of array i into register R2.
2. Load value of 0 into source register R1.
3. Store 0 in R1 into first element of array i[0].
4. Store 0 in R1 into i[n] using current effective address (EA) of autoindex register R2 plus offset 4. Then put this EA into R2.

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 **after** 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

(3) } 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.

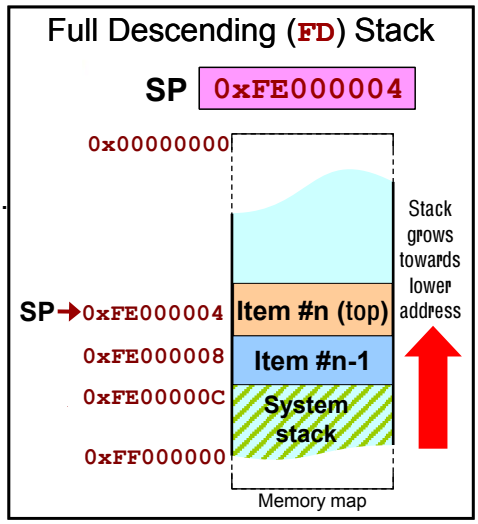
33

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"



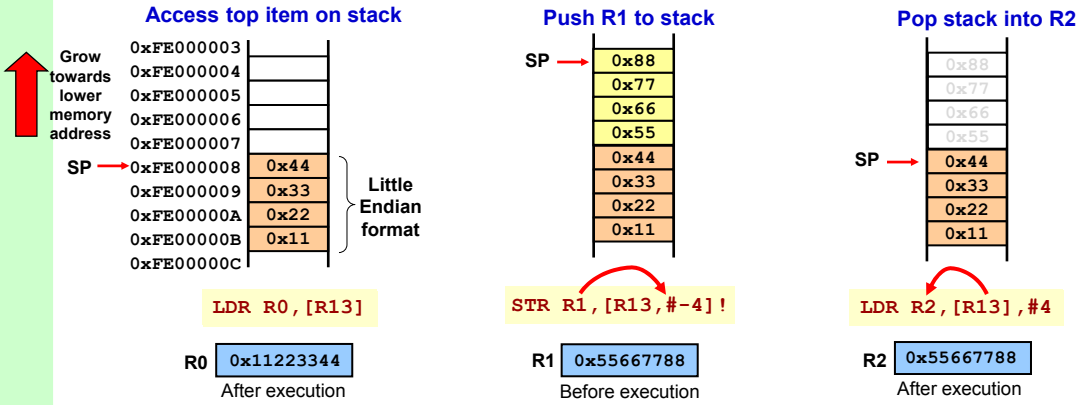
34

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 Ascending** (`FA`) stack implementation:

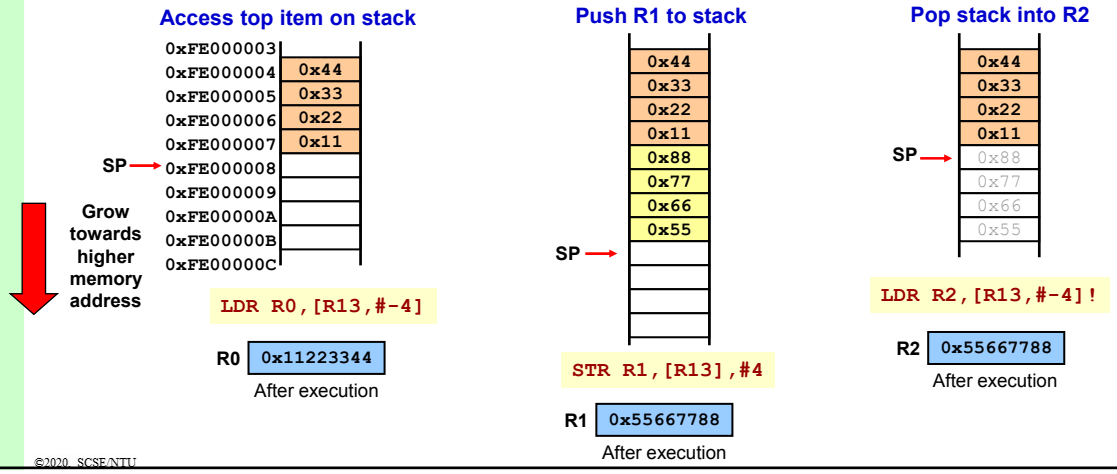


35

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)

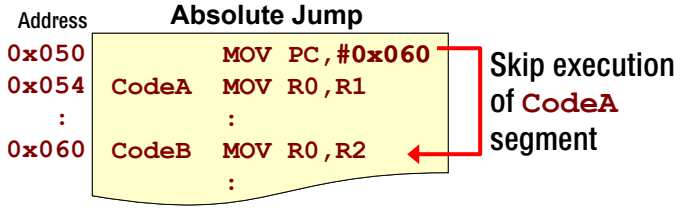
1. Describe difference between absolute & relative jump.
2. Describe the concept of position-independent code and how it is achieved.
3. Describe how data can be accessed using PC relative addressing

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`



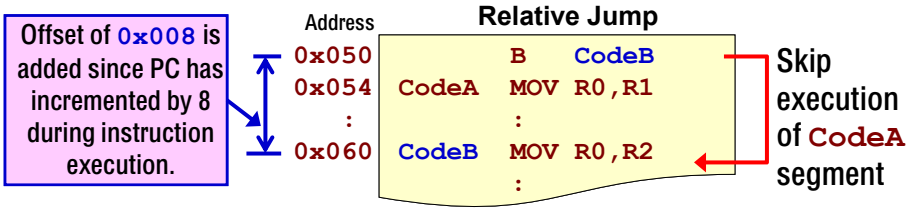
  - Absolute jump is not **position-independent**. This code can only execute correctly in this specific area of code memory.

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.

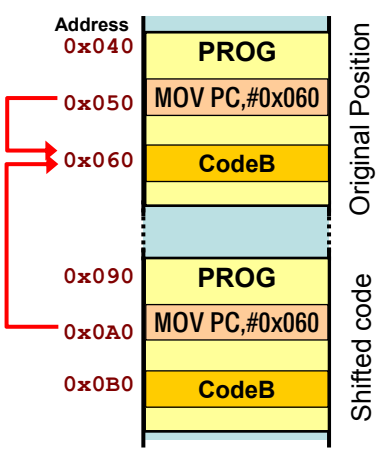
Note: In the ARM processor the PC points 8 bytes ahead of the current executed instruction due to its pipeline architecture.

41

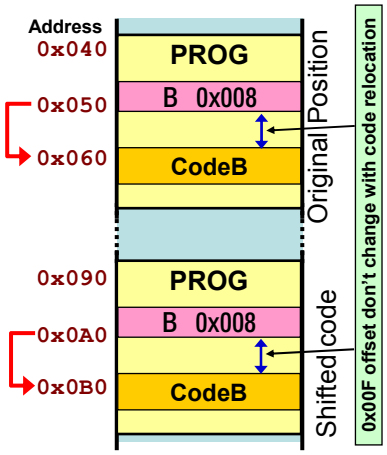
CZ1106 CE1106

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`



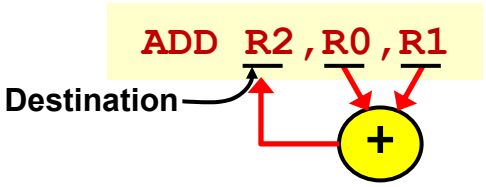
`B` still branch to **CodeB** after **PROG** is shifted to `0x090`

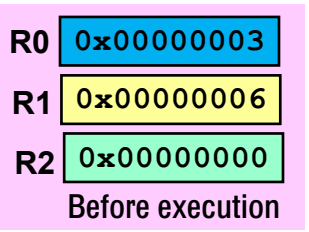
42

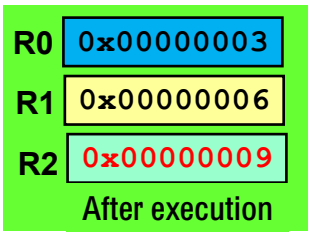
CZ1106 CE1106

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

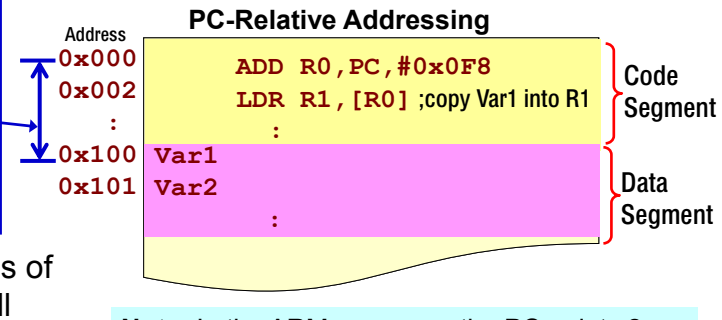
CZ1106 CE1106

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

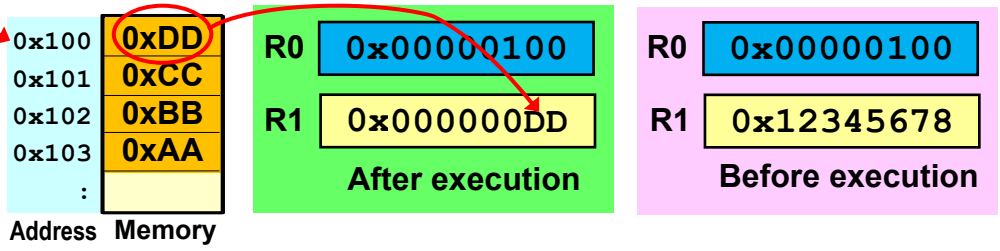
4

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

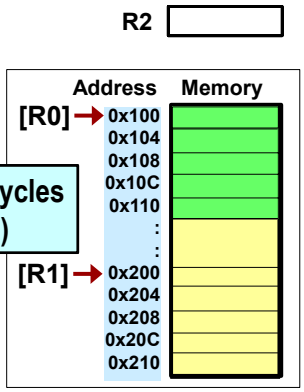
Copying a Block of Memory

- 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

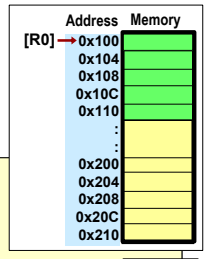
8

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**.




```
        MOV  R0,#0x100      ; setup source and destination pointer
loop    LDR  R2,[R0],#4     ; memory to register transfer
                            ; with post index autoindexing
        STR  R2,[R0,#0xFC]  ; register to memory transfer
                            ; with base plus offset of (0x200-0x100)+4=0xFC
```

Use one register less, save 1 cycle

20 cycles (4x5)

loop back 5 times

Offset range = ±4096

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)

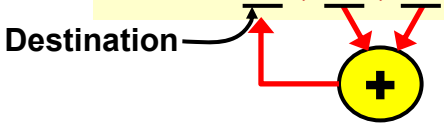
13

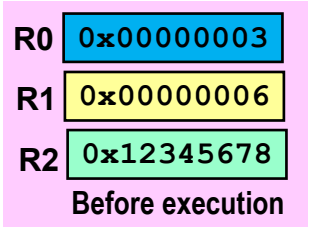
CZ1106 CE1106

ADD Instruction

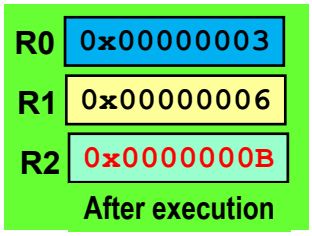
- Addition is a **commutative** operation that adds two source operands (order is immaterial).
  - But only **rightmost** operand can take on an **immediate** value. The other operands must be registers

```
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

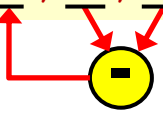
CZ1106 CE1106

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)

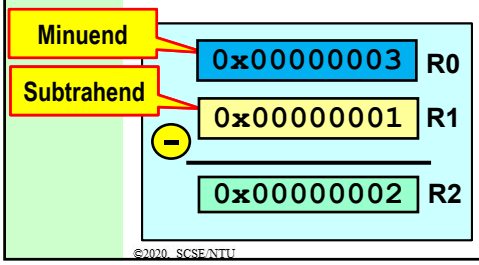
- Subtraction is done by **adding** the minuend to the **negated** subtrahend.

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

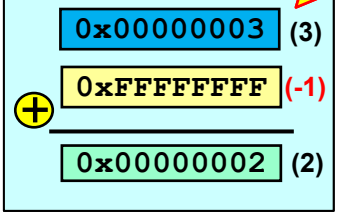
- **2's complement** is used to negate subtrahend.

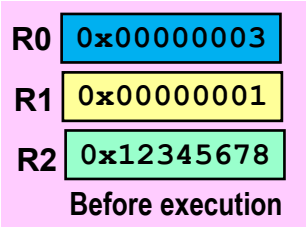
```
SUB R2,R0,R1 ; R2=R0–R1
```

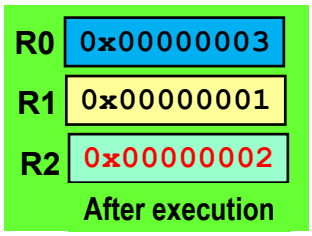
In binary
```
  ...0000 0011
+ ...1111 1111
  ...0000 0010
```



=





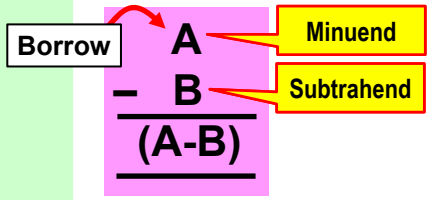


17

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
```

loop back 9 times

60 cycles (6x10)

1st –ve integer converted

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

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

loop back 9 times

R1 = R2 – R1
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

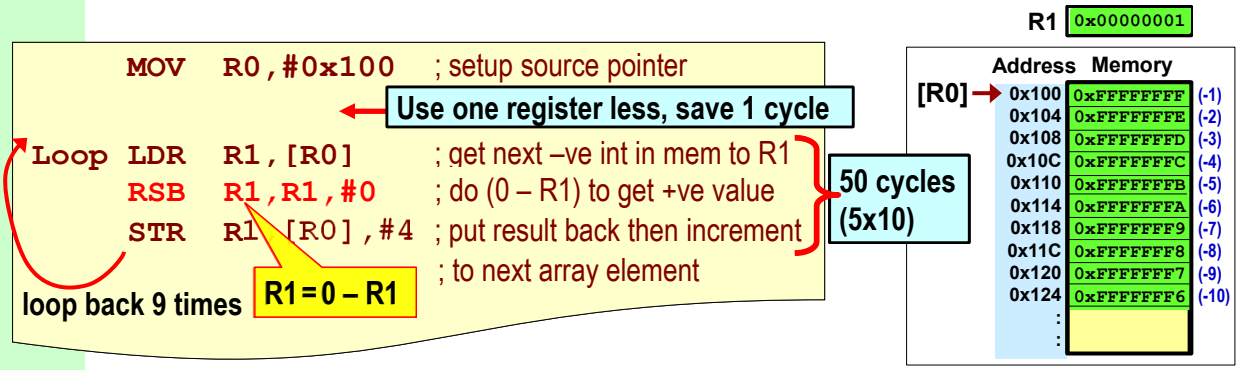
20

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.



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

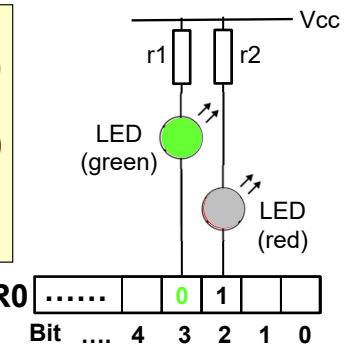
Program Example

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

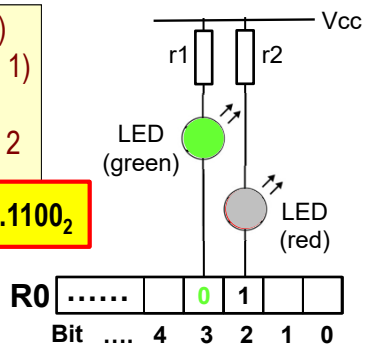
Program Example (Improved version)

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**)

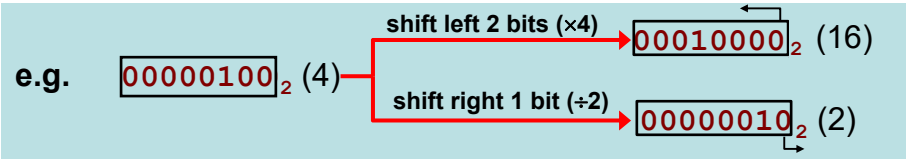
  - Rotate Right (**ROR**) and Rotate Right Extended (**RRX**).

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**).

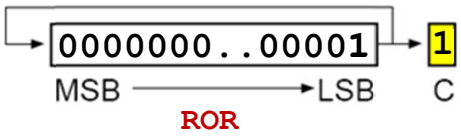
- The "**S**" suffix is used on the data processing operator to influence the **C** flag.

33

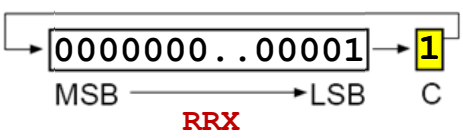
CZ1106 CE1106

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

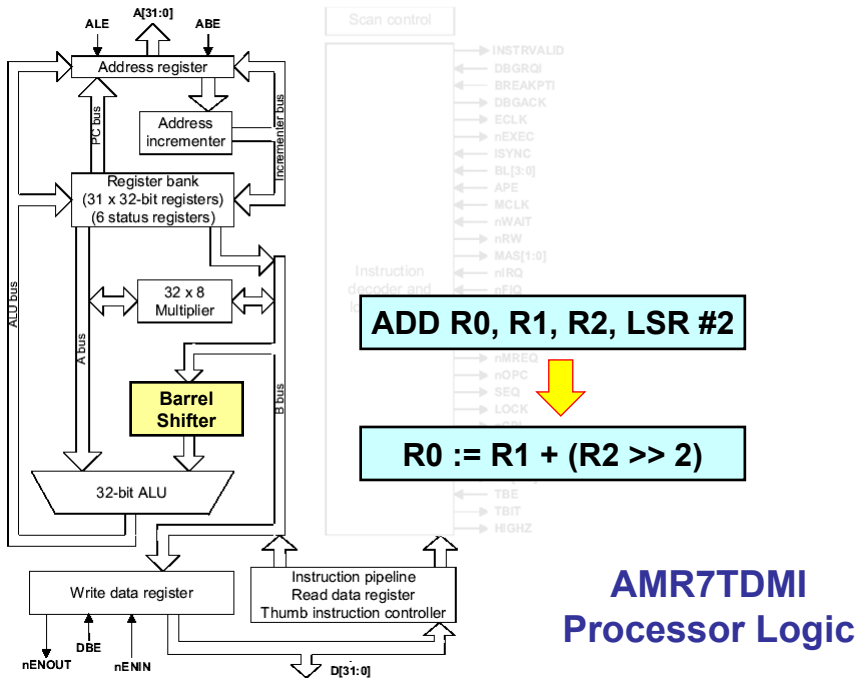
```
ADD  R2,R1,R0,LSR #2  ; R2=R1+R0>>1
```
ADD R1 with 2-bit right shifted R0. Put result in R2.

```
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



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

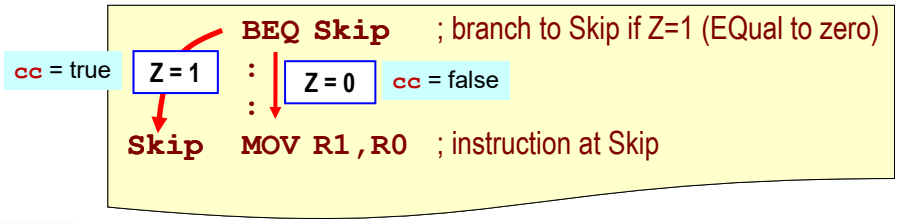
- These instructions facilitate the **disruption** of a program's normal **sequential** flow.
  - The disruption of sequential flow is implemented by modifying the contents of the Program Counter (**PC**).
  - The content of the **PC** can be modified **directly** or by using a **Branch** instruction.
  - 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

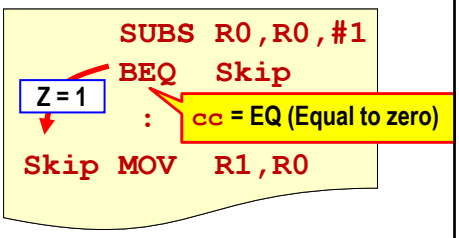
CZ1106 CE1106

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.



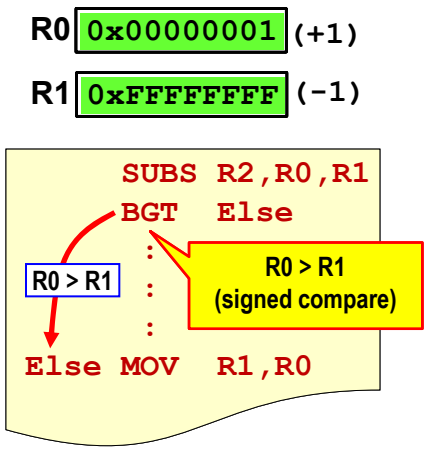
45

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