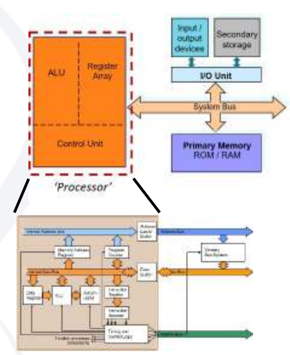
ENS1161 Computer Fundamentals

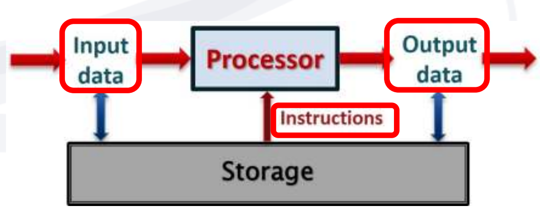
Module 3
Data and instruction formats

Moving forward..

- Last week the focus was on the hardware components of the system, and in particular the processor hardware and principles of operation



- This week, more details on the 'software' components
  - Data
  - Instructions



Module Objectives

On completion of this module, students should be able to:

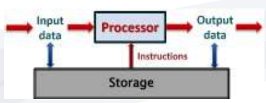
- Convert unsigned integers from decimal to binary and vice versa
- Convert signed integers from decimal to binary in 2's complement format and vice versa
- Describe how computers represent other types of numeric and non-numeric data
- Describe common instruction types and explain their function
- Explain what addressing modes are and the difference between data and program addressing modes
- Describe how common data addressing modes work
- Describe how common program addressing modes work

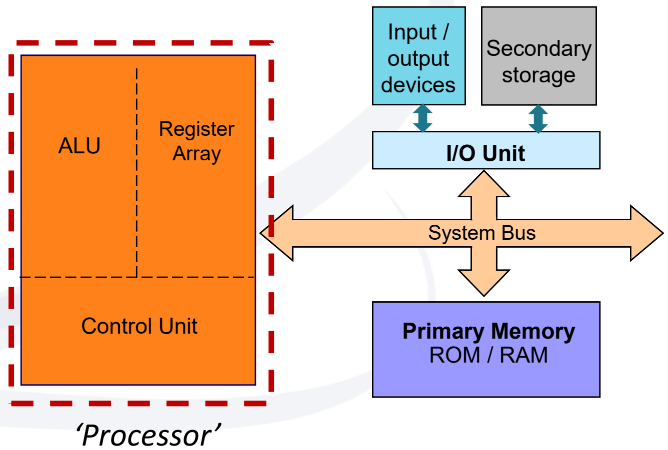
Introduction

- **Topic Scope**
  - Types of data and overview of data formats
  - Overview of instruction types
  - Operands and data addressing modes
  - Program flow control and related addressing modes

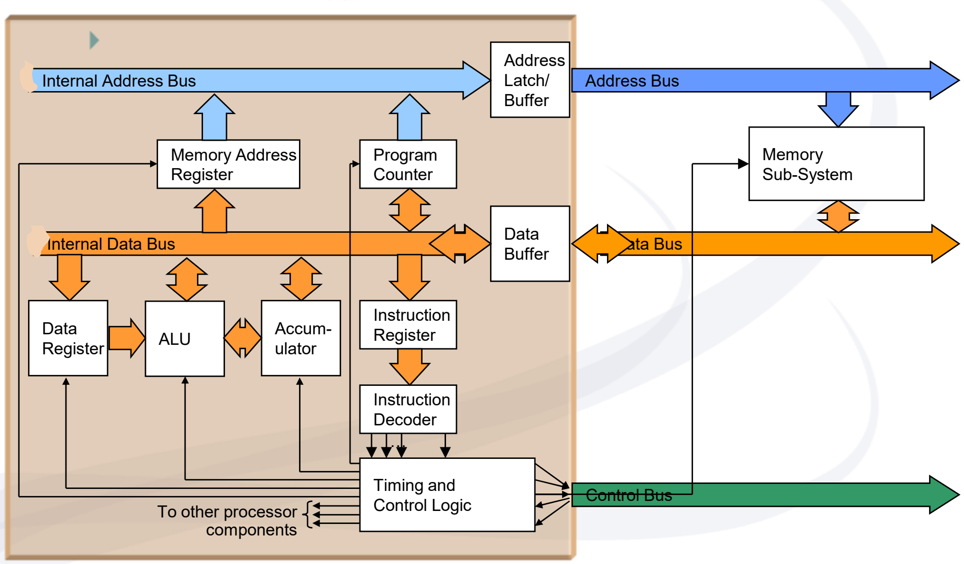
Basic Components of a Computer *(recap)*

- Every computer contains the same basic components:
  - Arithmetic logic unit (ALU)
  - Register array
  - Control unit
  - Memory
  - Input/Output (I/O) unit
  - System Bus

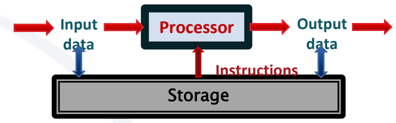




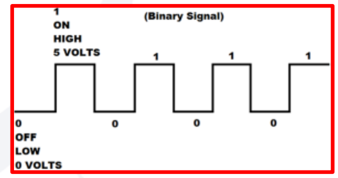
Simplified Diagram of a Processor *(recap)*



Data & Instructions *(recap)*



- Computer systems are digital systems
  - All **data** and **instructions** are in the form of binary signals
    - signals that have only two possible values
      - **1** or **0**
      - HIGH or LOW
      - ON or OFF



- These digital signals may be used to represent:
  - one bit of a binary number
  - one bit of a binary code
    - ASCII,BCD, instruction code, …
  - a control signal state, etc.

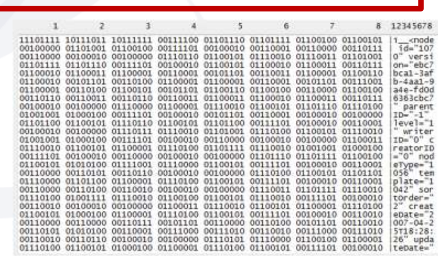
Data & Instructions *(recap)*

- Bytes stored in the memory unit of a computer can represent a number of different things:
  - Binary numerical data
  - Coded data (text, images, sound, video, etc)
  - Instruction codes
  - Operand or program branching addresses

Primarily input and output data (Binary numerical data, Coded data)

Related to program instructions (Instruction codes, Operand or program branching addresses)

- Difference lies only in context
  - How it is used / interpreted
- Else they cannot be differentiated
  - Just a bunch of bits



Numerical data

- **Integer data**
  - The set of integers: zero and the positive and negative "whole numbers"
    - i.e. have no fractional part.
    - E.g. 37, 1496, –28, –355, etc,
  - Computer science uses concepts of signed integers and unsigned integers
  - Unsigned integers are the non-negative integers
    - i.e. 0 and positive integers
    - Some applications involve positive whole numbers only.
  - Signed integers include all the integers
    - positive, negative and zero
  - A sign is needed to distinguish between positive and negative numbers

Unsigned integers in binary

- Each bit (place value) represents an increasing power of 2
- Each place value multiplied by 1 or 0
  - According to the bits in the number
  - same as ordinary binary numbers
  - may have extra zeros added in front to "pad out" to 8 bits or 16 bits
- Terminology:
  - *LSB* : **L**east **S**ignificant **B**it
    - The rightmost bit (Bit 0), represents 2^0
  - *MSB* : **M**ost **S**ignificant **B**it
    - Leftmost bit
    - Value depends on number of bits in number
    - For *n* bits, MSB represents 2^{n-1}

MSB							*LSB*
128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	1	0	1	0	1

32 + 16 + 4 +1 = **53**

Representation of unsigned integers

- Using 8 bits (1 byte) the range of values for unsigned integers is:
  - 0000 0000 to 1111 1111
    - 0 to 255 in decimal

- Using 16 bits (2 bytes) the range for unsigned integers is:
  - 0000 0000 0000 0000 to 1111 1111 1111 1111
    - 0 to 65535 in decimal

Signed 2s complement numbers

- Most common method of representing signed numbers in computers
- Split the binary combinations between positive and negative numbers
  - MSB = 0 for positive
  - MSB = 1 for negative
- 2s complement process
  1. *Invert all bits (complement)*
  2. *Add 1*
- The process converts numbers
  - +ve to –ve
  - -ve to +ve

| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | = **53** |

+ve

1 1 0 0 1 0 1 0 *Invert bits*

+1 *Add 1*

| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | = **-53** |

-ve

0 0 1 1 0 1 0 0 *Invert bits*

+1 *Add 1*

| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | = **53** |

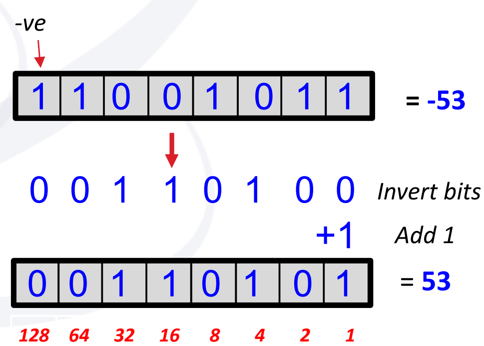
Representing negative integers

- To find the 2's complement representation of a negative integer *–m*:
  - drop the minus sign
  - convert the decimal number *m* to binary
    - 8-bit / 16-bit etc.
  - Invert all the bits (1's complement) and add 1

- *Example :*
  - Find the 2's complement representation of –95
  - The binary representation of 95 is 0101 1111
  - The 1's complement is: 1010 0000
  - Add 1 1
  - *2s complement representation of -95:* **1010 0001**

Interpreting 2's complement numbers

- To interpret a binary 2's complement number in decimal:
  - First check the sign bit (MSB)
  - If the sign bit is 0 (*positive* number)
    - Do normal binary to decimal conversion
  - If the sign bit is 1 (*negative* number)
    - Invert all the bits (1's complement) and add 1
      - I.e. change it to positive form
    - Convert to decimal integer (*m*)
    - Original number is *–m*



Representation of signed integers

- Using 8 bits (1 byte) the range of values for signed integers is:
  - 0000 0000 to 0111 1111 represent 0 to +127
  - 1000 0000 to 1111 1111 represent -128 to -1
    - MSB = 0 : positive
    - MSB = 1 : negative

Signed	Unsigned	
+5	5	0000 0101
+4	4	0000 0100
+3	3	0000 0011
+2	2	0000 0010
+1	1	0000 0001
0	0	0000 0000
-1	255	1111 1111
-2	254	1111 1110
-3	253	1111 1101
-4	252	1111 1100
-5	251	1111 1011

- Using 16 bits (2 bytes) the range of values for signed integers is:
  - 0000 0000 0000 0000 to 0111 1111 1111 1111 represent 0 to +32767
  - 1000 0000 0000 0000 to 1111 1111 1111 1111 represent -32768 to -1

Interpretation of integer data

- Integer data is stored in memory is in binary form
  - As is any other form of data
- What it means may vary depending on whether program interprets it as unsigned or signed
- If MSB is 0 - no difference
- If MSB is 1 – there is a difference
  - A large positive number or a negative number

Signed	Unsigned	2's Complement
+5	5	0000 0101
+4	4	0000 0100
+3	3	0000 0011
+2	2	0000 0010
+1	1	0000 0001
0	0	0000 0000
-1	255	1111 1111
-2	254	1111 1110
-3	253	1111 1101
-4	252	1111 1100
-5	251	1111 1011

Binary to hexadecimal *(review)*

- Easy for humans to make mistakes with long strings of 1s and 0s
- Hexadecimal (*base 16*) easy way to represent contents of memory / registers
  - Easier to convert from/to binary than decimal
  - Every *4 bits = 1 hex digit* and vice versa
  - E.g.

$$\underbrace{1010}_{A}\;\underbrace{1011}_{B}\;\;\underbrace{0010}_{2}\;\underbrace{1001}_{9}$$

16-bit binary → *4 hex digits*

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Numerical data in memory

- **Storing numerical data in memory**
  - One number may be larger than can be stored in one memory location
    - E.g. 16-bit number will occupy two 1-byte locations
  - **Little-endian vs big-endian**
    - Order in which bytes stored in memory may vary in different systems
  - "Little-endian" : low-order byte stored first
    - e.g. Intel x86
  - "Big-endian" : high-order byte stored first
    - e.g. Motorola 68xx family
    - E.g.: the 16-bit value $\boxed{AB}\boxed{29}$h

High-order byte   Low-order byte

Little-Endian

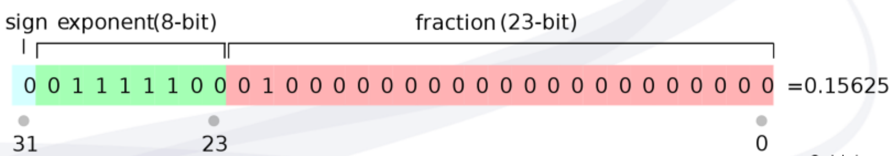
Memory Address (Hex)	Contents
C000	**0010 1001** (29h)
C001	**1010 1011** (ABh)

Big-Endian

Memory Address (Hex)	Contents
C000	**1010 1011** (ABh)
C001	**0010 1001** (29h)

Numerical data

- **Floating point numbers**
  - Used for non-integer numbers or very large integers
  - The number is divided into 3 parts
    - Sign, fraction and exponent
  - A common format is the *IEEE-754 32-bit Single-Precision Floating-Point Number*
    - Shown below

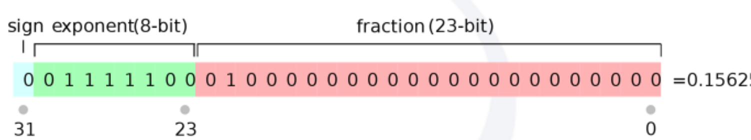


Codekaizen (https://commons.wikimedia.org/wiki/File:IEEE_754_Single_Floating_Point_Format.svg), „IEEE 754 Single Floating Point Format", https://creativecommons.org/licenses/by/3.0/legalcode

  - *Similar to scientific notation (e.g. 6.02 x 10^{23}) but in binary*

Numerical data

- **IEEE-754 format**



Codekaizen (https://commons.wikimedia.org/wiki/File:IEEE_754_Single_Floating_Point_Format.svg), „IEEE 754 Single Floating Point Format", https://creativecommons.org/licenses/by/3.0/legalcode

  - **Sign**: 0 = positive, 1 = negative
  - Biased 8-bit **exponent**
    - Exponent in *unsigned* binary
    - Add 127 *(bias)* to exponent so negative exponents can be represented
      - E.g. Above *01111100 = 124*, so actual exponent is *124 – 127 = -3*
  - **Fraction** (also called the ***mantissa***)
    - Assume that the first digit is 1 followed by binary point, and rest is what is in fraction part
      - Therefore number shown above is +1.01 x 2^{-3}
      - This is same as 0.00101 in binary = 0.15625 in decimal

Non-numeric data

- **Text**
  - One of the earliest and most common formats for textual data is ASCII
    - American Standard Code for Information Interchange
  - ASCII is a 7-bit code that has codes for characters, digits, as well as non-printable characters
    - Currently commonly stored as 8-bits with a leading 0
  - Another common text format is Unicode
    - an expansion of ASCII to cover non-English languages
  - Note: There is a difference between ASCII digits and integers
    - E.g.: Integer 15 is 0000 1111 (0F hex)
    - Text – 2 characters '1' and '5' (31 hex and 35 hex)

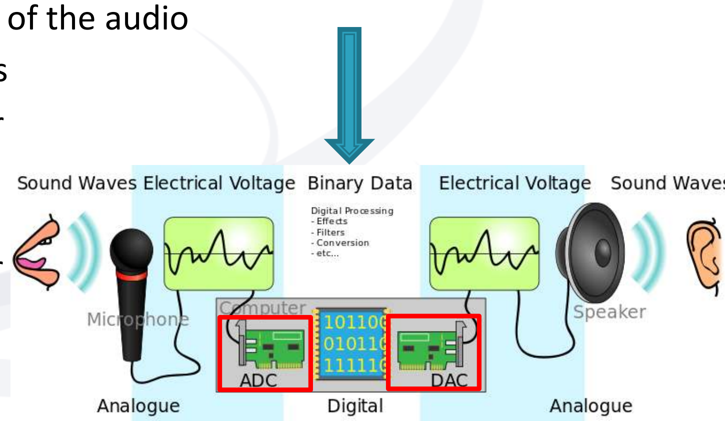
	$B_7B_6B_5$							
$B_4B_3B_2B_1$	000	001	010	011	100	101	110	111
0000	NULL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB		7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Non-numeric data

- **Non-text data**
  - There are various formats / standards for encoding data for use in computers
    - **Image:** JPEG, PNG, TIFF, BMP, GIF, SVG, etc.
    - **Audio:** WAV, PCM, AIFF, MP3, AAC, WMA, etc.
    - **Video:** AVI, MOV, MP4, WMV, etc.
  - All these data formats have a specific way of encoding the information in binary
  - Many of them also include *compression* techniques to save the amount of data to be stored or transmitted
  - To recreate the information from the bits, the binary data will need to be decoded
  - The devices or software to do this are called *coder-decoders* or *codecs* for short

Example: Audio data

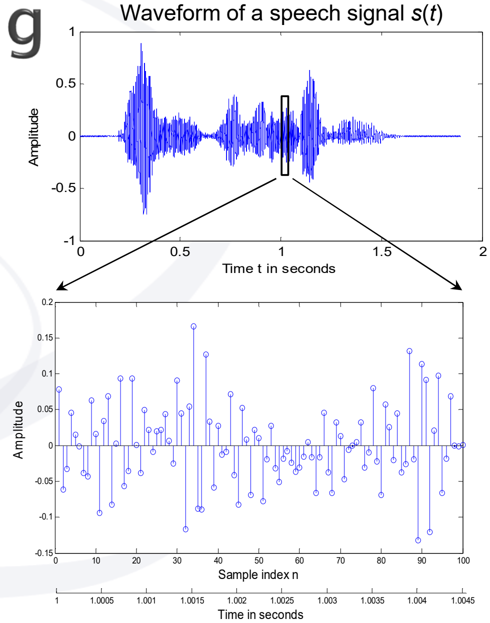
- ‘Real-world’ audio is an analogue (continuous) signal
- It needs to be converted to ***binary data*** that computers understand
- Reverse process during playback of the audio
- This is done using special devices
  - ADC – Analog to Digital Converter
    - Samples analogue signal
    - Produces binary data
  - DAC – Digital to Analog Converter
    - Takes the binary data
    - Converts it back to analogue signal



Milesjpool, 2016
https://commons.wikimedia.org/wiki/File:CPT-Sound-ADC-DAC.svg

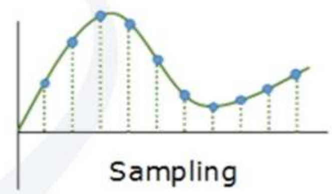
Example: Audio sampling

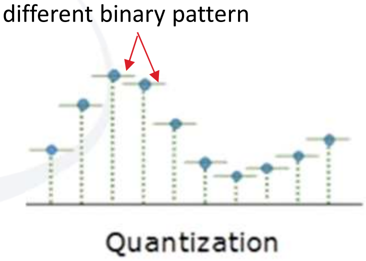
- Close-up of a section of the speech waveform stored in computer
  - shows discrete data points
  - called ***samples***
- Each vertical line ‘height’ is stored as one binary data point
- The closer the samples the better the representation of the waveform
- *More samples = Better quality sound*



Audio sampling and encoding

- ***Analog signals***
  - Signal whose amplitude can take on any value in a continuous range
    - E.g. speech signals
- ***Digital signals***
  - Values encoded in binary data of finite length
  - Therefore amplitude can only take on discrete values
  - *N* bits = 2^N levels
  - Some 'rounding errors' (*quantisation error*)
    - More levels = less rounding error
  - *∴ More bits per sample = better quality sound*





Instructions *(recap)*

- A processor can ONLY execute its predefined instruction set
  - This is *the set of operations a processor can perform*
  - **Different** for every processor family
- Each instruction (or variation of it) has a unique opcode
- Each instruction consists of two parts
  - an *operation*
  - an *operand* or *operand address*
    - data / location of the data on which to perform the operation

Operand address

Operations

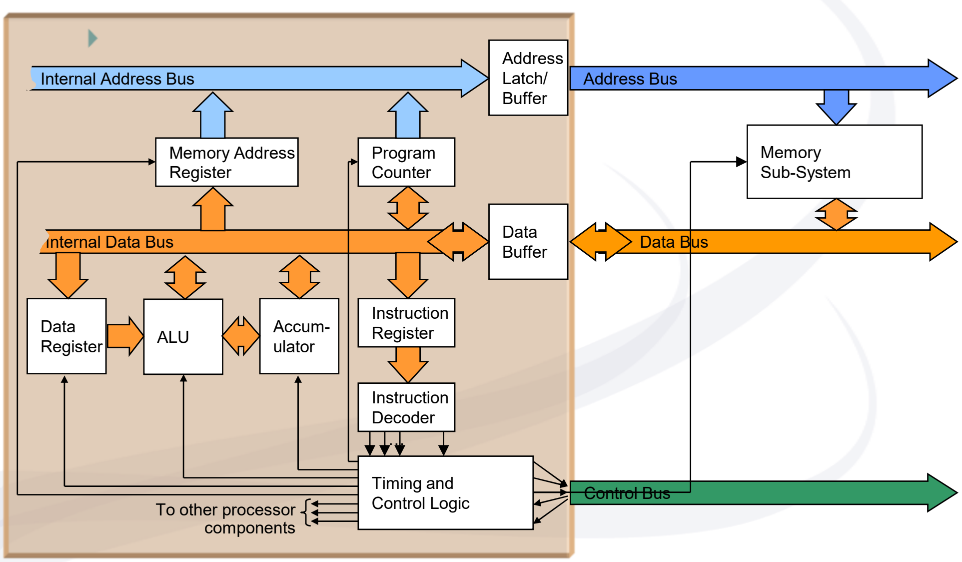
```
LDA  C100h   ; Load Accumulator with data from memory location
             ; C100 hex (or 1100 0001 0000 0000 in binary)
ADDI 2       ; Add immediate value (2) to the number in the Accumulator
```

Operand

Example Microprocessors

- Each processor has its own internal architecture (structure) and instruction set
- This unit will NOT attempt to cover all the details of any single processor
  - Instead focus will be on general principles that can relate to a variety of microprocessors and computer systems
  - In order to illustrate certain points, the following processors shall be referred to:
    - ***Hypothetical generic 8-bit processor***
      - Introduced in Module 2
    - ***Intel 8086***
      - The 'great-grandaddy' of processors in Windows-based systems, newer Macs and many Linux systems
    - ***Motorola M6800 / 68000***
      - Precursor / microprocessor used in Apple Macintosh
  - These processors have been chosen due to their relatively simple architecture

Generic 8-bit Processor



Common instruction types

1. **Data Manipulation Instructions**
   - Arithmetic instructions: *Add, subtract, multiply, etc.*
   - Logical instructions: *AND, OR, NOT, XOR, etc.*
   - Shift and rotation: *Shift right/left, rotate right/left, etc.*
2. **Data transfer instructions**
   - Transfer between processor and memory: *e.g. LDA, STA, PUSH, POP, etc.*
   - Transfer between registers within the processor
3. **Program control instructions**
   - Unconditional branching and function calls: *Jump, call, return (RET), etc*
   - Conditional branching: *Branch if equal, greater than, less than, etc.*

- *More in Module 5*

Instruction Format (recap)

- The instruction format will depend on the processor
- Even in within one processor, different operations may require different instruction formats
  - *Generic 8-bit processor* example:

Single-byte Instruction

	7 … 0	Op code – instruction that does not require explicit specification of an operand

Two-byte Instruction

Byte one		Op code – specifies a one byte operand
Byte two		Operand or operand address (8-bit)

Three-byte Instruction

Byte one		Op code – specifies a two byte operand
Byte two		1st byte of operand or operand address
Byte three		2nd byte of operand or operand address
	7 … 0	

Fetching the operand(s) *(recap Module 2)*

- If there are operands/operand addresses, they will need to be fetched from memory as part of the execution phase
  - This will be based on the result of the instruction decode
  - Control unit will then know how many bytes to fetch
- Process:
  1. Program counter value will be put on the address bus
  2. First byte of operand will be read from memory (via data bus) and stored at required location
  3. Program counter will be incremented
     - Ready pointing to next memory location
  4. Steps 1 – 3 repeated for as many bytes required

Memory address (Hex)	Memory contents (Hex)	
C000	B6	Opcode
C001	*C1*	2-byte Operand address
C002	*00*	

Instruction codes and formats

- How does the processor know if it needs to fetch extra bytes or not?
- How does it know how to use the operand?
- Each operation has a separate code
  - E.g. ADD, LDA, STA, NOT
- Each variation of an operation will also have its own code
  - Normally just 1 or 2 bits different
  - E.g. ADD *immediate value*, ADD *from memory*
  - *Example from generic 8-bit processor*

ADDI 2 *; **Add i**mmediate value (2) to the number in the ; accumulator* → 1011 1010 0000 0010

ADD C101h *; **Add** data at memory location C101h to the number ; in the accumulator* → 1011 1011 1100 0001 0000 0001

Difference in opcodes

Sample Instruction Set *(again)*

- Sample instruction set - for 'generic' processor

Instruction description	Mnemonic	Instruction Code (Hex)	Instruction Code (Binary)
Load Accumulator Immediate	LDI	B5	1011 0101
Load Accumulator data from memory	LDA	B6	1011 0110
Store Accumulator to memory	STA	B7	1011 0111
…	…	..	
Add Accumulator Immediate (8-bit)	ADDI	BA	1011 1010
Add Accumulator with data from memory (8-bit)	ADD	BB	1011 1011
…	…	…	…
Unconditional Jump (absolute address)	JMP	D0	1101 0000
Jump if Zero - Z flag set (relative)	JZ	D1	1101 0001
Jump if Not Zero - Z flag clear (relative)	JNZ	D2	1101 0010
Jump if Carry - C flag set (relative)	JC	D3	1101 0011
Jump if Not Carry - C flag clear (relative)	JNC	D4	1101 0100
Jump if oVerflow - V flag set (relative)	JV	D5	1101 0101
Jump if Not oVerflow - V flag clear (relative)	JNV	D6	1101 0110

Operand Order

- Operands can be:
  - Where to get data to be used by the operation (*source*)
  - Where the result of the operation goes (*destination*)
- The order in which the source and destination are specified varies
  - For example Intel specifies destination then source, whereas Motorola (Freescale) does the opposite
  - *Intel 8086* example
    - MOV AX, [0500] ; moves data from memory location 0500 to register AX

      destination source
  - *Motorola 68000* example
    - MOVE.W #$100, X ; moves 16-bit value 100 hex ($100) to register X

      source destination

Addressing modes

- The addressing mode specifies the way in which the operand(s) will be accessed by the processor
  - Each combination of instruction and addressing mode normally has its own unique opcode
- Common addressing modes are:
  - Implied mode
  - Register mode
  - Immediate mode
  - Direct mode
  - Indirect mode
  - Indexed mode
  - Relative mode

Implied addressing mode

- Where the instruction itself specifies the source, the destination or both
  - Also known as *implicit addressing*
  - *Examples from generic 8-bit processor*

LDA C100h ; ***Load Accumulator*** *with data from memory location C100 hex*

Implied destination

Source address

STA C102h ; ***Store Accumulator*** *contents in memory location C102 hex*

Implied source

Destination address

INCA ; ***Inc****rement* ***A****ccumulator i.e. A = A+1*

This operation has *no operands* specified. The source and destination (*accumulator*) and the value to be added (*1*) are all *implicit* (specified within the operation code)

Immediate addressing

- The operand is a constant given as part of the program
- i.e. the operand itself is the data and is fixed in the program code
  - The operand is lies in the memory area occupied by the program code (*code memory*)

Opcode	**Operand**

  - *Example from generic 8-bit processor*

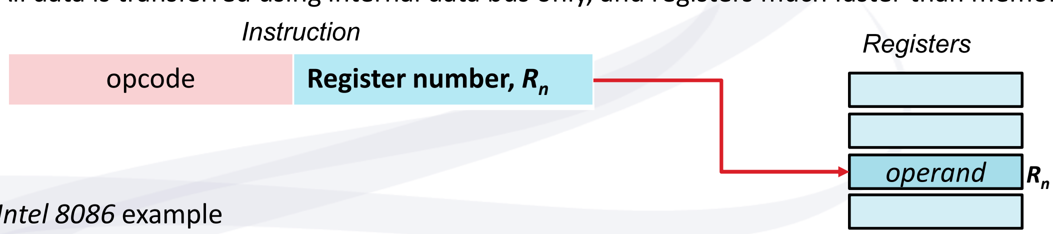
    ADDI 2 *; **Add I**mmediate value (2) to the number*
    ; in the accumulator

  - *Intel 8086* example
    - MOV AX, 0500 ; moves the number 500 to the register AX
  - *Motorola 68000* example
    - MOVE.W #$100,X ; moves 16-bit value 100 hex ($100) to register X

# indicates that the following is an immediate value

Register addressing

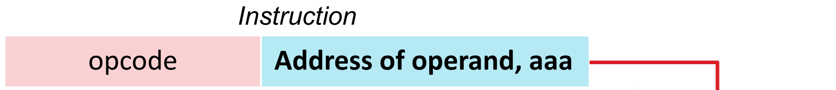
- The internal registers are specified as the source and/or destination
- Each register normally has some identifier
  - E.g. *AX, BX, CX*, etc. (Intel 8086) or *D0, D1, D2,..* (68000) in assembler code
  - Ultimately just binary codes for processor to understand
- Having both source and destination as registers is very common as it is fast
  - All data is transferred using internal data bus only, and registers much faster than memory

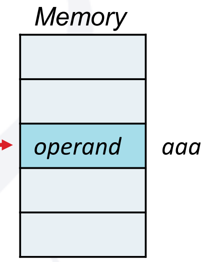


  - *Intel 8086* example
    - AND AX, DX ; Logical AND the contents of AX and DX and the answer goes back to AX (destination)

Direct addressing

- The address of the operand is specified in the instruction code





  - *Example from generic 8-bit processor*

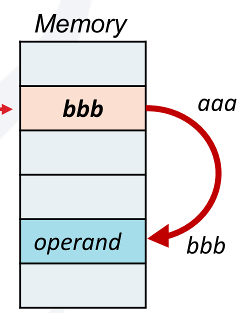
    ADD C101h   *; **Add** data at memory location C101h to the number*
    ; in the accumulator

  - *Intel 8086* example
    - MOV AX, [0500]   ; moves data from memory location 0500 to register AX

      [ ] indicates number within is the address from which to get operand (Intel standard)

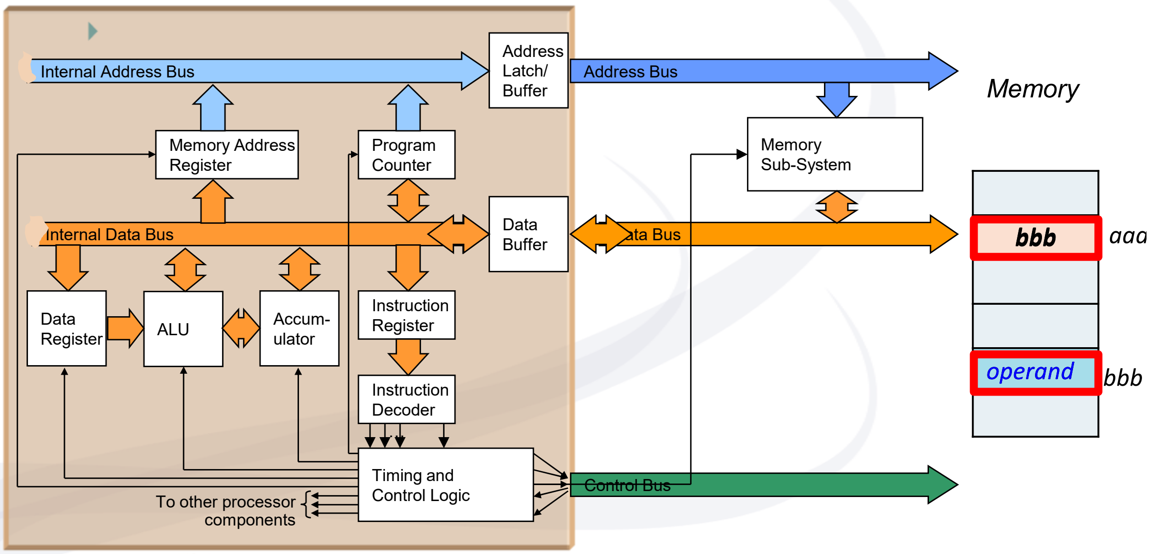
Indirect addressing

- The instruction has the address of a memory location that contains the address of the operand (pointer)



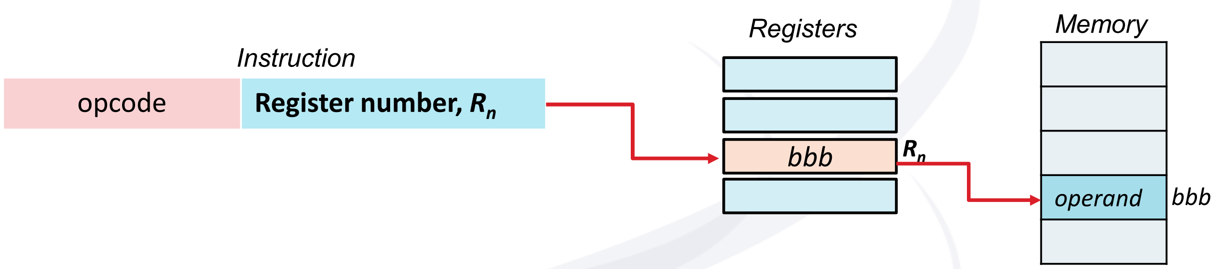
- Steps as follows:
  - Processor fetches the pointer address *(aaa)* from code memory
  - Pointer address is put on the address bus
  - Contents of pointer *(bbb)* is read into processor via data bus
  - Pointer value *(bbb)* is put on the **address bus**
  - *Operand* is read from **location** *bbb* into processor via data bus

Generic 8-bit Processor



Register Indirect addressing

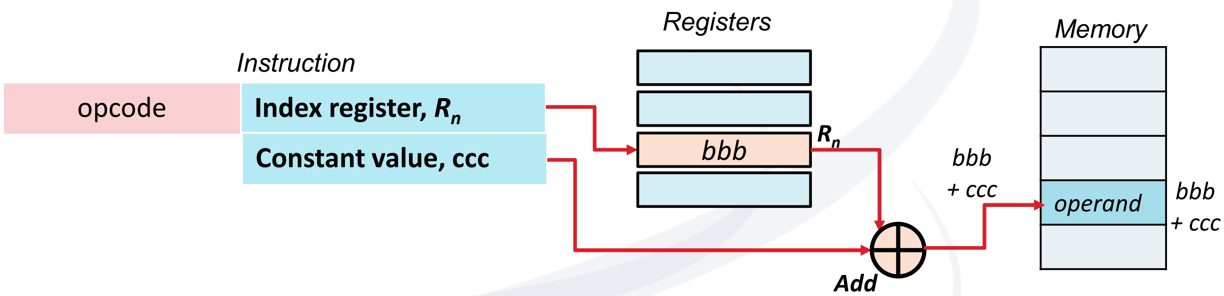
- Similar to indirect, except that the instruction specifies a **register** that contains the address of the operand



- *Intel 8086* example
  - MOV AX, [BX] ; takes value in BX register as address of a memory location
    ; moves data in that memory location to register AX

Indexed addressing

- The instruction contains a constant value and that is added to the contents of a register *(index register)* to work out the address of the operand



  - *Intel 8086* example
    - MOV AX, [SI + 10] ; takes value in SI register adds 10 and puts on address bus ; moves data in the selected memory location to register AX

Program execution

- A program is a series of instructions
  - stored in the memory sub-system
  - fetched and executed sequentially
- The sequential fetching is done using the Program Counter
  - After each fetch the Program Counter is incremented
    - PC ← PC + 1
    - Pointing to next address, ready for the next fetch
  - However, useful programs are generally not all sequential
    - May have loops, branches and function calls
    - There are instructions for changing the *flow* of the program

Memory address (Hex)	Memory contents (Hex)
C000	B6
C001	*C1*
C002	*00*
C003	BB
C004	*C1*
C005	*01*
C006	B7
C007	*C1*
C008	*02*

Data addressing vs program addressing

- Data addressing modes essentially are used to define where to obtain the data to be processed
  - The addressing modes described so far can be applied as data addressing modes
- Program addressing modes are used to define where the next instruction is to come from if not the default (next location)
  - Used by program flow control instructions like JMP, CALL, and conditional branching
  - These instructions change the contents of the Program Counter (PC)
  - The most common modes covered next
    - Absolute or Direct; Relative; Register Indirect

Absolute program addressing

- Also known as *direct* mode
- The instruction specifies a value that is written directly to the PC register

  - *Intel 8086* example
    - JMP 02A3H ; Program to **Jump** to location 02A3h

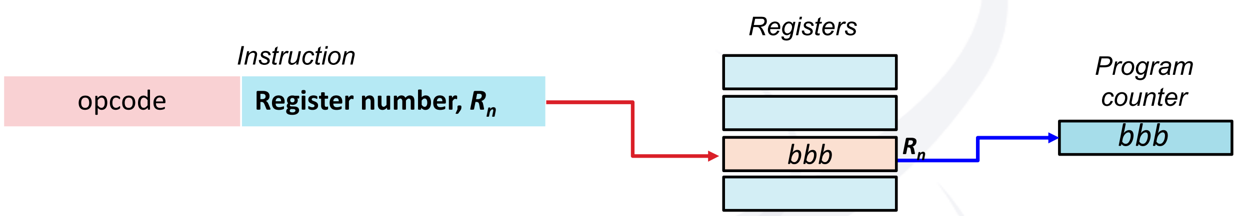
Relative program addressing

- The instruction specifies a value (positive or negative) that is added to the PC register
  - Causes the program to move forward or backward a certain number of bytes

  - *Intel 8086* example
    - JNZ 08 ; **Jump if** previous result is **Not Zero** forward 8 bytes

Indirect program addressing

- The program counter is loaded with the value from the register specified in the instruction



Summary

- Computers represent various kinds of data – all in binary
  - Integers, floating point numbers, text, images, audio, video, etc.
  - There are various formats and standards for each type
  - Data needs to be encoded and decoded appropriately for it to be used correctly
- Data addressing modes
  - Different methods by which the processor can find the operands for instructions
  - Common modes: Implied, Register, Immediate, Direct, Indirect, Indexed
- Program addressing modes
  - Methods by which program flow instructions get and use their operands
  - To have non-sequential fetching of instructions
  - Absolute/direct, relative, register indirect modes

Module Objectives

On completion of this module, students should be able to:

- Convert unsigned integers from decimal to binary and vice versa
- Convert signed integers from decimal to binary in 2's complement format and vice versa
- Describe how computers represent other types of numeric and non-numeric data
- Describe common instruction types and explain their function
- Explain what addressing modes are and the difference between data and program addressing modes
- Describe how common data addressing modes work
- Describe how common program addressing modes work

Unit Learning Outcomes

On completion of this unit, students should be able to:

- Describe the fundamental architecture and operating principles of a computer system.
- Interpret computer system specifications and standards and how they relate to system function.
- Compare different types of components and subsystems and their relative impacts on system function and performance.
- Make recommendations on the suitability of computer systems and components for a given function.
- Explain the interconnection between the software and hardware components of computer systems, and the processes involved in making them work together.

Next Module

Module	Topic
1	Introduction and overview
2	Basic Computer Architecture and Principles of Operation
3	Data and instruction formats
4	Basic computation in computers
5	Programming languages and tools
6	Memory devices and storage systems

Module	Topic
7	I/O devices and interfacing
8	I/O modes and BIOS
9	Operating Systems
10	Networks and the Internet
11	Embedded Systems and Cloud Computing
12	Review & Revision