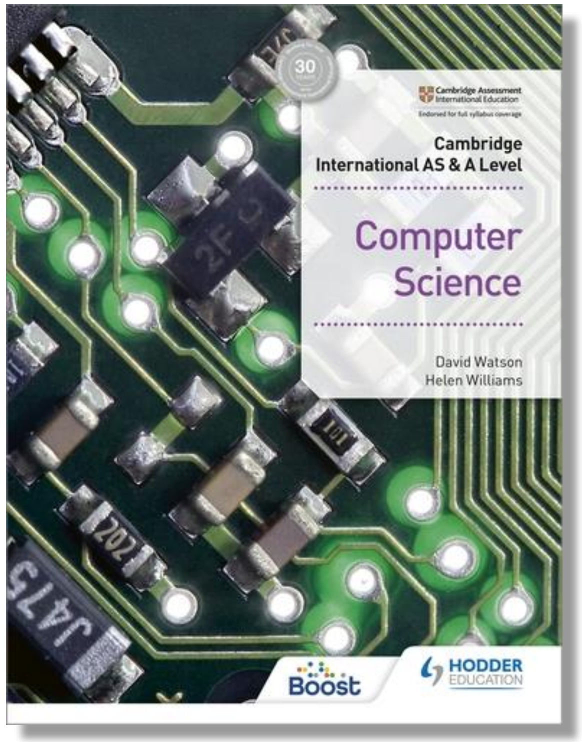
GCE AS LEVEL

In the context of CIE 9618 syllabus

CAMBRIDGE International Examinations

Chapter 4
Processor Fundamentals



4. Processor Fundamentals

LEARNING OBJECTIVES:

1. Von Neumann model of a computer system
2. The purpose and role of the registers PC, MDR, MAR, ACC, IX, CIR, and the status registers
3. The purpose and role of the arithmetic logic unit (ALU), control unit (CU), system clock and immediate access store (IAS)
4. Functions of the address bus, data bus and control bus
5. Factors affecting computer performance (such as processor type, bus width, clock speeds, cache memory and use of core processors)
6. The connection of computers to peripheral devices such as Universal Serial Bus (USB), high definition multimedia interface (HDMI) and Video Graphics Array (VGA)
7. The fetch-execute cycle and register transfers
8. The purpose of interrupts
9. The relationship between assembly language and machine code (such as symbolic, absolute and relative addressing)
10. Different stages for a two-pass assembler
11. Tracing sample assembly language programming code
12. Assembly language instruction groups (such as data movement, I/O operations, arithmetic operations, comparisons, and so on)
13. Addressing modes (immediate, direct, indirect, indexed and relative)
14. How to perform binary shifts (including logical, arithmetic, cyclic, left shift and right shift)
15. How bit manipulation is used to monitor/control a device.

4.1 Computers and their components

KEY TERMS: (1/4)

- Von Neumann architecture – computer architecture which introduced the concept of the stored program in the 1940s.
- Arithmetic logic unit (ALU) – component in the processor which carries out all arithmetic and logical operations.
- Control unit – ensures synchronisation of data flow and programs throughout the computer by sending out control signals along the control bus.
- System clock – produces timing signals on the control bus to ensure synchronisation takes place.
- Immediate access store (IAS) – holds all data and programs needed to be accessed by the control unit.
- Accumulator – temporary general purpose register which stores numerical values at any part of a given operation.
- Register – temporary component in the processor which can be general or specific in its use that holds data or instructions as part of the fetch-execute cycle.
- Status register – used when an instruction requires some form of arithmetic or logical processing.

4.1 Computers and their components

KEY TERMS: (2/4)

- Flag – indicates the status of a bit in the status register, for example, N = 1 indicates the result of an addition gives a negative value.
- Address bus – carries the addresses throughout the computer system.
- Data bus – allows data to be carried from processor to memory (and vice versa) or to and from input/output devices.
- Control bus – carries signals from control unit to all other computer components.
- Unidirectional – used to describe a bus in which bits can travel in one direction only.
- Bidirectional – used to describe a bus in which bits can travel in both directions.
- Word – group of bits used by a computer to represent a single unit.
- Clock cycle – clock speeds are measured in terms of GHz; this is the vibrational frequency of the clock which sends out pulses along the control bus – a 3.5 GHZ clock cycle means 3.5 billion clock cycles a second.

4.1 Computers and their components

KEY TERMS: (3/4)

- Overclocking – changing the clock speed of a system clock to a value higher than the factory/recommended setting.
- BIOS – basic input/output system.
- Cache memory – a high speed auxiliary memory which permits high speed data transfer and retrieval.
- Core – a unit made up of ALU, control unit and registers which is part of a CPU. A CPU may contain a number of cores.
- Dual core – a CPU containing two cores.
- Quad core – a CPU containing four cores.
- Port – external connection to a computer which allows it to communicate with various peripheral devices. A number of different port technologies exist.
- Universal Serial Bus (USB) – a type of port connecting devices to a computer.
- Asynchronous serial data transmission – serial refers to a single wire being used to transmit bits of data one after the other. Asynchronous refers to a sender using its own clock/timer device rather sharing the same clock/timer with the recipient device.

4.1 Computers and their components

KEY TERMS: (4/4)

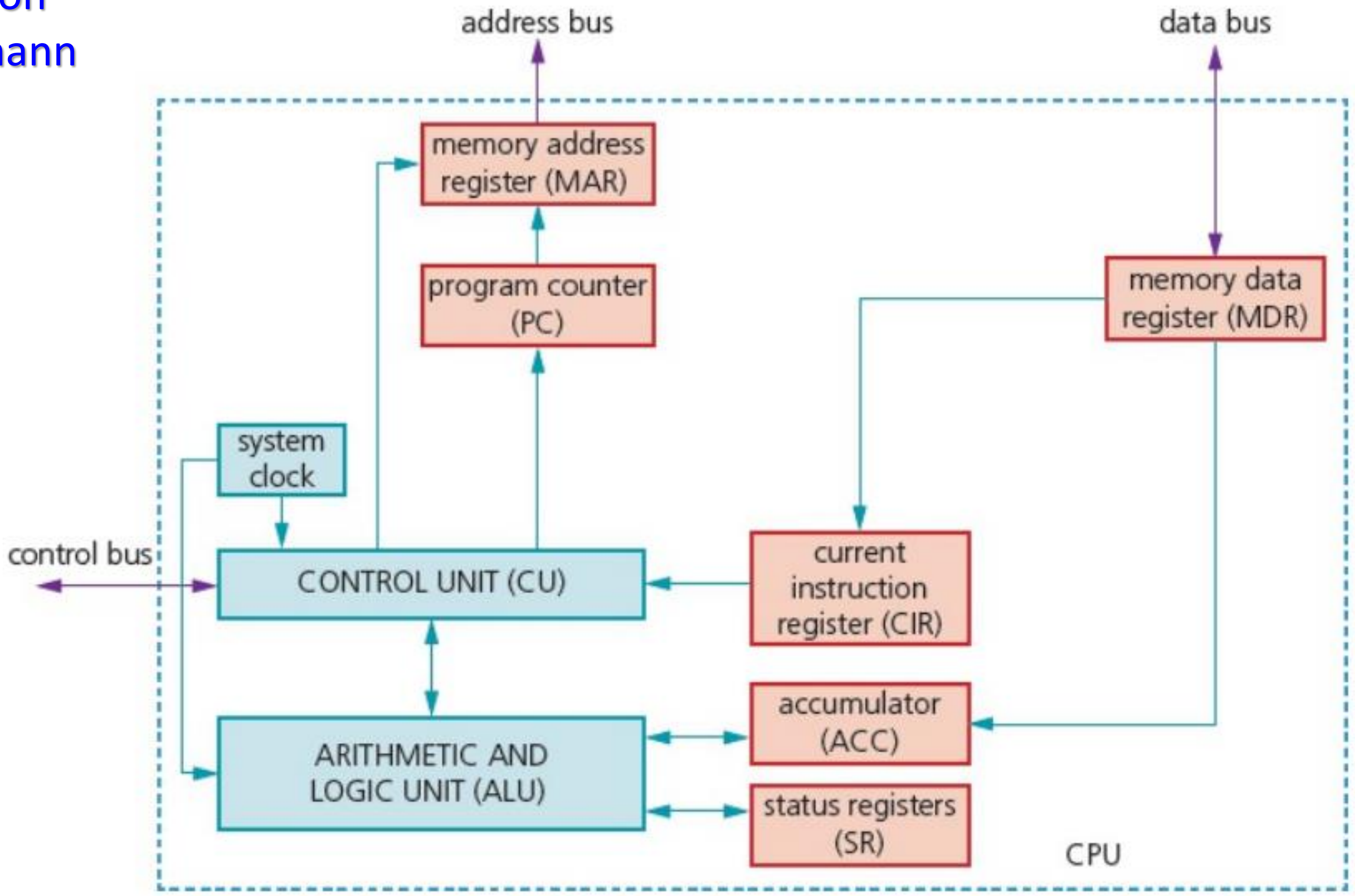
- High-definition multimedia interface (HDMI) – type of port connecting devices to a computer.
- Video Graphics Array (VGA) – type of port connecting devices to a computer. High-bandwidth digital copy protection (HDCP) – part of HDMI technology which reduces risk of piracy of software and multimedia.
- Fetch-execute cycle – a cycle in which instructions and data are fetched from memory and then decoded and finally executed.
- Program counter (PC) – a register used in a computer to store the address of the instruction which is currently being executed.
- Current instruction register – a register used to contain the instruction which is currently being executed or decoded.
- Register Transfer Notation (RTN) – short hand notation to show movement of data and instructions in a processor, can be used to represent the operation of the fetch-execute cycle.
- Interrupt – signal sent from a device or software to a processor requesting its attention; the processor suspends all operations until the interrupt has been serviced.
- Interrupt priority – all interrupts are given a priority so that the processor knows which need to be serviced first and which interrupts are to be dealt with quickly.
- Interrupt service routine (ISR) or interrupt handler – software which handles interrupt requests (such as 'printer out of paper') and sends the request to the CPU for processing.

4.1.1 Von Newmann Model

- Early computers: Fed data while running, no program or data storage, required human intervention.
- In the mid-1940s, John Von Neumann: Developed concept of stored program computer.
- Stored program computer: Basis of computer architecture for years.
- Von Neumann architecture: Main features -
  - Central Processing Unit (CPU): Also known as processor.
  - Processor can access memory directly.
  - Computer memories can store programs and data.
  - Stored programs: Comprised sequential instructions for execution.

4.1.1 Von Newmann Model

Representation of Von Neumann architecture


address bus
data bus
memory address register (MAR)
program counter (PC)
memory data register (MDR)
system clock
control bus
CONTROL UNIT (CU)
current instruction register (CIR)
accumulator (ACC)
ARITHMETIC AND LOGIC UNIT (ALU)
status registers (SR)
CPU

4.1.2 Components of the processor (CPU)

1. Arithmetic logic unit (ALU):

- ALU (Arithmetic Logic Unit):
  - Performs arithmetic and logic operations during program execution.
- Computer can have multiple ALUs:
  - One for fixed-point operations and another for floating-point operations.
- Multiplication and division:
  - Conducted through sequences of addition, subtraction, and left/right shifting operations.
- Accumulator (ACC):
  - Temporary register for ALU calculations.

4.1.2 Components of the processor (CPU)

2. Control unit (CU):

- Control Unit (CU):
  - Reads instruction from memory (address stored in Program Counter (PC)).
- Read instruction is interpreted
  - signals sent via control bus to guide other computer components.
- CU synchronizes data flow and program instructions across the computer.

4.1.2 Components of the processor (CPU)

3. System clock:

- System clock:
  - Generates timing signals on control bus for vital synchronisation.
- Clock essential:
  - Without it, computer would crash.

4.1.2 Components of the processor (CPU)

4. Immediate access store (IAS):

- IAS (Internal Storage):
  - Contains data and programs for CPU access.
- CPU transfers data and programs from backing store to IAS temporarily.
- IAS read/write operations faster than backing store.
- Key data stored temporarily in IAS to accelerate operations.
- IAS is primary (RAM) memory:
  - Also known as Internal Storage.

4.1.3 Registers

- Von Neumann system:
  - Includes registers as important parts.
- Registers:
  - Either general purpose or special purpose.
- General purpose:
  - Hold often-used data, like the accumulator.
- Special purpose:
  - Serve specific functions, holding program state.

4.1.3 Registers

Register	Abbreviation	Function/purpose of register
current instruction register	CIR	stores the current instruction being decoded and executed
index register	IX	used when carrying out index addressing operations (assembly code)
memory address register	MAR	stores the address of the memory location currently being read from or written to
memory data/buffer register	MDR /MBR	stores the address of the memory location currently being read from or written to
program counter	PC	stores the address where the next instruction to be read can be found
status register	SR	contain bits which can be set or cleared depending on the operation (for example, to indicate overflow in a calculation)

4.1.3 Registers

- All of the registers are used in the fetch-execute-cycle.
- Index registers are best explained when looking at addressing techniques in assembly code.
- A status register is used when an instruction requires some form of arithmetic or logic processing.
- Each bit is known as a flag.
- Most systems have the following four flags:
  - Carry flag (C)
    - is set to **1** if there is a CARRY following an addition operation
  - Negative flag (N)
    - is set to **1** if the result of a calculation yields a NEGATIVE value
  - Overflow flag (V)
    - is set to **1** if an arithmetic operation results in an OVERFLOW being produced
  - Zero flag (Z)
    is set to **1** if the result of an arithmetic or logic operation is ZERO

4.1.3 Registers

- Consider this arithmetic operation:

	0	1	1	1	0	1	1	1
+	0	0	1	1	1	0	0	0
	1	0	1	0	1	1	1	1

Flags:
N V C Z
1 1 0 0

- Since we have two positive numbers being added, the answer should not be negative.
- The flags indicate two errors:
  - a negative result
  - an overflow occurred.

4.1.3 Registers

- Now consider this operation:

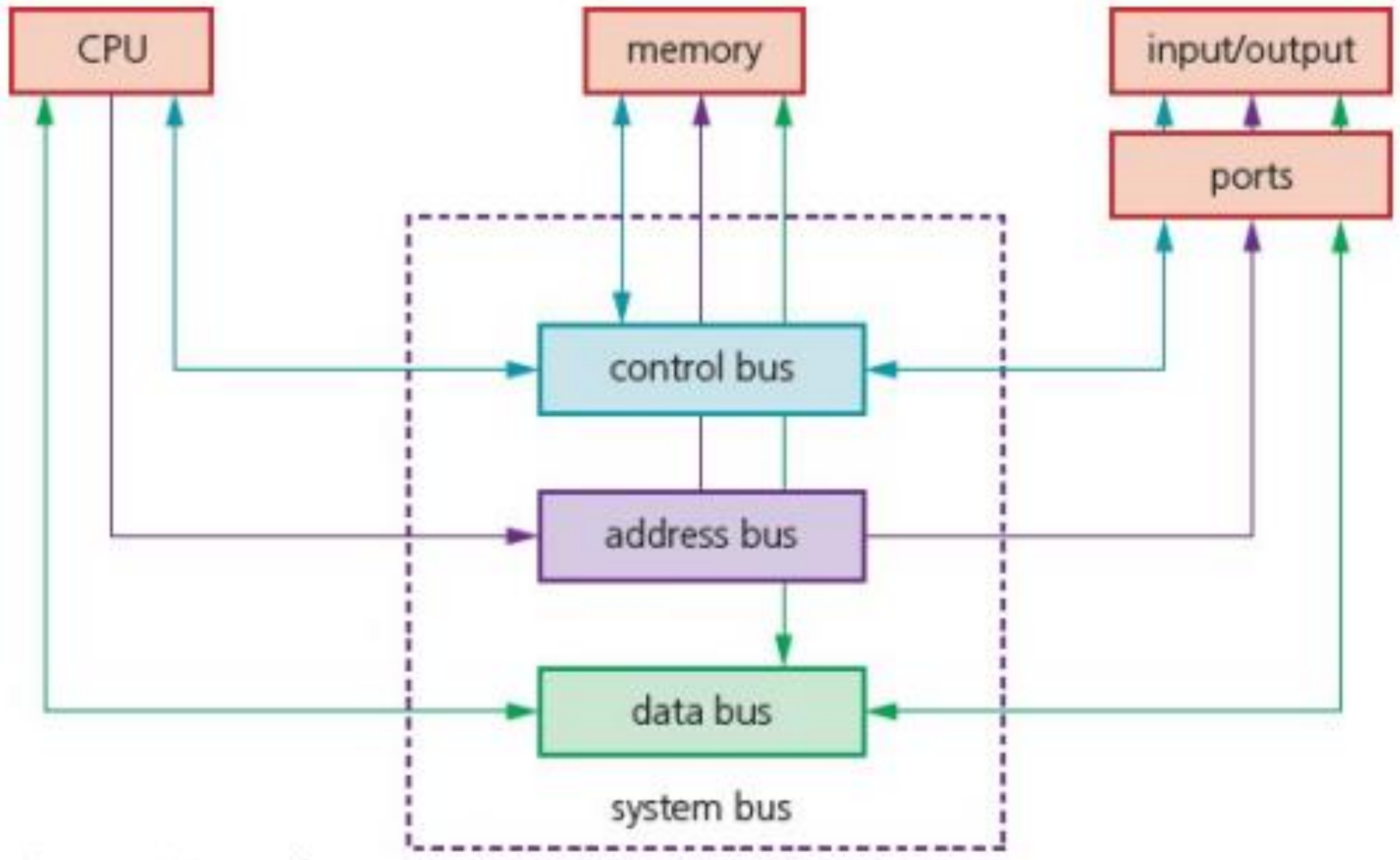
	1	0	0	0	1	0	0	0
+	1	1	0	0	0	1	1	1
1	0	1	0	0	1	1	1	1

Flags:
N V C Z
0 1 1 0

- Since we have two negative numbers being added, the answer should be negative.
- The flags indicate that two errors have occurred:
  - a carry has been generated
  - a ninth bit overflow has occurred.

4.1.4 System Buses

- System buses are used in computers as a parallel transmission component.
- Each wire in the bus transmits one bit of data.
- There are three common buses used in the Von Neumann architecture known as:
  - address bus
  - data bus
  - control bus



4.1.4 System Buses

1. Address Bus:

- Address bus:
  - Carries addresses in the computer system.
- Between CPU and memory:
  - Address bus is one-way to avoid issues (unidirectional).
- Bus width matters:
  - Wider bus means more memory locations can be addressed at once.
- Example:
  - 16-bit bus addresses 65,536 locations
  - 32-bit bus addresses 4,294,967,296 locations.

4.1.4 System Buses

2. Data Bus:

- Data bus:
  - Goes both ways (bidirectional).
- Moves data between CPU, memory, and input/output devices.
- Data can be:
  - address, instruction, or number.
- Data bus width matters:
  - Wider bus carries larger word lengths for better computer performance.

4.1.4 System Buses

3. Control Bus:

- The control bus is also bidirectional.
- It carries signals from the CU to all the other computer components.
- It is usually 8-bits wide since it only carries control signals.

4.1.4 System Buses

The role of the System Clock:

- Clock:
  - Sets the timing for computer tasks.
- Control bus:
  - Ensures synchronization.
- Higher clock speed:
  - Faster processing (like 3.5 GHz – billion cycles/sec).
- Faster clock ≠ Always better performance.

4.1.4 System Buses

Faster clock ≠ Always better performance:

Four other factors need to be considered:

1. Width of the address bus and data bus can affect computer performance.

2. Overclocking:
    - Change clock speed in BIOS settings.
    - Higher clock than design can cause issues:
        - Instructions exceed limits: Unsynced operations, crashes.
        - CPU overheating: Causes unreliable performance.

4.1.4 System Buses

Faster clock ≠ Always better performance:

Four other factors need to be considered:

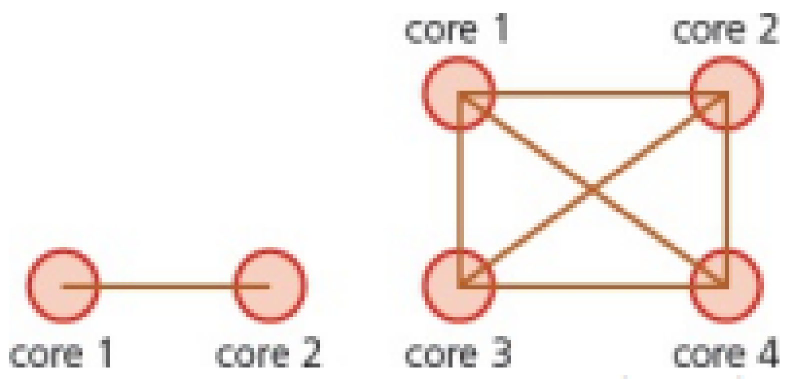
3. Cache Memory
   - Boosts processor performance.
   - Like RAM, lost when power's off.
   - Cache uses SRAM, faster than DRAM in main memory.
   - Faster access, no refreshing delay.
   - Processor checks cache first, then main memory.
   - Cache stores frequently used instructions/data, speeds up access.

4.1.4 System Buses

Faster clock ≠ Always better performance:

Four other factors need to be considered:

4. Cores:
    - More cores:
        - Improve computer performance.
    - Many have dual core (2 cores) or quad core (4 cores).
    - More cores reduce need to raise clock speeds.
    - Doubling cores ≠ Double performance:
        - CPU communication affects it.
    - Dual core:
        - One channel, CPU talks to both cores, reduces potential increase.
    - Quad core:
        - Six channels, CPU talks to all cores, reduces potential performance.



Two cores, one channel (left) and four cores, six channels (right)

4.1.5 Computer Ports

Computer Ports:

- Input and output devices:
  - Connected via ports.
- Control unit manages port interaction with devices.
- Commo

USB cable	HDMI cable	VGA cable

4.1.5 Computer Ports

USB ports:

- USB (Universal Serial Bus):
  - Method to transfer data between computer and devices.
- USB cable:
  - Four-wired shielded cable, power, earth, and data wires.
- Plugging in a device to a USB port:
  - Computer detects device: Voltage change on data wires.
  - Device recognized: Correct driver loaded for communication.
  - New device: Computer searches for matching driver; download if needed.

4.1.5 Computer Ports

USB ports:

ADVANTAGES	DISADVANTAGES
• Devices plugged in: Get detected and drivers loaded automatically. • Connectors fit only one way: Stops wrong connections. • Industry standard: Lots of support for users. • Different data rates supported. • Newer USB backward compatible: Works with older standards too.	• Current rate: Under 500 megabits per second. • Max cable length: About 5 meters. • Older USB standard: Might not be supported soon.

4.1.5 Computer Ports

High-definition multimedia interface (HDMI):

- HDMI ports: Send audio and video from computer to HDMI devices.
- Support high-definition signals: Enhanced or standard.
- Replaces VGA for better digital quality.
- Modern HD TVs have:
    - Widescreen format (16:9 ratio).
    - More pixels (usually 1920 x 1080).
    - Faster refresh rate (120 Hz or 120 frames/sec).
    - Wide range of colors (up to four million variations).
- HD TVs need more data faster (around 10 gigabits/sec).
- HDMI increases bandwidth: Provides data for sound and visuals.
- HDCP protection: Guards against piracy using authentication protocol.
- Example: Blu-ray player checks TV's key before transmitting data.

4.1.5 Computer Ports

High-definition multimedia interface (HDMI):

ADVANTAGES	DISADVANTAGES
• Current standard for TVs and monitors: • Fast data transfer. • Better security against piracy. • Works with modern digital systems.	• Connection not strong (breaks easily when moving). • Limited cable length for good signal. • Five cable/connection standards currently.

4.1.5 Computer Ports

Video Graphics Array (VGA):

- VGA introduced in late 1980s.
- VGA supports 640x480 pixels on screen.
- Refresh rate up to 60 Hz (60 frames/sec).
- For 16 colors, higher refresh rate.
- Lower pixel density (200x320) supports 256 colors.

4.1.5 Computer Ports

Video Graphics Array (VGA):

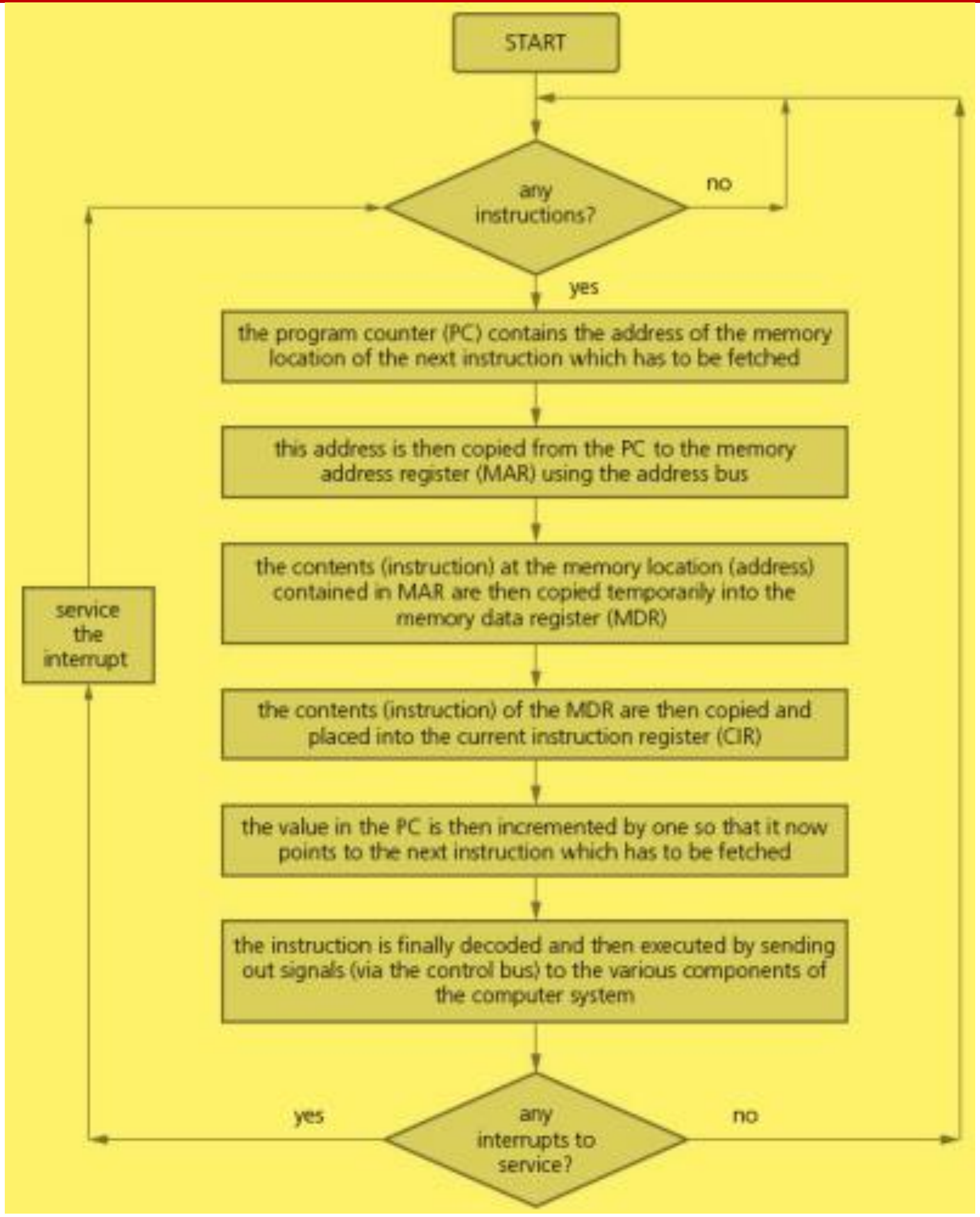
ADVANTAGES	DISADVANTAGES
• Simple technology.	• Old, outdated analog technology.
• Only one standard.	• Pins can bend easily when connecting.
• Easy to split signal for many devices.	• Need high-grade cables for clear signal.
• Very secure connection.	

4.1.6 Fetch-Execute Cycle

- Execution of Instructions:
  - Processor fetches data and instructions from memory, stores in registers.
  - Address bus and data bus used.
  - Instructions decoded before execution.
- Fetch:
  - Next instruction fetched from memory address in program counter (PC).
  - Stored in current instruction register (CIR).
  - PC incremented by 1 for next instruction.
  - Decoded for interpretation in next cycle.
- Execute:
  - Processor sends decoded instruction as control signals to system components.
  - Instructions carried out in logical sequence.

4.1.6 Fetch-Execute Cycle

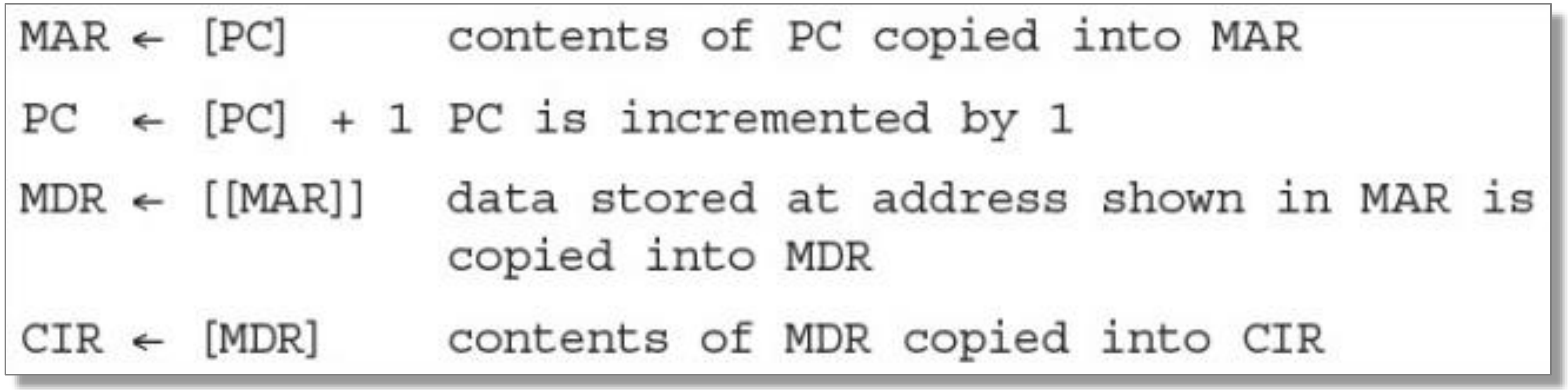
How the fetch-execute cycle is carried out in the Von Neumann computer model.



START
any instructions?
no
yes
the program counter (PC) contains the address of the memory location of the next instruction which has to be fetched
this address is then copied from the PC to the memory address register (MAR) using the address bus
the contents (instruction) at the memory location (address) contained in MAR are then copied temporarily into the memory data register (MDR)
the contents (instruction) of the MDR are then copied and placed into the current instruction register (CIR)
the value in the PC is then incremented by one so that it now points to the next instruction which has to be fetched
the instruction is finally decoded and then executed by sending out signals (via the control bus) to the various components of the computer system
service the interrupt
yes
any interrupts to service?
no

4.1.6 Fetch-Execute Cycle

- When registers are involved, it is possible to describe what is happening by using Register Transfer Notation (RTN).



MAR	←	[PC]	contents of PC copied into MAR
PC	←	[PC] + 1	PC is incremented by 1
MDR	←	[[MAR]]	data stored at address shown in MAR is copied into MDR
CIR	←	[MDR]	contents of MDR copied into CIR

- Double brackets are used in the third line because it is not MAR contents being copied into MDR but it is the data stored at the address shown in MAR that is being copied to MDR.

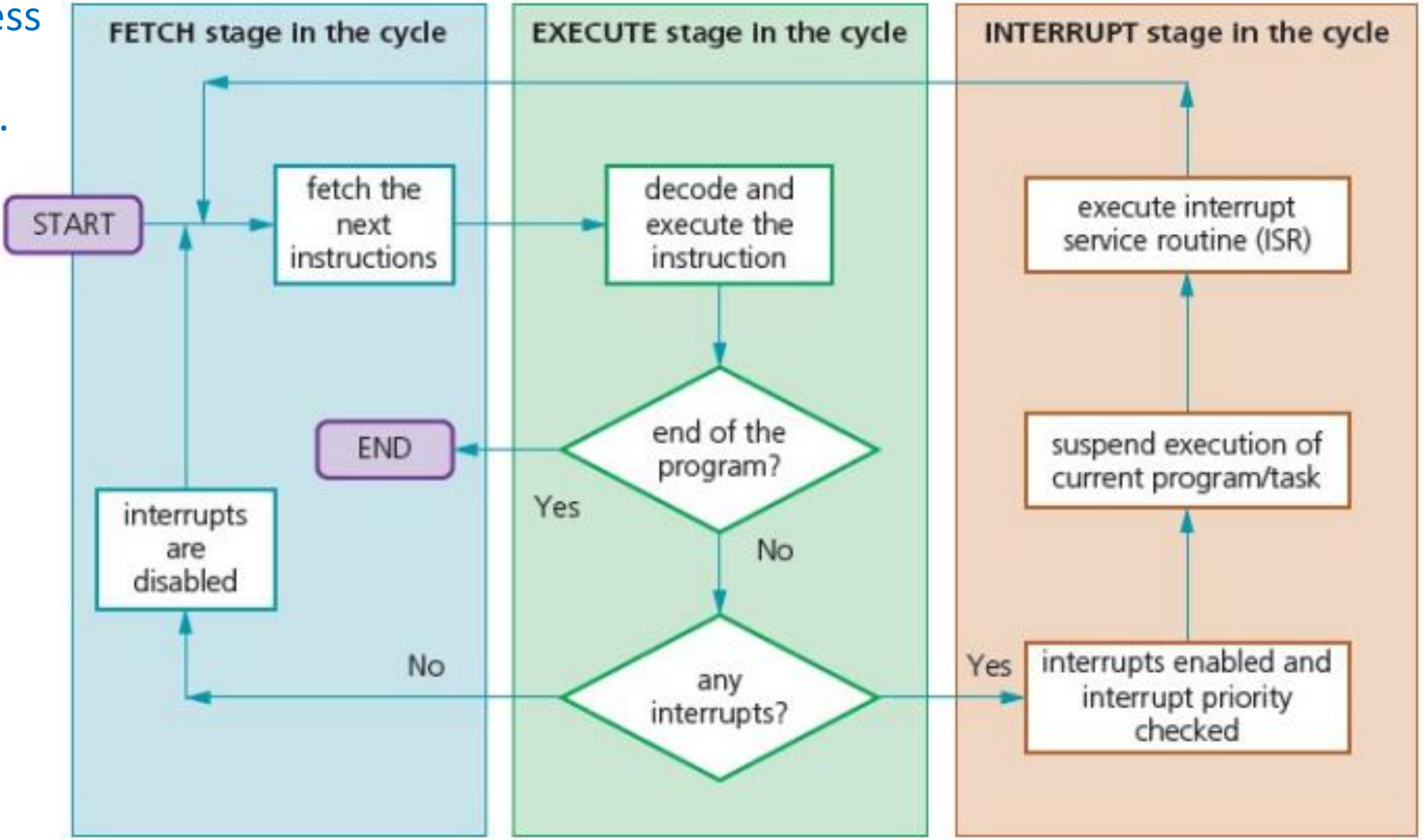
4.1.6 Fetch-Execute Cycle

How interrupts are specifically used in the fetch-execute cycle:

- Interrupt Register in fetch-execute cycle:
  - Interrupt can occur while CPU cycles.
  - Interrupt changes interrupt register bit status.
- Sequence when interrupt happens:
  - Next cycle: Interrupt register checked bit by bit.
  - Contents like 0000 1000 mean unresolved interrupt.
  - CPU services interrupt, depending on priority.
  - CPU halts task, saves register contents.
  - Control to interrupt handler.
  - After service, register reset, registers restored.

4.1.6 Fetch-Execute Cycle

The interrupt process during the fetch-execute cycle.


FETCH stage in the cycle
EXECUTE stage in the cycle
INTERRUPT stage in the cycle
START
fetch the next instructions
decode and execute the instruction
execute interrupt service routine (ISR)
END
end of the program?
Yes
No
suspend execution of current program/task
interrupts are disabled
No
any interrupts?
Yes
interrupts enabled and interrupt priority checked

4.1.7 Interrupts

- Interrupts in Simple Terms:
    - Signal from device or software to processor.
    - Makes processor pause and handle it.
- Interrupts can happen because of:
    - Timing signal.
    - Device ready (like a disk drive).
    - Hardware problem (like a printer jam).
    - User action (like pressing a key).
    - Software issue (like a missing file or a divide by zero).
- Processor then continues or stops to help.
- Interrupts let computers do many things at once.
- Example: Downloading while playing music.
- When handling an interrupt:
    - Save current task status.
    - Run interrupt routine.
    - After, go back to where it was before.

4.2 Assembly Language

KEY TERMS: (1/2)

- Machine code – the programming language that the CPU uses.
- Instruction – a single operation performed by a CPU.
- Assembly language – a low-level chip/machine specific programming language that uses mnemonics.
- Opcode – short for operation code, the part of a machine code instruction that identifies the action the CPU will perform.
- Operand – the part of a machine code instruction that identifies the data to be used by the CPU.
- Source code – a computer program before translation into machine code.
- Assembler – a computer program that translates programming code written in assembly language into machine code. Assemblers can be one pass or two pass.
- Instruction set – the complete set of machine code instructions used by a CPU.
- Object code – a computer program after translation into machine code.

4.2 Assembly Language

KEY TERMS: (2/2)

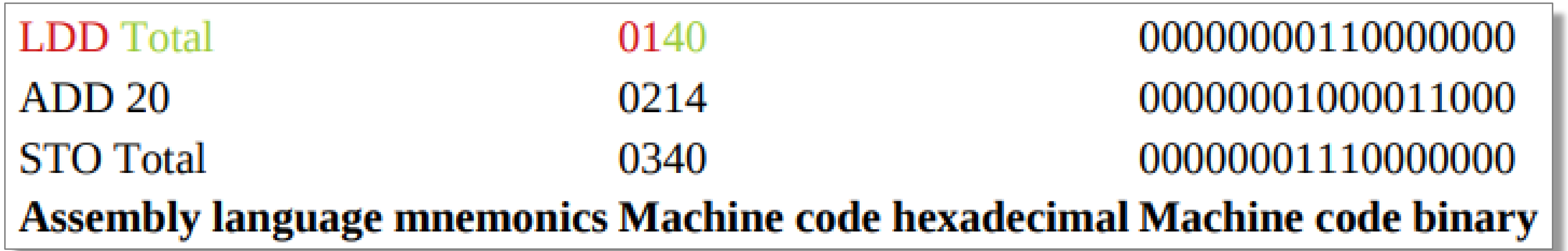
- Addressing modes – different methods of using the operand part of a machine code instruction as a memory address.
- Absolute addressing – mode of addressing in which the contents of the memory location in the operand are used.
- Direct addressing – mode of addressing in which the contents of the memory location in the operand are used, which is the same as absolute addressing.
- Indirect addressing – mode of addressing in which the contents of the contents of the memory location in the operand are used.
- Indexed addressing – mode of addressing in which the contents of the memory location found by adding the contents of the index register (IR) to the address of the memory location in the operand are used.
- Immediate addressing – mode of addressing in which the value of the operand only is used.
- Relative addressing – mode of addressing in which the memory address used is the current memory address added to the operand.
- Symbolic addressing – mode of addressing used in assembly language programming, where a label is used instead of a value.

4.2.1 Assembly Language and Machine Code

- Machine Code Basics:
  - CPU uses machine code language.
  - Each computer type has own instructions.
- Programs in Memory:
  - Computer program in memory is a series of instructions.
  - CPU follows these during fetch-execute cycle.
- Simple Tasks:
  - Each instruction does one simple task.
  - Like putting a value in a certain spot.
- Machine Code is Binary:
  - It's in 0s and 1s.
  - Sometimes shown as hexadecimal for programmers.
- Writing Machine Code:
  - Special task, takes lots of time.
  - Can be error-prone.
  - Testing is hard, need to run and see.
- Other Languages Developed:
  - Other languages easier for programmers.
- Translators Needed:
  - Programs not in machine code need to be translated.
  - Language translators were made for this.

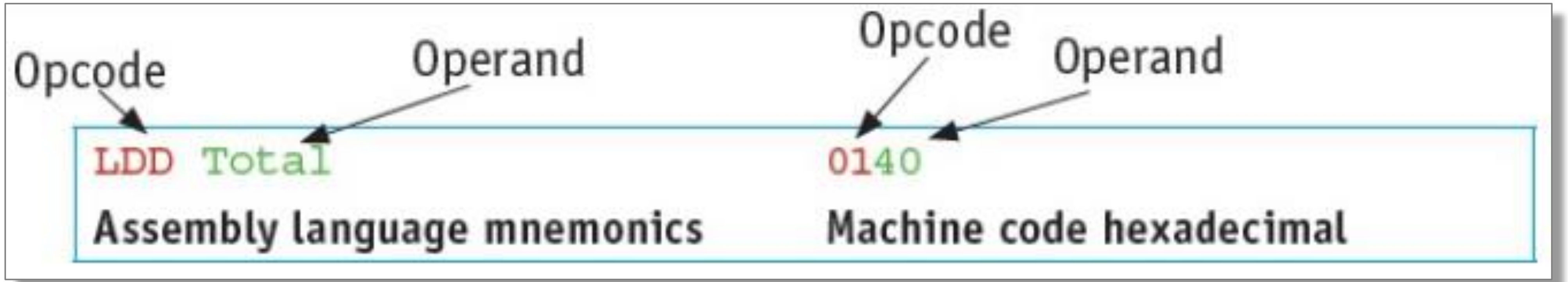
4.2.1 Assembly Language and Machine Code

- The first programming language to be developed was assembly language.
- This is closely related to machine code and uses mnemonics instead of binary.



LDD Total	0140	00000000110000000
ADD 20	0214	00000001000011000
STO Total	0340	00000001110000000
Assembly language mnemonics	**Machine code hexadecimal**	**Machine code binary**

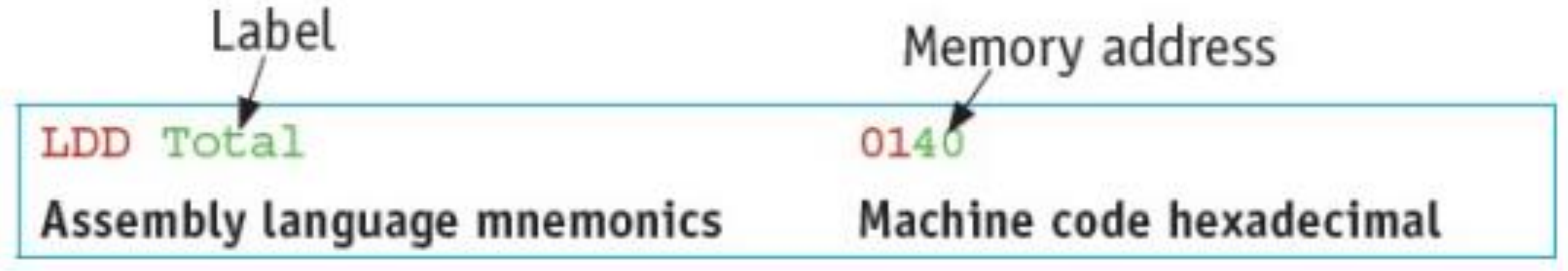
- The structure of assembly language and machine code instructions is the same.
- Each instruction has an opcode that identifies the operation to be carried out by the CPU.
- Most instructions also have an operand that identifies the data to be used by the opcode.



4.2.2 Stages of Assembly

- From Assembly to Machine Code:
  - Assembly code needs to become machine code.
  - Assembler program does the job.
- Assembler's Job:
  - Translates each assembly instruction to machine code.
  - Checks if assembly code is right.
  - Speeds up fixing errors before running.
- Types of Assemblers:
  - Single pass: Puts code in memory for direct use.
  - Two pass: Makes a machine code program, needs a loader.
- Two Pass Assembler:
  - Goes through source program twice.
  - Replaces labels with memory addresses.

4.2.2 Stages of Assembly



Pass 1

- Read the assembly language program one line at a time.
- Ignore anything not required, such as comments.
- Allocate a memory address for the line of code.
- Check the opcode is in the instruction set.
- Add any new labels to the symbol table with the address, if known.
- Place address of labelled instruction in the symbol table.

Pass 2

- Read the assembly language program one line at a time.
- Generate object code, including opcode and operand, from the symbol table generated in Pass 1.
- Save or execute the program.

4.2.2 Stages of Assembly

- The second pass is required as some labels may be referred to before their address is known.
- For example, Found is a forward reference for the JPN instruction.

```
Label     Opcode Operand
Notfound: LDD    200
          CMP    #0
          JPN    Found
          JPE    Notfound
Found:    OUT
```

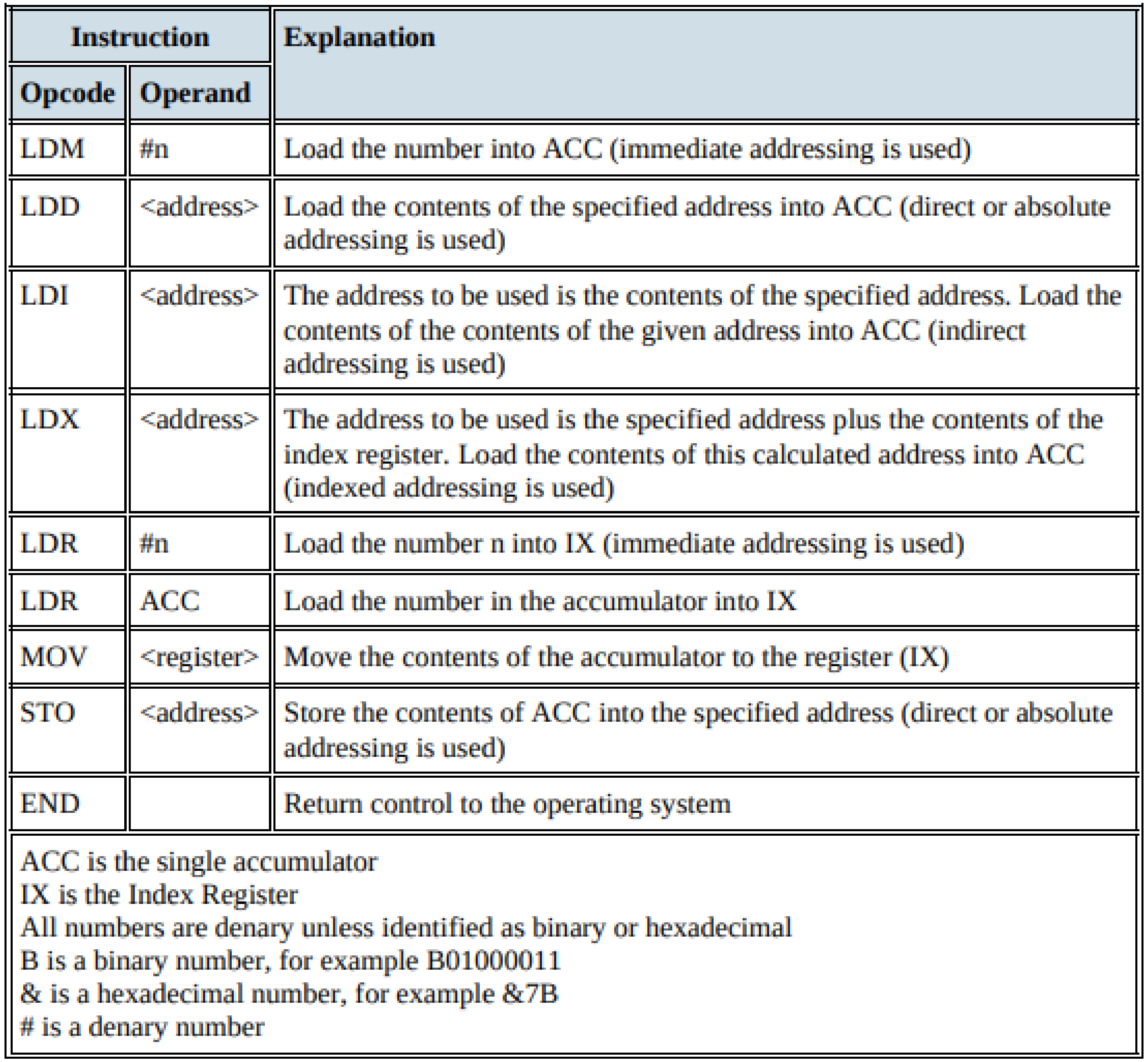
- If the program is to be loaded at memory address 100, and each memory location contains 16 bits, the symbol table for this small section of program would look like this:

Label	**Address**
Notfound	100
Found	104

4.2.3 Assembly Language Instructions

Data movement instructions:

- These instructions allow data stored at one location to be copied into the accumulator.
- This data can then be stored at another location, used in a calculation, used for a comparison or output.



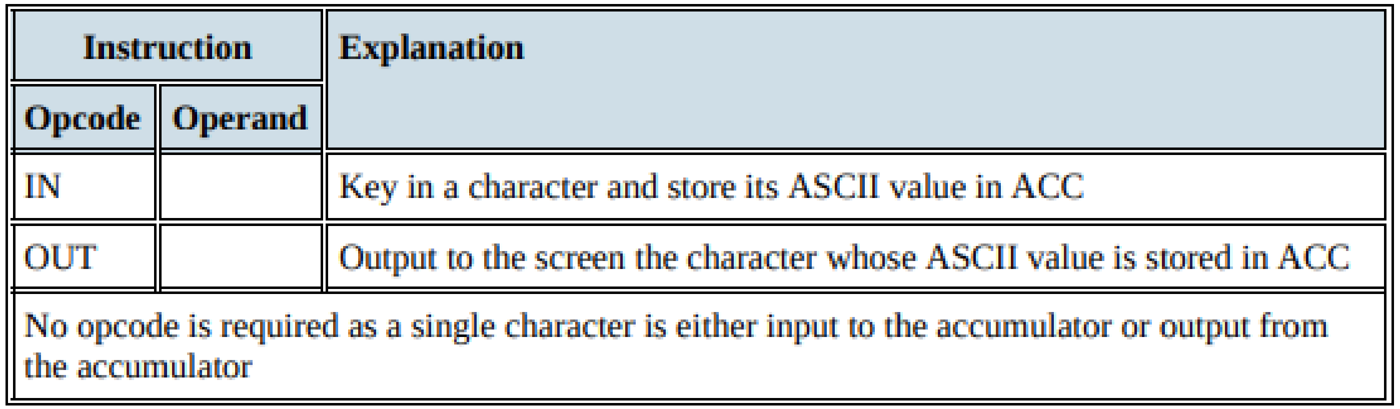
Instruction		Explanation
Opcode	**Operand**	
LDM	#n	Load the number into ACC (immediate addressing is used)
LDD	<address>	Load the contents of the specified address into ACC (direct or absolute addressing is used)
LDI	<address>	The address to be used is the contents of the specified address. Load the contents of the contents of the given address into ACC (indirect addressing is used)
LDX	<address>	The address to be used is the specified address plus the contents of the index register. Load the contents of this calculated address into ACC (indexed addressing is used)
LDR	#n	Load the number n into IX (immediate addressing is used)
LDR	ACC	Load the number in the accumulator into IX
MOV	<register>	Move the contents of the accumulator to the register (IX)
STO	<address>	Store the contents of ACC into the specified address (direct or absolute addressing is used)
END		Return control to the operating system

ACC is the single accumulator
IX is the Index Register
All numbers are denary unless identified as binary or hexadecimal
B is a binary number, for example B01000011
& is a hexadecimal number, for example &7B
# is a denary number

4.2.3 Assembly Language Instructions

Input and output of data instructions:

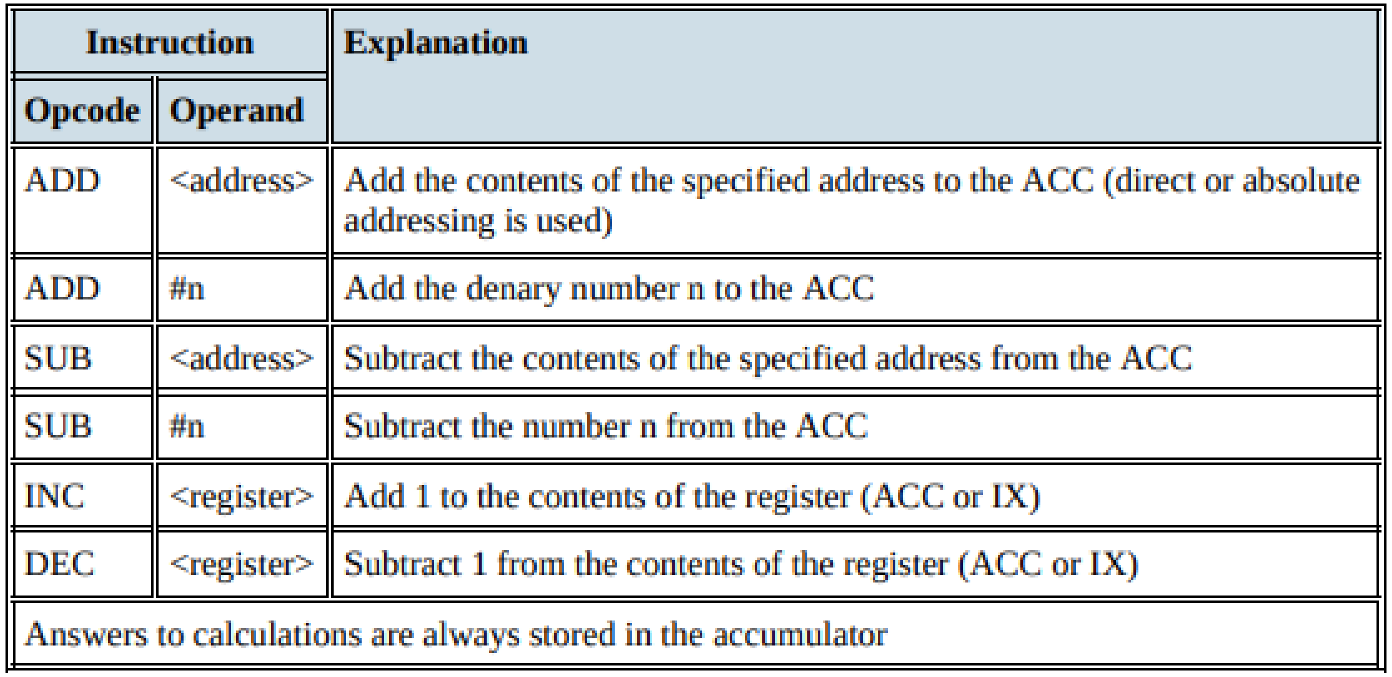
- These instructions allow data to be read from the keyboard or output to the screen.



Instruction		Explanation
Opcode	Operand	
IN		Key in a character and store its ASCII value in ACC
OUT		Output to the screen the character whose ASCII value is stored in ACC
No opcode is required as a single character is either input to the accumulator or output from the accumulator		

Arithmetic operation instructions:

- These instructions perform simple calculations on data stored in the accumulator and store the answer in the accumulator, overwriting the original data.



Instruction		Explanation
Opcode	Operand	
ADD	<address>	Add the contents of the specified address to the ACC (direct or absolute addressing is used)
ADD	#n	Add the denary number n to the ACC
SUB	<address>	Subtract the contents of the specified address from the ACC
SUB	#n	Subtract the number n from the ACC
INC	<register>	Add 1 to the contents of the register (ACC or IX)
DEC	<register>	Subtract 1 from the contents of the register (ACC or IX)
Answers to calculations are always stored in the accumulator		

4.2.3 Assembly Language Instructions

Unconditional and conditional instructions:

Instruction		Explanation
Opcode	Operand	
JMP	<address>	Jump to the specified address
JPE	<address>	Following a compare instruction, jump to the specified address if the comparison is True
JPN	<address>	Following a compare instruction, jump to the specified address if the comparison is False
END		Returns control to the operating system
Jump means change the PC to the address specified, so the next instruction to be executed is the one stored at the specified address, not the one stored at the next location in memory		

Compare instructions:

Instruction		Explanation
Opcode	Operand	
CMP	<address>	Compare the contents of ACC with the contents of the specified address (direct or absolute addressing is used)
CMP	#n	Compare the contents of ACC with the number n
CMI	<address>	The address to be used is the contents of the specified address; compare the contents of the contents of the given address with ACC (indirect addressing is used)
The contents of the accumulator are always compared		

4.2.4 Addressing Modes

Assembly language and machine code programs use different addressing modes depending on the requirements of the program:

- Absolute addressing:
  - The **contents** of the **memory location** in the **operand** are used.
  - For example, if the **memory location** with **address 200** contained the **value 20**, the assembly language instruction **LDD 200** would **store 20** in the **accumulator**.
- Direct addressing:
  - The **contents** of the **memory location** in the **operand** are used.
  - For example, if the **memory location** with **address 200** contained the **value 20**, the assembly language instruction **LDD 200** would **store 20** in the **accumulator**.
  - **Absolute** and **direct addressing** are the **same**.
- Indirect addressing:
  - The **contents** of the **contents** of the **memory location** in the **operand** are used.
  - For example, if the **memory location** with **address 200** contained the **value 20** and the **memory location** with **address 20** contained the **value 5**, the assembly language instruction **LDI 200** would **store 5** in the **accumulator**.

4.2.4 Addressing Modes

Assembly language and machine code programs use different addressing modes depending on the requirements of the program:

- Indexed addressing:
  - The **contents** of the **memory location** found by **adding** the **contents** of the **index register** (**IR**) to the **address** of the **memory location** in the **operand** are used.
  - For example, if **IR** contained the **value 4** and **memory location** with **address 204** contained the **value 17**, the assembly language instruction **LDX 200** would **store 17** in the **accumulator**.
- Immediate addressing:
  - The **value** of the **operand** only is used.
  - For example, the assembly language instruction **LDM #200** would **store 200** in the **accumulator**.
- Relative addressing:
  - The **memory address** used is the **current memory address** added to the **operand**.
  - For example, **JMR #5** would **transfer control** to the **instruction 5 locations after** the **current instruction**.

4.2.4 Addressing Modes

Assembly language and machine code programs use different addressing modes depending on the requirements of the program:

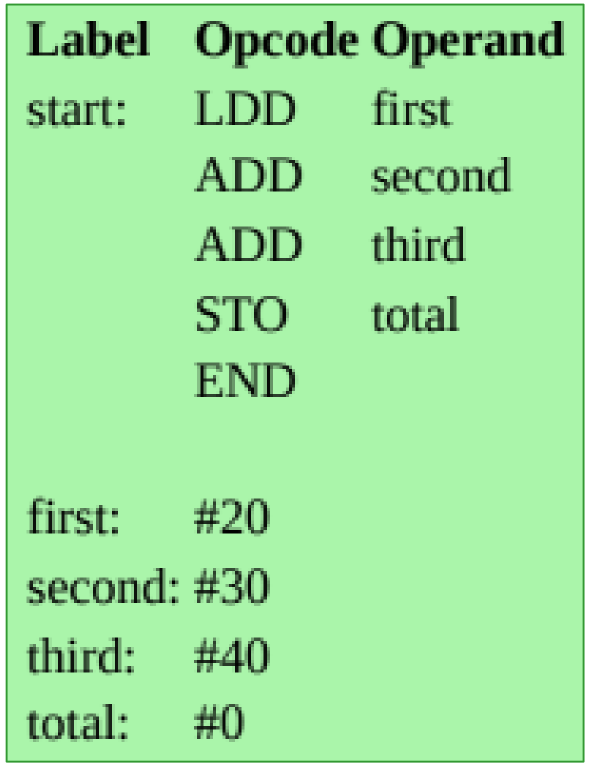
- Symbolic addressing:
    - Only used in **assembly language programming**.
    - A **label** is used **instead** of a **value**.
    - For example, if the **memory location** with **address** labelled **MyStore** contained the **value 20**, the assembly language instruction **LDD MyStore** would **store 20** in the **accumulator**.

4.2.5 Simple assembly language programs

- A program written in assembly language will need many more instructions than a program written in a high-level language to perform the same task.
- In a high-level language, adding three numbers together and storing the answer would typically be written as a single instruction:

    total = first + second + third

- The same task written in assembly language could look like this:

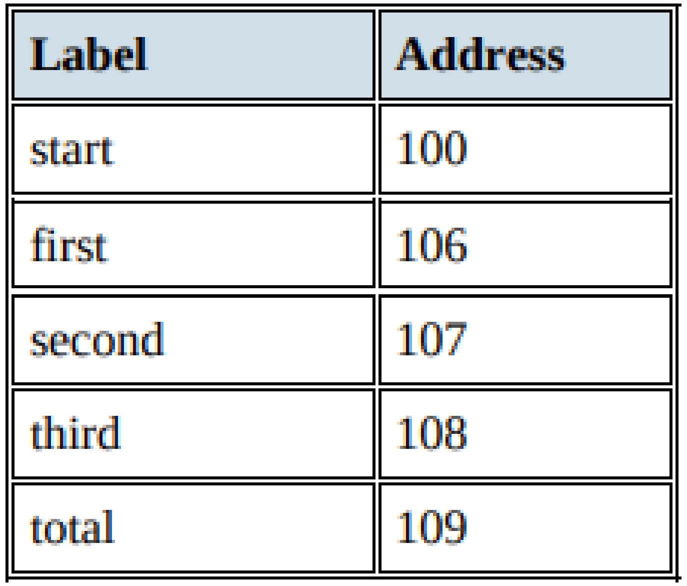


```
Label   Opcode Operand
start:  LDD    first
        ADD    second
        ADD    third
        STO    total
        END

first:  #20
second: #30
third:  #40
total:  #0
```

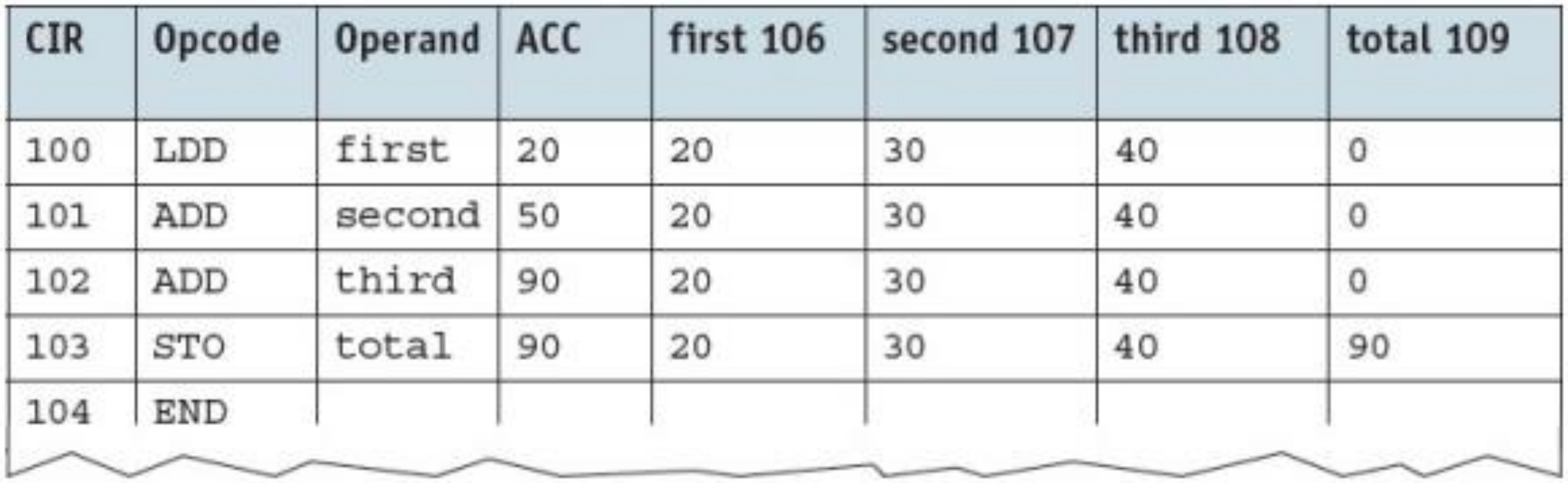
4.2.5 Simple assembly language programs

- If the program is to be loaded at memory address 100 after translation and each memory location contains 16 bits, the symbol table for this small section of program would look like this:



Label	Address
start	100
first	106
second	107
third	108
total	109

- When this section of code is executed, the contents of ACC, CIR and the variables used can be traced using a trace table.

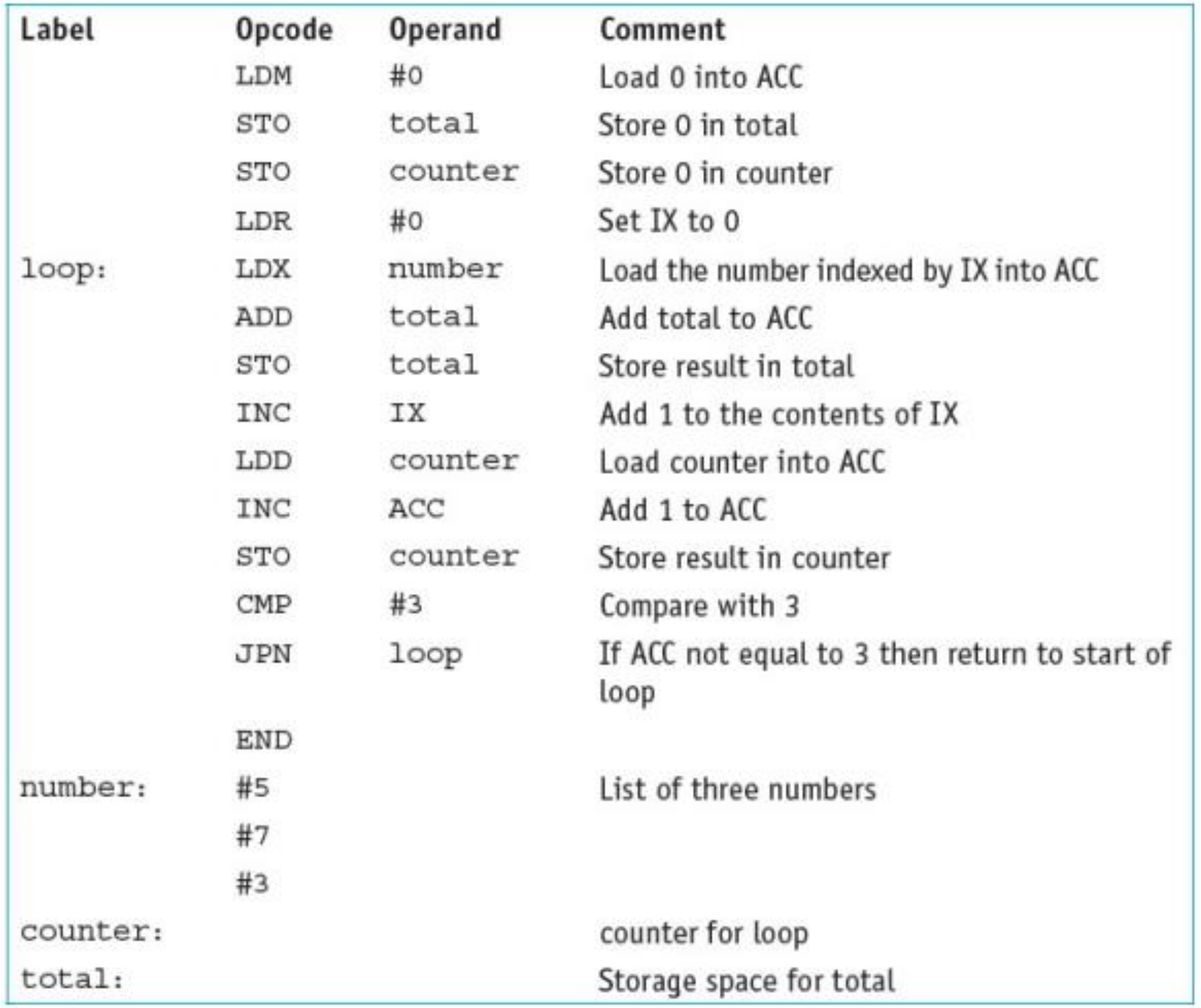


CIR	Opcode	Operand	ACC	first 106	second 107	third 108	total 109
100	LDD	first	20	20	30	40	0
101	ADD	second	50	20	30	40	0
102	ADD	third	90	20	30	40	0
103	STO	total	90	20	30	40	90
104	END						

4.2.5 Simple assembly language programs

- In a high-level language, adding a list of numbers together and storing the answer would typically be written using a loop.
- The same task written in assembly language would require the use of the index register (IX).
- The assembly language program could look like this:

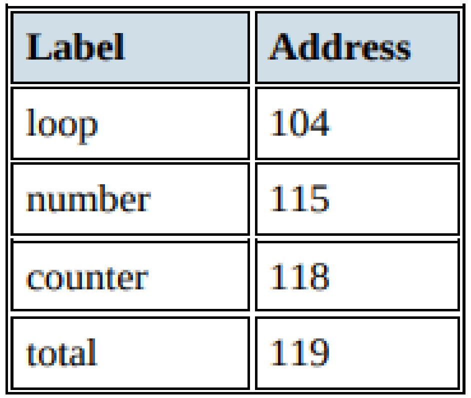
```
FOR counter = 1 TO 3
    total = total + number[counter]
NEXT counter
```



Label	Opcode	Operand	Comment
	LDM	#0	Load 0 into ACC
	STO	total	Store 0 in total
	STO	counter	Store 0 in counter
	LDR	#0	Set IX to 0
loop:	LDX	number	Load the number indexed by IX into ACC
	ADD	total	Add total to ACC
	STO	total	Store result in total
	INC	IX	Add 1 to the contents of IX
	LDD	counter	Load counter into ACC
	INC	ACC	Add 1 to ACC
	STO	counter	Store result in counter
	CMP	#3	Compare with 3
	JPN	loop	If ACC not equal to 3 then return to start of loop
	END		
number:	#5		List of three numbers
	#7		
	#3		
counter:			counter for loop
total:			Storage space for total

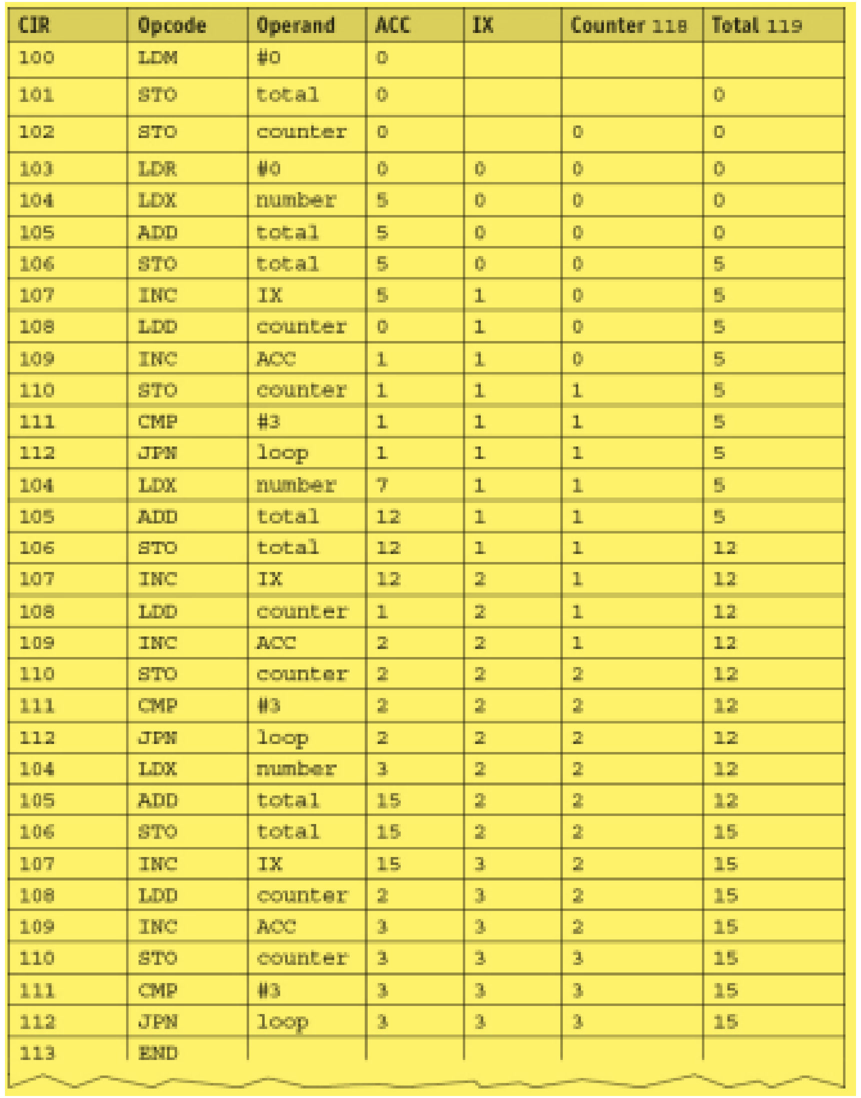
4.2.5 Simple assembly language programs

- If the program is to be loaded at memory address 100 after translation and each memory location contains 16 bits, the symbol table for this small section of program would look like this:



Label	Address
loop	104
number	115
counter	118
total	119

- When this section of code is executed the contents of ACC, CIR, IX and the variables used can be traced using a trace table:



CIR	Opcode	Operand	ACC	IX	Counter 118	Total 119
100	LDM	#0	0			
101	STO	total	0			0
102	STO	counter	0		0	0
103	LDR	#0	0	0	0	0
104	LDX	number	5	0	0	0
105	ADD	total	5	0	0	0
106	STO	total	5	0	0	5
107	INC	IX	5	1	0	5
108	LDD	counter	0	1	0	5
109	INC	ACC	1	1	0	5
110	STO	counter	1	1	1	5
111	CMP	#3	1	1	1	5
112	JPN	loop	1	1	1	5
104	LDX	number	7	1	1	5
105	ADD	total	12	1	1	5
106	STO	total	12	1	1	12
107	INC	IX	12	2	1	12
108	LDD	counter	1	2	1	12
109	INC	ACC	2	2	1	12
110	STO	counter	2	2	2	12
111	CMP	#3	2	2	2	12
112	JPN	loop	2	2	2	12
104	LDX	number	3	2	2	12
105	ADD	total	15	2	2	12
106	STO	total	15	2	2	15
107	INC	IX	15	3	2	15
108	LDD	counter	2	3	2	15
109	INC	ACC	3	3	2	15
110	STO	counter	3	3	3	15
111	CMP	#3	3	3	3	15
112	JPN	loop	3	3	3	15
113	END					

4.3 Bit Manipulation

KEY TERMS:

- Shift – moving the bits stored in a register a given number of places within the register; there are different types of shift.
- Logical shift – bits shifted out of the register are replaced with zeros.
- Arithmetic shift – the sign of the number is preserved.
- Cyclic shift – no bits are lost, bits shifted out of one end of the register are introduced at the other end of the register.
- Left shift – bits are shifted to the left.
- Right shift – bits are shifted to the right.
- Monitor – to automatically take readings from a device.
- Control – to automatically take readings from a device, then use the data from those readings to adjust the device.
- Mask – a number that is used with the logical operators AND, OR or XOR to identify, remove or set a single bit or group of bits in an address or register.

4.3.1 Binary Shifts

- Shift in Registers:
    - Move bits within a register.
    - Change positions by given number.
- Different Bit Uses:
    - Bits in register serve various purposes.
- Example - IR Register:
    - Each bit in IR identifies different interrupt.

4.3.1 Binary Shifts

Types of Shifts:

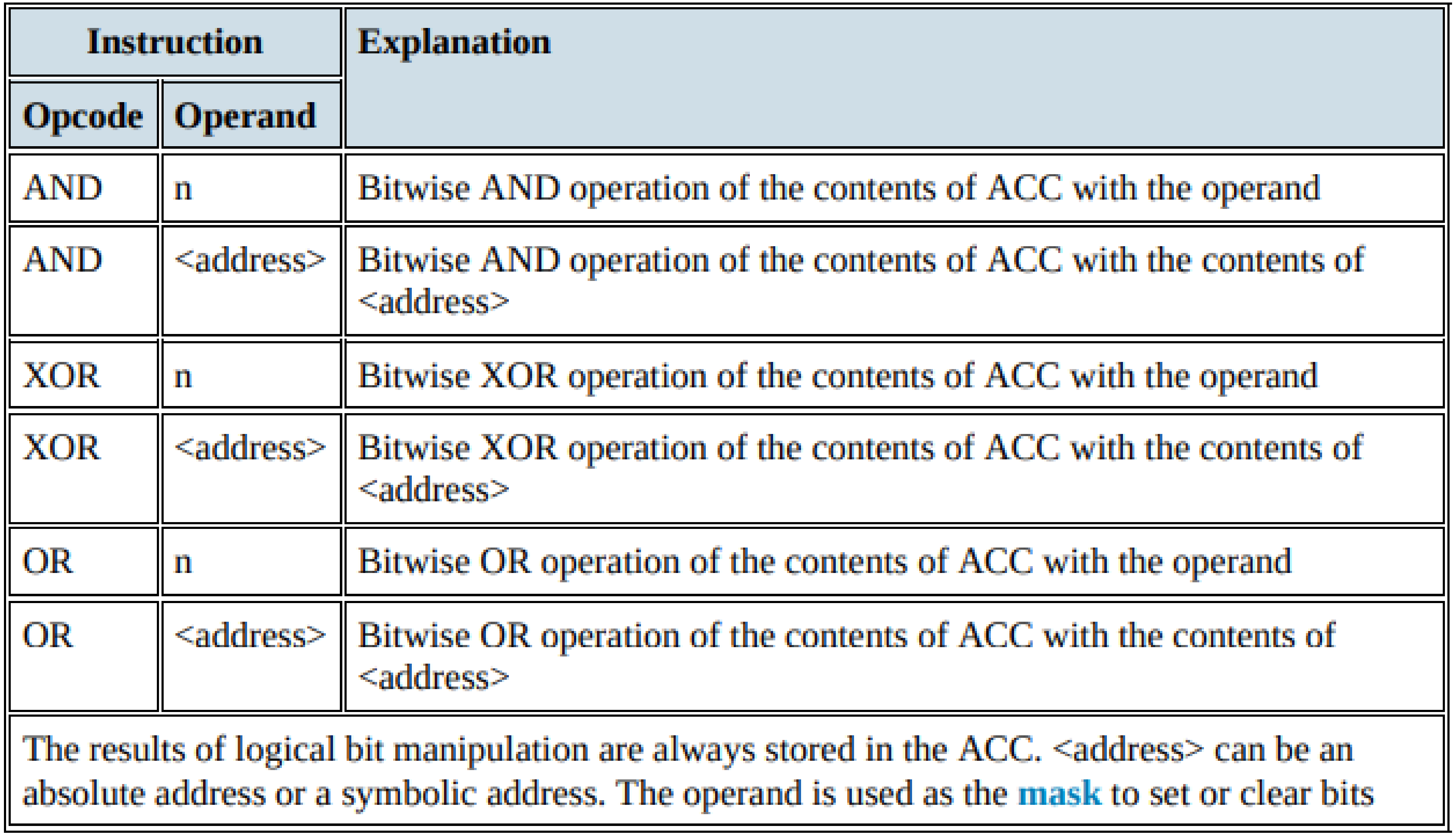
- Logical Shift: Bits out, replaced with zeros.
  - Example: 10101111 -> Shift left 3 -> 01111000.
- Arithmetic Shift: Keeps sign, shifts.
  - Example: 10101111 -> Shift right 3 -> 11110101.
  - Used for multiplication/division by powers of two.
- Cyclic Shift: No bits lost.
  - Bits out, wrap around to other end.
  - Example: 10101111 -> Shift left 3 -> 01111101.
- Left Shift:
  - Bits shifted left.
  - Direction for logical, arithmetic, cyclic shifts.
- Right Shift:
  - Bits shifted right.
  - Direction for logical, arithmetic, cyclic shifts.

4.3.2 Bit manipulation used in monitoring and control

- Monitoring and Control:
  - Bits as Flags: Test, set, or clear separately.
- Example - Sensor Data:
  - Eight sensors: Track processed data.
  - 8 bits in one memory location.
- Operations for Flags:
  - AND: Checks if bit is set.
  - OR: Sets the bit.
  - XOR: Clears set bit.

4.3.2 Bit manipulation used in monitoring and control

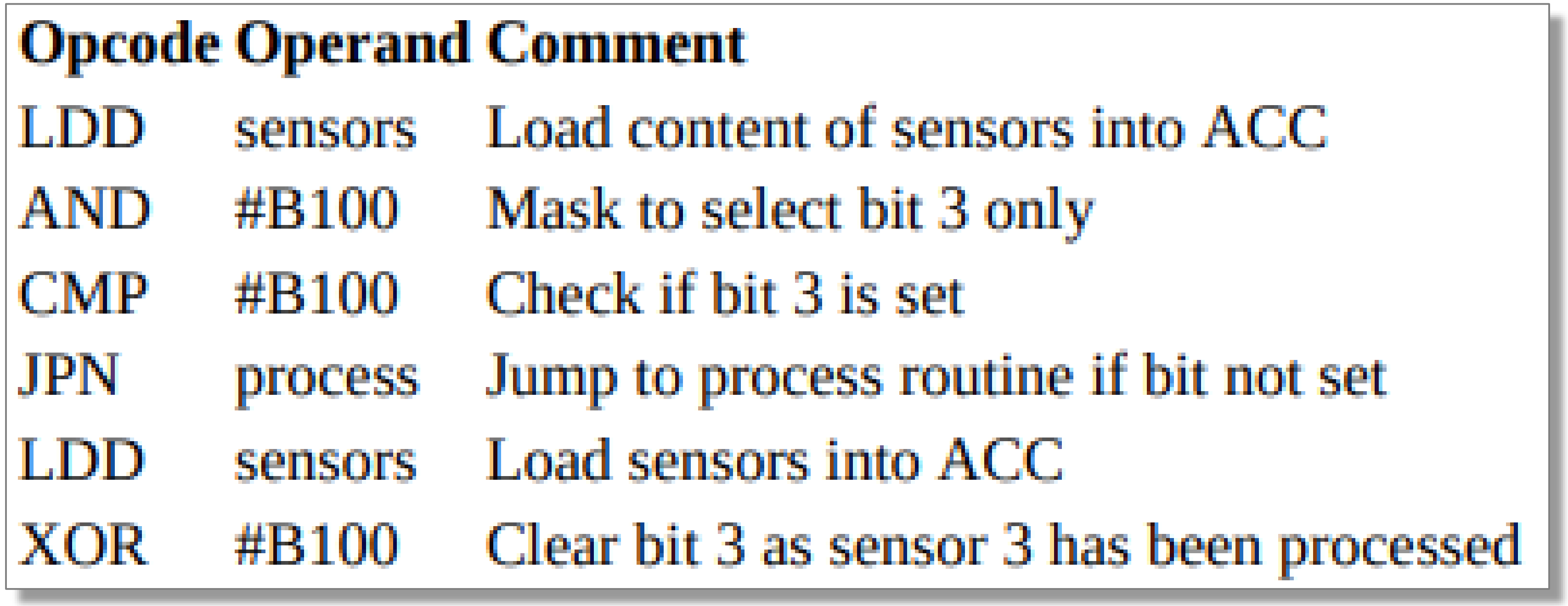
Instructions used to check, set and clear a single bit or group of bits:



Instruction		Explanation
Opcode	Operand	
AND	n	Bitwise AND operation of the contents of ACC with the operand
AND	<address>	Bitwise AND operation of the contents of ACC with the contents of <address>
XOR	n	Bitwise XOR operation of the contents of ACC with the operand
XOR	<address>	Bitwise XOR operation of the contents of ACC with the contents of <address>
OR	n	Bitwise OR operation of the contents of ACC with the operand
OR	<address>	Bitwise OR operation of the contents of ACC with the contents of <address>
The results of logical bit manipulation are always stored in the ACC. <address> can be an absolute address or a symbolic address. The operand is used as the mask to set or clear bits		

4.3.2 Bit manipulation used in monitoring and control

The assembly language code to test sensor 3 could be:



Opcode	Operand	Comment
LDD	sensors	Load content of sensors into ACC
AND	#B100	Mask to select bit 3 only
CMP	#B100	Check if bit 3 is set
JPN	process	Jump to process routine if bit not set
LDD	sensors	Load sensors into ACC
XOR	#B100	Clear bit 3 as sensor 3 has been processed