# ICS 3202:Assembly Language Programming

Lec: Mourice Ojijo

School of Computing and Engineering Sciences,Strathmore University

August 19, 2024

# Course content

Strathmore UNIVERSITY

1. Assembly language and processors
2. Assembly language concepts
3. Instruction set
4. Assembler directives
5. Macros
6. Programming techniques

- Assembly language learning helps in understanding the processor and memory functions. If the programmer is writing any program that needs to be a compiler, that means the programmer should have a complete understanding of the processor. Assembly language helps in understanding the work of processors and memory.
- Learning assembly language will teach you about the inner workings of the computer. You'll learn how the CPU/CPU registers work with memory addresses to achieve the end result one instruction at a time. This doesn't mean that you are going to begin coding everything in assembly.

# Assembler

Strathmore UNIVERSITY

- Almost every line of source coding in an assembly language source program translates directly into a machine.
- Therefore, the assembly language programmer must be familiar with both the assembly language and the processor for which he/she is programming.
  - **WHAT IS AN ASSEMBLER?**:An assembler is a software tool - a program – designed to simplify the task of writing computer programs.
  - Assembly language operation codes (opcodes) are easily remembered (MOV for move instructions, JMP for jump)
  - You can also symbolically express addresses and values referenced in the operand field of instructions

# Assembler

## What the assembler does

- To use the assembler, you first need a source program. The source program consists of programmer written assembly.
- Assembly language source programs must be in a machine-readable form when passed to the assembler
- Assembly language operation codes (opcodes) are easily remembered (MOV for move instructions, JMP for jump)
- The assembler program performs the clerical task of translating symbolic code into object code which can be executed by the 8080 and 8085 microprocessors.
- Assembler output consists of three possible files:
  - the object file containing your program translated into object code
  - the list file printout of your source code
  - the assembler generated object code, and the symbol table

# Assembler

## What the assembler does

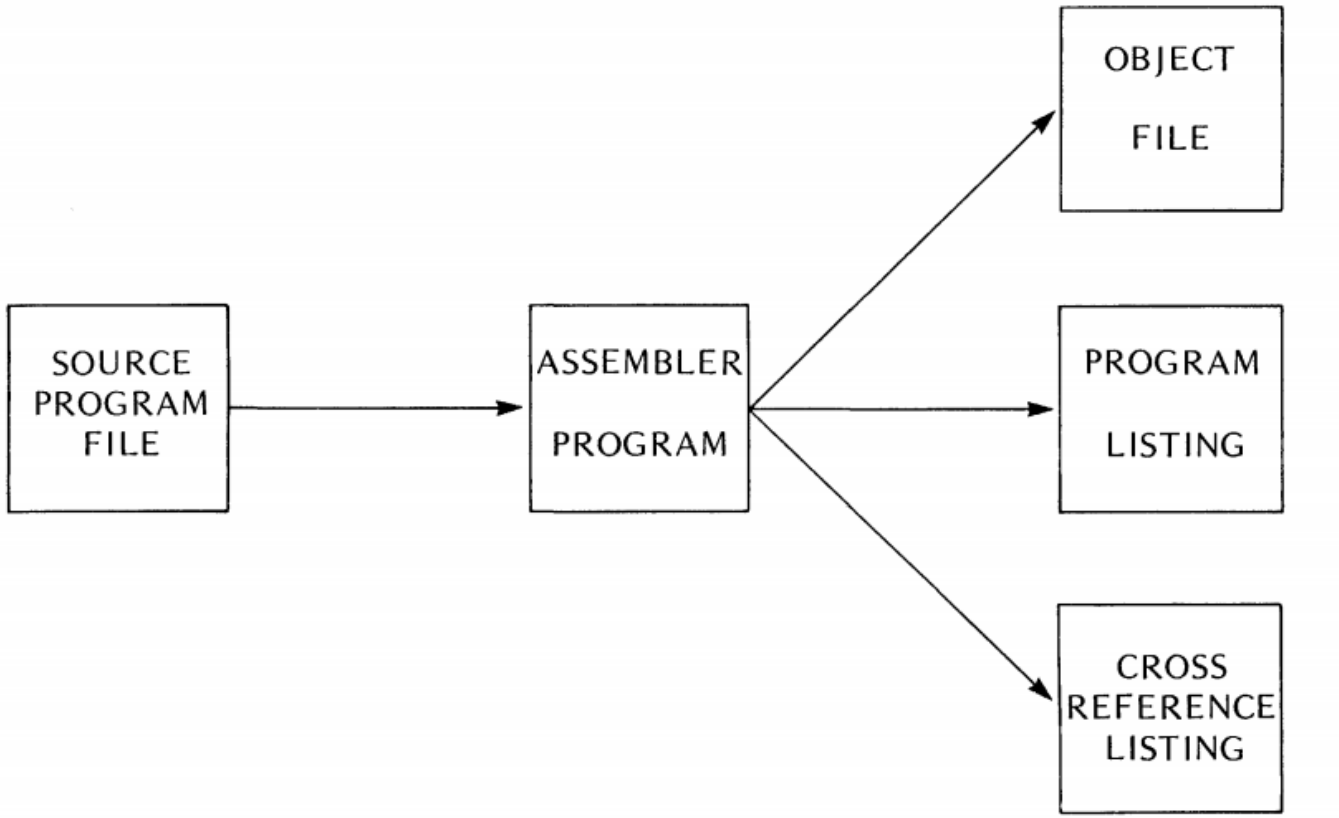


Figure 1: Assembler outputs

# Assembler

## Assembler outputs

- **Object cod**e:Object code is produced when an interpreter or a compiler translates source code into recognizable and executable machine code
- An assembler translates source code into object code, which is stored in object files.Object file examples include common object file format (COFF), COM files and ".exe" files.
- **The program listing** provides a permanent record of both the source program and the object code.
- The assembler also provides diagnostic messages for common programming errors in the program listing.
- **The symbol-cross-reference listing** is another of the diagnostic tools provided by the assembler
- Assume, for example, that your program manipulates a data field named DATE, and that testing reveals a program logic error in the handling of this data.

# 8085 Over view

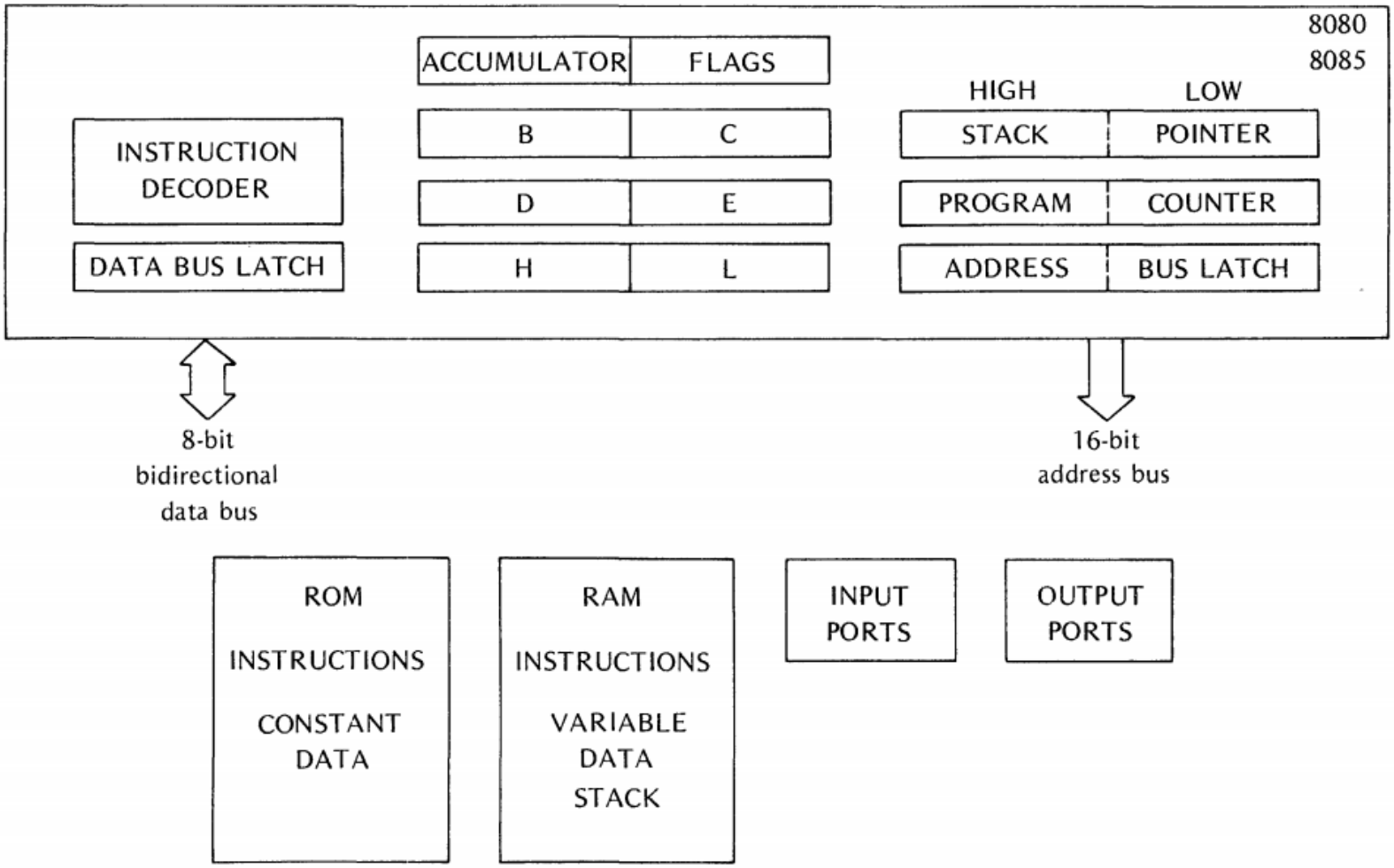


Figure 2: 8085 overview

# Work registers

- The 8085 provides an 8-bit accumulator and six other general purpose work registers, as shown in Figure 2.
- Programs reference these registers by the letters A (for the accumulator), B, C, D, E, H, and L. Thus, the instruction ADD B may be interpreted as 'add the contents of the B register to the contents of the accumulator
- Some instructions reference a pair of registers as shown in the following:

| *Symbolic Reference* | *Registers Referenced* |
|---|---|
| B | B and C |
| D | D and E |
| H | H and L |
| M | H and L (as a memory reference) |
| PSW | A and condition flags (explained later in this section) |

Figure 3: Work registers

# Conditional Flags

- **Carry Flag**:As its name implies, the carry flag is commonly used to indicate whether an addition causes a 'carry' into the next higher order digit. The carry flag is also used as a 'borrow' flag in subtractions, as explained under 'Two's Complement Representation of Data'
- The carry flag is also affected by the logical AND, OR, and exclusive OR instructions. These instructions set ON or OFF particular bits of the accumulator. See the descriptions of the ANA, ANI, ORA, ORI, XRA, and XRI

Example:

Addition of two one-byte numbers can produce a carry out of the high-order bit:

```
Bit Number:    7654 3210
AE=            1010 1110
+74=           0111 0100

               0010 0010 = 22 carry flag = 1
```

An addition that causes a carry out of the high order bit sets the carry flag to 1; an addition that does not cause a carry resets the flag to zero.

# Conditional Flags

- **Sign Flag**:A zero in bit 7 indicates a positive value; a one indicates a negative value. This value is duplicated in the sign flag so that conditional jump, call, and return instructions can test for positive and negative values
- **Zero Flag**:Certain instructions set the zero flag to one to indicate that the result in the accumulator contains all zeros. These instructions, reset the flag to zero if the result in the accumulator is other than zero. A result that has a carry and a zero result also sets the zero bit as shown below:

```
 1010 0111
+0101 1001

 0000 0000    Carry Flag = 1
              Zero Flag  = 1
```

Figure 5: Zero Flag

# Conditional Flags

Strathmore UNIVERSITY

- **Parity Flag**:Parity is determined by counting the number of one bits set in the result in the accumulator. Instructions that affect the parity flag set the flag to one for even parity and reset the flag to zero to indicate odd parity.
- Auxilliary Carry Flag: The auxiliary carry flag indicates a carry out of bit 3 of the accumulator. You cannot test this flag directly in your program; it is present to enable the DAA (Decimal Adjust Accumulator) to perform its function.
- **The auxiliary carry flag** and the DAA instruction allow you to treat the value in the accumulator as two 4-bit binary coded decimal numbers. Thus, the value 0001 1001 is equivalent to 19. (If this value is interpreted as a binary number, it has the value 25.) Notice, however, that adding one to this value produces a non-decimal result:

# Conditional Flags

- **The auxiliary carry Flag cont..**:Notice, however, that adding one to this value produces a non-decimal result:

```
 0001 1001
+0000 0001

 0001 1010 = 1A
```

The DAA instruction converts hexadecimal values such as the A in the preceding example back into binary coded decimal (BCD) format. The DAA instruction requires the auxiliary carry flag since the BCD format makes it possible for arithmetic operations to generate a carry from the low-order 4-bit digit into the high-order 4-bit digit. The DAA performs the following addition to correct the preceding example:

```
 0001 1010
+0000 0110

 0001 0000
+0001 0000  (auxiliary carry)

 0010 0000 = 20
```

Figure 6: auxiliary carry Flag

The auxiliary carry flag is affected by all add, subtract, increment, decrement, compare, and all logical AND, OR, and exclusive OR

# Stack pointer

Strathmore UNIVERSITY

- To understand the purpose and effectiveness of the stack, it is useful to understand the concept of a subroutine
- Assume that your program requires a multiplication routine. (Since the 8085 has no multiply instructions, this can be performed through repetitive addition.
- For example, 3x4 is equivalent to 3+3+3+3.) Assume further that your program needs this multiply routine several times.
- You can recode this routine inline each time it is needed, but this can use a great deal of memory. Or, you can code a subroutine:

# Stack and Subroutine

Strathmore UNIVERSITY

- To understand the purpose and effectiveness of the stack, it is useful to understand the concept of a subroutine
- Assume that your program requires a multiplication routine. (Since the 8085 has no multiply instructions, this can be performed through repetitive addition.
- For example, 3x4 is equivalent to 3+3+3+3.) Assume further that your program needs this multiply routine several times.
- You can recode this routine inline each time it is needed, but this can use a great deal of memory. Or, you can code a subroutine:
- When the call instruction is executed, the address of the next instruction (the contents of the program counter) is pushed onto the stack.
- The contents of the program counter are replaced by the address of the desired subroutine.

# Stack and subroutine

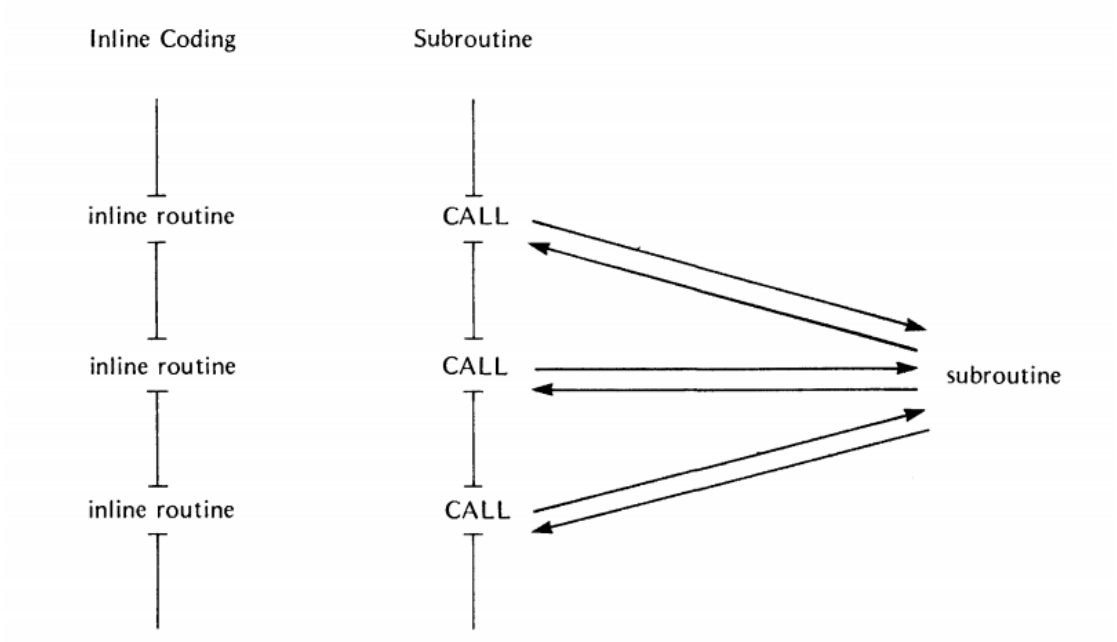


Figure 7: Subroutine and stack

-At the end of the subroutine, a return instruction pops that previously-stored address off the stack and puts it back into the program counter. Program execution then continues as though the subroutine had been coded inline

# Stack Operation

- The mechanism that makes subroutine possible is, of course, the stack. **The stack** is simply an area of random access memory addressed by the stack pointer.
- The stack pointer is a hardware register maintained by the processor. However, your program must initialize the stack pointer
- This means that your program must load the base address of the stack into the stack pointer.
- The base address of the stack is commonly assigned to the highest available address in RAM.
- This is because the stack expands by decrementing the stack pointer
- As items are added to the stack, it expands into memory locations with lower addresses. As items are removed from the stack, the stack pointer is incremented back toward its base address.

# Stack Operation

Strathmore UNIVERSITY

- Stack operations transfer sixteen bits of data between memory and a pair of processor registers. The two basic operations are PUSH, which adds data to the stack, and POP, which removes data from the stack.
- A call instruction pushes the contents of the program counter (which contains the address of the next instruction) onto the stack and then transfers control to the desired subroutine by placing its address in the program counter
- A return instruction pops sixteen bits off the stack and places them in the program counter. This requires the programmer to keep track of what is in the stack.
- If you call a subroutine and the subroutine pushes data onto the stack, the subroutine must remove that data before executing a return instruction.

# Saving Program Status

Strathmore UNIVERSITY

- It is likely that a subroutine requires the use of one or more of the working registers.
- However, it is equally likely that the main program has data stored in the registers that it needs when control returns to the main program.
- As general rule, a subroutine should save the contents of a register before using it and then restore the contents of that register before returning control to the main program.
- The subroutine can do this by pushing the contents of the registers onto the stack and then popping the data back into the registers before executing a return.
- Notice that the POP instructions must be in the opposite order of the PUSH instructions if the data is to be restored to its original location.

# Saving Program Status

```
SUBRTN:     PUSH    PSW
            PUSH    B
            PUSH    D
            PUSH    H

            subroutine coding

            POP     H
            POP     D
            POP     B
            POP     PSW
            RETURN
```

The letters B, D, and H refer to the B and C, D and E, and H and L register pairs, respectively. PSW refers to the program status word. The program status word is a 16-bit word comprising the contents of the accumulator and the five condition flags. (PUSH PSW adds three bits of filler to expand the condition flags into a full byte; POP PSW strips out these filler bits.)

Figure 8: Saving program status

# Input/Output Ports

- The 256 input/output ports provide communication with the outside world of peripheral devices. The IN and OUT instructions initiate data transfers
- The IN instruction latches the number of the desired port onto the address bus. As soon as a byte of data is returned to the data bus latch, it is transferred into the accumulator
- The OUT instruction latches the number of the desired port onto the address bus and latches the data in the accumulator onto the data bus
- The specified port number is duplicated on the address bus. Thus, the instruction IN 5 latches the bit configuration 0000 0101 0000 0101 onto the address bus.
- Notice that the IN and OUT instructions simply initiate a data transfer. It is the responsibility of the peripheral device to detect that it has been addressed.

# Instruction Set

## Addressing modes

- The 8080 incorporates a powerful array of instructions. This section provides a general overview of the instruction set.
- **Implied Addressing**. The addressing mode of certain instructions is implied by the instruction's function. For example, the STC (set carry flag) instruction deals only with the carry flag; the DAA (decimal adjust accumulator) instruction deals with the accumulator
- **Register Addressing.** Quite a large set of instructions call for register addressing. With these instructions, you must specify one of the registers A through E, H or L as well as the operation code. With these instructions, the accumulator is implied as a second operand. For example, the instruction CMP E may be interpreted as 'compare the contents of the E register with the contents of the accumulator.'

# Addressing modes

- **Immediate Addressing**. Instructions that use immediate addressing have data assembled as a part of the instruction itself. The instruction CPI C may be interpreted as 'compare the contents of the accumulator with C.Others include MVI D,OFFH, LXI SP,30FFH
- **Direct Addressing**. Jump instructions include a 16-bit address as part of the instruction. For example, the instruction JMP 1000H causes a jump to the hexadecimal address 1000 by replacing the current contents of the program counter with the new value 1000
- **Register Indirect Addressing**. Register indirect instructions reference memory via a register pair. Thus, the instruction MOV M,C moves the contents of the C register into the memory address stored in the H and L register pair. The instruction LDAX B loads the accumulator with the byte of data specified by the address in the Band C register pair

# Addressing modes

Strathmore UNIVERSITY

- **Combined Addressing Modes**. Some instructions use a combination of addressing modes. A CALL instruction, for example, combines direct addressing and register indirect addressing. The direct address in a CALL instruction specifies the address of the desired subroutine; the register indirect address is the stack pointer. The CALL instruction pushes the current contents of the program counter into the memory location specified by the stack pointer.
- Timing Effects of Addressing Modes. Addressing modes affect both the amount of time required for executing an instruction and the amount of memory required for its storage. For example, instructions that use implied or register addressing execute very quickly since they deal directly with the processor hardware or with data already present in hardware registers.

# Instruction Naming Conventions

The mnemonics assigned to the instructions are designed to indicate the function of the instruction.

- Data Transfer Group. The data transfer instructions move data between registers or between memory and registers.
  - MOV: Move
    MVI: Move Immediate
    LDA: Load Accumulator Directly from Memory
    STA: Store Accumulator Directly in Memory
    LHLD: Load Hand L Registers Directly from Memory
    SHLD: Store Hand L Registers Directly in Memory
  - An 'X' in the name of a data transfer instruction implies that it deals with a register pair:
    LXI :Load Register Pair with Immediate data
    LDAX: Load Accumulator from Address in Register Pair
    STAX: Load Accumulator from Address in Register Pair
    XCHG: The contents of DE are exchanged with HL
    XTHL: Exchange H and L with Top of Stack

# Instruction Naming Conventions

**Arithmetic Group.** The arithmetic instructions add, subtract, increment, or decrement data in registers or memory

| | |
|---|---|
| ADD | Add to Accumulator |
| ADI | Add Immediate Data to Accumulator |
| ADC | Add to Accumulator Using Carry Flag |
| ACI | Add Immediate Data to Accumulator Using Carry Flag |
| SUB | Subtract from Accumulator |
| SUI | Subtract Immediate Data from Accumulator |
| SBB | Subtract from Accumulator Using Borrow (Carry) Flag |
| SBI | Subtract Immediate from Accumulator Using Borrow |
| INR | Increment Specified Byte by One |
| DCR | Decrement Specified Byte by One |
| INX | Increment Register Pair by One |
| DCX | Decrement Register Pair by One |
| DAD | Double Register Add: Add Contents of Register Pair to H and L Register Pair |

# Instruction Naming Conventions

Strathmore UNIVERSITY

**Logical Group**. This group performs logical (Boolean) operations on data in registers and memory and on condition flags

The logical AND, OR, and Exclusive OR instructions enable you to set specific bits in the accumulator ON or OFF.

| | |
|---|---|
| ANA | Logical AND with Accumulator |
| ANI | Logical AND with Accumulator Using Immediate Data |
| ORA | Logical OR with Accumulator |
| ORI | Logical OR with Accumulator Using Immediate Data |
| XRA | Exclusive Logical OR with Accumulator |
| XRI | Exclusive OR Using Immediate Data |

The compare instructions compare the contents of an 8-bit value with the contents of the accumulator:

| | |
|---|---|
| CMP | Compare |
| CPI | Compare Using Immediate Data |

# Instruction Naming Conventions

Strathmore UNIVERSITY

**Rotate Instructions**. The rotate instructions shift the contents of the accumulator one bit position to the left or right:

The rotate instructions shift the contents of the accumulator one bit position to the left or right:

| | |
|---|---|
| RLC | Rotate Accumulator Left |
| RRC | Rotate Accumulator Right |
| RAL | Rotate Left Through Carry |
| RAR | Rotate Right Through Carry |

Complement and carry flag instructions:

| | |
|---|---|
| CMA | Complement Accumulator |
| CMC | Complement Carry Flag |
| STC | Set Carry Flag |

# Instruction Naming Conventions

**Branch Group**. The branching instructions alter normal sequential program flow, either unconditionally or conditionally. The unconditional branching instructions are as follows:

| | |
|---|---|
| JMP | Jump |
| CALL | Call |
| RET | Return |

# Instruction Naming Conventions

**Conditional branching** instructions examine the status of one of four condition flags to determine whether the specified branch is to be executed. The conditions that may be specified are as follows:

| | |
|---|---|
| NZ | Not Zero (Z = 0) |
| Z | Zero (Z = 1) |
| NC | No Carry (C = 0) |
| C | Carry (C = 1) |
| PO | Parity Odd (P = 0) |
| PE | Parity Even (P = 1) |
| P | Plus (S = 0) |
| M | Minus (S = 1) |

Thus, the conditional branching instructions are specified as follows:

| *Jumps* | *Calls* | *Returns* | |
|---|---|---|---|
| JC | CC | RC | (Carry) |
| JNC | CNC | RNC | (No Carry) |
| JZ | CZ | RZ | (Zero) |
| JNZ | CNZ | RNZ | (Not Zero) |
| JP | CP | RP | (Plus) |
| JM | CM | RM | (Minus) |
| JPE | CPE | RPE | (Parity Even) |
| JPO | CPO | RPO | (Parity Odd) |

# Instruction Naming Conventions

Strathmore UNIVERSITY

**Stack, I/O, and Machine Control Instructions.** The following instructions affect the stack and/or stack pointer:

| | |
|---|---|
| PUSH | Push Two Bytes of Data onto the Stack |
| POP | Pop Two Bytes of Data off the Stack |
| XTHL | Exchange Top of Stack with H and L |
| SPHL | Move contents of H and L to Stack Pointer |

The I/O instructions are as follows:

| | |
|---|---|
| IN | Initiate Input Operation |
| OUT | Initiate Output Operation |

The machine control instructions are as follows:

| | |
|---|---|
| EI | Enable Interrupt System |
| DI | Disable Interrupt System |
| HLT | Halt |
| NOP | No Operation |

# ASSEMBLY LANGUAGE CONCEPTS

Strathmore UNIVERSITY

Assembly language instructions and assembler directives may consist of up to four fields, as follows:

| Label: / Name | Opcode | Operand | ;Comment |
|---|---|---|---|

The fields may be separated by any number of blanks, but must be separated by at least one delimiter. Each instruction and directive must be entered on a single line terminated by a carriage return and a line feed. No continuation lines are possible, but you may have lines consisting entirely of comments.