CSE316: Microprocessors and Microcontrollers Sessional

Assembly Programming
Lecture 2

Nazmus Saquib

Lecturer
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

April 7, 2018

Introduction

- FLAGS register (ch. 5)[1]
- Branching (ch. 6)
- Bit manipulation (ch. 7)

[1] "Assembly Language Programming and Organization of the IBM PC" by Ytha Yu and Charles Marut

FLAGS Register

- A 16-bit register
- Six bits are used as **status flags**
- Three bits are used as **control flags**

status flags reflect the result of a computation

control flags enable/disable certain operations of the processor

We will focus on **status flags** only

FLAGS Register

- A 16-bit register
- Six bits are used as **status flags**
- Three bits are used as **control flags**

status flags reflect the result of a computation

control flags enable/disable certain operations of the processor

We will focus on **status flags** only

FLAGS Register

- A 16-bit register
- Six bits are used as **status flags**
- Three bits are used as **control flags**

status flags reflect the result of a computation

control flags enable/disable certain operations of the processor

We will focus on **status flags** only

FLAGS Register

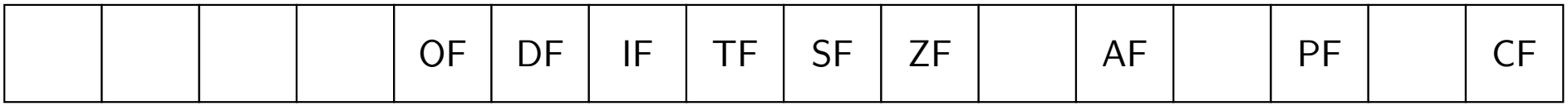
- A 16-bit register
- Six bits are used as **status flags**
- Three bits are used as **control flags**

status flags reflect the result of a computation

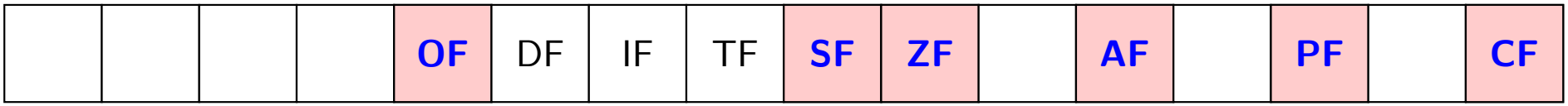
control flags enable/disable certain operations of the processor

We will focus on **status flags** only

FLAGS Register: Status Flags



FLAGS Register: Status Flags



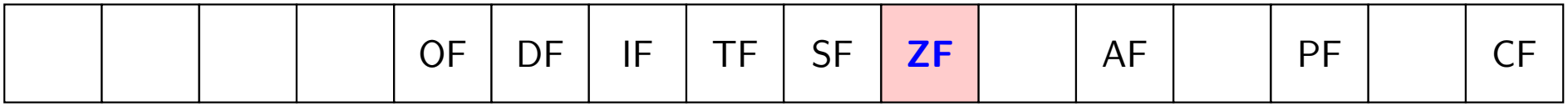
FLAGS Register: Status Flags *(Cntd.)*

- A single flag might be affected by a number of operations
- One operation might affect zero or more flags

Instruction	**Affected Flags**
MOV/XCHG	none
ADD/SUB	all
INC/DEC	all except CF
NEG	all[2]

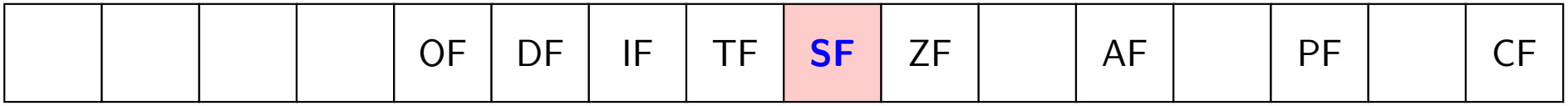
[2] CF=1 unless result is 0, OF=1 if word/byte operand is 8000h/80h

FLAGS Register: Zero Flag



- **ZF** is set if the result of certain operations is zero
- `SUB AX,AX`

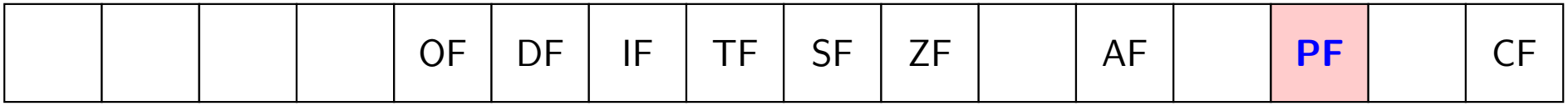
FLAGS Register: Sign Flag



- **SF** is set if the result of certain operations contains one in MSB

```
AL = 7FH          0111 1111
BL = 01H         +0000 0001
--------         ----------
ADD AL,BL         1000 0000
```

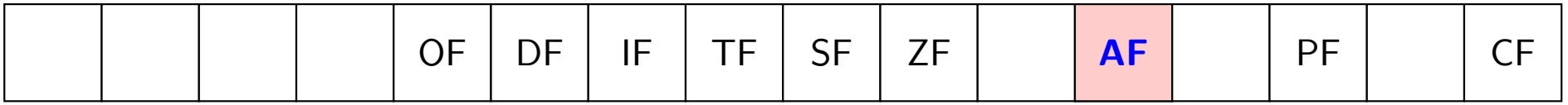
FLAGS Register: Parity Flag



- The parity flag denotes even parity
- If the *lower byte* result contains even number of 1's, PF is set

```
AL = 24H          0010 0100
BL = 48H         +0100 1000
--------         ----------
ADD AL,BL         0110 1100
```

FLAGS Register: Auxiliary Carry Flag



- AF comes into play only during BCD arithmetic
- To the interested readers: take a peek at Ch. 18
- Set when there is a carry out from lower to upper nibble

The Concept of Overflow

- The range of numbers a computer can represent is limited
- An 8-bit register can represent:
  - unsigned numbers from 0 to $2^8 - 1$ (0 to 255)
  - signed numbers from -2^7 to $2^7 - 1$ (-128 to 127)
- If the result falls outside the range, *overflow* occurs
- A truncated result is obtained that is incorrect

The Concept of Overflow

- The range of numbers a computer can represent is limited
- An 8-bit register can represent:
  - unsigned numbers from 0 to $2^8 - 1$ (0 to 255)
  - signed numbers from -2^7 to $2^7 - 1$ (-128 to 127)
- If the result falls outside the range, *overflow* occurs
- A truncated result is obtained that is incorrect

The Concept of Overflow

- The range of numbers a computer can represent is limited
- An 8-bit register can represent:
  - unsigned numbers from 0 to $2^8 - 1$ (0 to 255)
  - signed numbers from -2^7 to $2^7 - 1$ (-128 to 127)
- If the result falls outside the range, *overflow* occurs
- A truncated result is obtained that is incorrect

The Concept of Overflow

- The range of numbers a computer can represent is limited
- An 8-bit register can represent:
  - unsigned numbers from 0 to $2^8 - 1$ (0 to 255)
  - signed numbers from -2^7 to $2^7 - 1$ (-128 to 127)
- If the result falls outside the range, *overflow* occurs
- A truncated result is obtained that is incorrect

The Concept of Overflow

- The range of numbers a computer can represent is limited
- An 8-bit register can represent:
  - unsigned numbers from 0 to $2^8 - 1$ (0 to 255)
  - signed numbers from -2^7 to $2^7 - 1$ (-128 to 127)
- If the result falls outside the range, *overflow* occurs
- A truncated result is obtained that is incorrect

The Concept of Overflow

- The range of numbers a computer can represent is limited
- An 8-bit register can represent:
  - unsigned numbers from 0 to $2^8 - 1$ (0 to 255)
  - signed numbers from -2^7 to $2^7 - 1$ (-128 to 127)
- If the result falls outside the range, *overflow* occurs
- A truncated result is obtained that is incorrect

The Concept of Overflow *(Cntd.)*

- Overflow can be *signed* or *unsigned*
- They are represented by the **Overflow** and the **Carry** flag respectively

Overflow Type	**OF**	**CF**
None	0	0
Unsigned Only	0	1
Signed Only	1	0
Both	1	1

FLAGS Register: Overflow Flag and Carry Flag

$$OF = 0, CF = 0$$

	signed	**unsigned**
AL=02H	+2	2
BL=27H	+37	37
add	+39	39

```
 0000 0010
+0010 0101
----------
 0010 0111
```

FLAGS Register: Overflow Flag and Carry Flag

$$OF = 0,\ CF = 1$$

	signed	**unsigned**
AL=FFH	-1	255
BL=01H	+1	1
add	0	256

```
 1111 1111
+0000 0001
----------
10000 0000
```

FLAGS Register: Overflow Flag and Carry Flag

$$OF = 1, CF = 0$$

	signed	**unsigned**
AL=7FH	+127	127
BL=7FH	+127	127
add	+254	254

```
 0111 1111
+0111 1111
----------
 1111 1110
```

FLAGS Register: Overflow Flag and Carry Flag

$$OF = 1, CF = 1$$

	signed	**unsigned**
AL=80H	-128	128
BL=80H	-128	128
add	-256	256

```
 1000 0000
+1000 0000
----------
10000 0000
```

End of FLAGS

Branching

- Execute different instructions depending on conditions
- Execute a section of code repeatedly
- Branching can be conditional or unconditional
- The FLAGS register is extensively used for conditional instructions

Example Case: *If*

If AX>BX, store AX in DX

Introducing *CMP*

- `CMP dest,src`
- Performs subtraction (*dest* - *src*)
- The result is not stored (i.e. *dest* is not changed)
- FLAGS register affected

Introducing *CMP*

- `CMP dest,src`
- Performs subtraction (*dest* - *src*)
- The result is not stored (i.e. *dest* is not changed)
- FLAGS register affected

Introducing *CMP*

- `CMP dest,src`
- Performs subtraction (*dest* - *src*)
- The result is not stored (i.e. *dest* is not changed)
- FLAGS register affected

Introducing *CMP*

- `CMP dest,src`
- Performs subtraction (*dest* - *src*)
- The result is not stored (i.e. *dest* is not changed)
- FLAGS register affected

Example Case: *If (Cntd.)*

If AX>BX, store AX in DX

```
CMP AX,BX
JG STORE
JMP END_IF
STORE:
MOV DX,AX
END_IF:
```

```
CMP AX,BX
JLE END_IF
MOV DX,AX
END_IF:
```

Example Case: *If (Cntd.)*

If AX>BX, store AX in DX

```
CMP AX,BX
JG STORE
JMP END_IF
STORE:
MOV DX,AX
END_IF:
```

```
CMP AX,BX
JLE END_IF
MOV DX,AX
END_IF:
```

Example Case: *If (Cntd.)*

If AX>BX, store AX in DX

```
CMP AX,BX
JG STORE
JMP END_IF
STORE:
MOV DX,AX
END_IF:
```

```
CMP AX,BX
JLE END_IF
MOV DX,AX
END_IF:
```

Example Case: *If (Cntd.)*

If AX>BX, store AX in DX

```
CMP AX,BX
JG STORE
JMP END_IF
STORE:
MOV DX,AX
END_IF:
```

```
CMP AX,BX
JLE END_IF
MOV DX,AX
END_IF:
```

Example Case: *If (Cntd.)*

If AX>BX, store AX in DX

```
CMP AX,BX
JG STORE
JMP END_IF
STORE:
MOV DX,AX
END_IF:
```

```
CMP AX,BX
JLE END_IF
MOV DX,AX
END_IF:
```

Example Case: *If-Else*

If AX>BX, then SUB AX,BX; else ADD AX,BX

```
CMP AX,BX
JLE ELSE_
SUB AX,BX
JMP END_IF
ELSE_:
ADD AX,BX
END_IF:
```

Example Case: *If-Else*

If AX>BX, then SUB AX,BX; else ADD AX,BX

```
CMP AX,BX
JLE ELSE_
SUB AX,BX
JMP END_IF
ELSE_:
ADD AX,BX
END_IF:
```

Example Case: *If-Else*

If AX>BX, then SUB AX,BX; else ADD AX,BX

```
CMP AX,BX
JLE ELSE_
SUB AX,BX
JMP END_IF
ELSE_:
ADD AX,BX
END_IF:
```

Example Case: *If-Else*

If AX>BX, then SUB AX,BX; else ADD AX,BX

```
CMP AX,BX
JLE ELSE_
SUB AX,BX
JMP END_IF
ELSE_:
ADD AX,BX
END_IF:
```

Example Case: *If-Else*

If AX>BX, then SUB AX,BX; else ADD AX,BX

```
CMP AX,BX
JLE ELSE_
SUB AX,BX
JMP END_IF
ELSE_:
ADD AX,BX
END_IF:
```

Example Case: *Switch-Case*

AX>BX: CX=1; AX==BX: CX=0; AX<BX: CX=-1

```
CMP AX,BX
JG POSITIVE
JE ZERO
JL NEGATIVE
POSITIVE:
MOV CX,1
JMP END_CASE
ZERO:
MOV CX,0
JMP END_CASE
NEGATIVE:
MOV CX,-1
END_CASE:
```

Example Case: *Switch-Case*

AX>BX: CX=1; AX==BX: CX=0; AX<BX: CX=-1

```
CMP AX,BX
JG POSITIVE
JE ZERO
JL NEGATIVE
POSITIVE:
MOV CX,1
JMP END_CASE
ZERO:
MOV CX,0
JMP END_CASE
NEGATIVE:
MOV CX,-1
END_CASE:
```

Example Case: *Switch-Case*

AX>BX: CX=1; AX==BX: CX=0; AX<BX: CX=-1

```
CMP AX,BX
JG POSITIVE
JE ZERO
JL NEGATIVE
POSITIVE:
MOV CX,1
JMP END_CASE
ZERO:
MOV CX,0
JMP END_CASE
NEGATIVE:
MOV CX,-1
END_CASE:
```

Example Case: *Switch-Case*

AX>BX: CX=1; AX==BX: CX=0; AX<BX: CX=-1

```
CMP AX,BX
JG POSITIVE
JE ZERO
JL NEGATIVE
POSITIVE:
MOV CX,1
JMP END_CASE
ZERO:
MOV CX,0
JMP END_CASE
NEGATIVE:
MOV CX,-1
END_CASE:
```

Example Case: *Switch-Case*

AX>BX: CX=1; AX==BX: CX=0; AX<BX: CX=-1

```
CMP AX,BX
JG POSITIVE
JE ZERO
JL NEGATIVE
POSITIVE:
MOV CX,1
JMP END_CASE
ZERO:
MOV CX,0
JMP END_CASE
NEGATIVE:
MOV CX,-1
END_CASE:
```

Example Case: *Switch-Case*

AX>BX: CX=1; AX==BX: CX=0; AX<BX: CX=-1

```
CMP AX,BX
JG POSITIVE
JE ZERO
JL NEGATIVE
POSITIVE:
MOV CX,1
JMP END_CASE
ZERO:
MOV CX,0
JMP END_CASE
NEGATIVE:
MOV CX,-1
END_CASE:
```

Example Case: *AND Condition*

If CL represents an uppercase letter, then put 1 in CL; else put 0 in CL

```
CMP CL,'A'
JNGE ELSE_
CMP CL,'Z'
JNLE ELSE_
MOV CL,1
JMP END_IF
ELSE_:
MOV CL,0
END_IF
```

Example Case: *AND Condition*

If CL represents an uppercase letter, then put 1 in CL; else put 0 in CL

```
CMP CL,'A'
JNGE ELSE_
CMP CL,'Z'
JNLE ELSE_
MOV CL,1
JMP END_IF
ELSE_:
MOV CL,0
END_IF
```

Example Case: *AND Condition*

If CL represents an uppercase letter, then put 1 in CL; else put 0 in CL

```
CMP CL,'A'
JNGE ELSE_
CMP CL,'Z'
JNLE ELSE_
MOV CL,1
JMP END_IF
ELSE_:
MOV CL,0
END_IF
```

Example Case: *AND Condition*

If CL represents an uppercase letter, then put 1 in CL; else put 0 in CL

```
CMP CL,'A'
JNGE ELSE_
CMP CL,'Z'
JNLE ELSE_
MOV CL,1
JMP END_IF
ELSE_:
MOV CL,0
END_IF
```

Example Case: *AND Condition*

If CL represents an uppercase letter, then put 1 in CL; else put 0 in CL

```
CMP CL,'A'
JNGE ELSE_
CMP CL,'Z'
JNLE ELSE_
MOV CL,1
JMP END_IF
ELSE_:
MOV CL,0
END_IF
```

Example Case: *AND Condition*

If CL represents an uppercase letter, then put 1 in CL; else put 0 in CL

```
CMP CL,'A'
JNGE ELSE_
CMP CL,'Z'
JNLE ELSE_
MOV CL,1
JMP END_IF
ELSE_:
MOV CL,0
END_IF
```

Example Case: *AND Condition*

If CL represents an uppercase letter, then put 1 in CL; else put 0 in CL

```
CMP CL,'A'
JNGE ELSE_
CMP CL,'Z'
JNLE ELSE_
MOV CL,1
JMP END_IF
ELSE_:
MOV CL,0
END_IF
```

Example Case: *OR Condition*

If CL represents ‘y’ or ‘Y’, then put 1 in CL; else put 0 in CL

```
CMP CL,'y'
JE THEN
CMP CL,'Y'
JE THEN
MOV CL,0
JMP END_IF
THEN:
MOV CL,1
END_IF
```

Example Case: *OR Condition*

If CL represents ‘y’ or ‘Y’, then put 1 in CL; else put 0 in CL

```
CMP CL,'y'
JE THEN
CMP CL,'Y'
JE THEN
MOV CL,0
JMP END_IF
THEN:
MOV CL,1
END_IF
```

Example Case: *OR Condition*

If CL represents ‘y’ or ‘Y’, then put 1 in CL; else put 0 in CL

```
CMP CL,'y'
JE THEN
CMP CL,'Y'
JE THEN
MOV CL,0
JMP END_IF
THEN:
MOV CL,1
END_IF
```

Example Case: *OR Condition*

If CL represents ‘y’ or ‘Y’, then put 1 in CL; else put 0 in CL

```
CMP CL,'y'
JE THEN
CMP CL,'Y'
JE THEN
MOV CL,0
JMP END_IF
THEN:
MOV CL,1
END_IF
```

Example Case: *OR Condition*

If CL represents ‘y’ or ‘Y’, then put 1 in CL; else put 0 in CL

```
CMP CL,'y'
JE THEN
CMP CL,'Y'
JE THEN
MOV CL,0
JMP END_IF
THEN:
MOV CL,1
END_IF
```

Example Case: *OR Condition*

If CL represents ‘y’ or ‘Y’, then put 1 in CL; else put 0 in CL

```
CMP CL,'y'
JE THEN
CMP CL,'Y'
JE THEN
MOV CL,0
JMP END_IF
THEN:
MOV CL,1
END_IF
```

Example Case: *OR Condition*

If CL represents ‘y’ or ‘Y’, then put 1 in CL; else put 0 in CL

```
CMP CL,'y'
JE THEN
CMP CL,'Y'
JE THEN
MOV CL,0
JMP END_IF
THEN:
MOV CL,1
END_IF
```

The *LOOP* Instruction

- Performs instructions *N* number of times
- The value *N* is stored in **CX**
- Takes the form: `LOOP label`
- Decrements CX at each invocation, jumps to label if CX not 0

The *LOOP* Instruction: Example

Print the full alphabet in capital letters

```
MOV AH,2
MOV CX,26
MOV DL,65
TOP:
INT 21H
INC DL
LOOP TOP
```

The *LOOP* Instruction: Example

Print the full alphabet in capital letters

```
MOV AH,2
MOV CX,26
MOV DL,65
TOP:
INT 21H
INC DL
LOOP TOP
```

The *LOOP* Instruction: Example

Print the full alphabet in capital letters

```
MOV AH,2
MOV CX,26
MOV DL,65
TOP:
INT 21H
INC DL
LOOP TOP
```

The *LOOP* Instruction: Example

Print the full alphabet in capital letters

```
MOV AH,2
MOV CX,26
MOV DL,65
TOP:
INT 21H
INC DL
LOOP TOP
```

The *LOOP* Instruction: Example

Print the full alphabet in capital letters

```
MOV AH,2
MOV CX,26
MOV DL,65
TOP:
INT 21H
INC DL
LOOP TOP
```

The *LOOP* Instruction: Example

Print the full alphabet in capital letters

```
MOV AH,2
MOV CX,26
MOV DL,65
TOP:
INT 21H
INC DL
LOOP TOP
```

The *LOOP* Instruction: Example

Print the full alphabet in capital letters

```
MOV AH,2
MOV CX,26
MOV DL,65
TOP:
INT 21H
INC DL
LOOP TOP
```

The *LOOP* Instruction: Caveat

- The *LOOP* instruction will execute at least once
- This causes a problem when CX initially contains 0
- After decrementing the content becomes FFFFH (65535)
- So the instructions are executed 65535 more times
- The instruction *JCXZ* can be used to take care of this

Example Case: *While*

Count number of characters in input line

```
MOV DX,0
MOV AH,1
INT 21H
WHILE_:
CMP AL,0DH
JE END_WHILE
INC DX
INT 21H
JMP WHILE_
END_WHILE
```

Example Case: *While*

Count number of characters in input line

```
MOV DX,0
MOV AH,1
INT 21H
WHILE_:
CMP AL,0DH
JE END_WHILE
INC DX
INT 21H
JMP WHILE_
END_WHILE
```

Example Case: *While*

Count number of characters in input line

```
MOV DX,0
MOV AH,1
INT 21H
WHILE_:
CMP AL,0DH
JE END_WHILE
INC DX
INT 21H
JMP WHILE_
END_WHILE
```

Example Case: *While*

Count number of characters in input line

```
MOV DX,0
MOV AH,1
INT 21H
WHILE_:
CMP AL,0DH
JE END_WHILE
INC DX
INT 21H
JMP WHILE_
END_WHILE
```

Example Case: *While*

Count number of characters in input line

```
MOV DX,0
MOV AH,1
INT 21H
WHILE_:
CMP AL,0DH
JE END_WHILE
INC DX
INT 21H
JMP WHILE_
END_WHILE
```

Example Case: *While*

Count number of characters in input line

```
MOV DX,0
MOV AH,1
INT 21H
WHILE_:
CMP AL,0DH
JE END_WHILE
INC DX
INT 21H
JMP WHILE_
END_WHILE
```

Example Case: *While*

Count number of characters in input line

```
MOV DX,0
MOV AH,1
INT 21H
WHILE_:
CMP AL,0DH
JE END_WHILE
INC DX
INT 21H
JMP WHILE_
END_WHILE
```

Example Case: *While*

Count number of characters in input line

```
MOV DX,0
MOV AH,1
INT 21H
WHILE_:
CMP AL,0DH
JE END_WHILE
INC DX
INT 21H
JMP WHILE_
END_WHILE
```

Example Case: *While*

Count number of characters in input line

```
MOV DX,0
MOV AH,1
INT 21H
WHILE_:
CMP AL,0DH
JE END_WHILE
INC DX
INT 21H
JMP WHILE_
END_WHILE
```

Example Case: *Repeat*

Take input until 'N' is pressed

```
MOV AH,1
TOP:
INT 21H
CMP AL,'N'
JNE TOP
```

Example Case: *Repeat*

Take input until 'N' is pressed

```
MOV AH,1
TOP:
INT 21H
CMP AL,'N'
JNE TOP
```

Example Case: *Repeat*

Take input until 'N' is pressed

```
MOV AH,1
TOP:
INT 21H
CMP AL,'N'
JNE TOP
```

Example Case: *Repeat*

Take input until 'N' is pressed

```
MOV AH,1
TOP:
INT 21H
CMP AL,'N'
JNE TOP
```

Example Case: *Repeat*

Take input until 'N' is pressed

```
MOV AH,1
TOP:
INT 21H
CMP AL,'N'
JNE TOP
```

Properties of Jump Instructions

- Jump instructions do not affect the flags
- Conditional jumps cannot jump:
  - more than 126 bytes before the jump instruction
  - more than 127 bytes after the jump instruction
- This restriction can be bypassed through an unconditional jump

Properties of Jump Instructions *(Cntd.)*

- There are three types of conditional jumps:
  - signed jump
  - unsigned jump
  - single flag jump
- See Table 6.1 for details

Signed VS Unsigned Jump

- Every signed jump has a corresponding unsigned jump
- For example JG and JA
- Suppose AL=7FH and BL=80H
- In signed representation AL>BL
- However, in unsigned representation AL<BL

End of Branching

AND, OR, XOR

Effect on flags:

- SF, ZF, PF reflect the result
- AF undefined
- CF,OF=0

Common usage:

- **AND** is used to clear specific bits
- **OR** is used to set specific bits
- **XOR** is used to toggle specific bits

AND, OR, XOR

Effect on flags:

- SF, ZF, PF reflect the result
- AF undefined
- CF,OF=0

Common usage:

- **AND** is used to clear specific bits
- **OR** is used to set specific bits
- **XOR** is used to toggle specific bits

AND, OR, XOR

Effect on flags:

- SF, ZF, PF reflect the result
- AF undefined
- CF,OF=0

Common usage:

- **AND** is used to clear specific bits
- **OR** is used to set specific bits
- **XOR** is used to toggle specific bits

NOT

- Single operand instruction
- Performs 1's complement
- Does not affect the status flags

TEST

- Performs AND of the two operands
- However, does not store the result
- Affects the status flags the same way AND does
- Helps to find out whether a specific bit is set or not

Shift Instructions: SHL/SAL

- `SHL dest,1` or `SHL dest,CL`
- Every bit shifts left, MSB shifted to CF, 0 shifted into LSB
- Effect on flags:
  - SF, PF, ZF reflect the result
  - AF is undefined
  - CF = last bit shifted out
  - OF = 1 if result changes sign on last shift

Shift Instructions: SHR/SAR

- `SHR dest,1` or `SHR dest,CL`
- Every bit shifts right, LSB shifted to CF, 0 shifted into MSB
- Effect on flags is similar to that of SHR
- SAR retains the MSB

Rotate Instructions: ROL

- `ROL dest,1` or `ROL dest,CL`
- Every bit shifts left, MSB shifted to LSB and to CF
- All rotate instructions affect the flags in similar manner:
  - SF, PF, ZF reflect the result
  - AF is undefined
  - CF = last bit shifted out
  - OF = 1 if result changes sign on the last rotation

Rotate Instructions: ROR

- `ROR dest,1` or `ROR dest,CL`
- Every bit shifts right, LSB shifted to MSB and to CF

Rotate Instructions: RCR/RCL

- `RCR dest,1` or `RCR dest,CL`
- Similar to ROR/ROL, but with CF in the loop

Conclusion

- Read the chapters thoroughly (you may exclude section 5.4)
- Try out all the example codes in the book
- Solve all the exercises
- EMU8086 allows you to examine the values of registers
- The emulator also has a single step execution mode
- Use these to understand how different operations affect the flags

Conclusion

- Read the chapters thoroughly (you may exclude section 5.4)
- Try out all the example codes in the book
- Solve all the exercises
- EMU8086 allows you to examine the values of registers
- The emulator also has a single step execution mode
- Use these to understand how different operations affect the flags

Conclusion

- Read the chapters thoroughly (you may exclude section 5.4)
- Try out all the example codes in the book
- Solve all the exercises
- EMU8086 allows you to examine the values of registers
- The emulator also has a single step execution mode
- Use these to understand how different operations affect the flags

Conclusion

- Read the chapters thoroughly (you may exclude section 5.4)
- Try out all the example codes in the book
- Solve all the exercises
- EMU8086 allows you to examine the values of registers
- The emulator also has a single step execution mode
- Use these to understand how different operations affect the flags

Conclusion

- Read the chapters thoroughly (you may exclude section 5.4)
- Try out all the example codes in the book
- Solve all the exercises
- EMU8086 allows you to examine the values of registers
- The emulator also has a single step execution mode
- Use these to understand how different operations affect the flags

Conclusion

- Read the chapters thoroughly (you may exclude section 5.4)
- Try out all the example codes in the book
- Solve all the exercises
- EMU8086 allows you to examine the values of registers
- The emulator also has a single step execution mode
- Use these to understand how different operations affect the flags

Questions?