Arithmetic and Logic Instructions

Thorne : Chapter 6, 7, 9

(Irvine, Edition IV : 4.1, 4.2, 6.2 7.2, 7.3, 7.4)

Breakdown of Intel 8086 Assembly Instructions

1. **Data transfer**: copy data among state variables (registers, memory and I/O ports)
   - Their execution do not modify FLAGS
   - eg. MOV instruction
2. **Data manipulation**: modify state variable values
   - Executed within the ALU data path
   - Their execution do modify the FLAGS
   - eg. Arithmetic : ADD, SUB, CMP
   - eg. Logical : AND, OR, NOT, XOR
   - eg. Shift, Rotate : SHL, SAR, RCL, ROL
3. **Control-flow**: determine “next” instruction to execute
   - Their execution allow non-sequential execution
   - eg. JMP and JE

Data Manipulation Instructions

Compute new value … modify flags

Some common flag results :

ZF = zero flag   set iff result = 0

CF = carry flag   reflect carry value

set = 1, clear = 0

SF = sign flag   set iff result < 0

assumes 2's complement encoding!

iff ==If-and-only-if

OF = overflow flag

set iff signed overflow

specific use of "**overflow**" – not the same as the general concept!

Signed versus Unsigned Arithmetic

- These operations perform a bitwise add/subtract of values of width n to give a result of width n

- 8-bit Unsigned Integer Examples:

$$
\begin{array}{rrcl}
 & 117_{10} & = & 0111\ 0101_2 \\
+ & 99_{10} & = & 0110\ 0011_2 \\
\hline
 & 216_{10} & = & 1101\ 1000_2
\end{array}
$$

$$
\begin{array}{rrcl}
 & 133_{10} & = & 1000\ 0101_2 \\
- & 51_{10} & = & 0011\ 0011_2 \\
\hline
 & 82_{10} & = & 0101\ 0010_2
\end{array}
$$

Arithmetic Operations : Binary Addition and Subtraction

8-bit Signed Integer Examples:

- The computer does exactly the same thing for 2's complement signed integers! ☺

Signed is now being used to mean 2's complement signed

$$
\begin{array}{rrcl}
 & -117_{10} & = & 1000\ 1011_2 \\
+ & 99_{10} & = + & 0110\ 0011_2 \\
\hline
 & -18_{10} & = & 1110\ 1110_2
\end{array}
\quad (\ 0001\ 0001_2 + 1 = 12h = 18d)
$$

$$
\begin{array}{rrcl}
 & -32_{10} & = & 1110\ 0000_2 \\
- & 5_{10} & = - & 0000\ 0101_2 \\
\hline
 & -37_{10} & = & 1101\ 1011_2
\end{array}
\quad (\ 0010\ 0100_2 + 1 = 25h = 37d)
$$

Arithmetic Operations : Binary Addition and Subtraction

Computers often implement subtraction using "negate and add"

$$X - Y = X + (-Y)$$

Example : 32 –65 = 32 + (-65)

$$
\begin{array}{rcll}
 & 32_{10} = & 0010\ 0000_2 & \\
+ & -\ 65_{10} = & +\ 1011\ 1111_2 & (\ 0100\ 0000_2 + 1 = 41h = 65d) \\
\hline
 & -\ 33_{10} = & 1101\ 1111_2 & (\ 0010\ 0000_2 + 1 = 21h = 33d)
\end{array}
$$

Overflow : The Concept

Overflow : Result of operation **outside the range** that can be represented

- Problem arising due to limited range of fixed-width representation.
- Result is still produced; the result is just meaningless.

- 8-bit Unsigned Integer Example:

What is the range of an 8-bit unsigned number ?

We need 9 bits to represent the result

$$
\begin{array}{rrcr}
 & 255_{10} & = & 1111\ 1111_2 \\
+ & 1_{10} & = & 0000\ 0001_2 \\
\hline
256\ ?? & 0_{10} & = & (1)\ 0000\ 0000_2
\end{array}
$$

CARRY

- In this case (with fixed 8-bits) : OVERFLOW OCCURRED!
  CF=1
- In this case (unsigned) : A carry @ MSB is important in the INTERPRETATION of the result.

Addition and Subtraction Overflow

Is that the only interpretation of the example?

$$
\begin{array}{rl}
 & 1111\ 1111_2\ (=-1_{10}) \\
+ & 0000\ 0001_2\ (=+1_{10}) \\
\hline
(1) & 0000\ 0000_2\ (=0_{10})
\end{array}
$$

Same binary pattern !

- What if the values are interpreted as 8-bit signed integers ?
  - The result is correct (– 1 + 1 = 0). There is no overflow.
    OF=0 (CF=1)
  - In this case (signed), the carry at MSB still occurs but is not important to the interpretation!
  - ZF = ?   SF = ?

Overflow:

- Carry flag (CF) for unsigned numbers
- Overflow flag (OF) for singed numbers

Addition and Subtraction Overflow

Another example : With Borrow

$$
\begin{array}{lrcl}
\text{8-bit result} & 32_{10} & = & 0010\ 0000_2 \\
- & 65_{10} & = & 0100\ 0001_2 \\
\hline
- & 33_{10} & = & 1\quad 1101\ 1111_2 \ (= +223_{10} : \text{unsigned}) \\
 & & & (= -33_{10} : \text{signed})
\end{array}
$$

8-bit result

- If the values are interpreted as unsigned, the borrow implies overflow (actually, underflow) : the result is WRONG.
- If the value are interpreted as signed, there is no overflow; ignore the borrow : the result is CORRECT.

CF = 1
OF = 0
SF = 1
ZF = 0

Addition and Subtraction Overflow

Overflow depends on the *interpretation* of the values.

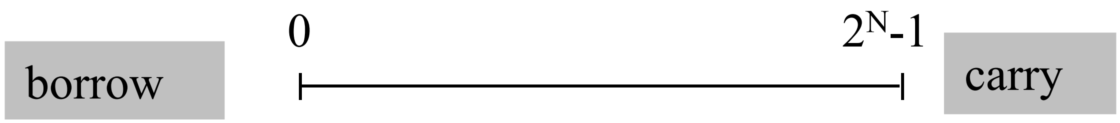
Another example:	**unsigned**	**signed**
$0111\ 1111_2$	127	127
$+\ 0000\ 0001_2$	+ 1	+ 1
$1000\ 0000_2$	128	– 128
	CORRECT	WRONG

CF = 0 (CF → unsigned number)
OF = 1 (OF → signed number)
SF = 1

OVERFLOW … even though there is no carry outside of fixed width!

Overflow Cookie Cutters

Unsigned: Carry or borrow means overflow



Signed : Ignore carry or borrow

Overflow possible if :

positive + positive = negative

(positive – negative = negative)

negative + negative = positive

(negative – positive = positive)

Overflow impossible if :

positive + negative (positive – positive)

negative + positive (negative – negative)

Data Manipulation : ADD

Example: Suppose that AL contains 73H, when it is executed :

execute: **ADD AL, 40H**

73 H + **40** H

= **B3**H carry?

results: **AL** := **B3**H (= 1011 0011 B)

ZF := 0 result ≠ 0

SF := 1 result is negative (signed)

CF := 0 (no carry out of msbit)

OF := 1 +ve + +ve = –ve

Correct result for unsigned number (CF = 0)

Wrong result for signed number (OF = 1)

Problem : Write a program to perform the following operation :

z = x + y where x = 55667788h and y = 99669988h

Solution:

```
.data
x DW  7788h
  DW  5566h
y DW  9988h
  DW  9966h
z DW  ?
  DW  ?
```

	1	111
	5566	7788
+	9966	9988
	EECD	1110

	5566	7788
+	9966	9988
	EECC	1110
	(BX)	(AX)

```
.code
  MOV AX, x
  MOV BX, x+2
  ADD AX, y      → CF=1
  ADD BX, y+2    → ADC BX, y+2 ⇨ BX:= BX+(y+2)+CF
  MOV z, AX
  MOV z+2, BX
END
```

Control Flow : JMP instructions

Four types of JUMP instructions:

Unary (unconditional) jumps : always execute

JMP target

Conditional Jumps

Simple jumps: jump is taken when a specific status flag is set

JC target (Jump if CF=1)

Unsigned jumps: jumps are taken when a comparison or test of unsigned numbers results in a specific combination of status flag

JA target (Jump if above)

Implication : Preceded by an instruction that alters the appropriate flags

Signed jumps: jumps are taken when a comparison or test of signed quantities results in a specific combination of status flags

JG target (Jump if greater than)

Signed and Unsigned Conditional Instructions

- The processor provides status flags to reflect results of (binary) manipulation under both signed and unsigned interpretations
- For this reason, there are separate conditional jump instructions for signed and unsigned
  - Semantically equivalent but implementation tests the flags appropriate to the data type.

Unsigned		Signed	
JA	Above	JG	Greater
JAE	Above or Equal	JGE	Greater or Equal
JB	Below	JL	Less
JBE	Below or Equal	JLE	Less or Equal

- There are also instructions for Not conditions too!

Example : Conditional Branches

Suppose AL contains 7FH:

Unsigned Scenario	Signed Scenario
CMP AL, 80h	CMP AL,80h
JA Bigger	JG Bigger

In each scenario, is the jump taken? Why?

Programmer MUST know how binary values are to be interpreted!
(e.g. value in AX above)

Limitation of J* Instructions

- Conditional jump are restricted to 8-bit signed relative offset!
  - IP := IP + (offset sign-extended to 16-bits)
  - Can't jump very far! – 128 ←→ +127 bytes
- Example:

```
            JL      Less
            ....
     Less:  MOV     . . .
```

maximum possible distance = 127 bytes

- One possible workaround if distance is greater than 127 bytes (but not the only one!):

```
            JNL     Continue
            JMP     Less
Continue:   ....
Less:               MOV   . . .
```

JMP can have 16-bit relative offset

distance can now be > 127

Example Write a code fragment showing how you would implement the following pseudocode

boolean done = FALSE;

while (! done)

{ .... }

Solution:

```
TRUE equ 1
FALSE equ 0
.code
MOV AL, FALSE  ; AL= register variable done
notDone: CMP AL, TRUE
         JE    amDone
         ; ....;  Somewhere : MOV AL, TRUE
         JMP   notDone
amDone: …
```

LOOP Instruction

- Useful when you have an action repeated a given number of times
- C++ analogy

  for (int i=max; i > 0; i--)

```
        MOV   CX, max
DoLoop:       . . .
        SUB   CX, 1
        JNZ           DoLoop
```

Functionally equivalent

Different performance & code size

```
        MOV CX, max
DoLoop:       . . .
        LOOP  DoLoop
```

- LOOP automatically decrements CX
- Only works with CX

Data Manipulation : DIV

Unsigned Integer Division

- Syntax : **DIV** **src**
- Semantics : Performs an integer division : Accumulator / src
  - The size of *divisor* (8-bit or 16-bit) is determined by size of **src**
  - **src** may be specified using register, direct or indirect mode but not immediate mode

$$\frac{\textbf{16}\text{-bit dividend}}{\textbf{8}\text{-bit divisor}}$$

- <u>8-bit division</u> : DIV *src* where src = 8-bit operand
  - Semantics divide *src* into 16-bit value in AX
  - AL := AX ÷ src (unsigned divide) — Integer result
    AH := AX mod src (unsigned modulus) — Integer remainder
    Two 8-bit results
  - The flags are undefined after DIV (values may have changed, no meaning)

Data Manipulation : DIV

32-bit dividend / 16-bit divisor

- 16-bit division : DIV *src* where src = 16-bit operand
  - Semantics divide *src* into 32-bit value obtained by concatenating DX and AX (written DX:AX)

Two 16-bit results

Integer result

Integer remainder

AX := DX:AX ÷ src (unsigned divide)

DX := DX:AX mod src (unsigned modulus)

  - The flags are undefined after DIV (values may have changed, no meaning)

Question : In either case, what if the result is too big to fit in destination?

- eg : AX ÷ 1 ?? AL = ??
- overflow trap – more later!

Data Manipulation : Logical Operations

Syntax : LOGICAL_MNEUMONIC dest, src

Semantics : dest = dest LOGICAL_MNEUMONIC src

Example : AND AL, 80h

Example : .data

control DB ?

.code

OR control, BH

(where BH=02h)

		AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Example : XOR AX, AX

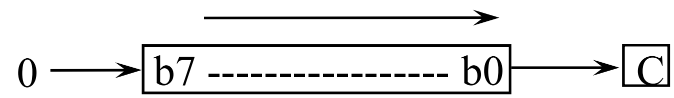
XOR AX, 0FFh

Operation bit by bit!

Data Manipulation : Shift

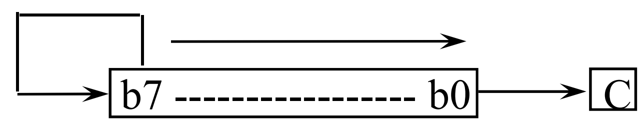
- Versions for : Left/Right and Arithmetic/Logical

Logical Shift Right



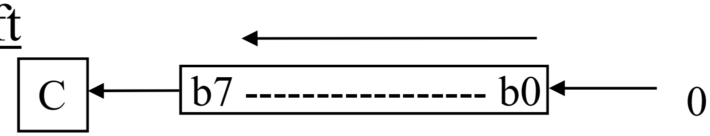
SHR AL, 1

Arithmetic Shift Right



MOV CL, 2
SAR AL, CL

Logical or Arithmetic Shift Left



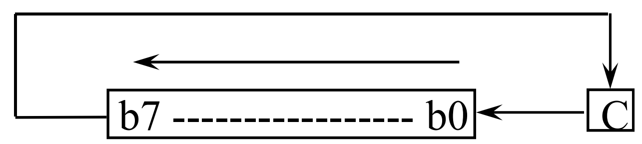
SHL AL, 1
SAL AL, 1

Data Manipulation : Rotate

- Versions for : Left/Right and with/out carry

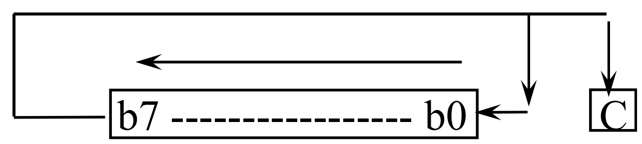
Rotate-Carry-Left

RCL AL, 1



Rotate Left

MOV CL, 4
ROL AL, CL



Example Write a code fragment to test whether a variable is divisible by 4, leaving the boolean result in AX.
Solution: A number divisible by 4 would have the least significant two bits equal 0s.

```
FALSE       equu      0
TRUE        equ       1
.data
variable    dw        1922h
.code
      MOV   AX, variable
      AND   AX, 03h
      JZ    yes
      MOV   AX, FALSE
      JMP   continue
yes:  MOV   AX, TRUE
continue:
      …
```

Alterative :

TEST variable, 03h

Example Suppose a robot has four motors, each of which can be off, on in forward direction or on in reverse direction. The status of these motors are written by the robot into a status word, say called "motors" in the following bitmap formation.

7	6	5	4	3	2	1	0
Motor1		Motor2		Motor3		Motor4	

where the two bits for each motor are set according

01	forward
10	reverse
11	off

Write a code fragment that waits until motor1 is off before continuing on.

…

Solution:

```
            .data
            motors db    ?
            .code
waiting:    MOV AL, motors
            AND AL, 0C0h
            CMP AL, 0C0h
            JNZ waiting
            …
```