Outline

• Status flags
  * Zero flag
  * Carry flag
  * Overflow flag
  * Sign flag
  * Auxiliary flag
  * Parity flag

• Arithmetic instructions
  * Addition instructions
  * Subtraction instructions
  * Multiplication instructions
  * Division instructions

• Application examples
  * PutInt8
  * GetInt8

• Multiword arithmetic
  * Addition
  * Subtraction
  * Multiplication
  * Division

• Performance: Multiword multiplication

Arithmetic Flags and Instructions

Chapter 6
S. Dandamudi
Status Flags

- Six status flags monitor the outcome of arithmetic, logical, and related operations

Status Flags

- Carry Flag (CF)
- Parity Flag (PF)
- Zero Flag (ZF)
- Sign Flag (SF)
- Overflow Flag (OF)
- Auxiliary Carry Flag (AF)

Control Flags

- Direction Flag (DF)
- Trap Flag (TF)
- Interrupt Flag (IF)
- Virtual Interrupt Flag (VIF)

System Flags

- Nested Task (NT)
- Resume Flag (RF)
- Virtual 8086 Mode (VM)
- Alignment Check (AC)
- Virtual Interrupt Pending (VIP)
- ID Flag (ID)

Status Flags (cont’d)

- Status flags are updated to indicate certain properties of the result
  * Example: If the result is zero, zero flag is set
- Once a flag is set, it remains in that state until another instruction that affects the flags is executed
- Not all instructions affect all status flags
  * add and sub affect all six flags
  * inc and dec affect all but the carry flag
  * mov, push, and pop do not affect any flags
Status Flags (cont’d)

• Example

; initially, assume ZF = 0
mov AL, 55H ; ZF is still zero
sub AL, 55H ; result is 0
; ZF is set (ZF = 1)
push BX ; ZF remains 1
mov BX, AX ; ZF remains 1
pop DX ; ZF remains 1
mov CX, 0 ; ZF remains 1
inc CX ; result is 1
; ZF is cleared (ZF = 0)

Zero Flag

∗ Indicates zero result
– If the result is zero, ZF = 1
– Otherwise, ZF = 0

∗ Zero can result in several ways (e.g., overflow)

mov AL, 0FH | mov AX, 0FFFH | mov AX, 1
add AL, 0F1H | inc AX | dec AX

» All three examples result in zero result and set ZF

∗ Related instructions

jz jump if zero (jump if ZF = 1)
jnz jump if not zero (jump if ZF = 0)
Status Flags (cont’d)

• Uses of zero flag
  ∗ Two main uses of zero flag
    » Testing equality
      – Often used with `cmp` instruction
        ```
cmp   char, ’$’  ; ZF = 1 if char is $
        ```
        ```
cmp   AX, BX
        ```
    » Counting to a preset value
      – Initialize a register with the count value
      – Decrement it using `dec` instruction
      – Use `jz/jnz` to transfer control

• Consider the following code

  ```
  sum := 0
  for (I = 1 to M)
    for (j = 1 to N)
      sum := sum + 1
  end for
end for
  ```

• Assembly code

  ```
  sub   AX,AX  ; AX := 0
  mov   DX,M
  outer_loop:
    mov   CX,N
  inner_loop:
    inc   AX
    loop  inner_loop
  dec   DX
  jnz   outer_loop
  exit_loops:
    mov   sum,AX
  ```
Status Flags (cont’d)

- Two observations
  * **loop** instruction is equivalent to
    
    ```
    dec   CX
    jnz   inner_loop
    ```
  
    » This two instruction sequence is more efficient than the **loop** instruction (takes less time to execute)
  
    » **loop** instruction does not affect any flags!
  
  * This two instruction sequence is better than initializing
    CX to zero and executing
    
    ```
    inc   CX
    cmp   CX, N
    jle   inner_loop
    ```

- Carry Flag
  
  * Records the fact that the result of an arithmetic operation on unsigned numbers is out of range
  
  * The carry flag is set in the following examples
    
    ```
    mov   AL, 0FH        mov   AX, 12AEH
    add   AL, 0F1H       sub   AX, 12AFH
    ```

  * Range of 8-, 16-, and 32-bit unsigned numbers
    
    | size  | range            |
    |-------|------------------|
    | 8 bits| 0 to 255 (2^8 – 1) |
    | 16 bits| 0 to 65,535 (2^16 – 1) |
    | 32 bits| 0 to 4,294,967,295 (2^32–1) |
Status Flags (cont’d)

* Carry flag is not set by \texttt{inc} and \texttt{dec} instructions
  » The carry flag is not set in the following examples
    \begin{align*}
    \text{mov} & \quad \text{AL}, 0\text{FFH} & \quad \text{mov} & \quad \text{AX}, 0 \\
    \text{inc} & \quad \text{AL} & \quad \text{dec} & \quad \text{AX}
    \end{align*}

* Related instructions
  \begin{itemize}
  \item \texttt{jc} jump if carry (jump if CF = 1)
  \item \texttt{jnc} jump if no carry (jump if CF = 0)
  \end{itemize}

* Carry flag can be manipulated directly using
  \begin{itemize}
  \item \texttt{stc} set carry flag (set CF to 1)
  \item \texttt{clc} clear carry flag (clears CF to 0)
  \item \texttt{cmc} complement carry flag (inverts CF value)
  \end{itemize}

Status Flags (cont’d)

• Uses of carry flag
  * To propagate carry/borrow in multiword addition/subtraction
    \begin{align*}
    1 & \leftarrow \text{carry from lower 32 bits} \\
    x & = 3710 \ 26A8 \ 1257 \ 9AE7H \\
    y & = 489B \ A321 \ FE60 \ 4213H \\
    \hline
    7FA8 \ C9CA \ 10B7 \ DCFAH
    \end{align*}
  
  * To detect overflow/underflow condition
    » In the last example, carry out of leftmost bit indicates overflow
  
  * To test a bit using the shift/rotate instructions
    » Bit shifted/rotated out is captured in the carry flag
    » We can use \texttt{jc/jnc} to test whether this bit is 1 or 0
Status Flags (cont’d)

• Overflow flag
  * Indicates out-of-range result on signed numbers
    – Signed number counterpart of the carry flag
  * The following code sets the overflow flag but not the carry flag
    \[
    \begin{align*}
    \text{mov} & \quad \text{AL, 72H} \quad ; \quad 72H = 114D \\
    \text{add} & \quad \text{AL, 0EH} \quad ; \quad 0EH = 14D
    \end{align*}
    \]
  * Range of 8-, 16-, and 32-bit signed numbers

<table>
<thead>
<tr>
<th>size</th>
<th>range</th>
<th>(2^7) to ((2^7 - 1))</th>
<th>(2^{15}) to ((2^{15} - 1))</th>
<th>(2^{31}) to ((2^{31} - 1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>–128 to +127</td>
<td>(2^7) to ((2^7 - 1))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 bits</td>
<td>–32,768 to +32,767</td>
<td>(2^{15}) to ((2^{15} - 1))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 bits</td>
<td>–2,147,483,648 to +2,147,483,647</td>
<td>(2^{31}) to ((2^{31} - 1))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Status Flags (cont’d)

• Signed or unsigned: How does the system know?
  * The processor does not know the interpretation
  * It sets carry and overflow under each interpretation

Unsigned interpretation

\[
\begin{align*}
\text{mov} & \quad \text{AL, 72H} \\
\text{add} & \quad \text{AL, 0EH} \\
\text{jc} & \quad \text{overflow}
\end{align*}
\]

no_overflow:

(no overflow code here)

overflow:

(overflow code here)

Signed interpretation

\[
\begin{align*}
\text{mov} & \quad \text{AL, 72H} \\
\text{add} & \quad \text{AL, 0EH} \\
\text{jo} & \quad \text{overflow}
\end{align*}
\]

no_overflow:

(no overflow code here)

overflow:

(overflow code here)
Status Flags (cont’d)

* Related instructions

  - **jo**  jump if overflow (jump if OF = 1)
  - **jno**  jump if no overflow (jump if OF = 0)

* There is a special software interrupt instruction

  - **into**  interrupt on overflow

  Details on this instruction in Chapter 12

- Uses of overflow flag

  * Main use
    - To detect out-of-range result on signed numbers

---

Status Flags (cont’d)

- **Sign flag**

  * Indicates the sign of the result
    - Useful only when dealing with signed numbers
    - Simply a copy of the most significant bit of the result

* Examples

<table>
<thead>
<tr>
<th>mov AL, 15</th>
<th>mov AL, 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>add AL, 97</td>
<td>sub AL, 97</td>
</tr>
<tr>
<td>clears the sign flag as</td>
<td>sets the sign flag as</td>
</tr>
<tr>
<td>the result is 112</td>
<td>the result is −82</td>
</tr>
<tr>
<td>(or 0111000 in binary)</td>
<td>(or 10101110 in binary)</td>
</tr>
</tbody>
</table>

* Related instructions

  - **js**  jump if sign (jump if SF = 1)
  - **jns**  jump if no sign (jump if SF = 0)
Status Flags (cont’d)

• Usage of sign flag
  * To test the sign of the result
  * Also useful to efficiently implement countdown loops

• Consider the count down loop:
  for (i = M downto 0)
      <loop body>
  end for

• If we don’t use the jns, we need cmp as shown below:

  \[
  \begin{align*}
  \text{cmp} & \quad \text{CX}, 0 \\
  \text{jl} & \quad \text{for_loop}
  \end{align*}
  \]

  The count down loop can be implemented as

  \[
  \begin{align*}
  \text{mov} & \quad \text{CX}, M \\
  \text{for_loop}: & \\
  \text{dec} & \quad \text{CX} \\
  \text{jns} & \quad \text{for_loop}
  \end{align*}
  \]

Status Flags (cont’d)

• Auxiliary flag
  * Indicates whether an operation produced a carry or borrow in the low-order 4 bits (nibble) of 8-, 16-, or 32-bit operands (i.e. operand size doesn’t matter)

  * Example

    \[
    \begin{align*}
    \text{mov} & \quad \text{AL}, 43 \\
    & \quad 43D = 0010 \ 1011B \\
    \text{add} & \quad \text{AL}, 94 \\
    & \quad 94D = 0101 \ 1110B \\
    & \quad 137D = 1000 \ 1001B
    \end{align*}
    \]

    » As there is a carry from the lower nibble, auxiliary flag is set
Status Flags (cont’d)

* Related instructions
  » No conditional jump instructions with this flag
  » Arithmetic operations on BCD numbers use this flag
    aaa ASCII adjust for addition
    aas ASCII adjust for subtraction
    aam ASCII adjust for multiplication
    aad ASCII adjust for division
    daa Decimal adjust for addition
    das Decimal adjust for subtraction
  – Chapter 11 has more details on these instructions

* Usage
  » Main use is in performing arithmetic operations on BCD numbers

Status Flags (cont’d)

- Parity flag
  * Indicates even parity of the low 8 bits of the result
    – PF is set if the lower 8 bits contain even number 1 bits
    – For 16- and 32-bit values, only the least significant 8 bits are considered for computing parity value
  * Example
    mov AL, 53     53D = 0011 0101B
    add AL, 89     89D = 0101 1001B
    142D = 1000 1110B
  » As the result has even number of 1 bits, parity flag is set

* Related instructions
  jp    jump on even parity (jump if PF = 1)
  jnp   jump on odd parity (jump if PF = 0)
Status Flags (cont’d)

* Usage of parity flag
  » Useful in writing data encoding programs
  » Example: Encodes the byte in AL (MSB is the parity bit)

```asm
parity_encode  PROC
  shl  AL
  jp   parity_zero
  stc
  jmp  move_parity_bit
parity_zero:
  clc
move_parity_bit:
  rcr  AL
parity_encode  ENDP
```

Arithmetic Instructions

- Pentium provides several arithmetic instructions that operate on 8-, 16- and 32-bit operands
  - Addition: `add, adc, inc`
  - Subtraction: `sub, sbb, dec, neg, cmp`
  - Multiplication: `mul, imul`
  - Division: `div, idiv`
  - Related instructions: `cbw, cwd, cdq, cwde, movsx, movzx`

* There are few other instructions such as `aaa, aas`, etc. that operate on decimal numbers
  » See Chapter 11 for details
Arithmetic Instructions (cont’d)

• Addition instructions
  * Basic format
    
    \[ \text{add} \quad \text{destination}, \text{source} \]

    » Performs simple integer addition

    \[ \text{destination} := \text{destination} + \text{source} \]

  * Five operand combinations are possible

    \begin{align*}
    &\text{add} \quad \text{register}, \text{register} \\
    &\text{add} \quad \text{register}, \text{immediate} \\
    &\text{add} \quad \text{memory}, \text{immediate} \\
    &\text{add} \quad \text{register}, \text{memory} \\
    &\text{add} \quad \text{memory}, \text{register}
    \end{align*}

• Basic format

  \[ \text{adc} \quad \text{destination}, \text{source} \]

  » Performs integer addition with carry

  \[ \text{destination} := \text{destination} + \text{source} + \text{CF} \]

  * Useful in performing addition of long word arithmetic

  * The three carry flag manipulating instructions are useful

    \begin{align*}
    &\text{stc} \quad \text{set carry flag (set CF to 1)} \\
    &\text{clc} \quad \text{clear carry flag (clears CF to 0)} \\
    &\text{cmc} \quad \text{complement carry flag (inverts CF value)}
    \end{align*}
Arithmetic Instructions (cont’d)

* The final instruction `inc` requires a single operand

```
inc    destination
```

- Performs increment operation
- `destination := destination + 1`
- The operand is treated as an unsigned number
- Does not affect the carry flag
- Other five status flags are updated
- In general
  - `inc BX` is better than `add BX,1`
  - Both take same time but `inc` version takes less space

---

Arithmetic Instructions (cont’d)

- Subtraction instructions

```
sub    destination,source
```
- Performs simple integer subtraction
  - `destination := destination - source`

```
sbb    destination,source
```
- Performs integer subtraction with borrow
  - `destination := destination - source - CF`

```
dec    destination
```
- Performs decrement operation
  - `destination := destination - 1`
Arithmetic Instructions (cont’d)

• Subtraction instructions (cont’d)

  \texttt{neg destination}
  \hspace{1cm} » Performs sign reversal
  \hspace{1cm} \texttt{destination := 0 - destination}
  \hspace{1cm} » Useful in signed number manipulation

  \texttt{cmp destination, source}
  \hspace{1cm} » Performs subtraction without updating \texttt{destination}
  \hspace{1cm} \texttt{destination - source}
  \hspace{1cm} » Updates all six status flags to record the attributes of the result
  \hspace{1cm} » The \texttt{cmp} instruction is typically followed by a conditional jump instruction

Arithmetic Instructions (cont’d)

• Multiplication

  * More complicated than \texttt{add/sub}
  \hspace{1cm} » Produces double-length results
  \hspace{1cm} \hspace{1cm} - E.g. Multiplying two 8 bit numbers produces a result that requires 16 bits
  \hspace{1cm} » Cannot use a single multiply instruction for signed and unsigned numbers
  \hspace{1cm} \hspace{1cm} - \texttt{add} and \texttt{sub} instructions work both on signed and unsigned numbers
  \hspace{1cm} \hspace{1cm} - For multiplication, we need separate instructions
  \hspace{1cm} \hspace{1cm} \\texttt{mul} \hspace{0.5cm} for unsigned numbers
  \hspace{1cm} \hspace{1cm} \\texttt{imul} \hspace{0.5cm} for signed numbers
### Arithmetic Instructions (cont’d)

- **Unsigned multiplication**

  \[ \text{mul source} \]

  » Depending on the source operand size, the location of the other source operand and destination are selected

<table>
<thead>
<tr>
<th>Source Size</th>
<th>AL</th>
<th>AH</th>
<th>Low-order 8 bits</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td>AL</td>
<td>AH</td>
<td>AL</td>
<td>AL</td>
</tr>
<tr>
<td>16-bit</td>
<td>AX</td>
<td>DX</td>
<td>AX</td>
<td>AX</td>
</tr>
<tr>
<td>32-bit</td>
<td>EAX</td>
<td>EDX</td>
<td>EAX</td>
<td>EAX</td>
</tr>
</tbody>
</table>

- **Example**

  ```
  mov AL, 10
  mov DL, 25
  mul DL
  ```

  produces 250D in AX register (result fits in AL)

- **The \text{imul} instruction can use the same syntax**

  » Also supports other formats

  - **Example**

    ```
    mov DL, OFFH ; DL := -1
    mov AL, 0BEH ; AL := -66
    mul DL
    ```

    produces 66D in AX register (again, result fits in AL)
Arithmetic Instructions (cont’d)

• Division instruction
  * Even more complicated than multiplication
    » Produces two results
      – Quotient
      – Remainder
    » In multiplication, using a double-length register, there will not be any overflow
      – In division, divide overflow is possible
        Pentium provides a special software interrupt when a divide overflow occurs
  * Two instructions as in multiplication

    div source \hspace{1em} \text{for unsigned numbers}
    idiv source \hspace{1em} \text{for signed numbers}

16-bit dividend
\[ \begin{array}{c|c|c|c|}
\text{Source} & \text{Quotient} & \text{Remainder} \\
\hline
\text{AX} & \text{AL} & \text{AH} \\
\hline
\text{Divisor} & & & \\
\end{array} \]

8-bit

32-bit dividend
\[ \begin{array}{c|c|c|c|}
\text{Source} & \text{Quotient} & \text{Remainder} \\
\hline
\text{DX} & \text{AX} & \text{DX} \\
\hline
\text{AX} & & & \\
\hline
\text{Divisor} & & & \\
\end{array} \]

16-bit

64-bit dividend
\[ \begin{array}{c|c|c|c|}
\text{Source} & \text{Quotient} & \text{Remainder} \\
\hline
\text{EDX} & \text{EAX} & \text{EDX} \\
\hline
\text{EAX} & & & \\
\hline
\text{Divisor} & & & \\
\end{array} \]

32-bit
### Arithmetic Instructions (cont’d)

- **Example**
  
  ```
  mov    AX, 00FBH    ; AX := 251D
  mov    CL, 0CH      ; CL := 12D
  div    CL
  ```
  
  produces 20D in AL and 11D as remainder in AH

- **Example**
  
  ```
  sub    DX, DX       ; clear DX
  mov    AX, 141BH    ; AX := 5147D
  mov    CX, 012CH    ; CX := 300D
  div    CX
  ```
  
  produces 17D in AX and 47D as remainder in DX

### Signed division requires some help

- We extended an unsigned 16 bit number to 32 bits by placing zeros in the upper 16 bits
- This will not work for signed numbers
  - To extend signed numbers, you have to copy the sign bit into those upper bit positions

* Pentium provides three instructions in aiding sign extension
  - All three take no operands
    
    ```
    cbw    converts byte to word (extends AL into AH)
    cwd    converts word to doubleword (extends AX into DX)
    cdq    converts doubleword to quadword
            (extends EAX into EDX)
    ```
Arithmetic Instructions (cont’d)

* Some additional related instructions
  » Sign extension
    cwde  converts word to doubleword
          (extends AX into EAX)

  » Two move instructions
    movsx  dest,src  (move sign-extended src to dest)
    movzx  dest,src  (move zero-extended src to dest)
  » For both move instructions, dest has to be a register
  » The src operand can be in a register or memory
     – If src is 8-bits, dest has to be either a 16 bit or 32 bit register
     – If src is 16-bits, dest has to be a 32 bit register

• Example
  mov    AL,0A1H    ; AL := -95D
  cbw               ; AH = FFH
  mov    CL,0CH     ; CL := 12D
  idiv   CL
  produces −7D in AL and −11D as remainder in AH

• Example
  mov    AX,0EBE5    ; AX := -5147D
  cwd                ; DX := FFFFH
  mov    CX,012CH    ; CX := 300D
  idiv   CX
  produces −17D in AX and −47D as remainder in DX
Application Examples

• **PutInt8** procedure
  * To display a number, repeatedly divide it by 10 and display the remainders obtained
  
<table>
<thead>
<tr>
<th>quotient</th>
<th>remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>108/10</td>
<td>10</td>
</tr>
<tr>
<td>10/10</td>
<td>1</td>
</tr>
<tr>
<td>1/10</td>
<td>0</td>
</tr>
</tbody>
</table>

  * To display digits, they must be converted to their character form
    » This means simply adding the ASCII code for zero (see line 24)

  line 24: add AH, ‘0’

• **GetInt8** procedure
  * To read a number, read each digit character
    » Convert to its numeric equivalent
    » Multiply the running total by 10 and add this digit

<table>
<thead>
<tr>
<th>Input digit</th>
<th>Numeric value (N)</th>
<th>Number := Number*10 + N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial value</td>
<td>--</td>
<td>0</td>
</tr>
<tr>
<td>‘1’</td>
<td>1</td>
<td>0 * 10 + 1 = 1</td>
</tr>
<tr>
<td>‘5’</td>
<td>5</td>
<td>1 * 10 + 5 = 15</td>
</tr>
<tr>
<td>‘8’</td>
<td>8</td>
<td>15 * 10 + 8 = 158</td>
</tr>
</tbody>
</table>

1998 © S. Dandamudi Arithmetic: Page 37
To be used with S. Dandamudi, “Introduction to Assembly Language Programming,” Springer-Verlag, 1998.
Multiword Arithmetic

- Arithmetic operations (\texttt{add}, \texttt{sub}, \texttt{mul}, and \texttt{div}) work on 8-, 16-, or 32-bit operands.
- Arithmetic on larger operands require multiword arithmetic software routines.
- Addition/subtraction
  - These two operations are straightforward to extend to larger operand sizes.
  - Need to use \texttt{adc/sbb} versions to include the carry generated by the previous group of bits.
  - Example addition on the next slide.

Multiword Arithmetic (cont’d)

;-----------------------------------------------
;Adds two 64-bit numbers in EBX:EAX and EDX:ECX.
;The result is returned in EBX:EAX.
;Overflow/underflow conditions are indicated
;by setting the carry flag.
;Other registers are not disturbed.
;-----------------------------------------------
add64 PROC
  add    EAX,ECX
  adc    EBX,EDX
  ret
add64 ENDP
Multiword Arithmetic (cont’d)

• Multiplication
  ∗ We consider two algorithms
    » Longhand multiplication
      – Uses the method that we are familiar with
      – Needs addition operations only
      – Examines each bit in the multiplier and adds the
        multiplicand if the multiplier bit is 1
        → Appropriate shifting is required
    » Using the `mul` instruction
      – Chops the operand into 32-bit chunks and applies `mul`
        instruction
      – Similar to the addition example seen before

Multiword Arithmetic (cont’d)

∗ Longhand multiplication
  → Final 128-bit result in P:A

  \[ P := 0; \text{count} := 64 \]
  \[ A := \text{multiplier}; B := \text{multiplicand} \]
  \[ \text{while (count} > 0) \]
      \[ \text{if (LSB of} A = 1) \]
      \[ \text{then } P := P+B \]
      \[ \text{CF := carry generated by } P+B \]
      \[ \text{else } CF := 0 \]
    \[ \text{end if} \]
    \[ \text{shift right } CF:P:A \text{ by one bit position} \]
  \[ \text{count} := \text{count} - 1 \]
  \[ \text{end while} \]
Multiword Arithmetic (cont’d)

* Using the **mul** instruction
  » A 64-bit number is treated as two 32-bit numbers
    – A is considered as consisting of A1A0 (similarly B)
    – Left shift operation replaces zeros on the right

<table>
<thead>
<tr>
<th>temp := A0 × B0</th>
<th>temp := A0 × B1</th>
</tr>
</thead>
<tbody>
<tr>
<td>result := temp</td>
<td>temp := left shift temp by 32 bits</td>
</tr>
<tr>
<td>temp := A1 × B0</td>
<td>result := result + temp</td>
</tr>
<tr>
<td>temp := left shift temp by 32 bits</td>
<td>temp := A1 × B1</td>
</tr>
<tr>
<td>result := result + temp</td>
<td>temp := left shift temp by 32 bits</td>
</tr>
<tr>
<td></td>
<td>result := result + temp</td>
</tr>
</tbody>
</table>

**Division**

* To implement n-bit division (A by B), we need an additional n+1 bit register P
* Core part of the algorithm
  » Test the sign of P
  » if P is negative
    – left shift P:A by one bit position
    – P := P+B
  » else
    – left shift P:A by one bit position
    – P := P–B
Multiword Arithmetic (cont’d)

Division Algorithm

\[ \begin{align*}
P &: = 0; \text{ count } &: = 64 \\
A &: = \text{ dividend} \\
B &: = \text{ divisor} \\
\text{while } (\text{count } > 0) & \\
\text{if } (P \text{ is negative}) & \\
\text{then } & \text{ shift left } P:A \text{ by 1 bit position} \\
P &: = P + B \\
\text{else } & \text{ shift left } P:A \text{ by 1 bit position} \\
P &: = P - B \\
\text{end if}
\end{align*} \]

\[ A = \text{ quotient}, \ P = \text{ remainder} \]

\[ \begin{align*}
\text{if } (P \text{ is negative}) & \\
\text{then set low-order } & \text{ bit of } A \text{ to 0} \\
\text{else set low-order } & \text{ bit of } A \text{ to 1} \\
\text{end if} \\
\text{count } &= \text{ count-1} \\
\text{end while} \\
\text{if } (P \text{ is negative}) & \\
P & = P + B \\
\text{end if}
\end{align*} \]

Performance: Multiword Multiplication

- Longhand version versus \texttt{mul} version

\[ \begin{align*}
\text{To multiply } 2^{64} - 1 \times 2^{64} - 1
\end{align*} \]
Performance: Multiword Multiply (cont’d)

- Using **add** versus **mul** instruction
  - Certain special cases of multiplication can be done by a series of additions (e.g. power of 2 by shift operations)
  - Example: Multiplication by 10
  - Instead of using **mul** instruction, we can multiply by 10 using **add** instructions as follows (performs $AL \times 10$):

```
sub AH, AH   ; AH := 0
mov BX, AX   ; BX := x
add AX, AX   ; AX := 2x
add AX, AX   ; AX := 4x
add AX, BX   ; AX := 5x
add AX, AX   ; AX := 10x
```

Performance: Multiword Multiply (cont’d)

- Multiplication of $2^{32} - 1$ by 10

![Graph showing execution time for multiplication](image-url)