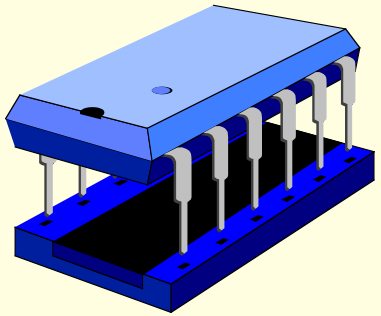
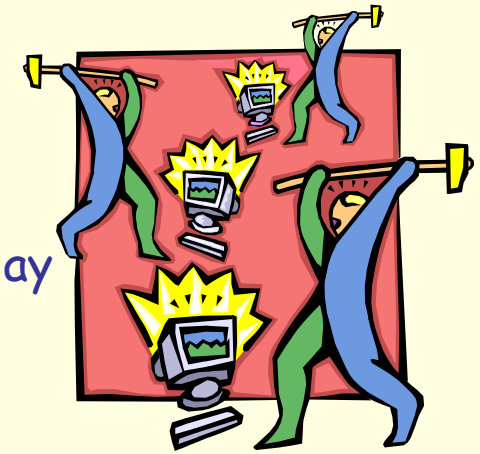
Pentium Architecture:

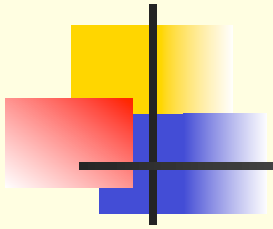
Introductory Programming

Kin K. Leung
kin.leung@imperial.ac.uk
www.commsp.ee.ic.ac.uk/~kkleung/



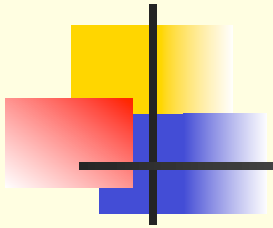
Heavily based on materials by Naranker Dulay





Topics

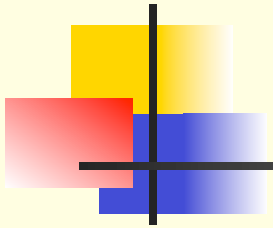
- Expressions
- Overflow and Divide by Zero
- Booleans & Comparison
- If Statements & Loops
- **Methods**
  - Parameter Passing
  - Local Variables



Integer Addition & Subtraction

Instruction	Operation	Notes
`add dst, src`	`dst = dst + src`	add
`sub dst, src`	`dst = dst - src`	subtract
`cmp dst, src`	`dst - src`	compare & set flag bits
`inc opr`	`opr = opr + 1`	increment by 1
`dec opr`	`opr = opr - 1`	decrement by 1
`neg opr`	`opr = - opr`	negate

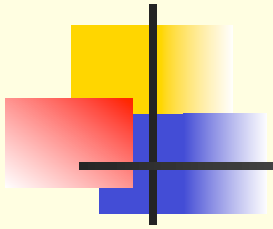
- Operands can be **byte**, **word** or **doubleword** sized
- Arithmetic instructions also set flag bits, e.g. the **zero flag** (zf), the **sign flag** (sf), the **carry clag** (cf), the **overflow flag** (of). Flags are used by branching instructions.



Integer Multiply

Instruction	*Operation*
`imul destreg, srcopr`	*destreg = destreg * srcopr*
`imul destreg, srcreg, immediate`	*destreg = srcreg * immediate*
`imul destreg, memopr, immediate`	*destreg = memopr * immediate*

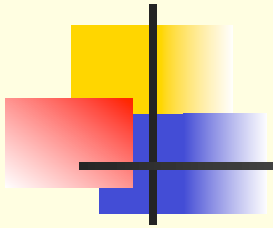
Operands can be **word** or **doubleword** sized



Integer Divide

Instruction	Operation	Notes
`idiv opr`	`al = ax / opr` `ah = ax mod opr`	Word/Byte
	`ax = (dx:ax) / opr` `dx = (dx:ax) mod opr`	Doubleword/Word
	`eax = (edx:eax) / opr` `edx = (edx:eax) mod opr`	Quadword/ Dword

Operands must be registers or memory operands



More Instructions

Instruction	Operation	Notes
`sal dst, n`	`dst = dst * 2^n`	`shift arithmetic left`
`sar dst, n`	`dst = dst / 2^n`	`shift arithmetic right`

`sal` and `sar` are quick ways of multiplying/dividing by powers of 2 where n must be a constant (immediate value) or the byte register `cl`.

```
cbw             ax = al          convert byte to word
cwde            eax = ax         convert word to doubleword
cdq             edx:eax = eax    convert double to quadword
```

`cbw`, `cwde` & `cdq` extend a signed integer by filling the extra bits of destination with the sign bit of the operand (i.e. preserve value of result)

Expressions

```
int alpha=7, beta=4, gamma=-3          /* global variables */

alpha = (alpha * beta + 5 * gamma) * (alpha - beta)
...
```

- In this example we'll represent integers as 16-bit 2's complement values and use global variable and direct addressing.

```
alpha           dw      7
beta    dw      4
gamma   dw      -3
```

Example

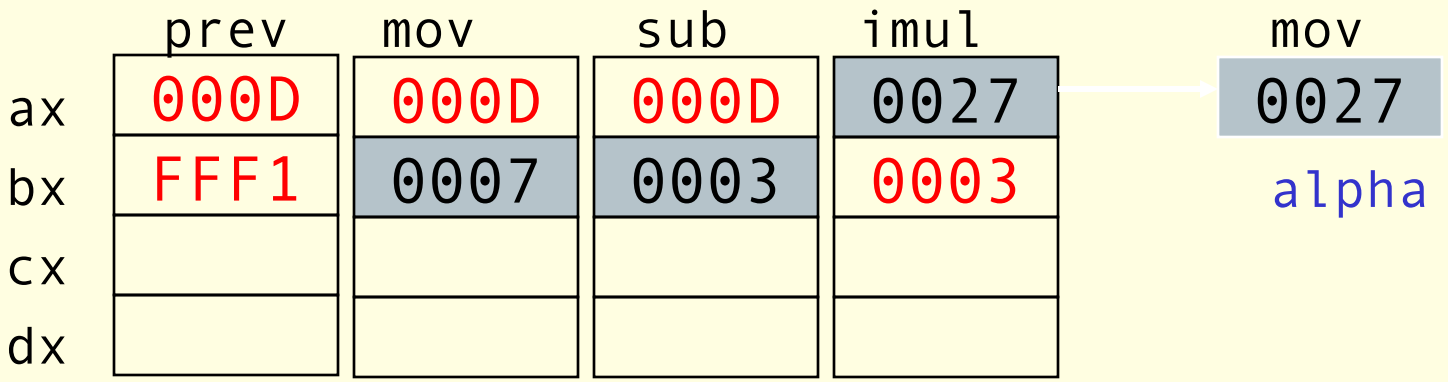
```
; int alpha = 7;  beta = 4; gamma = -3
; alpha = (alpha * beta + 5 * gamma) * (alpha – beta)
mov  ax, [alpha]        ; ax = alpha
imul ax, [beta] ; ax = alpha * beta
mov  bx, 5              ; bx = 5
imul bx, [gamma]        ; bx = 5 * gamma
add  ax, bx             ; ax = 5 *  gamma + alpha * beta
```

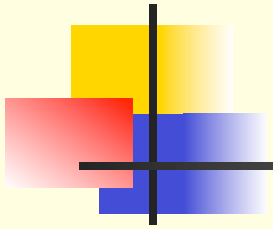
	mov	imul	mov	imul	add
ax	0007	001C	001C	001C	000D
bx			0005	FFF1	FFF1
cx					
dx					

regs shown in hex

Example Continued

```
; alpha = 7;  beta = 4; gamma = -3
; alpha = (alpha * beta + 5 * gamma) * (alpha – beta)

mov  bx, [alpha]        ; bx = alpha
sub  bx, [beta] ; bx = alpha – beta
imul ax, bx             ; ax = ax * (alpha-beta)

mov  [alpha], ax        ; alpha := final value
```





Integer Overflow

- Most arithmetic operations can produce an overflow, for example for signed byte addition

  if A + B > 127 or A + B < -128

- Instructions which result in an overflow set the overflow flag in the `eflags` register, which we can test, e.g.

Overflow Test

```
        add ah, bh    ; add, will set FLAGS.OF on overflow

        jo  ov_label  ; jump to ov_label if overflow
        ...
ov_label:             ; handle overflow condition somehow?
```

Integer Divide by Zero

- Another erroneous condition is division by zero which causes an interrupt to occur (we will cover interrupts later in the course).
- We can guard against this occurring by explicitly checking the divisor before division, e.g.

Divide by Zero Test

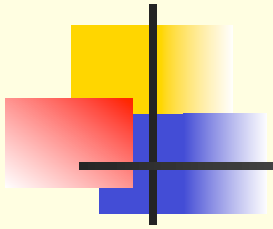
```
            cmp     bh, 0   ; compare divisor with zero
            je      zero_div        ; jump if (divisor) is equal to
zero
            idiv    bh      ; else perform division

zero_div:   ....    ; handle divide by zero somehow?
```

"LOGICAL" (Bit-level) Instructions

Instruction	Operation	Notes
`and dst, src`	`dst = dst & src`	bitwise and
`test dst, src`	`dst & src`	bitwise and, also set flags
`or dst, src`	`dst = dst | src`	bitwise or
`xor dst, src`	`dst = dst ^ src`	bitwise xor
`not opr`	`opr = ~ opr`	bitwise not

Typical Uses

`and` is used to **clear specific bits** (the 0 bits in src) in dst.
`or` is used to **set specific bits** (the 1 bits in src) in dst.
`xor` is used to **toggle/invert specific bits** (the 1 bits in src) in dst.
`test` is used to **test specific bit patterns**.



Booleans

- We'll use a full byte to represent a boolean value with the following interpretation:

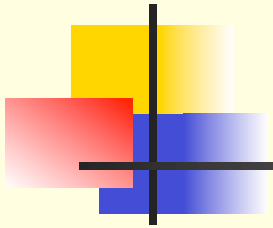
  False = 0, True = Non-Zero

- **boolean** man, rich, okay

  ...

  ; okay = (man && rich) || (! man)

```
mov  al, [man]    ; al = man
and  al, [rich]         ; al = man && rich
mov  ah, [man]    ; ah = man
not  ah           ; ah = not man
or   al, ah             ; al = (man && rich) || ! man
mov  [okay], al   ; okay = al
```



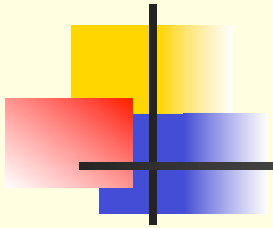
JUMP Instructions

Jump instructions take the form **OPCODE label**, e.g. JGE next

Opcode	Flag Conditions	Notes
jmp	unconditional	jump
je or jz	zf = 1	jump if equal or jump if zero (==)
jne or jnz	zf = 0	jump if not equal or jump if not zero

Signed Comparisons

jg	zf = 0 and sf = 0	jump if greater than (>)
jge	sf = 0	jump if greater than or equal (>=)
jl	sf = 1	jump if less than (<)
jle	zf = 1 or sf = 1	jump if less than or equal (<=)



More JUMP Instructions

Unsigned Comparisons

```
ja    zf = 0 and cf = 0        jump if above ( > )
jae   cf = 0                   jump if above or equal ( >= )
jb    cf = 1                   jump if below ( < )
jbe   zf = 1 or cf = 1         jump if below or equal ( <= )
```

Miscellaneous

```
jo    of = 1                   jump if overflow
jno   of = 0                   jump if no overflow, ditto
for ...
```

If Statement

```
if (age < 100) {
    statements
}
```

```
if: cmp word[age],100
       jl   stats
       jmp  endif
stats:
       ; statements
endif:
```

```
if: cmp word[age],100
       jge  endif
       ; statements
endif:
```

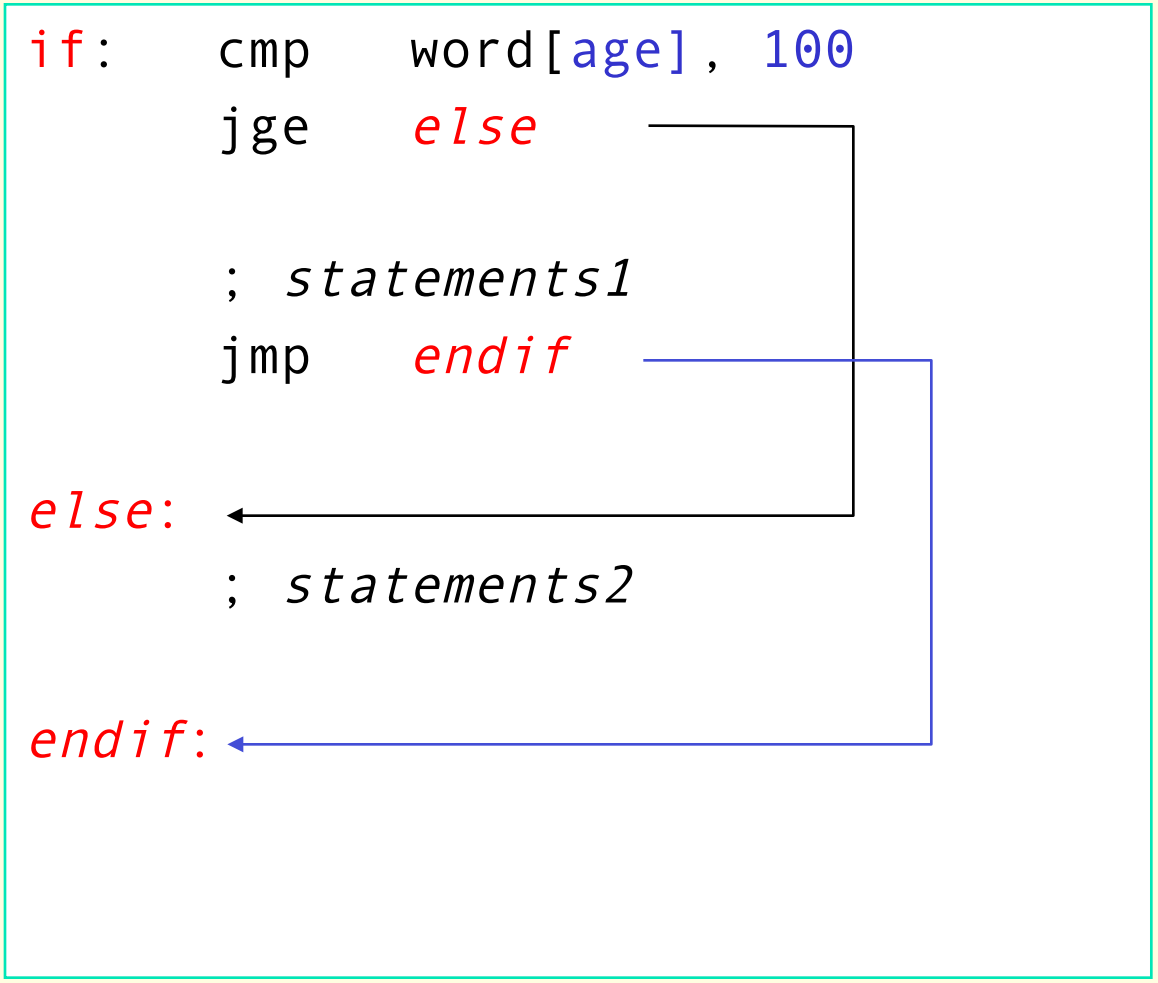
If Statement Contd

```
if (age >= 21) && (age <= 65) {
    statements
}
```

```
if:     cmp     word[age], 21
        jl      endif
        cmp     word[age], 65
        jg      endif

        ; statements
endif:
```

IF-Then-Else Statement

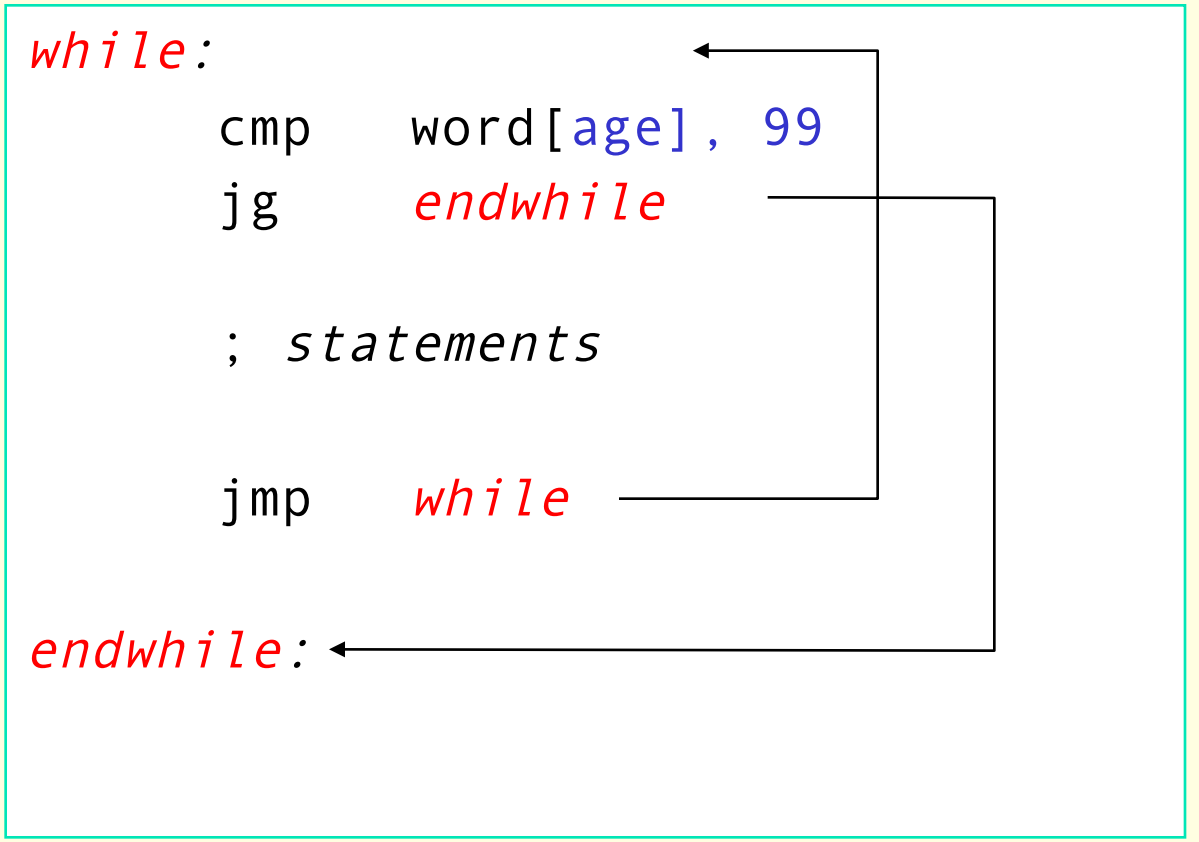
```
If (age < 100) {
    statements1
} else {
    statements2
}
```



While Loop

```
while (age <= 99) {
    statements
}
```

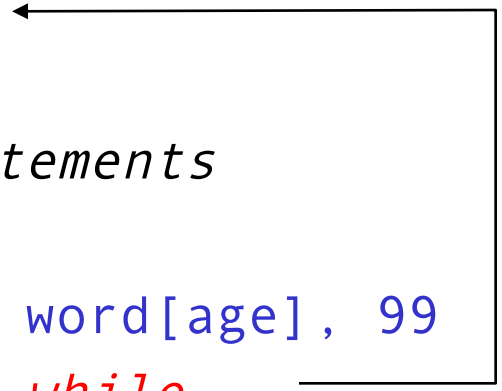
```
while:
        cmp     word[age], 99
        jg      endwhile

        ; statements

        jmp     while

endwhile:
```



Do-While Loop

```
do {
    statements
} while (age <= 99)
```

```
while:

        ; statements

        cmp    word[age], 99
        jle    while

endwhile
```



For Loop

```
for (age = 1; age<=99; age++)
    statements
}
```

```
for:   mov     word[age], 1

next:
       cmp     word[age], 99
       jg      endfor

       ; statements

       inc     word[age]
       jmp     next

endfor:
```